

# Scaling Google Cloud Platform

Run Workloads Across Compute, Serverless PaaS, Database,  
Distributed Computing, and SRE



Swapnil Dubey

# Scaling Google Cloud Platform

Run Workloads Across Compute, Serverless PaaS, Database,  
Distributed Computing, and SRE

The lower half of the book cover features a network diagram with a central Google Cloud logo (a cloud shape divided into four colored segments: red, yellow, green, and blue). The logo is surrounded by a web of grey nodes connected by thin lines. Various colored dots (red, yellow, blue, green) are scattered throughout the network. In the bottom left corner, there is a small red rectangular box containing the author's name. The background of this section is white with faint, light grey diagonal lines. The top left and bottom right corners of the entire cover are decorated with abstract, overlapping shapes in shades of purple and magenta.

Swapnil Dubey

# Scaling Google Cloud Platform

---

*Run Workloads Across Compute,  
Serverless PaaS,  
Database, Distributed Computing,  
and SRE*

---

**Swapnil Dubey**



[www.bpbonline.com](http://www.bpbonline.com)

Copyright © 2023 BPB Online

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

**Group Product Manager:** Marianne Conor

**Publishing Product Manager:** Eva Brawn

**Senior Editor:** Connell

**Content Development Editor:** Melissa Monroe

**Technical Editor:** Anne Stokes

**Copy Editor:** Joe Austin

**Language Support Editor:** Justin Baldwin

**Project Coordinator:** Tyler Horan

**Proofreader:** Khloe Styles

**Indexer:** V. Krishnamurthy

**Production Designer:** Malcolm D'Souza

**Marketing Coordinator:** Kristen Kramer

**First published:** 2023

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

**UK | UAE | INDIA | SINGAPORE**

ISBN 978-93-55512-840

[www.bpbonline.com](http://www.bpbonline.com)

# **Dedicated to**

*Two strong ladies in my life*

*My Mom **Sushma***

*My Wife **Vartika***

*&*

*My Kids – **Sayasha & Stavya***

# About the Author

The author of the book is **Swapnil Dubey**. He has a Master's Degree in Data Analytics, along with a total of 14 years of experience in various domains such as eCommerce, Ad Serving, Oil & Gas, and BFSI, using technologies like Kubernetes, Apache Spark, and Apache Airflow. He has also spoken at multiple national and international conferences, both in corporate as well as in academia. He is also a certified Professional Architect for Google Cloud and Microsoft Azure. Dubey currently hold the position of Cloud Architect, at Schlumberger India Technology Center – Pune.

# About the Reviewer

**Peeyush Maharshi** is seasoned enterprise architect with thought leadership and diverse background.

He provides technical leadership on architecture and engineering standards through reference architectures and best practices. Peeyush is involved in bootstrapping a world class engineering and application development team, that is focused on lean engineering practices and cloud integration/migration, and building customer-centric solutions using his expertise in Technology, Enterprise Architecture, Data Strategy and Engineering.

Peeyush has extensive experience on large cloud migration execution on AWS, Azure and GCP, using factory-based model delivery, including data discovery, assessment and disposition strategy finalization, actual migration execution with adoption of DevOps principles and applying different modernization themes.

He has working experience with USA, Europe, Middle East and Australian customers. Peeyush has also played the role of technical reviewer for many books such as “The Java EE Architect Handbook” 2nd Ed., “Microservices for Java Architects” and so on.



# Acknowledgement

Any accomplishment requires the effort of many people, and this work is no different. First and foremost, I would like to thank my family (especially my father figure, mentor and guardian, Mr. N.R. Tiwari) for continuously encouraging and supporting me in writing the book — I could have never completed this book without their support.

I am grateful to the Courses and the Enterprises which gave me support throughout the learning and understanding process of the Google Cloud Platform. Thank you for all the hidden support provided. I gratefully acknowledge Mr. Peeyush Maharshi for his kind technical scrutiny of this book.

My gratitude also goes to the team at BPB Publications for being supportive and patient during the editorial review of the book.

A big thank you to the Tech Center Manager (Rashmi Kumat), Tech Center HR Manager (Shamecka Ferrnandis) and Intellectual Property team of Schlumberger for providing the clearance to publish the book.

# Preface

The book will cover the basic building blocks of cloud scaling in general, and will give good insights into some key indicators using which, enterprises can start their journey and measure success on the cloud. It also covers the right heuristics to measure cost and the need for governance to control changes. This book maps a lot of theoretical jargon in the industry with real-world examples and scenarios.

The book will describe each GCP component that is scalable on the Google Cloud Platform, describe the architecture and internal details of components with the intention of giving the audience the feel of potential areas inside components (managed offering from GCP) where scaling makes sense, and will eventually go ahead to describe them in depth with sufficient real-world scenarios, theory, and hands-on content. All aspects of manual scaling, predictive scaling, and autoscaling are covered wherever applicable.

This book has a section on the SRE practices in the industry and how these industry practices can be implemented and hosted on Google Cloud. It covers the out-of-the-box as well as custom scenarios of the SRE world. At the end, you will investigate the two most common architectures – Microservices and Bigdata – and will see how you can do reliability engineering for them on GCP.

This book is divided into **16 chapters**. The details are listed as follows:

**Chapter 1, Basics of Scaling Cloud Resources:** This chapter will give users an introduction to concepts of scaling infrastructure resources in the cloud world. It will be a generic chapter (not specific to GCP), which will introduce concepts and terminologies used across the book.

**Chapter 2, KPI for Cloud Scalability:** This chapter will talk about the KPIs one should look into, to identify when and in what amount to scale. Breach of any of these metrics means a high risk of not meeting the SLAs.

**Chapter 3, Cloud Elasticity:** This chapter will introduce the audience to the term cloud elasticity. It will introduce key elasticity parameters to be taken into consideration, while designing workloads for the cloud.

**Chapter 4, Challenges of Infrastructure Complexity and the Way Forward:** This chapter will introduce to the audience the infrastructural complexity and its challenges. It will make the reader think about the importance of strict governance needed while migrating or developing the workloads on the cloud.

**Chapter 5, Scaling Compute Engine:** This chapter describes the ins and outs of scaling GCP compute VMs.

**Chapter 6, Scaling Google Kubernetes Engine:** This chapter will help the audience understand the concepts involved in scaling container workloads.

**Chapter 7, Scaling VMware Engine:** This section will educate audiences on strategies to scale the VMware Engine cloud offering of GCP.

**Chapter 8, Scaling App Engine:** This chapter will enable readers to understand in and out of the App engine infrastructure footprint, and how they can configure the App engine to scale up and down in a cost-effective manner.

**Chapter 9, Scaling Google Cloud Function and Cloud Run:** This section will introduce to the audience at a high level, the workload expected to be handled by these components and will build an expert-level understanding of how to configure scalability aspects.

**Chapter 10, Configuring Bigtable for Scale:** This chapter will introduce the concepts required for deciding

and configuring the right parameters for scaling Big Table. The chapter will start by describing the workload that Bigtable handles along with the related architecture. These 2 explanations will be used in further topics.

**Chapter 11, Configuring Cloud Spanner for Scale:** This section will introduce the readers to details of scaling of cloud spanner instances. The chapter will start by quickly describing the right use case for Spanner and describing the infrastructural footprint (architecture). These will be used as key aspects while discussing the rest of the topics.

**Chapter 12, Scaling Google Composer 2:** This chapter will help the audience understand the scalability aspects of Google composer 2. The chapter will start with one basic introduction to Google composer showing one workflow. The same workflow will be used to showcase the rest of the headings in the chapter.

**Chapter 13, Scaling Google Dataproc:** This chapter introduces the nuances, challenges and solutions to scale the Dataproc cluster for big data computer workloads written in Hadoop or Spark. The chapter will start with an introduction to Dataproc, with one simple Spark job code explanation. The same job will be used for the rest of the headings as well.

**Chapter 14, Scaling Google Dataflow:** This chapter introduces the concepts of Google dataflow scaling aspects for production-grade deployments and workloads. The structure of the chapter remains the same with an introduction to an example, uniformly used across the rest of the topics.

**Chapter 15, Site Reliability Engineering:** This chapter will introduce the concepts of site reliability engineering on GCP. Apart from end-to-end technical implementation, the behavioral aspects of the SRE will also be described.

**Chapter 16, SRE Use Cases:** This chapter will present examples/use cases of what is presented in the preceding chapter.

# Code Bundle and Coloured Images

Please follow the link to download the **Code Bundle** and the **Coloured Images** of the book:

**<https://rebrand.ly/ec456b>**

The code bundle for the book is also hosted on GitHub at **<https://github.com/bpbpublications/Scaling-Google-Cloud-Platform>**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **<https://github.com/bpbpublications>**. Check them out!

## Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**[errata@bpbonline.com](mailto:errata@bpbonline.com)**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.bpbonline.com](http://www.bpbonline.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: [business@bpbonline.com](mailto:business@bpbonline.com) for more details.

At [www.bpbonline.com](http://www.bpbonline.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

## **Piracy**

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [business@bpbonline.com](mailto:business@bpbonline.com) with a link to the material.

## **If you are interested in becoming an author**

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit [www.bpbonline.com](http://www.bpbonline.com). We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## **Reviews**

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit [www.bpbonline.com](http://www.bpbonline.com).





# Table of Contents

## **1. Basics of Scaling Cloud Resources**

Introduction

Structure

Objectives

What is cloud scalability?

*Horizontal scalability (Scale up and down)*

*Vertical scalability (Scale in and out)*

*Auto scalability*

*Diagonal scaling*

Benefits of cloud scaling

*Flexibility and speed*

*Ease of use and maintenance*

*Cost saving*

*Disaster Recovery*

*Global presence*

When to scale?

*Scenario 1*

*Scenario 2*

*Scenario 3*

How to scale?

*Manual scaling*

*Scheduled scaling*

*Automatic scaling*

Key challenges of scaling

*Cloud native and hybrid deployments*

*Load balancing*

*Housekeeping services*

Scale versus cost relationship

Risks of improper scaling

Conclusion

## **2. KPI for Cloud Scalability**

Introduction

Structure

Objectives

Defining KPIs

Basic cloud scalability metrics

Performance

Use case 1: Big data

Use case 2: Microservices

Use case 3: REST API

Reliability

Mean time to failure

Mean time to repair

Rate Of Occurrence Of Failure

Probability Of Failure On Demand

Costs

Total cloud cost

Forecasted cost

Availability

Indirect KPI impact of cloud scalability

Innovations

Software development and operational KPIs impacts

Customer satisfaction

Advanced metrics

Response time

Data in and out

Request per second

Average response time

Peak response time

Infrastructure utilization

Latency

Average end-to-end latency

Number of slow end-to-end transactions

Number of very slow end to end latency times

Throughput

Conclusion

Points to remember  
Questions

### **3. Cloud Elasticity**

Introduction

Structure

Objectives

Defining cloud elasticity.

Example 1

Example 2

Example 3

Example 4

Benefits of elasticity.

Painless and optimal scaling.

Justified costs

More redundancy and flexibility

Considerable capacity

High availability

Simple management

Elasticity and cost relationship

Key challenges

Identifying the right attributes/metrics to track

Identifying the right scaling measurement value

Defining the minimum and maximum limits

Cost spikes

Difference between scalability and elasticity.

Use cases

eCommerce application

Song streaming application

Conclusion

Points to remember

Questions

Answers

## **4. Challenges of Infrastructure Complexity and the Way Forward**

Introduction

Structure

Objectives

Defining multi-cloud and hybrid-cloud deployments

Redundant deployments

Hybrid environment

Business continuity multi-cloud or hybrid cloud storage

Cloud bursting

Distributed deployments

Tiered hybrid

Partitioned multi-cloud

Analytics hybrid/multi-cloud

Multi-cloud deployment model

Hybrid-cloud deployment model

Need of multi-cloud deployments and hybrid-cloud deployments

Reducing dependency/avoiding lock in

Heterogeneous deployments within an organization

Regulatory and data sovereignty

Redundant deployment for high availability

Performance improvements

Cost optimization

Challenges of multi-cloud deployments and hybrid cloud deployments

Increased operational complexities

Increased data management complexities

Data protection challenges

Increased architectural complexities

IaaS vs PaaS for multi-cloud and hybrid cloud deployments

Redundant deployment

Distributed deployments

IaaS vs PaaS

[Docker and Kubernetes](#)  
[Hadoop cluster](#)  
[OpenShift Container Platform](#)  
[Infrastructure as code](#)  
[Governance and way out](#)  
[Creating guidelines](#)  
[Effective communication](#)  
[Effective planning.](#)  
[Proper auditing.](#)  
[Cloud agnostic automation - benefits and risks](#)  
[Conclusion](#)  
[Points to remember](#)  
[Questions](#)

## **5. Scaling Compute Engine**

[Introduction](#)  
[Structure](#)  
[Objectives](#)  
[Interacting with GCP](#)  
[Using the console\UI portal](#)  
[Using GCloud commands](#)  
[REST APIs](#)  
[Introduction to instance groups](#)  
[Managed Instance Group](#)  
[Creating a Managed Instance Group](#)  
[Unmanaged instance group](#)  
[Autoscaling groups of VMs](#)  
[Scaling](#)  
[Autoscaling.](#)  
[Predictive scaling](#)  
[Scale-in controls](#)  
[Maximum Allowed Reduction](#)  
[Trailing Time Window](#)  
[Autoscaling in Action](#)  
[Developing and managing autoscalers](#)

[Scaling Based on CPU utilization](#)  
[Scaling based on load balancing serving capacity](#)  
[Scaling Based on cloud monitoring metrics](#)  
[Configuring auto scaling for per instance metric](#)  
[Configuring auto scaling for per group metric](#)  
[Scaling based on schedules](#)  
[Scheduling based on prediction](#)  
[Creating autoscaling policy based on multiple signals](#)  
[CRUD operations on autoscalers](#)  
[Describing an Autoscaler](#)  
[Updating a scalar](#)  
[Turning off a scalar](#)  
[Deleting an autoscaler](#)  
[Autoscaling node groups](#)  
[Reserving resources for effective auto scaling](#)  
[Single project zonal reservations](#)  
[Shared project zonal reservations](#)  
[Consuming reservations](#)  
[Consuming instances from any matching reservation](#)  
[Consuming a specific shared reservation](#)  
[Creating instances without consuming reservations](#)  
[Load balancing](#)  
[Adding instance group to load balancer](#)  
[Aligning backend service with an MIG](#)  
[Adding a Managed Instance Group to a target pool](#)  
[Configuring multi regional external load balancer](#)  
[Cross regions load balancing](#)  
[Conclusion](#)  
[Points to remember](#)  
[Questions](#)  
[Answers](#)

## **6. Scaling Kubernetes Engine**

[Introduction](#)

Structure

Objectives

Building and packaging an application on Kubernetes

Kubernetes architecture

Building and deploying a web app

Scaling an application

Scale-up mechanism

Scale-down mechanism

Configuring horizontal pod scaling.

Metric threshold definition

Configuring multiple metrics

Thrashing.

Issuing horizontal scaling requests

Autoscaling on resource utilization

Autoscaling on external metric

Autoscaling on custom metric

Configuring vertical pod scaling.

Configuring multi-dimensional pod scaling.

Exponential scaling of fault tolerant workloads

Using Spot Pods

Using Spot VMs

Using preemptible VMs

Cluster autoscaler

Scaling limits

Key considerations

Node pool configurations

Network policies for scale

Load balancing.

Storage

Conclusion

Points to remember

Questions

## **7. Scaling VMware Engine**

Introduction



[Structure](#)  
[Objectives](#)  
[VMware Engine](#)  
[Creating a private cloud](#)  
[Configuring autoscaling policies](#)  
[Conclusion](#)  
[Points to remember](#)  
[Questions](#)  
[Answers](#)

## **8. Scaling App Engine**

[Introduction](#)  
[Structure](#)  
[Objectives](#)  
[App Engine under the hood](#)  
[Standard App Engine vs. flex App Engine](#)  
[Standard App Engine](#)  
[\*Configuring basic scaling\*](#)  
[\*Configuring manual scaling\*](#)  
[\*Configuring autoscaling scaling\*](#)  
[Flex App Engine](#)  
[\*Configuring manual scaling\*](#)  
[\*Configuring autoscaling\*](#)  
[Conclusion](#)  
[Points to remember](#)  
[Questions](#)  
[Answers](#)

## **9. Scaling Google Cloud Function and Cloud Run**

[Introduction](#)  
[Structure](#)  
[Objectives](#)  
[Cloud Run](#)  
[\*Nature of workloads\*](#)  
[\*Infrastructural footprint\*](#)

[Autoscaling Container Instances](#)

[Configuring CPU allocation](#)

[Configuring maximum concurrency](#)

[Configuring minimum and maximum Container Instances](#)

[Cloud Functions](#)

[Nature of workloads](#)

[Configuring memory](#)

[Configuring maximum and minimum instances](#)

[Addressing traffic spikes above max limits](#)

[Conclusion](#)

[Points to remember](#)

[Questions](#)

## **[10. Configuring Bigtable for Scale](#)**

[Introduction](#)

[Structure](#)

[Objectives](#)

[Nature of data and its handling](#)

[Voluminous datasets](#)

[High throughput](#)

[Fast writes](#)

[Versioning changes](#)

[Strong consistency](#)

[Atomic writes](#)

[Selection of data](#)

[Bigtable infrastructural footprint](#)

[Scaling Bigtable options](#)

[Autoscaling triggers](#)

[Autoscaling](#)

[When to Autoscale](#)

[Manual node allocation](#)

[Programmatically Autoscaling](#)

[Limitations of Autoscaling](#)

[Conclusion](#)

Points to remember

Questions

Answers

## **11. Configuring Cloud Spanner for Scale**

Introduction

Structure

Objectives

Nature of workload

Cloud Spanner infrastructural footprint

Manual scaling

Autoscaling using Autoscaler

Autoscaler Architecture

Cloud scheduler

Poller cloud function

Scaler Cloud Function

End to end working.

Autoscaler deployment topology.

Deployment of Autoscaler per project

Centralized deployment topology.

Distributed deployment

Scaling strategies as per load

Stepwise scale up

Linear scale up

Direct scale up

Conclusion

Points to remember

Multiple choice questions

Answers

## **12. Scaling Google Composer 2**

Introduction

Structure

Objectives

Introduction to Composer

Options for horizontal scaling

Adjusting minimum and maximum number of workers

Adjusting number of schedulers

Options for vertical scaling

Adjusting worker, scheduler, web server scale and performance parameters

Adjusting environment size

Composer Autoscaling

Role of Airflow worker set controller

Factors affecting Composer autoscaling.

Composer Autoscalars

Horizontal Pod scalar

Cluster Autoscalar

Node auto provisioning.

Optimizing the Airflow environment

Start with environment pre-set

Run your DAGs

Observe the environment

Monitoring the scheduler CPU and memory.

Monitoring total parse time of DAGs

Monitoring worker Pod evictions

Monitoring active workers

Monitoring workers CPU and memory usage

Monitoring running and queued tasks

Monitoring the database CPU and memory usage

Monitoring the task scheduling latency.

Monitoring web server CPU and memory.

Commands to perform the preceding changes

Conclusion

Points to remember

Questions

Answers

## **13. Scaling Google Dataproc**

Introduction

[Structure](#)  
[Objectives](#)  
[Introduction to Dataproc](#)  
[Manual scaling](#)  
[Auto scaling](#)  
[Autoscaling deep dive](#)  
[Introducing Autoscaling Policies API](#)  
[Autoscaling policy resource](#)  
[BasicAutoscalingAlgorithm Resource](#)  
[BasicYarnAutoscalingConfig Resource](#)  
[InstanceGroupAutoscalingPolicyConfig Resource](#)  
[CRUD on Autoscaling policies](#)  
[Applying Autoscaling policies to Dataproc cluster](#)  
[Limitations of scale](#)  
[Graceful decommissioning of clusters](#)  
[Using preemptible VMs to scale](#)  
[Conclusion](#)  
[Points to remember](#)  
[Questions](#)  
[Answers](#)

## **14. Scaling Google Dataflow**

[Introduction](#)  
[Structure](#)  
[Objectives](#)  
[Introduction to Dataflow](#)  
[Apache Beam pipelines](#)  
[Wordcount Dataflow job](#)  
[Fusion optimization](#)  
[Combine optimizations](#)  
[Dataflow Shuffle service](#)  
[Dataflow streaming engine](#)  
[Dataflow Prime](#)  
[Configuring infrastructure](#)  
[Disk size](#)

[Machine type](#)  
[Disabling public IPs](#)  
[Selecting right regions](#)  
[Dataflow job lifecycle](#)  
[Distribution and parallelization](#)  
[Execution graph](#)  
[Combining optimizations](#)  
[Fusion optimization](#)  
[Dataflow autotuning](#)  
[Horizontal autoscaling](#)  
[Scaling Dataflow for batch jobs](#)  
[Scaling Dataflow for streaming jobs](#)  
[Horizontal scaling of streaming pipeline](#)  
[Vertical auto scaling](#)  
[Dynamic work rebalancing](#)  
[Autoscaling algorithms](#)  
[NONE](#)  
[BASIC](#)  
[THROUGHPUT\\_BASED](#)  
[Scaling and Dataflow Prime right fitting](#)  
[Limiting max nodes](#)  
[Scaling the persistent disk](#)  
[Optimizing data shuffle using Dataflow shuffle](#)  
[Conclusion](#)  
[Points to remember](#)  
[Questions](#)  
[Answers](#)

## **15. Site Reliability Engineering**

[Introduction](#)  
[Structure](#)  
[Objectives](#)  
[Introduction to SRE process](#)  
[Defining a typical SRE process](#)  
[Defining SLO, SLI and SLAs](#)

- [Service Level Objectives \(SLO\)](#)
- [Service Level Indicators \(SLI\)](#)
- [Service Level Agreement \(SLA\)](#)
- [Service monitoring using Google Cloud Monitoring](#)
- [Selecting metrics for SLIs](#)
  - [Using the out of box SLI metrics](#)
  - [Log based metrics](#)
  - [Key SLIs](#)
- [Setting SLO](#)
  - [Creating SLIs](#)
  - [Creating SLO](#)
- [Tracking error budgets](#)
- [Creating alerts](#)
- [Probes and uptime checks](#)
- [Aggregating logs to set up cloud monitoring dashboard](#)
- [Responsibilities of SRE](#)
  - [Incident management](#)
  - [Playbook maintenance](#)
  - [Drills](#)
- [Automating SRE actions](#)
- [Conclusion](#)
- [Points to remember](#)
- [Questions](#)
  - [Answers](#)

## **16. SRE Use Cases**

- [Introduction](#)
- [Structure](#)
- [Objectives](#)
- [GCP service grouping](#)
  - [Request response services](#)
  - [Data storage and retrieval services](#)
  - [Data processing services](#)
- [SRE practices in the microservices world](#)
  - [Availability](#)

*Latency*

SRE practices big data world

*Correctness SLI*

*Freshness SLI*

*Coverage as an SLI*

*Throughput as an SLI*

Conclusion

Points to remember

Questions

*Answers*

**Index**



# CHAPTER 1

## Basics of Scaling Cloud Resources

### Introduction

In this chapter, we will look into some key concepts for scaling infrastructure on the cloud. The concepts mentioned here generally apply to all major public cloud providers, such as, **Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)** and **Microsoft Azure** platform, although with some strategic differences in varying offerings. We will deep dive into the what, why, and how of cloud scalability and expand the discussion to challenges, risks, and costs associated with scaling.

The concepts discussed here will act as building blocks of our future chapters.

### Structure

In this chapter, we will discuss the following topics:

- What is cloud scalability?
- Benefits of cloud scaling
- When to scale?
- How to scale?
- Key challenges of scaling
- Scale versus cost relationship
- Risks of improper scaling

## **Objectives**

This chapter will look into the purpose and need of having a workload on the cloud. We will also look into the types, as well as benefits of hosting workloads in cloud platforms. When we develop applications, it is essential to look into the scalability aspects, because when these applications are hosted in the cloud, they bring new types of challenges. If this is not done correctly, there are severe cost implications. We will then look into some real-world scenarios, to understand the must-haves for cloud scalability. At the end of this chapter, the audience will understand the mentioned aspects conceptually and will be able to apply this to any cloud platform implementation.

## **What is cloud scalability?**

Cloud scalability refers to the capability of scaling up and down/scaling in and out of infrastructure needs (compute, storage and network needs) of applications, deployed in the cloud, based on changing demands. Scalable infrastructure is one of the key driving factors for organizations to adopt cloud platforms for their end-to-end digital transformation journeys. The ability to handle the sudden spike in data, experimentation with new technology, as well as commissioning and decommissioning of infrastructure, are key benefits which support today's agile way of developing software.

In yesteryears, when the applications were deployed primarily on-premises, increasing the infrastructure was not a trivial activity. It involved multiple teams - Software teams to raise requests for more infra, management team to approve, and IT infra team to place orders. Finally, when the hardware resources are delivered, it has to integrate with the rest of the hosted infrastructure. This whole cycle of making more resources available for a software application,

has multiple parties involved, who use their own time to complete the processes. This risk associated with such a cycle can be mitigated up to some level by proper planning.

Maintaining an on-prem infrastructure brings in a lot more maintenance responsibilities for organizations, and maintenance comes with due costs. A more significant infrastructure needs more manpower to support it, not just for normal day-to-day activities but also for a lot of energy in Disaster Recovery, security, and availability and reliability.

**Disaster Recovery** is the ability of services you manage, to recover from data center/infrastructure going down. An inefficient disaster recovery plan will affect the availability of your application, and in case your application is generating revenue, this implies a loss of money for the organization.

**Security** means securing your infrastructure from attacks on data (encryption at rest and encryption in transit) and restricting the hackers from doing any infrastructure triggers – such as creating **Virtual Machines (VMs)**. A non-secure application will lower customers' confidence in using the services.

**Availability** of software systems is defined as the availability of your services. Without availability, there is no practical need for scaling. **Reliability** means that the application did not fail, in case of adverse conditions. For example, even if data grows multifold, the application can process (taking a long time) with accurate results.

Today, organizations believe in starting small and growing gradually. The applications are becoming more and more data-intensive, and processing involves vast datasets. However, the **Service Level Agreements (SLAs)** have not increased. Moreover, technology evolution is very fast and to pace with this evolution, an organization needs to spend significantly in innovation. A delay in performing these new

flavors of workloads, slows down the entire time to market strategy of end-to-end digital transformation journey of the enterprises.

Adopting the cloud system allows organizations to quickly scale infrastructural needs without compromising security, reliability, and availability. Over the years, these cost and scale models have matured well in all public clouds, and thus, it makes a lot of sense to decide on the adoption of models at the start of the development of the applications.

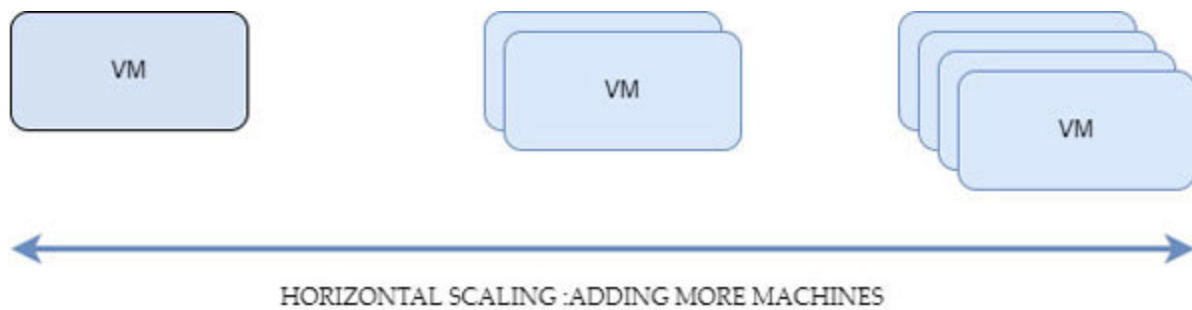
Scaling up is easy, but so is scaling down, as well. The providers offer manual scaling and auto-scaling options, making them cost-optimal for new generation workloads, which further require the system to scale up and down, based on the data spikes in workloads. All public clouds have a pay-as-you-go model for costs, which means that although the infrastructure is not required, shutting them down will also cost. Cloud platforms offer **Platform as a Service (PaaS)**, **Infrastructure as a Service (IaaS)**, and **Software as a Service (SaaS)** models for different workloads.

There are primarily 4 different strategies of scaling available: horizontal scalability, vertical scalability, auto scalability and diagonal scalability.

## **Horizontal scalability (Scale up and down)**

Horizontal scaling means adding more resources to your pool of resources. For example, if an application is deployed on a 2 vCores, 2 GB RAM, and 10 GB hard disk machines, and we need to scale it up, we will add one more device with similar capabilities. What is critical here, is that we do not modify the earlier running instance. Instead, the scale-up (as seen in [Figure 1.1](#)) is attributed to adding one new machine (scale up). The newly added machine could be of

similar or different capabilities. These new and old instances are placed behind a load balancer which result in no changes for the end users. Load balancer handles the workload by distributing the workload among the old and new instances. [Figure 1.1](#) features a diagram of horizontal scaling:



**Figure 1.1:** Horizontal scaling

This is applicable in cases where your processing needs are not expected to increase with the scale of data. That means, if we used to process 1 GB of data on one machine, and now we want to process 2 GB of data, the application developed is smart enough to divide processing responsibilities into 2 devices, making the situation similar to before, that is, 1 GB per machine.

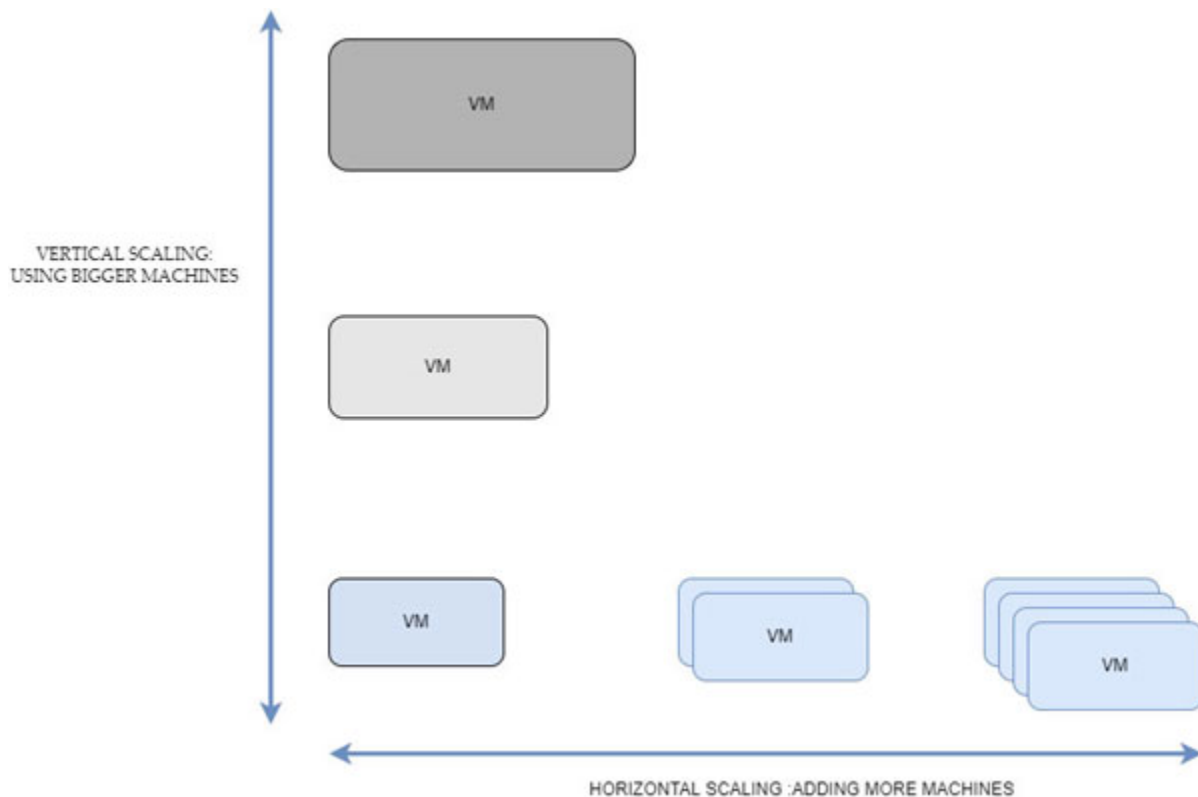
One obvious advantage of this scaling is that scaling down is also simple, with similar complexity as scaling up. Nonetheless, not every application is capable of scaling like this by default. Applications developed need to be a category of first-class functions to support this.

One famous example which fits this scaling strategy is the Hadoop and Spark jobs. In Hadoop and Spark, the data loaded into the file system **Hadoop Distributed File System (HDFS)**, is broken into blocks (HDFS blocks) of fixed size, and each block is processed in 1 virtual machine/container. So, an increase in data means more blocks and hence more containers processing it.

Another example is HTTP microservices with short response times, and decorated with load balancers. Here too, a request lifetime is small, and one request can execute independently from other requests in the system.

## **Vertical scalability (Scale in and out)**

Vertical scalability means increasing the size of machines (scale out), on which the application is hosted for scale needs. For example, if an application is deployed on a 2 vCores, 2 GB RAM, and 10 GB hard disk machines, and we need to scale it up ([Figure 1.2](#)), we will use a more giant device - 4 vCores, 4GB RAM, and 20 GB hard disk. In comparison to horizontal scaling, the pre-scaling VM has to be abandoned, and new infrastructure has to be used. [Figure 1.2](#) features a diagram for vertical scaling:



**Figure 1.2:** Vertical scaling

This is applicable in cases where your processing needs are expected to increase with the scale of data. If we used to process 1 GB data on one machine of 2 vCores, 2GB RAM, and 10 GB Hard Disk, and now we want to process 2 GB data, the application will need a device double in size, that is, with 4 vCores, 4GB RAM and 20 GB Hard Disk.

One advantage of this type of scaling is that all applications, by default, can support this. Cloud providers provide specialized type of virtual machines as per the nature of workload. The disadvantage is that there will be an upper limit to this. For example, in GCP, currently, we can have machines up to a max of 416 vCores and 28.5 GB RAM.

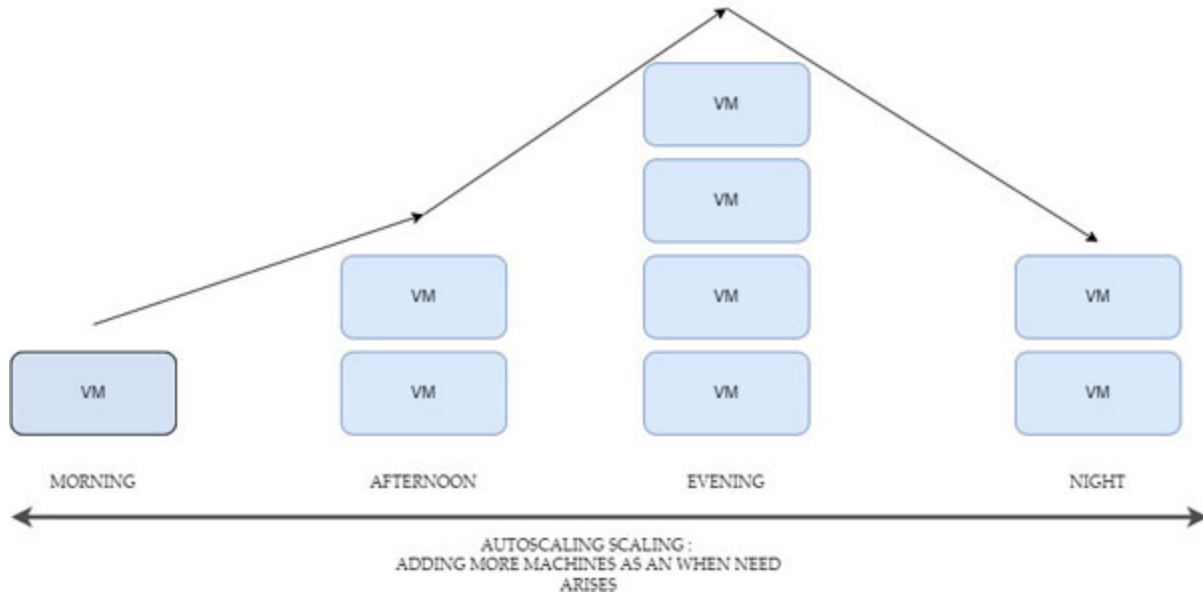
**Link:**

**<https://cloud.google.com/compute/docs/machine-types>**

An application processing a file cannot be distributed to multiple machines. For example, the processing of images to identify patterns cannot be broken and processed in parallel. In this situation, if a 1 GB file needs 2 vCores, 2GB RAM and 10 GB Hard Disk, a 2 GB file will need 4 vCores, 4 GB RAM and 20 GB Hard Disk.

## **Auto scalability**

Auto scalability as a scaling strategy is available on cloud platforms, allowing organizations to scale up or scale down applications, based on some identified parameters. For example, a web application can scale, based on the number of incoming requests. Similarly, an application acting as a subscriber to queues can scale based on the number of messages unread in a Queue. The same workflow can trigger multiple size of infrastructure; for example in [Figure 1.3](#), the same application spins 1 VM in the morning, 2 in the afternoon, 4 in the evening and 2 at night:



**Figure 1.3:** Auto scaling

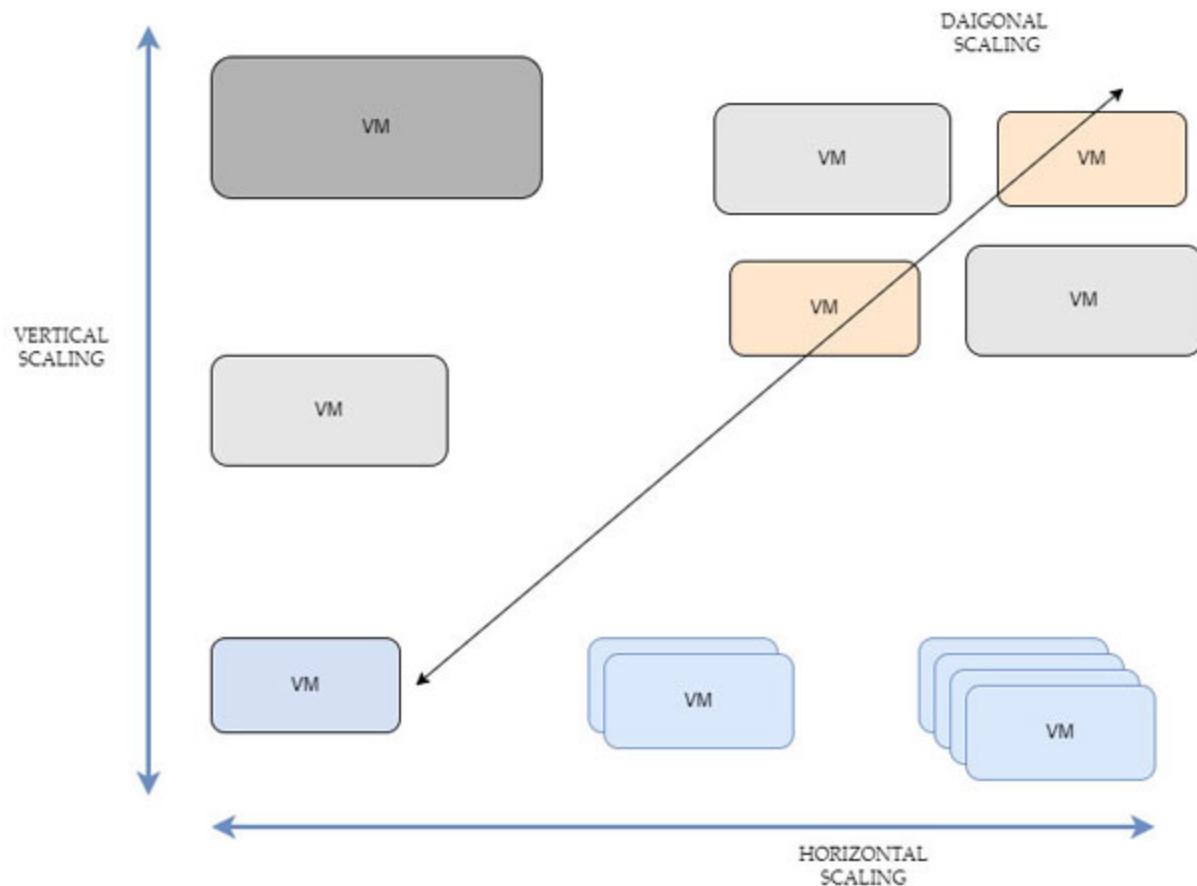
Advantages are obvious: no effort is required for maintaining the scale up and scale down of application. So, in terms of cost, it is very effective. You only pay for what you have used. However, the disadvantage is that such services provided by cloud providers are their prosperity services available for their cloud. Hence, in hindsight, it is a use case where multi-cloud or hybrid deployment will need efforts.

Most often, any application/example which supports horizontal scaling are good candidates for this scaling too.

## Diagonal scaling

Diagonal scaling combines horizontal scaling and vertical scaling (as seen in [Figures 1.1](#) and [1.2](#) respectively). It constitutes adding resources to a single server, until the server reaches maximum capability or up to a cost-effective threshold, and then adding more nodes (horizontal) in the current configuration to the deployment. [Figure 1.4](#) features diagonal scaling:





**Figure 1.4:** Diagonal scaling

This term was coined by Flickr's Operation Manager *John Allspaw*, who told how *Flickr* replaced 67 dual-CPU boxes with 18 dual quad-core machines and recovered almost 4x rack space, and reduced costs by about 50 percent.

In short, a delicate balance is needed between vertical and horizontal scaling for optimal utilization of resources. High horizontal scaling, that is, 1 vCore/less than GB RAM, will cost a lot when stacked on RACKs in data centers. On the other hand, having a very high machine, that is, 416 vCore/30 GB RAM, will cost more money saved in data centers. However, these RACKs are primary concern factors in On-premises. In Cloud, you need not worry about Infra RACKs management; although, depending on your workload, you may require diagonal scaling.

## Benefits of cloud scaling

Every organization – large or small enterprise – uses one of the cloud platforms for at least some percentage of their use case, if not a complete 100%. There are significant advantages to adopting the cloud, and if things are done correctly as per the best practices, cloud can catalyze an organization's digital journey. In this section, let us look at the key benefits of cloud resource scaling.

### Flexibility and speed

In today's software world, businesses change priority at a fast pace. Decisions are driven by aspects of customer needs and satisfaction, as well as work done by competitors. Such fragile requirements produce demands for the appropriate IT infrastructure as well. Cloud scalability empowers IT to respond to the changes quickly.

The software team does not need to wait for the procurement and readiness of infrastructure before planning and committing to the deliverables. A team working on software whose backlog is defined and stable might not see many of these demands. A team working on a new project could have many changing needs. Mid-size and big organizations with significant funds can make an investment upfront for infrastructure and can absorb the pressure better. However, small organizations with lower funds might not be able to do it. The cloud pay-as-you-go model bridges this gap.

In addition to the above flexibility, teams can choose the infrastructure, the VM capability, the storage capability, the topology of the infrastructure, and so on, for their projects. They can also select the version of the software. For example, a team can get **Airflow 1.x** installed and used, and others can have **Airflow 2.x** installed and used. Teams

can start with a class of infrastructure and version of the software. The upgrade/downgrade of both is not difficult.

## **Ease of use and maintenance**

IT administrators can quickly spin up storage and compute components with few clicks and without much delay in the cloud. Teams are entirely abstracted away from worries of physical setup and hardware maintenance. They can concentrate on actual work, that is, the development of features/functionalities.

IT teams have both options - either to use the pre-configured configurations of infrastructure or to use the custom configurations for infrastructure, for the needs of an organization. Thanks to the virtualization strategy behind the entire infrastructure provisioning on the cloud, these customizations are possible. A lot of valuable IT time is saved and hence the cost.

Multiple '**Infrastructure as Code**' tools like *Ansible* and *Terraform* are available, supporting all major cloud providers. If this is not the scenario, then the cloud provider has its own tool for infrastructure as code, helping the IT infra team further develop infra as per need and maintain the state of infra. Such code makes it easy for new deployments. For example, an application which needs to be deployed in multiple regions, can use this script to easily handle identical deployments across regions.

## **Cost saving**

There are four key areas where the non-cloud deployments were burning much cash, and cloud deployments had improved it.

- **Reduction in the amount of expensive hardware**

There is no need to buy costly hardware in the cloud pay-as-you-go model. When a use case needs a hardware component, it could be acquired, and once the work is complete, could be terminated.

- **Reduction in labor and maintenance costs**

As there is a reduction in actual hardware hosting in on-prem data centers, the cost to install and maintain reduces drastically. The responsibility to manage it moves to the cloud provider side.

- **Higher productivity**

Software teams are abstracted away from the restriction of limited availability of resources for development and testing. If the resources are less, it could be very easily scaled up. Cloud deployments result in the high productivity of teams.

- **High returns on investments**

Earlier, a single experiment was costly. For example, running a machine learning use case requiring a GPU would have cost very high. Now, such a request could be made quickly. Whether an infrastructure component solves, the purpose can be identified without purchasing them.

## **Disaster Recovery**

With scalable cloud computing, we can deploy workloads across multiple regions. In the case of one region going down or being unavailable, the business continuity remains intact, that is, the entire data and compute capability does not suffer.

This is achieved by the cloud provider's redundant deployments of storage and compute resources. The workloads spanned across multiple regions have a higher

cost, but **Mean Time to Recover (MTTR)** is critical for production use cases.

To do a similar setup in a non-cloud environment, means setting up data centers across 2 locations. This will bring huge investments and operational costs.

## Global presence

Various use cases in industry have regional regulations on data, both at rest and in transit. You cannot process certain datasets outside a particular region. For example, a bank in US can have rules to process data in US region. Data cannot move out, and for such regional constraints, cloud scaling can be leveraged to set up a processing platform in a particular region at scale.

## When to scale?

There are three broad scenarios which need scaling up or down as per the need. These situations could be directly driven from business or indirectly impacting business.

## Scenario 1

Each workflow/workload/user journey is solving a problem statement. The solution's effectiveness depends on the results produced at the right time. In the software world, that right time is known as **Service Level Agreement (SLA)**. SLA is defined by people having the acumen to assess the proper value of SLA.

With the data growing exponentially, it is crucial to keep an eye on whether your applications are meeting SLAs or not. Teams enforce monitoring strategies to track and analyze the breach of SLAs. There would be multiple reasons for such violations, for example, network glitches and intermittent non-availability of resources. However, if the

SLA breaches are frequent and the system's throughput is the same as before, it is a clear indication to scale infra for your application.

For example, if the *REST API* has an SLA for 1 second and with everything else constant, the APIs have started taking 2 seconds; one strong reason could be that the server is not free enough to process the requests.

Another example could be a scenario where your Spark jobs used to take less than an hour to process the hourly data generated for a use case. Now, it has started taking more time to analyze the growth in data size and increase infrastructure appropriately.

On the other hand, if the processing is getting completed much before expectation, it might be a use case to scale down your infrastructure, as a bigger infrastructure means higher cost.

In this situation, applications do not need a frequent scale down, as scale down means that usage of the system has decreased due to a reduction in business. However, there can still be a need to reduce the infrastructure. This scenario does look to be a good fit for manual scaling.

## **Scenario 2**

Each software application has its own need for infrastructure. For example, a *REST Microservice* can be deployed in a container, managed by the *Kubernetes orchestration engine*, and scales based on the number of incoming HTTP requests. Similarly, there can be a data science job (data science jobs do a lot of iterative processing) that has a sudden need to acquire infrastructure to run 1000 parallel code flows. Once the execution completes, there is a need to scale down the infrastructure received.

The point is that each application has its own need to run a workload efficiently, and those needs can have transient scaling needs. By transient, we need to scale up the infra, do the processing and then scale down the infra.

In this kind of situation, it is crucial to not only scale up but also scale down. If the system does not scale down, it can cost very high. Since the frequency of scaling up and down is high, manual scaling is not possible. Either we can leverage the autoscaling provided by cloud providers, which scale up and down based on need in a time-optimized manner; or we can write custom scripts, which scale up and down infra before and after the run. However, such strategies have a risk of failure.

### **Scenario 3**

The third scenario where it is vital to scale, is to manage some ad hoc/temporary workloads. For example, we are doing some performance tests for the application. It is crucial to test the current workload expectations as well as expectations for the next couple of years.

Similarly, you may have a big data application processing GBs of data every hour, producing hourly reports. One day, it was observed that the data did not arrive at the right time for one of the hours, and we needed to re-process the data. Then using the already configured infrastructure in its original capacity will delay current hourly processing. In this situation, we can either scale up the existing infra or create an altogether new infra to execute such workflows.

This scenario refers to exceptional conditions in projects that need a temporary increase in devices, making it a candidate for manual scaling up and down.

### **How to scale?**

When it comes to scaling strategies on cloud, there are 3 common strategies available across all cloud platforms:

- Manual scaling
- Scheduled scaling
- Automatic scaling

## **Manual scaling**

As the name suggests, manual scaling is manually running commands to increase or decrease infrastructure. It sounds simple, but has some hidden issues. First among them might be: how will somebody know the correct number to scale to? Another concern could be the time when this action has to be taken. Yet another downside to this strategy is identifying and making sure that we downsize infra in off-peak hours. Otherwise, we can see an unnecessary increase in cost.

Even with so many downsides of this approach, this strategy is the starting point for the scale of your application, since both migrated from on-prem or developed new. This strategy could work for some time, but it is advised to work and implement better strategies of scale provided by cloud providers.

It might look naive, but it has some obvious advantages compared to on-prem, and those are an inexpensive upfront investment, with a short duration of scale, and manageable upgrades to infra.

## **Scheduled scaling**

Scheduled scaling is precisely similar to manual scaling with one difference. Instead of manually taking actions, there are cloud native offerings as well as scripts, that can be written to schedule the scale up/down of the system. Identifying the right time and correct quantity of scale still holds true. The



advantages of inexpensive investments, short duration, and manageable upgrades also hold true.

There is an added risk in this approach. The cron job/scheduled job running these scale-up scripts might fail and, because of its automated nature, could get missed from the team, resulting in a breach of SLAs. Teams utilizing this strategy implement monitoring of these scheduled jobs to track failures.

## **Automatic scaling**

In this strategy, the cloud providers provide an advanced way of scaling up and down, not based on a prediction, but on attributes. For example, a microservice deployed as a container in the Kubernetes orchestration platform can scale from one to two, based on the number of incoming requests.

This is just one example. Similar strategies are available for components, available in each cloud. Generally, this strategy can take into consideration the following parameters:

- CPU usage
- Memory usage
- Disk usage
- Number of incoming HTTP requests

Cloud providers develop and manage these strategies, and therefore, once configured, these will not fail quickly. Another advantage is that these strategies are not based on predictions, but rather on concrete system parameters, and hence it makes more sense to use them.

To implement the strategy at the right level, it is essential to analyze the application and identify the correct scale parameter. There is also a slight delay between the arising

need to scale and the actual scale happening, which must be handled while architecting the same.

## **Key challenges of scaling**

Scaling is one of the major reasons why enterprises move to the cloud. However, movement to cloud becomes complex when there is a need to span across multiple clouds like AWS and GCP, or GCP and Azure, and so on. In such a situation, one common strategy to scale does not work in all providers. These discrepancies in strategies can be broadly classified into a few sub points, as described in the following sub-sections.

## **Cloud native and hybrid deployments**

The business requirements of deploying workloads to multiple clouds - public, private, and in-house - are becoming common. Sometimes these situations are pushed by customers. For example, a particular client x has a tie-up with GCP, and hence they want to use GCP. Other cases could be forced. For example, GCP is not available in China, or few banking companies allow public cloud, while others do not. There can be multiple combinations of these situations.

But whenever we have such a situation, there is a need to deploy the same software to multiple cloud providers. This brings in the following complexities:

1. At the application code level, we have to ensure it is written well enough to interact with cloud-native APIs. For example, for storage needs, applications can interact with S3 in the case of AWS, and cloud storage in the case of GCP.
2. Added complexities to trivial tasks. Provisioning a VM instance in a cloud is trivial. However, provisioning VMs

in multiple clouds together can become cumbersome. This, when spread across all the components used by an application, becomes even more complex.

3. Apart from basic setup infrastructure, the individual components have different strategies to scale driven by each cloud provider.

## Load balancing

Load balancing available on one cloud provider usually does not support the load balancing needs of other cloud providers. For example, **Elastic Load Balancing (ELB)** by AWS cannot distribute the load on services deployed on GCP and vice versa.

One obvious solution could be to have a self-managed custom load balancer, balancing loads across the clouds, could be set up. But in that case, the management, compatibility, and upgrades become the responsibility of the IT team.

## Housekeeping services

As was the case of incompatible load balancing, this case of housekeeping services is for monitoring, alerting, centralized logging, and so on. This again brings us to developing and supporting components native to the cloud. These days, a famous technology stack to handle this is *Prometheus* and open tracing with *Jaeger* and *Grafana*, alongside *Thanos* to maintain metrics.

However, this too has to be managed and maintained by a team, bringing us back to where we started - self-managed to unmanaged.

Apart from technology, other supporting workflows like **Site Reliability Engineering (SRE)** have to evolve accordingly.

## Scale versus cost relationship

The decision to move to the cloud is primarily taken at benefits of scalability and agility, faster time to market, and enabler for innovation. The aspects of cost are often theoretically read, and more often than not, one comparison of components is analyzed. For example, let us say a company plans 100 VM for a workflow. Based on how much time it takes to run the VMs in on-prem vs. cloud, a decision is taken. There are a few key areas in cloud which do incur cost and are left unnoticed initially. However, they should be taken into consideration upfront, and are as follows:

- Data retrieval and egress costs
- Workload and allocated resources
- Volume of data

Ideally, your cost should be a linear function of your usage, which could be true considering all the best practices of scaling native to the cloud are implemented. When it comes to the actual deployment of applications managed by enterprises, the equation is not that simple, and companies find themselves struggling to reduce the **Total Cost to Ownership (TCO)**.

Major reasons behind this are as follows:

- **Skill gaps/Learning curves**

Each cloud provider has its own defined best practices and hence, there is often a learning curve involved when we start adopting the cloud platform. Identifying the right parameters to scale is important, right from day one of the design process.

- **Aggressive business priorities**

Business priorities keep on changing and so do the need for iterative development of applications. These requirements are often overoptimistic. In this situation,

even if the **Non-Functional Requirements (NFR)** of scalability were originally defined, it gets de-prioritized by teams.

- **Inappropriate allocation of assets**

No matter how smart we are while allocating resources, it could never be 100% optimized. Oversized VMs, high performance storage and orphaned assets are few common examples. Multiple surveys have reported this reason for 30-40% higher bills.

- **Low emphasis on capacity planning and appropriate monitoring**

Often, these aspects are ignored at the start of the cloud journey, and organizations start giving importance to these aspects only when they start receiving huge bills. By the time we start giving due importance to this, we are already in the middle of the journey. Now bringing in even a small change for all the applications becomes a costly affair in itself, and then comes the aspect of somehow retro fitting these aspects. This might give initial indication of lower cost, but in the long run no major advantages are observed.

This love-hate relationship of scale-cost on cloud needs to be handled diligently, following good practices from day 1 in the team. Cloud providers also provide some key features, resulting in lower overall costs. You have to make sure to use them to the fullest. A few key ones are:

- **Use of reserved and dedicated instances:** Reserved instances are a discounted billing concept if you commit to the use of VM for a specified amount of time. Even if you do not use the VM, you are committed to paying the cost.
- **Use of spot instances/preemptible VMs:** These are spare VM instances provided at discounted prices

(generally  $\frac{1}{3}$  of the actual cost). If the need arises, cloud providers will take those back without intimation.

- **Appropriate transition strategy from hot to cold storage:** Hot storage refers to storage power that is higher in cost but offers fast storage and retrieval. On the other hand, cold storage is lower in price with lower **Input/Output Operations Per Second (IOPS)**. Instead of keeping the complete data in hot storage, historical data can be moved to cold storage for audit purposes.
- **Using pre-owned licenses:** You can use the pre-owned license to reduce the cost of components on the cloud. For example, if you already have licenses of MySQL used on-prem and plan to create a MySQL instance in the cloud, you can use the same license; in that case, cloud providers reduce the cost of cloud host MySQL.
- **Use serverless options:** You can use the various serverless options provided by each cloud provider. Though there is going to be some initial extra work, in long run the lesser cost associated with serverless systems will cover up.

## Risks of improper scaling

Scaling is essential; however, proper scaling is even more critical. Scaling can affect the other applications hosted on the cloud if not done correctly.

- **Cross application impacts**

Whenever we define the scale-up configurations/script, it is essential to mention the upper limit to scaling. The addition of upper limits of all applications should be less than the quota on the cloud. Quota in cloud terms

means the maximum limit on computing and storage that could be spun up.

- **Scaling down**

When the applications are scaled up, the system should scale down, and thus the workflow fails. In the case of autoscaling, cloud providers take care; however, in the case of scheduled and manual scaling, its strategy should have a very clear definition and implementation to scale down in case of failures.

- **Inflated costs**

Improper scaling could result in high costs. Monitoring the price and generating alerts based on consumption of the allocated budget is recommended.

## **Conclusion**

Having a clear scalability plan is essential to tackle the ever-increasing workloads due to the amount of data we produce these days. Cloud's pay-as-you-go model has acted as an enabler for organizations to scale quickly as per customer needs, experiment with new technologies, and reduce overall time to market, by providing virtually infinite infrastructure ready to be used with minimal management.

Cloud providers provide capabilities to perform scaling based on multiple aspects of an application. IT could be system metrics like CPU or memory usage or external material like the number of requests and unread messages. However, if scaling is not leveraged correctly, it could lead to high costs and situations where an application can impact other applications.

# CHAPTER 2

## KPI for Cloud Scalability

### Introduction

The **Key Performance Indicator (KPI)** is defined and tracked by all modern organizations to assess whether a company is hitting or missing its north star growth and profitability goals. To meet the goals, organizations must be open to change as per the market's needs. Cloud adoption makes companies scalable to adopting new business, innovation, and quick turnarounds to market needs/customer satisfaction aspects.

Selecting a cloud provider depends on how well the cloud provider's offering fits with that of an organization's KPIs. It is essential to define business KPIs, and it is also vital to define KPIs for the enabler, that is, the selected cloud provider. Scaling being one of the central pillars, cloud scaling KPIs became important.

This chapter will look into Key Performance Indicators for cloud scalability and how one should track metrics in cloud platforms.

### Structure

In this chapter, we will discuss following topics:

- Defining KPI
- Basic cloud metrics
  - Performance
  - Reliability
  - Cost
  - Availability



- Indirect KPI Impact of cloud scalability
- Advanced Metrics
  - Response time
  - Latency
  - Throughput

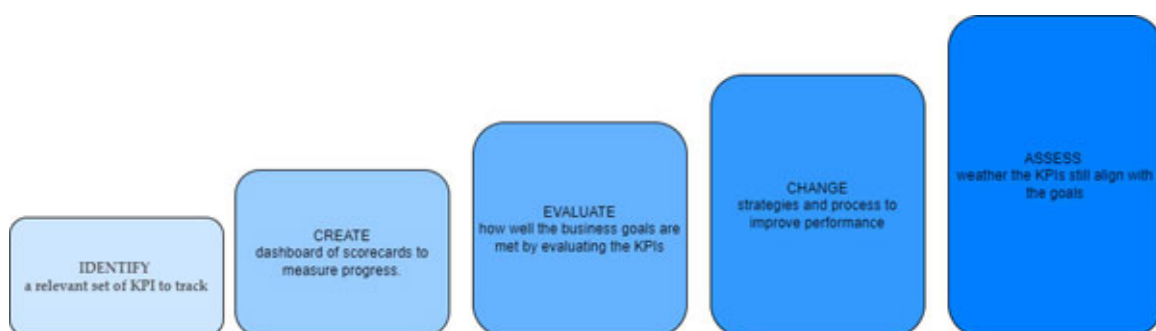
## Objectives

After studying this unit, the reader should be able to understand all about Key Performance Indicators, in context of public cloud platforms. We will be looking into all the key metrics that can be defined on cloud platforms and how these key metrics can be treated as KPI. In the last section, we will look into some advanced KPIs that enterprises use for tracking the progress of cloud deployments/adoption.

## Defining KPIs

The term '*metric*' is derived from the term measurement. Metric is some measurable attribute of the system. This attribute has a current value. For example, my software can process a 1 GB file in 1 minute. 1 GB per minute is the measurable attribute and hence, a metric. When we define targets for these metrics, it becomes KPI. For example, a business can specify a target: they want to achieve a processing capability of 2 GB per minute.

Figure 2.1 is an overview of various stages in KPI management, starting from inception of KPI to achieving it:



**Figure 2.1:** Stages of KPI management

When an organization selects a cloud provider, it defines some critical KPIs. There is a business need to deploy essential features/fixes as soon as possible. Imagine a hypothetical situation: an organization's deployment frequency was once a day on-premise, and project management wants to make it once in an hour. Each deployment needs a specific testing infrastructure for automated tests, which runs with each build and release.

In this case, deployment frequency becomes the metric. Once per day in on-prem, becomes the current value. And once per hour becomes the target value. While adopting a cloud provider, we need to assess whether selecting a cloud will enable us to achieve the deployment frequency goal defined for us by the management (business owners).

The organization has two options - Cloud X and Cloud Y. Cloud X offers a deployment mechanism where the infrastructure increases automatically, based on the number of deployments triggered. In Cloud Y, there is no autoscaling for deployment agents. Instead, it has manual scaling, and there is an upper limit to scaling. Cloud X and Cloud Y offer to scale up; hence both can serve the purpose. However, Cloud X becomes an obvious choice due to auto scaling and max upper limit limitations.

In the preceding example, to meet the business need, we could define the Cloud scalability KPI - *"How quickly the system scales up to promoted deployments with automated tests?"* This could be further broken down into KPIs of *"Fast scaling for testing infra"* and *"Fast scaling of deployment agents infra."*

Similar to the example stated above, a business KPI percolates into a technical KPI; when it comes to scaling infrastructure on the cloud, there are multiple industry-level metrics that mostly every organization wants to track and define a target value for them (KPI).

These metrics with a target value/KPI can be divided into two categories:

- Simple metrics
- Compound metrics

**Simple Metrics** can be metrics that are not driven by other metrics. For example, **Input/Output Operations Per Second (IOPS)** for an SSD disc offered by the cloud is 3000 in cloud X and 4000 in cloud Y. These metrics are entirely technical.

**Compound Metrics** are metrics that are combinations of two or more simple metrics. For example, some databases hosted on Cloud X can run 500 queries, and the same database on Cloud Y gives the ability to run 1000 queries per second. Compound metrics are closer to business and make more sense for the software application.

The things to remember while defining KPIs are as follows:

- KPIs have to be **S.M.A.R.T.**, that is, **Specific, Measurable, Achievable, Relevant, and Time-Based**. If any KPI misses out on any of the preceding five parameters, it will not be effective. For example, a KPI could be Specific, Measurable, Achievable, and Relevant, but there is no timeline associated. It makes no sense.
- Efficiency is critical when defining KPIs. This is because, let us say there is KPI to bring down the cost by 10% in the next six months. However, if the software adoption increases simultaneously, it would mean more infra and hence more money. Instead, here, a better KPI would be to define lowering the cloud cost by 10% per active user in the next six months.
- KPI should be backed up by data or evidence-based assumptions and not intuitions and guesses.
- A KPI should be defined in the context of some business needs. There is no point in defining a KPI with no business impact.

The heading of this chapter, “*KPI for Cloud Scalability*,” means deciding the metric to track, defining a target value (improved), and trying to achieve a targeted value in time for applications that leverage cloud scalability.

## **Basic cloud scalability metrics**

All the previously-defined KPIs are an intelligent aggregation of 3 basic Metrics, that is, performance, cost, and uptime. We can measure them accurately, and all KPIs can probably be built on top of it.

### **Performance**

An essential aspect of developing software is to identify the key bottlenecks and deadlocks in your system that affect the overall performance of the system. Cloud providers provide a level of performance, but to leverage supported/claimed performance capabilities, the Engineering teams are expected to develop software that will adhere to the practices suggested by cloud providers. A team not adhering to those offered practices will create an application that does not perform well in leveraging the scalability aspects.

It is good to collect and monitor a wide variety of metrics (infrastructural and behavioral); however, it is even more critical to accurately collect specific metrics as per the situation. These metrics depend on a case-by-case basis. For example, if you are trying to monitor a SQL query system, a few standard metrics which will help are:

- **URLs:** The URLs in the web application that make use of the query.
- **Average time:** Average time taken by query to revert with results.
- **Calls:** Number of calls to a query.

Another important aspect is to make the observability of performance numbers. Usually, cloud providers provide

portal/dashboard for the services deployed on one particular cloud. For instance, on Google Cloud Platform we have *GCP Monitor*, and on AWS we have *CloudWatch*. However, in case of hybrid or multi cloud deployments, there are monitoring tools available, that integrate with multiple clouds. A few famous tools available in the market are *AppDynamic*, *Datadog* and *New Relic*.

The performance aspects vary widely as per the nature of the application. However, the heuristics to optimize an application on cloud or on-premises, conceptually remains the same and are as follows:

- Maximizing usage of memory
- Reduced disk I/O
- Reducing data transfer over the network
- Parallel processing
- Appropriate class of infrastructure

Let us look into a few everyday performance considerations in Big Data, Microservice and REST API environments on clouds:

### **Use case 1: Big data**

For designing any big data applications, the following points need to be taken care of:

- Apply filtering of data in the starting steps of data processing. This will reduce the size of data an application has to process at the very start.
- Optimize joining 2 datasets to reduce data transfer over networks.
- Use recommendations from cloud providers. The recommendations could vary for the same tool used across different cloud providers.
- Strategize to implement auto scaling or at least scheduled scaling for your workloads.

There are cloud providers that have their solution available, and require near zero maintenance of infra. For example, GCP has *Google Dataflow*, which they claim runs fastest on GCP infrastructure. However, other big data tools are also available under the name *Google Dataproc*. Teams that want to support hybrid or multi cloud deployment generally do not opt for such solutions. Instead, they choose a technology available across all cloud providers, such as *Apache Spark*. In this case, management of infrastructure will be the responsibility of the IT team.

## **Use case 2: Microservices**

For designing any microservice, it is important to keep the following basic rules in mind:

- Loosely coupled architecture.
- Easy to scale, based on demand. Use of serverless architecture wherever possible.
- Small individual services, based on business functionality.
- The technology stack of each microservice can be different.
- Separate datastore for each microservice.

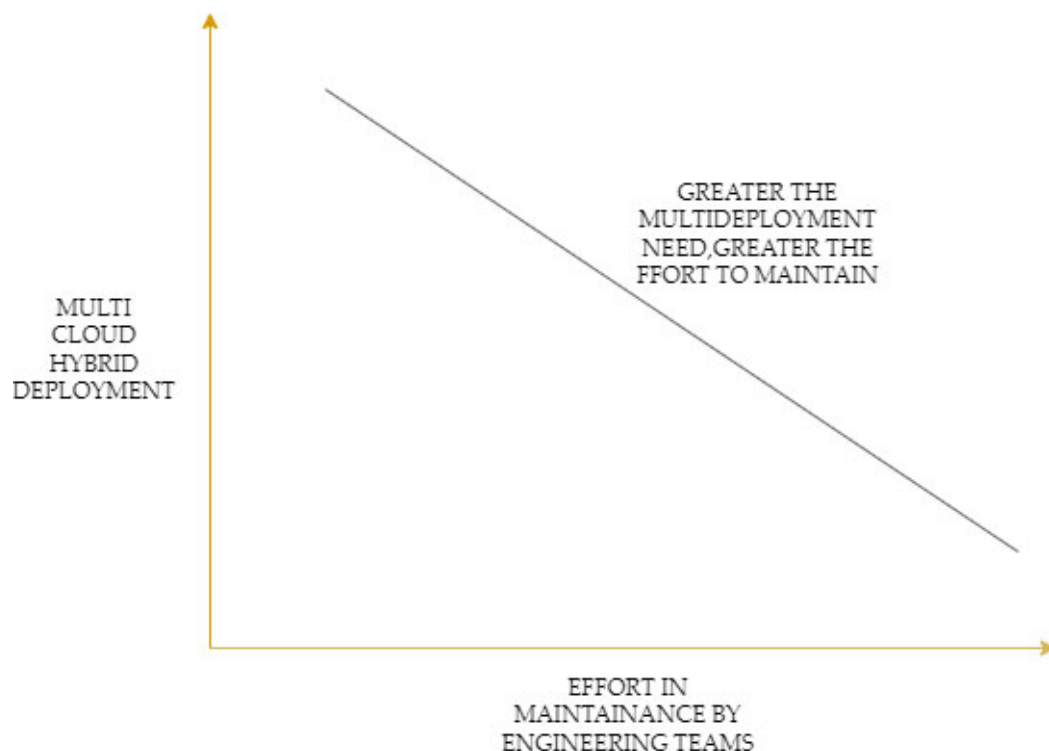
As was the case with Big Data Applications, two ways of deployment are possible. The first is to use a cloud specific PaaS to host the microservice; but then, that microservice will only run on one cloud. There is multi cloud stack available as well. The caveat is that the management has to be taken care of by the engineering team in the second case, whereas the cloud provider managed everything in the first approach.

## **Use case 3: REST API**

The design principles remain the same in the case of REST APIs, just like for the previous two scenarios as well. The key things to be kept in mind for the REST API are as follows:

- **Persistence support:** Open the connection once and reuse the connection for multiple requests saving handshake cost with REST API calls.
- Parallelism
- **Throttle:** Support for minimum number of requests.
- **Low latency responses:** Response within a time frame.
- Selecting load balancing and content delivery network options.

There is also a cloud-specific hosting method (using cloud provider proprietary components such as App Engine GCP) and a hybrid way of hosting the REST API. The hybrid strategy will need the scaling aspects to be defined entirely and managed by engineering teams. It is much easier to configure in utilizing solutions specific to the cloud. Consider the [Figure 2.2](#) for the relation between multi cloud deployment and IT team efforts. The bigger is the multi cloud deployment need, the better will be the effort to maintain it. Please refer to the following figure:



**Figure 2.2:** Multicloud deployment vs IT team efforts

## **Reliability**

Reliability of a software system means the ability of the system to perform consistently according to specifications. Software is reliable if it passes all test cases and consistently achieves what it is intended to serve.

For instance, if a batch job processing a file of 1 GB takes an hour, with data and infrastructure remaining similar, the job should take an hour in multiple runs. There could be numerous failures, for example, the network going down, VM getting unresponsive, IOPS not happening as per expectations, and so on. The key is how the system recovers from it and performs at similar levels.

Cloud scalability should add more reliability to your application simply because if any component in the cloud fails, a new one could be easily created. Some key metrics are used while defining reliability, and are as follows:

### **Mean time to failure**

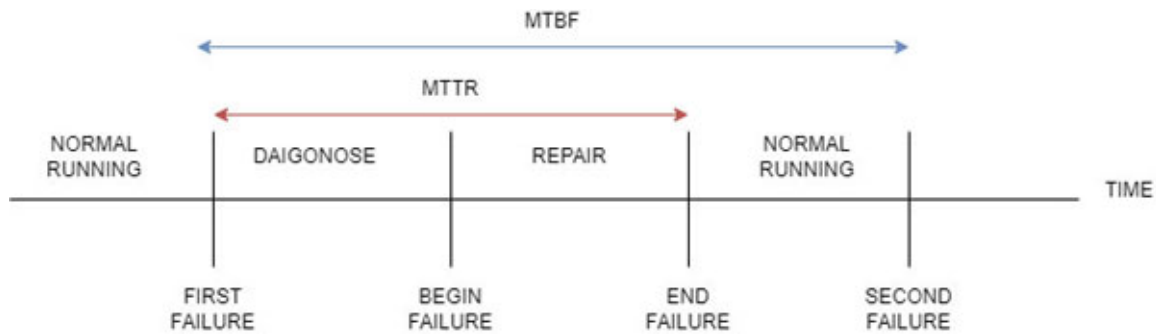
**Mean time to failure** is defined as the time interval between 2 failures. An MTTF value of 1000 means that there might be one failure for each 1000-time unit.

An organization can measure the current MTTF value and aim for a high mean time to failure as KPI. The MTTF value should improve when scaling the system up or down, on the cloud.

### **Mean time to repair**

**Mean time to repair** measures the average time taken to track the errors, causing the failure and then fixing them. An organization can measure the current MTTR value and aim for a low mean time to repair as KPI. Lower MTTR means less time to identify and fix the bug in the bug lifecycle, as illustrated in [\*Figure 2.3\*](#). When scaling the system up or down on the cloud, the MTTR value should reduce. Please refer to the following figure:





**Figure 2.3:** Defining MTTR

## Rate Of Occurrence Of Failure

**Rate of Occurrence of Failure (ROCOF)** measures the number of failures appearing in a unit time interval. When we scale up the system, it should not increase ROCOF.

## Probability Of Failure On Demand

**Probability of Failure on Demand (POFOD)** is the probability that the system will fail when a service is requested. Scale-up and scale-down should not have any effect on the POCOF.

## Costs

Cost is a metric on which many decisions depend. From the first decision of selecting a cloud provider to defining the max range for scalability, cost plays an important role.

In large organizations, typically, when a project starts on the cloud, the cost is not given due importance. However, soon this becomes the single most crucial aspect because of the scale of the team, scale of use case, and scale of customers.

Cost depends on a lot of factors, both controllable and uncontrollable. A few key controllable factors are as follows:

- Cost of unused resources.
- Usage of reservations and discounts.

- Capacity planning your workload. Strategizing the scale mechanism.

On the other hand, the uncontrollable factors are as follows:

- Cost increase due to new customers on boarding.
- Cost due to team immaturity.

To check cloud costs and improve efficiency, organizations rely a lot on automation, mainly to keep track of resources and identify unused resources. A few key policies implemented in the automation are as follows:

- **Notify:** Notification to inform when the monthly budget is consumed.
- **Suspend:** If a policy detects that a VM is created not according to central governance policies, suspend the launch of the VM.
- **Terminate:** Terminate resources that are lying orphaned in the system. For example, no IOPS on a storage service or CPU activity for a while.
- **Revoke:** Access to any account logged in from a non-conforming IP address.
- **Schedule:** Periodic shutdowns (especially weekends and nights) for non-production assets to avoid wasting resources.

Almost all cloud providers understand the importance of providing transparent spending for each customer, and hence they give a pretty detailed report, giving complete insight into where the money was spent. Such reports are a good source for understanding if resource consumption matches the expectations. Otherwise, a deeper dive is needed into why a cost has been incurred. For example, if a team thinks they should be charged for 100 hours of VM time, but the price says it is 150 hours, it is a clear indication that VMs were running when they are not supposed to.

Cost KPIs are one of the most important KPIs for cloud scaling. It feels impressive that a system can handle the load by scaling on the cloud. It is also imperative to keep in mind how the money is burnt in doing so. A scaling trigger where a business earns \$1 by spending \$2, is a loss of profit.

The way cost KPIs are defined depends on use case to use case, but few widely used ones are as follows:

### **Total cloud cost**

This refers to the total cost of your production and non-production environment. Ideally speaking, scaling should primarily impact total cost on production. The increased adoption of applications will impact the production costs going up. Your non-production environments will show an increase in case of team size growing or some environmental impacts like increased performance testing.

We can add a bit more context to the KPI definitions to make it more effective and achievable.

### **Defining KPI for cost in production environments**

Rather than tracking total cost, a more appropriate measurement will be measuring the cost rate.

$$KPI = Total\ Cost / Environmental\ Factor$$

The denominator here - “*Environmental Factor*” could be anything. For example, the number of environments (KPI becomes cost per environment), number of users (KPI becomes cost per user), number of runs (KPI becomes cost per execution), GBs of data, and so on.

### **Defining KPI for cost in non-production environments**

In non-production environments, there could be two primary reasons for the increase in cost:

- Team size and usage of resources

- Stage of software

The non-production cost increases as the size of the team grows. If there is DEV and QA environment, the DEV environment cost will be directly proportional to members of the dev team, and similarly, the QA environment cost will be proportional to QA team size.

Another aspect that could affect the cost is the stage of the software. If the project has started new, the development cost is expected to be less. Instead, if the project is in a scene where the plan has too many performance runs/reliability runs, the price is expected to increase.

### **Forecasted cost**

Forecasting the cost of your cloud usage is vital in regards to making the right decisions. For example, if the prediction is that platform adoption will increase by 10 X, a better decision could be taken in terms of scaling parameters.

There is the business forecast of the usage of the application, and then the technical management defines the non-functional requirements to achieve the forecasting done by the business. It then comes to entirely technical decisions like committing for usage, using Spot VMs, class of machine, and so on. With every scalable aspect (technical and non-technical), measurement of the cost will help identify profitable decisions.

### **Money saved on committed discounts and reservations**

Committed discounts are discounts provided by cloud providers, if an organization uses its platform. To get discounts, there has to be a substantial deal (in terms of money).

On the other hand, reservations are the commitments to use a specific infrastructure class. For example, we can say that we will use a series of VM for six months. And when we do so, we can get the VM reserved for six months only for our use case.

The advantage is that such VM assignments will cost less, and the disadvantage is that even if you do not use them, you are committed to using them for a specified time.

If cloud resource scaling could be done so that more and more reserved instances are being used, it could lower the cost. Similarly, if more and more scaling activity involves the discounted price infrastructure, the cost will be under check.

A meaningful KPI here could be the cost reduction when reserved and discounted infrastructure is used.

There could be many ways/aspects a company can look into the above cost KPI and define use case-specific KPIs. For example, it can be the percentage change in the price of cloud resources over time (%) due to reserved instances, the percentage of infrastructure running on-demand vs. covered by discount or Spot, and so on.

## **Availability**

Availability refers to the ability of an IT service to perform its function. It is measured as the percentage of time the service is available. It is a report of the past and a prediction of the future. It tells how well the service is performed in a period of time.

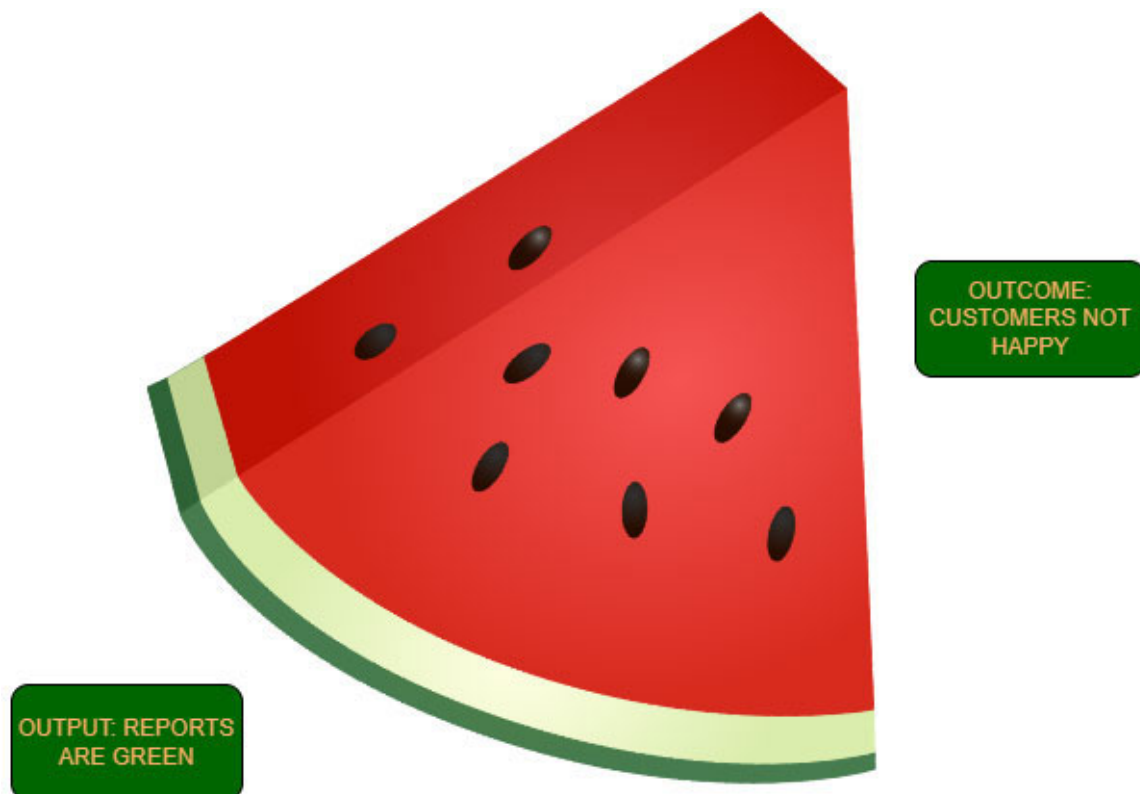
Scalability in the cloud facilitates improved availability of systems. Availability describes how well the system provides resources over a while. With everything getting digitized, availability has become the most crucial aspect of generating revenue. For example, think of a situation where the eCommerce portal Amazon is down for 5 minutes. It straight away means a huge loss of business.

Cloud providers that provide PaaS service, typically offer a 99.xx% availability, with few even committing 0 downtimes. The primary idea behind making the system available, is redundant deployment across multiple locations with an efficient switchover.

All public clouds provide cross geographical scaling and easy redundancy and remove **Single Point Of Failures (SPOFs)** from the system.

One might think that the availability metrics and customer satisfaction are tied, that is, a customer will be happy with high availability. It is sometimes true, although not always, as customer satisfaction depends on the preferred outcomes of customers. For instance, let us assume you are an eCommerce provider with 99.9% availability (weekly) (.1% or 10 minutes of downtime in a week). But the .1% downtime occurs during high usage events, such as when there are maximum discounts offered or there is a campaign for up-selling in progress. We can reach the availability targets, but the customer is not happy.

In [Figure 2.4](#), we see the classic watermelon pattern, which is green (good) on the outside and red (bad) inside. Your output meets the defined target but not the desired outcome. Please refer to the following figure:



**Figure 2.4:** Watermelon pattern

Availability is critical for customer satisfaction and to ensure that, redundant developments are on cloud used. There is a term used these days, known as auto-healing (primarily in the context of cloud). In auto-healing, the infrastructure is configured to support minimum components, that is, let us say we configure 10 VMs to be running at any given point of time. Suddenly one VM goes down. Then, the system will create the VM to ensure that we have a minimum of 10 VMs running.

## **Indirect KPI impact of cloud scalability**

Cloud scalability can impact some KPIs in your project indirectly. By indirectly, we mean that these KPIs can severely get affected if the scaling of the cloud platform is done right and in time.

## **Innovations**

Cloud scalability allows software teams to scale infrastructure quickly for innovation, which helps them create new workflows and features. Not only that, but because of the scalability of infrastructure, teams can quickly try new technologies for their innovations. In the traditional scenario, on prep system, this was a significant bottleneck because scalability was a bottleneck.

KPIs like *Innovation Rate* ( $= \text{number of innovations} / \text{number of products} * 100$ ) and *Degree Of Innovation* ( $= \text{newness of the purpose-medium combination}$ ) will get impacted due to this.

## **Software development and operational KPIs impacts**

The comfort that new infrastructure is readily available, or the infrastructure can scale, can significantly impact the general software development and operational KPIs. For example, software developer productivity KPIs (the most popular ones

being speed, cycle, and response time), will optimize. Also, operational KPIs that measure the software's stability in terms of production and maintenance efficiency, will have a positive impact.

## **Customer satisfaction**

Cloud scalability has a positive impact on the customer satisfaction KPIs like **Customer Satisfaction Score (CSAT)**. Customer Satisfaction Score is a metric used by companies to gauge how satisfied a customer is, with a particular interaction or their overall experience. CSAT is expected to improve positively with faster lead time and better SLAs.

## **Advanced metrics**

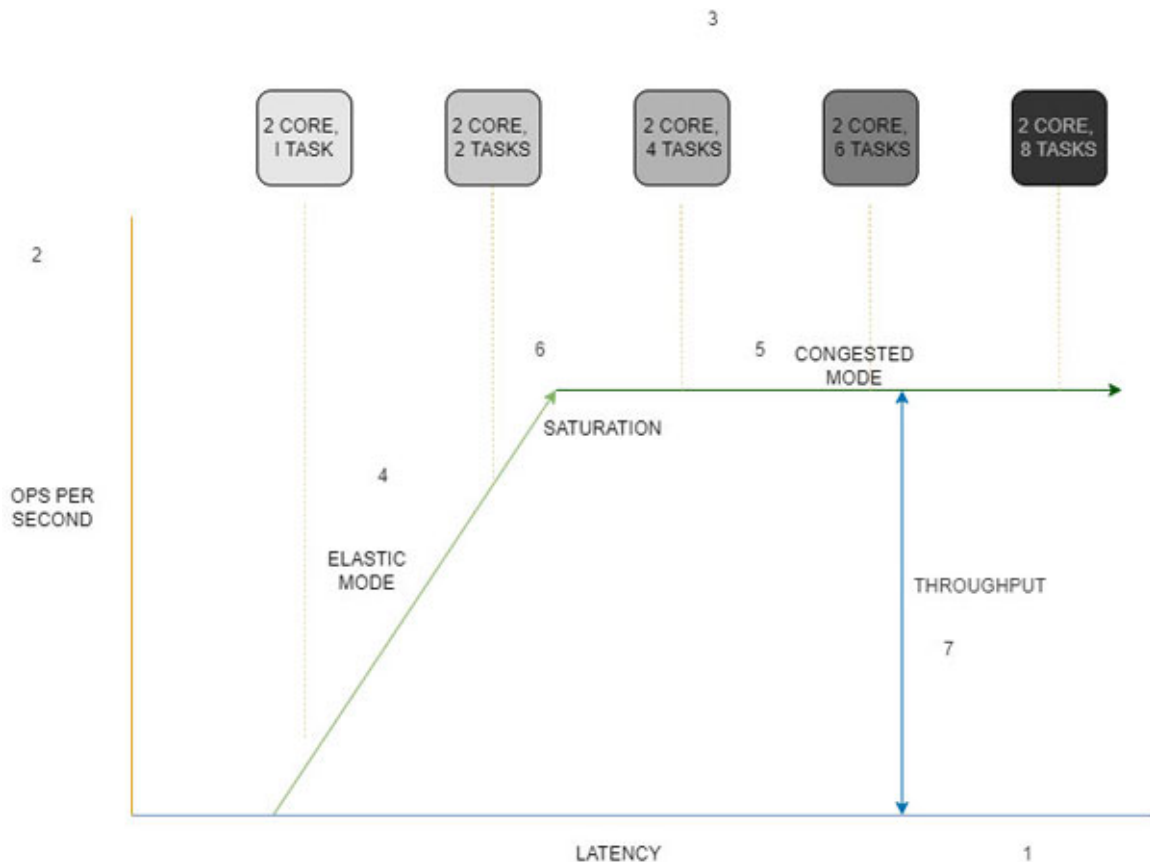
We looked into the primary metric to calculate KPIs on cloud scalability. This section will look into some advanced metrics used across the industry to define KPIs.

[Figure 2.5](#) represents a scenario where the load/transactions on the system increase progressively. Refer to the numbers in the figure for the following explanation:

1. The X-axis represents the latency of your system.
2. The Y-axis represents the operations per second on the system.
3. This means that the progressive operations trigger into the system. For example, it starts with 1 task on two core machines, then raises to 2 tasks on two cores and then 4, 6 and tasks on two cores.
4. Represents the elastic zone, that is, the system will utilize more and more infra, to present a linear increase in latency with the number of operations.
5. Represent the congested mode. The infrastructure is constant, and the number of tasks launched is high. The system is doing many contexts switching, to handle the workloads.



6. Represents the saturation point where the time taken for processing one operation starts increasing with more and more tasks launched. Rather than being linear with a positive slope (throughput), the curve is not a flat line with a slope of zero.
7. The system's throughput is constant after the saturation point, no matter how many tasks are launched. The system is saturated now to show a further increase in throughput. Please refer to the following figure:



**Figure 2.5:** OPS per second vs. latency

Ideally speaking, the application needs to increase the throughput as the load on the application increases. It means that anything on the left of the **Saturation Point** works well in favor of the application. Cloud scalability makes sure that the infrastructure is increased as and when needed, to not cross **Saturation Point** and go into consumption mode.

## Response time

Applications accept requests from users and revert with a response. The time the application takes to respond, determines the efficiency, and the greater the efficiency, the greater customer satisfaction is. There are five different response metrics that we can measure, and they are as follows:

### Data in and out

The metric captures the size of each batch of requests to the server, as well as the number of responses the server creates. For example, if we have 1000 requests and only 100 responses come back, the data in data out ratio is 10:1. As the load increases, this ratio might need to be more aggressive, and hence it means a capable backend catering to such requests.

Cloud scalability enhances this capability of handling a wavy load. The better the cloud scalability, better is the metric.

### Request per second

It measures the number of requests a server receives every second. More requests per second can lead to slower responses. However, cloud scalability can help scale the infrastructure based on the number of incoming requests to maintain a critical response time.

### Average response time

It measures the amount of time a server takes to respond to a request. Lower response time means better performance, as the server will take less time to respond. Under heavy load, this metric might deteriorate, and hence, cloud scalability can help. In this situation, applications can scale based on CPU usage or Memory usage, to maintain a consistent **Average Response Time (ART)**.

## **Peak response time**

Peak response time measures the longest response time from the server. This is nothing but an outlier in your ART calculation. Cloud scalability can help have very few outliers, and even if the deviation is seen, that deviation from ART should not be much.

## **Infrastructure utilization**

This metric measures how much infrastructure power a request consumes, to produce a response. An application under heavy load will increase the infra consumption, and cloud scaling can help keep them lower.

## **Latency**

The response time metric described above, provides performance information about transactions. However, in the case of asynchronous applications, it may not indicate everything.

Let us consider a situation where a method spawns another thread and then returns control to the calling thread. The return of control stops the clock for the response time metric; however, logical processing continues behind the scenes. For such situations, we rely on latency metric. Latency metrics reflect the response time for such asynchronous transactions. The key metrics under this heading are as follows:

### **Average end-to-end latency**

It is the average time taken by all the asynchronous processing done in the system. Obviously, a low average time here means the processing took time, which means that the cloud scalability can help lower these metrics with efficient scaling.

### **Number of slow end-to-end transactions**

This is the number of end-to-end transactions which took more than the expected value of processing in a time period. For example, there is an eCommerce application, and the asynchronous process for a customer checking out the cart takes 1 minute, but in certain cases, it takes 2 minutes. Then the count of cases where it took 2 minutes becomes the value for this metric.

### **Number of very slow end to end latency times**

This represents the maximum amount of time taken in an asynchronous process. These maximum times will represent the worst latency times.

Generally, we start with a value supported by the application (current value); the business people define the expected value (target value) based on customer input, and then the tech teams try to bridge the gap between current and expected values in a phased manner, by leveraging auto scaling/scheduled scaling features of the cloud.

Cloud scalability should bring down the latency of the asynchronous process by scaling up the infrastructure when needed.

### **Throughput**

Throughput is an indicator of performance. It is the quantity of an activity that a system accomplishes in a unit of time. For example, this system has a throughput of 300 transactions per second.

Rules of the game remain the same; we start with an initial value and define a target value with inputs from the business. We use cloud scaling to meet the business goals.

### **Conclusion**

To meet the goals of digit journey, organizations must be open to change as per the market's needs. Cloud adoption makes

companies scalable to adopting new business, innovation, and quick turnarounds to market needs and customer satisfaction aspects. Modern organizations define multiple KPIs in which they want to track the overall success of their cloud adoption/migration journeys. Traditional KPIs include performance, reliability, cost, and availability. However, this is not the complete list; it depends on use case to use case.

Identifying efficient and valid KPIs and their achievements are essential to measuring the effectiveness of cost adoption.

## **Points to remember**

1. PaaS, IaaS, and SaaS offerings from cloud providers look very attractive upfront but identifying correct metrics and the target values (KPI) are the key to defining the success of cloud journeys. Before adopting, do not forget to specify and track them.
2. It is crucial to identify the business SLAs to utilize cloud infrastructure appropriately. For example, the data availability constraints will define the single zone vs. multizone solutions.
3. There could be multiple ways of accomplishing the same tasks on each cloud. It is vital in identifying the centralized governance of cloud actions.

## **Questions**

1. What are some key qualities of efficient KPIs?
2. Why are cloud KPIs important to define and track?
3. Which is one key KPI for your project and why?

# CHAPTER 3

## Cloud Elasticity

### Introduction

An aspect that organizations want to look at, apart from reliability, cost, availability, and security, before selecting a cloud provider, is the scalability and elasticity offered by cloud platforms. Scalability traditionally has been an act of manually increasing or decreasing resources on infra. However, there is a different scaling strategy in the cloud world, apart from traditional manual scaling, that is, auto-scaling. Auto-scaling aims to spin up the exact number of resources needed at a given time, to tackle the workload without manual intervention.

We discussed the auto-scaling of cloud resources in [\*Chapter 1, Basics of Scaling Cloud Resources\*](#), and we are going to deep dive into it in the current chapter. Autoscaling is just another name for cloud elasticity.

Autoscaling is the scaling of computing, network, memory, and storage either supported by default by a cloud provider or configured explicitly.

### Structure

In this chapter, we will discuss the following topics:

- **Defining cloud elasticity:** Defining scale in and scale-out
- Benefits of elasticity
- Elasticity and cost relationship
- Key challenges

- Difference between cloud scalability and cloud elasticity
- Use cases
  - E-commerce (batch job)
  - Song streaming (stream job)

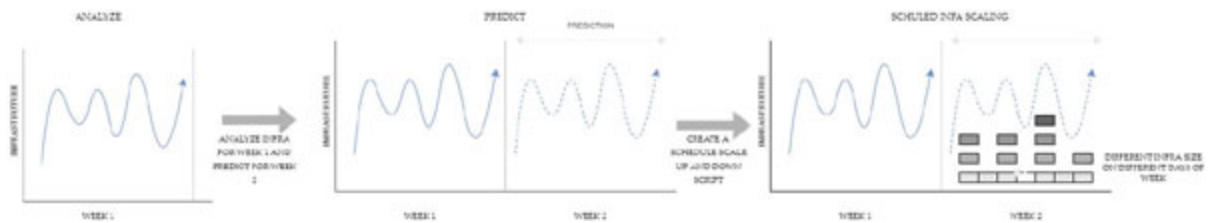
## **Objectives**

After reading this chapter, you should be able to understand elasticity, its types, forms, advantages, and its relationship with cost. It is essential to understand the flaws of traditional scaling and how cloud providers tend to overcome issues of over provisioning and under provisioning in a cost-effective manner.

## **Defining cloud elasticity**

Cloud elasticity (Autoscaling) is the capability to dynamically scale up/down (Horizontal scaling) or scale in/out (vertical scaling) resources, as per the need of the application. Dynamically implies that no explicit triggers to scale up and down are needed. Traditionally, when we used the term scaling, it was an external action taken to trigger the spinning up of infrastructure. There are two strategies when we talk about scaling – scheduled and manual.

In [\*Figure 3.1\*](#), teams used to analyze the infrastructural need for one of the weeks (Analyze phase). Based on the analysis, a similar workload is predicted for future weeks (Predict phase). Cron jobs/scripts made upscaling and downscaling as per the expected load (from Predict phase) in week 2 (scheduled infra scaling). Please refer to the following figure:



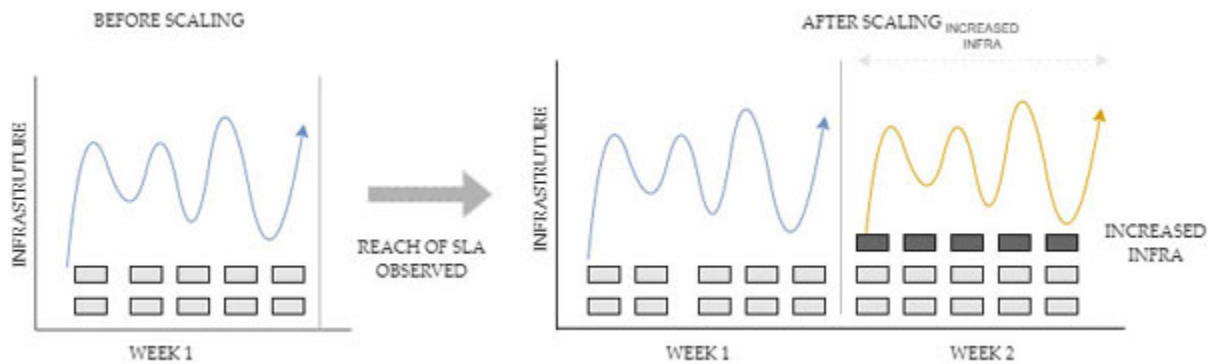
**Figure 3.1:** Scheduled scaling

The success of this strategy is dependent on the prediction of workload. We can schedule a scale-up and a scale-down in non-peak hours if we know the peak hours. In this particular use case, after frequently scaling up and down as needed, it is impossible to take the actions manually. It becomes an excellent example for writing cron jobs which increase the infra as per configurations for the peak hour. The previously-described strategy is known as *scheduled scaling* or *predictive scaling*.

There can be multiple situations where this *predictive scaling* strategy might not suffice. This strategy could fail any day due to deviations in the expected load for an hour, which could happen due to multiple reasons, such as data arriving late due to choking of ingestion pipeline in case of data platforms, or consumers hitting more APIs due to some campaign launched by the sales team in case of an eCommerce application. Even if the workload is tackled well and configurations are done well, there is a need to check whether the cron job is running correctly, which again implies additional monitoring and effort.

Another similar use case could be that of manual scaling. Consider [Figure 3.2](#). There is a workload running with some infra capacity in week 1, and it was observed that there were many breaches in SLAs. The IT team was requested to perform manual scaling (over the under provisioned), and infrastructure was increased by 50%. Now, the SLAs are being met again. Please refer to the following figure:





**Figure 3.2:** Manual scaling

In manual scaling, the infrastructure is scaled up and down manually, but the frequency of such triggers is very low. Since the infrastructure remains mostly constant, and the scale is determined by meeting the SLA of peak hours, a significant infra remains in an over-provisioned state, in off-peak hours (over provisioned).

In the preceding two approaches of scheduled and manual scaling, the decision to scale up was external, that is, some persona (IT team or the engineering team) took a call on the amount of infrastructure to be spun at any time. No matter how diligently we try to develop the proper infra, it will not be optimal (there is no over provisioning and under provisioning). That is because of multiple factors that are both known and unknown.

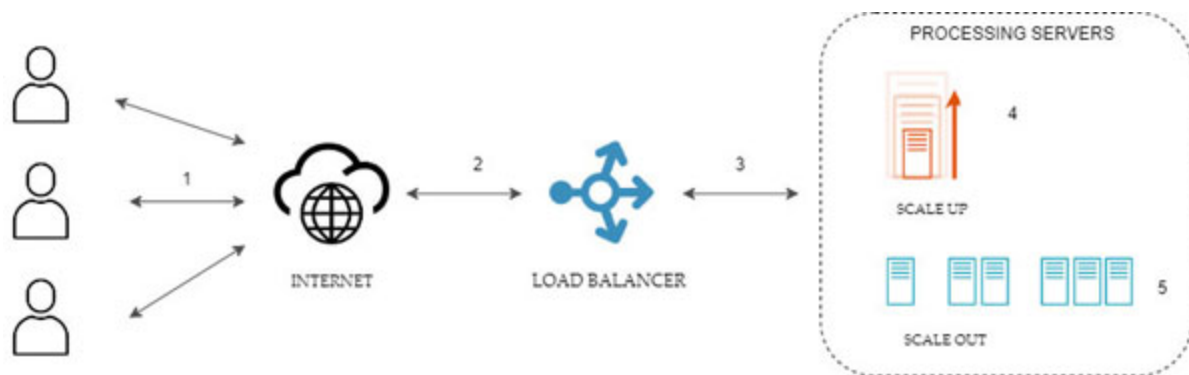
Categories of unknown factors include examples related to data. For instance, we are running a distributed commuting job, which does a group by operation, and in specific runs, there is a spike in the key to the group by operation. There will be apparent delays due to long shuffling time. As another example, let us say we have an application with memory leak issues. No matter how well you predict the infra, it will take a longer time or might even fail under certain situations.

Hence, either manual or scheduled or autoscaling, scaling decisions depend on known factors. The magnitude of these

factors cannot be generalized across runs with pinpoint accuracy, and instead, it depends on a run to run. For example, each run has its CPU utilization or memory utilization. If we want the correct scaling, it is imperative to scale up and down, based on metrics in each run. Let us have a look at a few common examples available across cloud providers in general.

## Example 1

Consider the following [Figure 3.3](#), which depicts a front-end application, where scaling can happen based on the number of incoming requests. Please refer to the following figure:

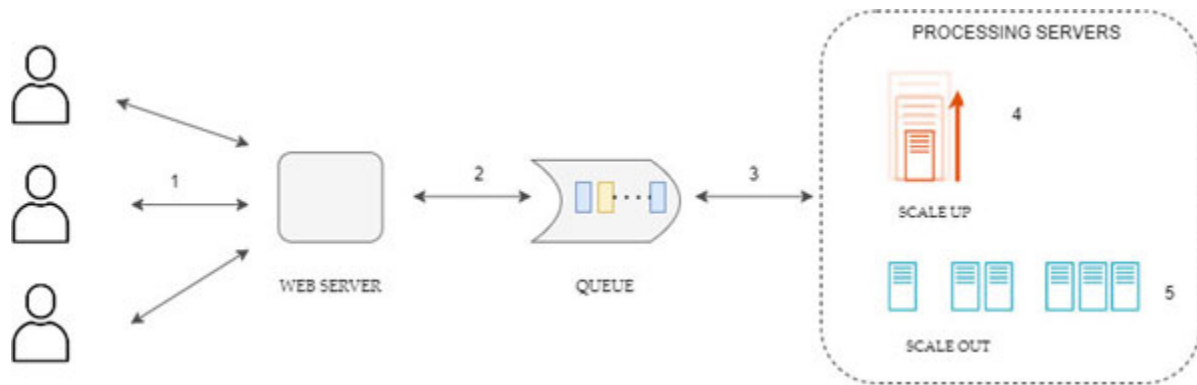


**Figure 3.3:** Scaling front end application

In the [Figure 3.3](#), users are sending requests over the internet (1), and internet requests are transferred to the load balancer of the application (2). Load balancer passes the request to appropriate servers (3). Servers handling the request could be configured for scaling up (4) or scale-out (5).

## Example 2

Consider [Figure 3.4](#). It shows an application acting as a subscriber to a queue. Scaling of the application can happen on the number of unread messages in the queue. Please refer to the following figure:



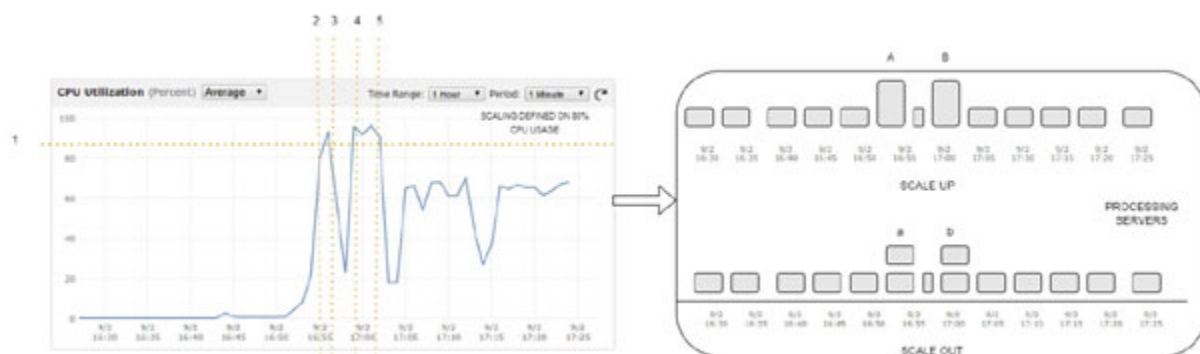
**Figure 3.4:** Scaling a subscriber to a queue

In the [Figure 3.4](#), users submit requests to a web server (1). Web Servers place the user requests in a queue (2). These user requests (messages in queue) are pulled by the application or pushed by the queue to processing servers (3). Based on the statistics of the number of messages, the processing servers can scale up (4) or scale-out (5).

## **Example 3**

### **Internal factors: Scaling based on CPU and memory spikes**

Consider the [Figure 3.5](#), which demonstrates infrastructure scaling based on CPU usage:



**Figure 3.5:** Scaling based on CPU usage

The left half showcases the average CPU usage for an application. The scaling configuration is set to be 80% (1), that is, whenever the CPU usage goes above 80%, the

system will scale (up or out and vice versa). For the assumption, let us say that the time between 16:30 to 16:55 needs one CPU. At 16:55, there is a spike in average CPU usage (**2**), and hence the system scales up (**A**) to a larger machine or systems scale out (**a**) to 2 machines. There is a dip (**2**) and a spike (**3**) in average CPU usage, and because of that, the system scales down (**B**)/scales in(**b**) and scales up(**C**)/scales out (**c**).

The example uses average CPU usage for the demonstration of strategy. We can use average Memory spikes in place of CPU usage, and the picture will remain similar.

## **Example 4**

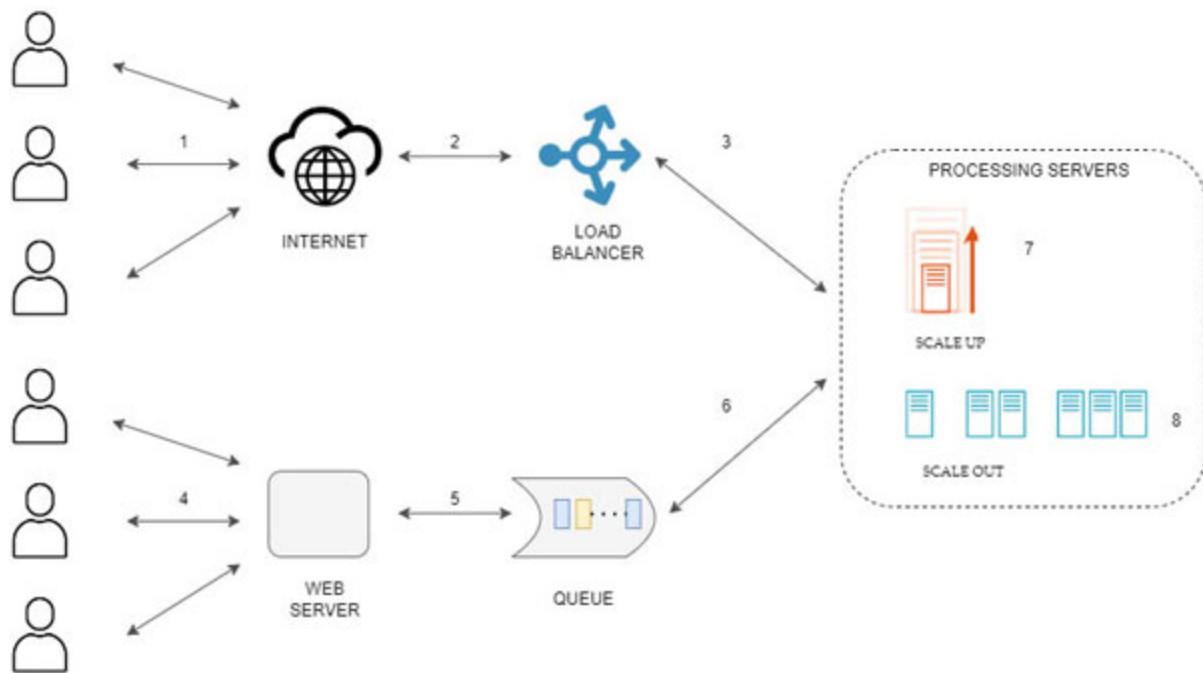
### **Combination of two or more internal and external factors**

The preceding three scaling up/out strategies are provided mainly by all the cloud providers. However, due to application complexity, there might be a need to have custom scale up/scale out situations and hence custom scaling metrics. The custom scaling metric here could be a combination of two or more metrics defined.

Assume that we have an application that combines [Figures 3.3](#) and [3.4](#). The application accepts triggers in two ways: one where users can submit processing requests directly by an HTTP call over the internet (**1**, **2** and **3** in the [Figure 3.6](#)), or the users can submit a request to a front-facing web server, which then places the processing request in a queue and application takes one task/message from the queue and processes it (**4**, **5** and **6** in [Figure 3.6](#)).

One obvious advantage of the first approach is that the user requests get processed right away. For the second approach, the advantage is a more resilient request submission. On the contrary, the disadvantage of approach 1 is that the user's requests might fail due to system unavailability. For

approach 2, there will be a delay in the request submission and actual processing. Approach 1 is generally used when the human's actions trigger application processing, whereas approach 2 is typically taken when the users are system users (another application starting processing). [Figure 3.6](#) features scaling based on multiple factors:



**Figure 3.6:** *Scaling based on multiple factors*

Since the application accepts processing triggers via HTTP over the internet and via Queue messages, an appropriate scale-up/out could not solely depend on one metric, that is, number of HTTP requests or number of unread messages in a queue.

In this situation, we can define a custom scale metric in two ways:

- **OR condition:** Scale when the number of messages increases to more than 5 in a queue or scale up when the number of incoming requests exceeds 20. Though it is technically possible to have such a configuration, it might generally not be an ideal strategy.

- **AND condition:** Scale when a combination of a number of requests and number of messages reach a threshold value. For example,  $a * (\text{number of the incoming HTTP request}) + b * (\text{number of unread messages}) > 80$ , where a and b are multiplying factors depending on the type of application and SLA defined. The greater the value of a or b, the more aggressive the scaling.

Manual scaling, as well as schedule scaling, both have their pros and cons. To overcome those, cloud providers provide better elasticity strategies.

- SaaS services automatically come with a default strategy implemented, which could be configured properly as per need. For example, the app engine and cloud functions in GCP both have a default scaling configuration based on the load at run time, that is, even if nothing is done by the engineering team to support autoscaling, it is already implemented. We will deep dive more into this topic in the upcoming chapters.
- SaaS service, where an application deployed has no default scaling strategy in place. The complete onus lies on the engineering team, to enable scaling of applications. Kubernetes services available in GCP (that is, **Google Kubernetes Engine (GKE)**) or Azure (that is, **Amazon Kubernetes Service (AKS)**) have, by default, no runtime scale-up configuration configured, for horizontal and vertical scaling of applications. It has to be done by the engineering team.

## Benefits of elasticity

We have discussed multiple aspects of the elasticity of hosting applications on the cloud. We also looked into the advantages and disadvantages of scaling, in general, on the cloud. It is time to be a little more specific and discuss the

various advantages and disadvantages of elasticity. Aspects in this section have already been discussed before when we discussed scaling in general. Here we will highlight the benefits that autoscaling brings to the scheme of things.

## **Painless and optimal scaling**

The elasticity gives a better scaling experience for applications. With elasticity, teams can ensure optimal use of spun resources, that is, whatever is being spun is actually in use – no over/under provisioning. In contrast to elasticity, manual scaling is scaling up and down predicted, based on historical loads or the scaling used to meet SLAs in peak load.

## **Justified costs**

With all cloud providers supporting the pay-as-you-go budget, there is no upfront cost associated with increasing infrastructure size. But if it is not handled with care, an over-provisioned infra could result in unnecessarily high costs. As the application matures, it begins to bring even the slightest of changes. So, it is crucial to understand application scaling needs during architecture design, and implement it accordingly. Auto-scaling provides the most optimized solution for cost, by reducing the possibilities of over/under provisioned systems.

## **More redundancy and flexibility**

IT teams can host infrastructure across multiple geographical regions to control disruptions happening in one region. For example, the computer infrastructure could be configured so that if one region cannot spin up VMs, the VMs get spun in another region automatically. This cross-regional strategy is more frequently used while saving mission-



critical datasets. Datasets stored across regions are supported with flexibility, reliability, and automated recovery options.

Public cloud providers support features which facilitate redundancy and flexibility well in the following manner:

### **Considerable capacity**

We get almost infinite compute and storage capability, as needed. In some situations, a cloud provider might have some restrictions which need to be checked on official documentation, but generally, those restrictions are hand full.

### **High availability**

The ability to spin up additional redundant resources, helps reduce unnecessary slowdowns and disruptions. Strategies like canary and blue-green deployment, could be very easily used on the cloud. Canary deployment is a strategy where we deploy code for a small subset of users, let them use it, and once confirmed that the functionality works as expected, changes could be rolled out to all users.

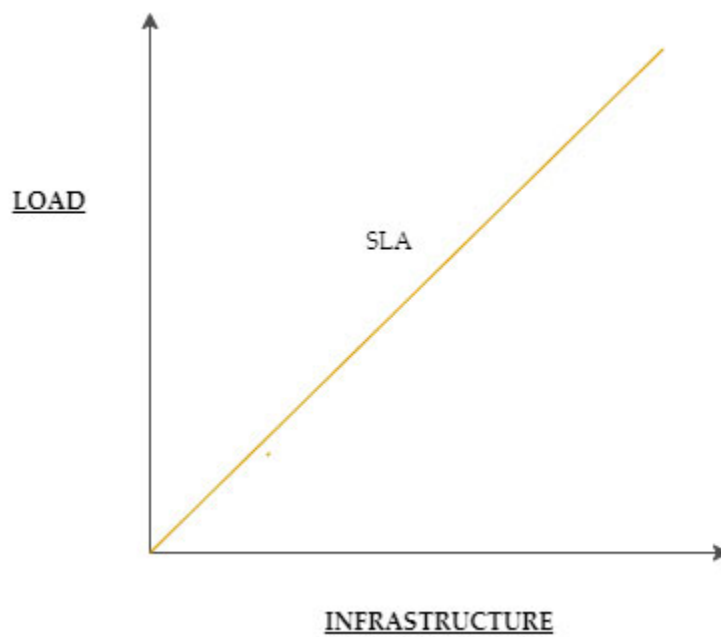
### **Simple management**

IT teams do various activities to manage and update the infrastructure. This includes patching operating systems, updating software, and making sure that the infrastructure is secure and robust under high load. To ensure that the infrastructural components are working well, IT teams monitor and alert issues in the infrastructure 24\*7. With cloud providers supporting elastic infrastructure, all these aspects are supported automatically. The whole management of such activities is abstracted away from the IT team.



## Elasticity and cost relationship

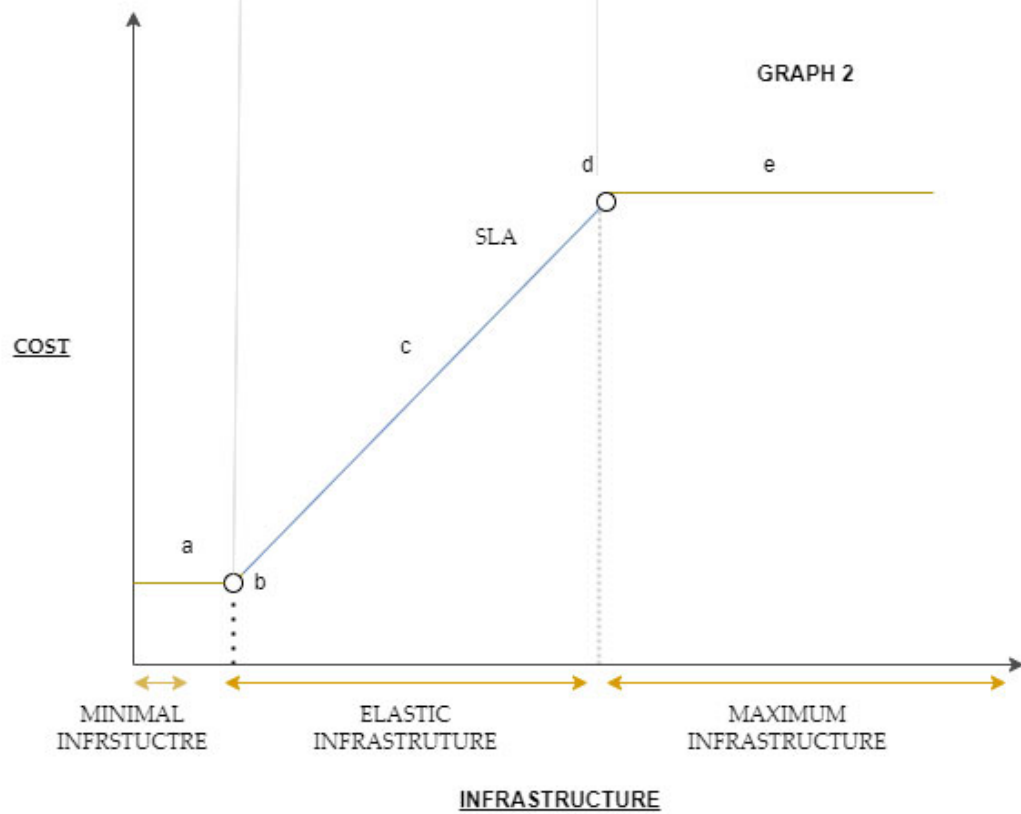
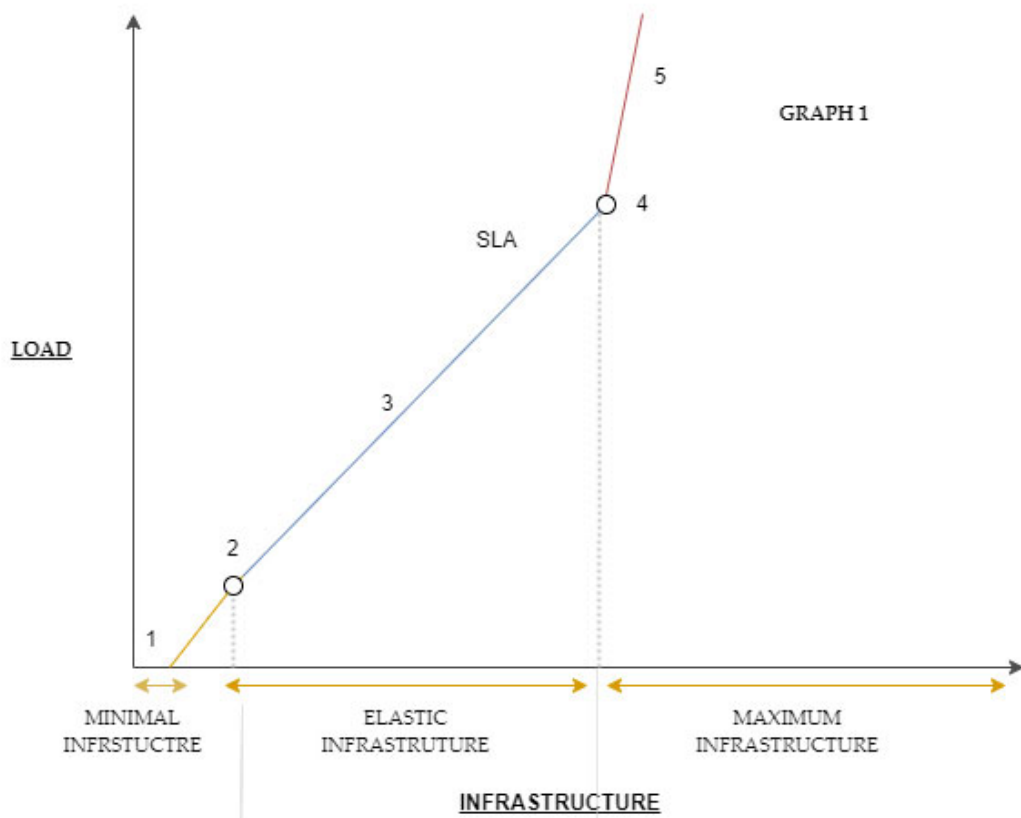
As per the discussions till now, it is easy to appreciate the fact that elasticity enables dynamic scaling intending to save teams from over provisioning and under provisioning of cloud resources. We want the spike in data or processing to be a function of infrastructure. Consider [Figure 3.7](#), which shows the relationship between infrastructure and load. The greater the load, the greater is the infra; the same SLA is the mantra. Please refer to the following figure:



**Figure 3.7:** Ideal load, infra and SLA relationship

In [Figure 3.7](#), we can see that SLA is a function of infrastructure and load. If we can ensure that infrastructure increases proportionally with an increase in load, the SLA or the slope will remain constant. The value of SLA is defined by business, but to achieve that is the responsibility of the engineering team in all conditions. This is an ideal scenario and only possible if the application is hosted on truly infinitely scalable infrastructure. Under infinite load, applications will scale infinitely to meet SLA and vice versa.

The phrase “*Infinite scalability*” looks fancy, but each application should be defined with a lower and an upper bound. A lower bound makes sure that the application responds quickly, and an upper bound makes sure that one application does not take all the resources and start impacting other applications. Consider [Figure 3.8](#):



**Figure 3.8:** Relationship between load, elasticity, SLA and cost

Graph 1 represents a more realistic load vs. infrastructure mapping. Graph 2 illustrates the cost vs. infrastructure mapping. Application is constantly running with minimum load (1), and hence there is always a minimal cost associated (a). When the load increases, it is tackled by the minimal infrastructure only up to a particular limit (2 in Graph 1) and hence the cost remains constant for the initial load (b in Graph 2). Once the threshold is reached, the system starts scaling up (3), and so does the increase in cost (c). Once the maximum configured limit is reached (d and 4), the SLA starts breaching (5) while the price remains constant (e).

As we can see, when the system is leveraging the elasticity of the cloud within a minimum and maximum limit, the cost is directly proportional to infrastructure and is directly proportional to load. Minimum and maximum prices remain constant.

## **Key challenges**

The biggest challenge to optimally achieving elasticity is that there cannot be one common general strategy for applications. Just as the number of applications can differ, so can the permutation and combination of metrics and values for elasticity. Generally, engineering teams start with an educated guess of strategy and a logical guess of one set of values for metrics. However, an application might take several tuning iterations to achieve the proper elasticity configurations. In this section, let us look at some key hurdles.

## **Identifying the right attributes/metrics to track**

Identifying a metric whose value will define the scale up and down scenarios might be straightforward in cases where the triggers of load increase are clearly defined. For example, when we already know the load definition for our application, for instance, in the case of one of our previous scenarios ([Figure 3.3](#)), in the HTTP REST services use case, the load could be the number of incoming requests. However, in cases where we combine two or more metrics, it becomes problematic. For example, let us assume that we have an application that can be triggered via REST call and by placing messages in a queue ([Figure 3.6](#)). Here, it is difficult to identify the correct contributions of each load increase aspect. For simplicity, let us assume we defined a custom metric-based linear relationship of REST calls and messages in the queue.

$$\text{CUSTOM SCALE ATTRIBUTE} = a * \text{Number of Requests} + b * \text{Number of unread messages.}$$

Identifying the correct value of a and b is complicated.

## **Identifying the right scaling measurement value**

Once we get the right attributes/metrics to track, another vital question is what the scale-up and scale-down threshold conditions should be. An example threshold condition is scaling up when the average CPU usage is above 80%. The question that arises is how did we arrive at the number 80%? Nobody can give a perfectly optimized number. Engineering should start with a conservative number and slowly start optimizing it with an eye on SLAs. The moment a breach of SLA is seen, revert to the previous step.

## **Defining the minimum and maximum limits**

Defining the minimum and maximum limits is tricky, not from the aspect of the engineering team, but more from the business side. If you ask any business person or product owner, they will always want an ideal situation ([Figure 3.7](#)). However, the engineering team discusses and tries to align expectations of business person to that of what can be offered. Since multiple parties are involved, this becomes a bit challenging in aligning everybody to a common understanding.

## **Cost spikes**

Till the time-optimal metrics and their value are identified, engineering teams should start with slightly oversized infrastructure. An undersized infrastructure will mean breaching SLA, which means loss of business and credibility in the worst cases. It is also important to quickly correct the oversized infrastructure because we are unnecessarily paying extra cloud costs. Maintaining a balance between elasticity and cost is a challenging exercise.

## **Difference between scalability and elasticity**

Elasticity and scalability are often used interchangeably by teams deploying workloads on the cloud. Both might sound similar, but there are some subtle differences between them. [Table 3.1](#) talks about those differences:

Cloud Elasticity	Cloud Scalability
Elasticity is to scale up/down and scale in/out automatically, as per the load on the system.	Scalability on the other hand is to scale up/down and scale in/out, performed explicitly by the IT team/engineering team.
It is primarily used for use cases where the workload and demand	Scalability is primarily used where there is a consistent need for increased infrastructure.

increase only for a specific amount of time.	
This is an exercise which might need multiple iterations of performance runs to configure properly.	Main driving factor here is the meeting of SLAs. Generally easy to achieve in lesser iterations.
Cost effective, as the infrastructure is not over provisioned or under provisioned at any given point in time.	Can lead to unused infrastructure, and hence exaggerated costs.
Each cloud provider has a different strategy and underlying implementations to support elasticity. In situations of multi cloud deployment or hybrid deployments, it becomes difficult to host apps leveraging each cloud provider's elasticity facilities to full potential.	In case of manual and scheduled scaling, achieving both in a hybrid and multi cloud environment is easier. A similar strategy could be easily implemented.

**Table 3.1:** Elasticity vs. scalability

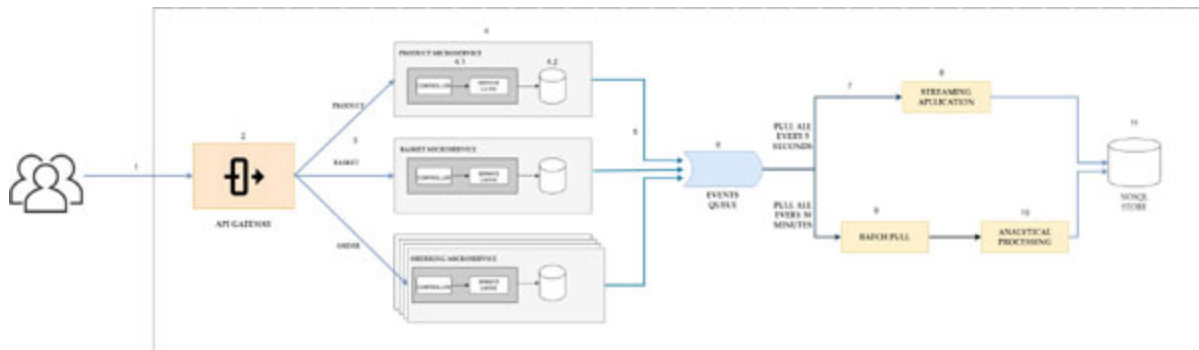
## Use cases

In this section, we are going to have a look at 2 use cases, firstly of an eCommerce application whose nature is batch job and secondly, a use case of song streaming application, whose nature is streaming. The use case description will primarily describe the scalability and elasticity needs of these two scenarios.

## eCommerce application

Let us take for an example, architecture of a typical eCommerce application. This only represents a part of an eCommerce application; actual architecture could vary organization to organization. In the following eCommerce application architecture, micro service architecture has been followed for service responding to user requests. In addition to that, the data coming via different microservices is logged on the queue and is further analyzed by streaming as well as batch jobs. Let us look at each component one by

one and discuss the aspects of elasticity, as shown in [Figure 3.9](#):



**Figure 3.9:** eCommerce application

Consider the preceding numerically labelled figure (eCommerce Application) with the following subsequent numerically labelled explanations:

1. Users trigger various API via a front-facing UI - either web-based applications or mobile applications. These requests go via API Gateway servers.
2. **API Gateway Servers:** An API Gateway is a component that takes in all the requests from outside users and does two main activities - requesting multiple microservices and triggering more than one microservices, aggregating the result, and responding. There are API gateway servers available as PaaS across all cloud providers. All the PaaS gateways are auto-scalable, that is, as the number of requests increases, the underlying infrastructure is scaled up and vice versa.
3. API Gateway servers route the request to individual microservices. For example, a product microservice has all CRUD operations for a product.
4. Each logical entity can be combined as one microservice. Cloud-specific technology is available for hosting such microservices (labelled as 4.1 in the [Figure 3.8](#)). For example, in GCP, we can use an app engine or



cloud run to host microservices. In Azure, we can do the same on Azure cloud services. If we select these options, we will be able to leverage the full capability of elasticity offered by the cloud. However, it could become challenging to manage when we have a hybrid or multi-cloud deployment.

In such situations, we mostly opt for container-based deployment on microservices. Kubernetes has proved to be the most widely used container-based orchestrator, managing the containers. In Kubernetes, however, there is no default autoscaling available. An application developer has to develop the configuration to do so. The auto-scaling metric could be the number of incoming requests in this scenario.

It is also preferred to have the database separate for individual microservices (Labelled as 4.2 in [Figure 3.8](#)), as multiple microservices concurrently scaling using the same database could lead to scaling bottlenecks. These databases are available on the cloud as PaaS, like MySQL available in GCP and Azure as PaaS. Cloud providers do the complete management. In some cloud solutions available, cloud providers expect an SLA, how many MB of data is read and written, and cloud providers auto scale behind the scenes to support the SLA.

5. The events generated by different microservices are pushed to a queue.
6. Queues are available as PaaS, which is auto scalable. As the number of messages increases, the underlying infrastructure also scales up. While there is no minimum limit to the number of messages, some solutions on certain clouds also expect a maximum limit. For example, there is a limit to a maximum size of a queue in the *Azure Service Bus*. Similarly, there are limitations on the number of subscribers and the number of topics

that depend on the class of services used. Before selecting a queue solution, it is crucial to analyze these restrictions to minimize the surprises.

7. The messages in the queue (*point 6* in [Figure 3.8](#)) could be pulled with different strategies. For a streaming application, the pulling of messages from the queue could be high in frequency, like once every 5 seconds. It could be different for batch jobs, a more infrequent one, such as once every 30 minutes.
8. It is a streaming application where multiple technologies can be used. It could be PaaS or SaaS services. Their management effort depends on the selected choice. However, here the critical auto-scaling strategy will be the number of unread messages in the queue. Streaming applications have very short SLAs, like calculation in 2 seconds; if the number of messages pulled by a streaming application at an average is 25 and it can meet the SLA when the number of messages becomes 50, 2 instances are required to complete the SLA.
9. This component is not always running but gets triggered every 30 minutes to pull all the unread messages after the last pull. As this is a job that gets started once in half an hour, hosting this as a persistent application does not make sense. Cloud providers do provide mechanisms for such use cases. For example, in GCP, we can set up a cloud scheduler API, which triggers a cloud run (serverless) every 30 minutes. The infrastructure requirements of cloud run will be taken care of by GCP.
10. It is a batch job that runs on half-hour data and produces reports. Batch jobs could be written in a distributed framework like *Apache Spark* or *Apache Hadoop*, which are open-sourced, or in a cloud-specific distributed framework like *Azure Data Factory* or *GCP*

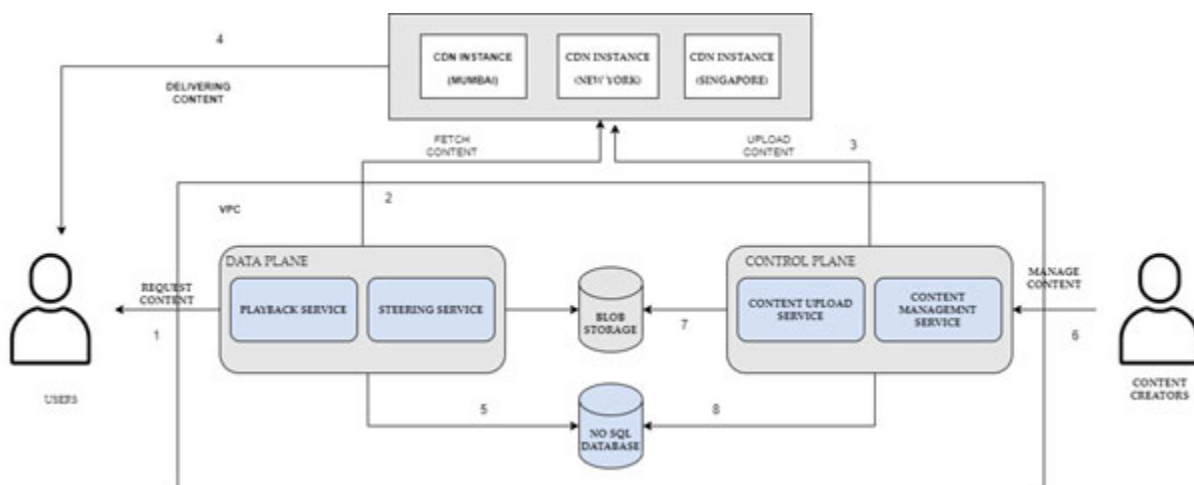
*Dataflow*. Cloud-specific solutions are serverless and hence ideal from an elasticity point of view.

11. The data results of streaming application (**8**) and batch application (**10**) could be saved in a no SQL columnar/document database. Cloud providers have their own hosted solutions for both the flavours. For example, in GCP, we have *Cloud Bigtable*, which is nothing, but GCP managed *HBase* (a columnar DB) or *Google Datastore* (which is a document DB).

## Song streaming application

Let us look at another use case of song streaming application. In a song streaming application (described in [Figure 3.10](#)), the preceding described microservices, real time streaming, and batch processing use cases do exist and the elasticity thought is precisely the same. In the big data use case, we primarily concentrated on compute elasticity. What we are going to discuss here, is the elasticity of network and storage.

In song streaming or for that matter any streaming, the main business is to deliver audio and video fragments to users with low latency. [Figure 3.10](#) features a song streaming application:



**Figure 3.10:** *Song streaming application*

Consider the preceding numerically labelled figure with the following subsequent numerically labelled explanations:

1. User request for an audio content or video content from data plane services. The data plane services follow the microservice architecture and align with elastic scenarios described in the eCommerce use case, point no 4.
2. The data lane service tries to fetch the content from **Content Delivery Network (CDN)**. CDN is a geographically distributed group of servers that deliver the blob contents. Having the servers distributed across multiple regions is another flavour of elasticity. If a need arises to support users from a new region, a CDN environment setup is needed specific to that region.
3. If the CDN servers do not have the requested media file, they use control plane services to fetch the data from the blob location and update the data to the regional CDN server.
4. Once the data reaches the CDN server, it is served to the end user.
5. All metadata related to these activities are logged in a Database.
6. Just as we have the users served via data plane, another set of platform users known as content creators can use a family of microservices referred to as control plane.
7. These control plane services upload content to a central repository, called the regional blob storage. This blob storage is auto replicated across regions and has all the features of disaster recovery implemented.
8. The Control plane service also logs all their metadata of activities to a database.

## Conclusion

In this chapter, we deep dived into the concepts of elasticity on public clouds. Elasticity becomes a very important aspect on cloud platforms simply because of the pay-as-you-go model. On cloud, more infrastructures mean more cost, and hence making sure the price does not blow up depends on leveraging the elasticity provided by cloud platforms, and this could really help teams achieve more.

In the next chapter, we will look into the challenges of infrastructural complexity, arising due to multi cloud and hybrid cloud deployments and the way such situations can be tackled in an enterprise.

## Points to remember

- All public cloud providers give efficient support elasticity of compute and memory. We should always try to use the options available in cloud, out of the box, since the management of such critical activity will remain with the cloud provider team, rather than us handling it.
- Leveraging elasticity on public cloud providers is easy, but using elasticity efficiently is difficult. You will have to investigate multiple aspects before taking decisions. Do not forget: cost and elasticity go hand in hand. Unjustified elasticity could lead to unjustified costs.
- The metric to scale up/down can vary application to application, and it is advised to think about this during the architectural design phase of functionality.

## Questions

1. **Among the following statements, which is false for cloud elasticity?**

- a. The property of a cloud's capacity to grow or shrink for CPU, memory, and storage resources to adapt to the changing demands of an organization.
  - b. Cloud elasticity is often associated with horizontal scaling.
  - c. It provides businesses and IT organizations the ability to meet any unexpected jump in demand, dynamically. No stand by infrastructure for peak load, rather scaling happens as and when needed.
  - d. Elasticity is easy to use and you should always define maximum possible limits of elasticity for your application to make it infinitely scalable.
- 2. What are the challenges in cloud elasticity?
  - 3. What are the benefits of cloud elasticity?
  - 4. How does cloud elasticity work?

## **Answers**

- 1. **d.** It is not easy to use cloud elasticity optimally. Ideally, every application/workload architected on cloud should have aspects of elasticity covered in design.

# **CHAPTER 4**

## **Challenges of Infrastructure Complexity and the Way Forward**

### **Introduction**

There are many reasons why enterprises opt for multi-cloud or hybrid cloud deployment. When organizations start their journey on a cloud, complete workloads are not available right away on the cloud. Instead, they start small, that is, one application after another is migrated to the cloud. However, that migrated application needs to interact with the other applications that are running on a different cloud or on-premise environment, where they were originally hosted before migration.

Another reason why it is inevitable to use this deployment model is because companies do not want vendor locking or the application needed for a client on the cloud of their choice. Public cloud providers have different regulatory and data sovereignty capabilities.

There could be more reasons to support this form of deployment; however, some key aspects must be considered when an organization decides on such strategies. These concerns are security, operation, and governance. This chapter will discuss the elements that need to be considered before designing and adopting this strategy.

### **Structure**

In this chapter, we will discuss following topics:

- Defining multi-cloud and hybrid-cloud deployments
- Multi-cloud deployment model
- Hybrid cloud deployment model
- Need Of multi-cloud deployments and hybrid-cloud deployments
- Challenges of multi cloud deployments and hybrid cloud deployments
  - Security
  - APIs
  - Logging
- **IaaS vs PaaS**: Benefits and challenges of choosing one over the other
- Governance and way out
  - Effective communication
  - Effective planning
  - Proper auditing
- Cloud agnostic automation: benefits and risks

## Objectives

In this chapter, we will dive deep into the needs of multi-cloud/hybrid deployments and investigate some key challenges that modern organizations face. We will also assess whether to use IaaS or PaaS from a multi-cloud perspective. In the aforementioned process, we will also see how to ensure governance of critical components and the cloud automation strategies, to ensure successful management of elements adhering to this model.



## **Defining multi-cloud and hybrid-cloud deployments**

Multi-cloud and hybrid clouds are distributed deployment models, where one part of the application runs in the cloud and the other runs either on a different cloud (multi-cloud) or on-premise (hybrid).

There are two modes of deployment: redundant deployment and distributed deployment.

### **Redundant deployments**

In redundant deployment, the complete application is deployed across multiple platforms. For example, a complete application is deployed on one public cloud in multi-cloud deployment, and a full copy of the application is deployed on the other. In this, not a lot of code changes or refactoring is needed for the business logic. Some examples of redundant deployment are hybrid environment, business continuity multi-cloud or hybrid cloud storage and cloud bursting.

### **Hybrid environment**

Non-production workloads like development, testing, and performance runs, happen on the public cloud. Production workflows are executed either on-prem or in a public cloud, unlike non-production environments.

### **Business continuity multi-cloud or hybrid cloud storage**

To avoid single points of failures in data storage, backups, archives, and standby systems are deployed redundantly across multiple clouds.

## **Cloud bursting**

A sudden spike in the workload can be handled by delegating the processing capability to a redundant deployment, present on other cloud environments.

## **Distributed deployments**

In a distributed model, a part of the application is deployed on one platform and another on a different platform. For example, in multi-cloud deployment, a part of the application is deployed on public cloud one, and another is deployed on public cloud two. Some examples of distributed deployment are tiered hybrid, partitioned multi-cloud and analytics hybrid/multi-cloud.

### **Tiered hybrid**

An example of this can be a big data use case batch job, running in the cloud on an elastic Hadoop cluster, that results being served via API deployed on-prem.

### **Partitioned multi-cloud**

In this, we deploy the application in multiple regions. It could be due to customer business priorities and obligations; it could also enable processing close to the customer to reduce latency.

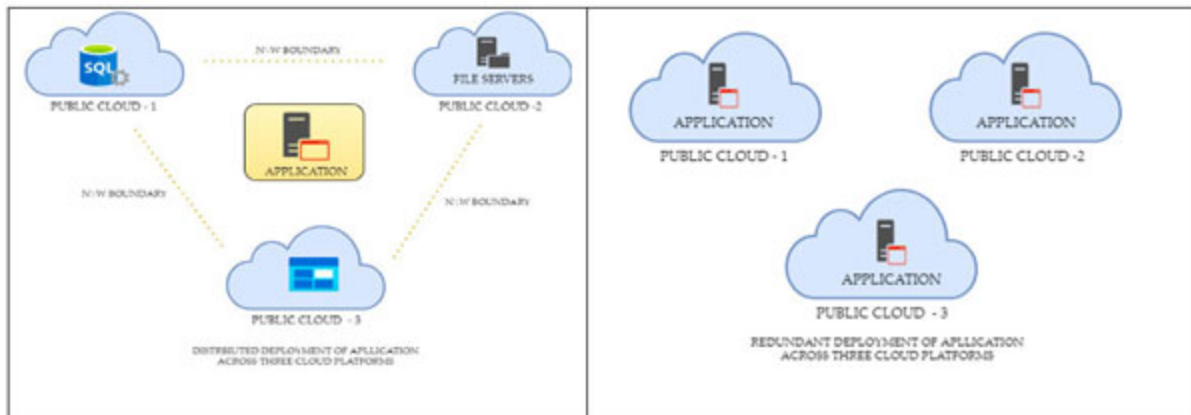
### **Analytics hybrid/multi-cloud**

Processing happens in an on-prem environment, and the result has been pushed to cloud infrastructure for consumption.

## **Multi-cloud deployment model**

It is the deployment of applications on IaaS, PaaS, and SaaS solutions across multiple cloud platforms. [\*Figure 4.1\*](#)

features a diagram of the multi-cloud model:

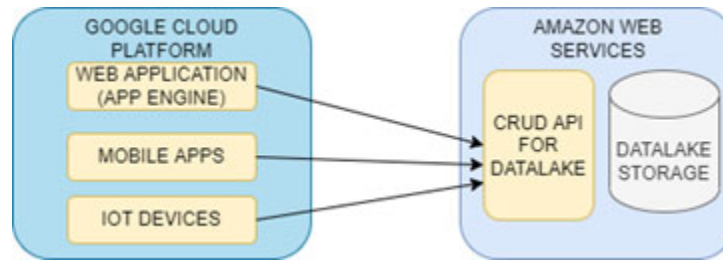


**Figure 4.1:** Multi-cloud deployment

In this figure depicting multi-cloud deployment (left image), an application uses SQL database hosted in **PUBLIC CLOUD -1**, file servers in **PUBLIC CLOUD -2**, and blob store in **PUBLIC CLOUD -3**. The application resides within the virtual network boundary, spanning multiple cloud providers. This is distributed deployment of a multi-cloud strategy. On the right side, the complete application is deployed on **PUBLIC CLOUD -1**, **PUBLIC CLOUD -2**, and **PUBLIC CLOUD -3**. This is the redundant model for multi-cloud deployment.

A multi-cloud strategy helps avoid vendor locking, improves business continuity by being less susceptible to **Distributed Denial-of-Service (DDoS)** attacks and **Single Point of Failure (SPOF)** incidents, and provides flexibility for teams and clients in terms of options to choose from.

An application in a multi-cloud environment needs to be developed so that the same business logic successfully interacts with APIs from multiple clouds. Consider the following [Figure 4.2](#), which showcases a real-world example of multi cloud deployment:

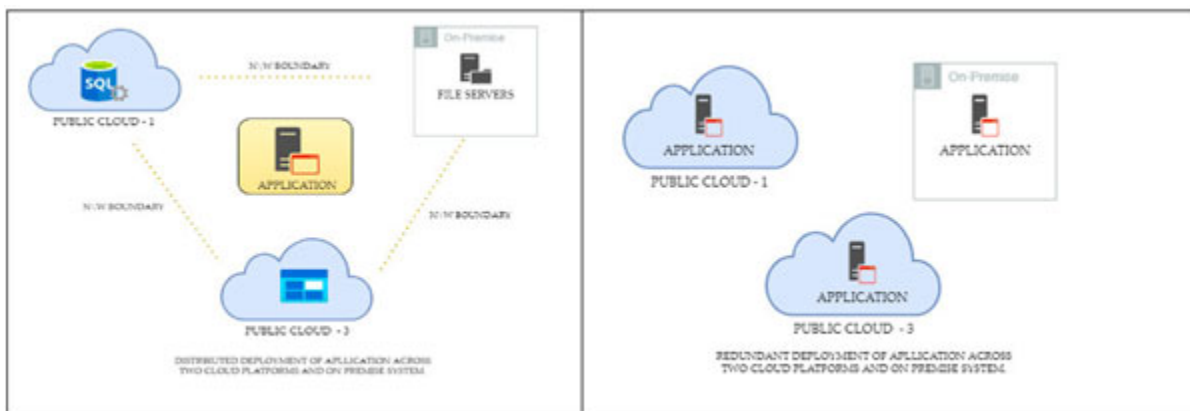


**Figure 4.2:** Multi-cloud deployment example

In the preceding image, a data lake, and its crud APIs are hosted in Amazon Web Services. The applications consuming this data lake reside in Google Cloud Platform.

## Hybrid-cloud deployment model

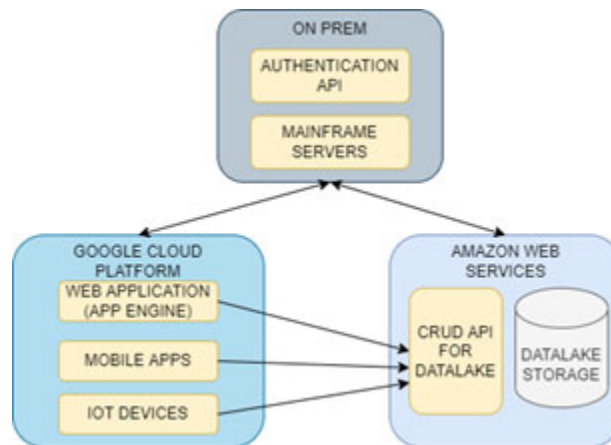
It is the deployment of applications on IaaS, PaaS, and SaaS Solutions across multiple cloud platforms and at least one on-prem environment. [Figure 4.3](#) features a hybrid-cloud model:



**Figure 4.3:** Hybrid deployment

In this figure depicting hybrid deployment, in the left section, an application uses SQL database hosted in **PUBLIC CLOUD -1**, file servers in On-Prem, and blob store in **PUBLIC CLOUD - 3**. The application resides within the virtual network boundary, spanning multiple cloud providers. This is distributed deployment of a hybrid strategy. On the right side, the complete application is

deployed on **PUBLIC CLOUD -1, On-Prem,** and **PUBLIC CLOUD -3**. This is a redundant model for multi-cloud deployment. Consider the following [Figure 4.4](#), which shows a real-world example of hybrid deployment. This is an extension of the example shown in [Figure 4.4](#):



**Figure 4.4:** Hybrid deployment example

In the preceding figure, the data lake and CRUD APIs reside in AWS and its consuming application resides in GCP. However, the authentication APIs are hosted on-prem. Also, the mainframe server of the organization is on-prem.

In hybrid cloud deployment, on-prem infrastructure can be extended to public clouds. Public clouds being scalable can help applications handle use cases like sudden burst in load, backup and archiving of data, performance runs, and so on. The most common implementation of this use case is the migration of on-prem to the cloud, where the application is migrated in small chunks to the cloud, without disrupting business needs.

All major public cloud providers appreciate the need for this deployment model, which is why cloud providers support multiple tools to implement it. For AWS, a hybrid deployment can use *Storage Gateway* and many other services. Similarly, for Azure, we can use *StorSimple*.

## **Need of multi-cloud deployments and hybrid-cloud deployments**

Let us look at some key business drivers due to which organizations adopt multi-cloud or hybrid deployment.

### **Reducing dependency/avoiding lock in**

Hybrid and multi-cloud deployment models reduce the compulsion to depend on one cloud provider. High dependency reduces the flexibility for clients in terms of the adoption of applications, as most clients have their preferred cloud vendors. So, if the application is available to get deployed on the selected cloud vendor for customers, it makes adoption easy and smooth. The multi-cloud environment does provide an opportunity to innovate faster, as different cloud providers have different strengths and weaknesses.

### **Heterogeneous deployments within an organization**

There are multiple teams whose projects come together to deliver value for an organization, within an organization. These teams can have different cloud provider implementations. When such a situation occurs, the multi-cloud strategy helps all the applications integrate and deliver value quickly.

### **Regulatory and data sovereignty**

Each cloud provider may not have the same level of maturity in terms of a particular country's regulations and data sovereignty laws. Data sovereignty means that each country has its own rules and regulations for capture and

process data. Every cloud provider might not adhere to all the rules set in that particular country.

Data locality and data residency are two key components where each country working on confidential data puts regulations. The local government rules data locality on moving the data outside of a specific region. Data residency refers to rules imposed by the government while storing data in some regions. Multi-cloud and hybrid environments are the answers to such situations.

## **Redundant deployment for high availability**

Redundant deployment across multiple clouds helps the customer choose a public cloud of their preference. This is a key aspect as most enterprises have selected a cloud provider and do contract to use them. Redundant deployment of applications across multiple regions (within a cloud or across multiple clouds) improves the application's business continuity numbers. Business continuity means how well your system continues to work in case of infra failures. With redundant deployments across the cloud, applications become less susceptible to DDoS attacks as well as SPOF incidents.

## **Performance improvements**

Multi-cloud deployment models help applications run geographically closer to the user locations, reducing processing time. There are use-cases where massive datasets are processed, with frequent transfer of data across regions while processing. In such cases, the time to the process increases, and hence users can see delays more than expected. In case of use cases where SLA for response time is in a millisecond, regional cross movement of calls via

network might take time and also result in breaches within the SLA.

## Cost optimization

It might seem that costs are similar across significant cloud providers. However, careful multi-cloud and hybrid deployment can significantly reduce costs without affecting the results. Multi-cloud facilitates processing closer to user location, while hybrid cloud processing when on-prem infra is free, can save cost. These are just two scenarios; there could be many more depending on a use case to use case.

## Challenges of multi-cloud deployments and hybrid cloud deployments

Though multi-cloud and hybrid deployments are essential, there are multiple inherent challenges to managing them effectively. These challenges are both technical as well as governance based. Let us look at the critical challenges of this deployment strategy.

## Increased operational complexities

Hybrid and multi-cloud deployments increase the operational complexity of the whole project. The greater the platform's diversity, the bigger are the operational complexities to be handled. The following are a few common ones:

- **Different cloud providers have different ways of authentication and authorization:** For example, in *GCP Identity and Access Management* groups, which are different from Azure, we do similar control using AD groups. The on-prem situation could be completely



different. Not just authentication and authorization, but we need to ensure that the consistent behavior for auditing, logging, and policies across computing environments are different as well.

- **Using consistent tooling and processes to limit complexity:** Each provider has its own tools and technologies. For example, the logging framework in GCP is different from that of Azure. The same is true for monitoring and alerting as well. Engineering teams can select a generic tool and technologies to bridge the gap.
- **Providing visibility across environments:** It is vital to have visibility across environments, to see if everything is working as expected. None of the services serving business use cases are working as expected. If not done, it could lead to SLA breaches and customer escalations.

Ops/DevOps team has to learn and evolve centrally governed policies for each of the concerns described above to minimize risks.

## **Increased data management complexities**

Applications deployed with multi-cloud model and hybrid model frequently need database management solutions to store and manage data. While sometimes these applications use classic options like Oracle and MySQL, there is multiple purpose-built cloud SaaS database, with advanced capabilities like memory processing, Map-reduce, and binary object storage. There is an obvious reason to use these databases for better performance and efficiency. However, if they are not handled properly, it can lead to issues stated as follows:

- **Data redundancy:** Enterprises try to create a copy of data for ease of use and performance on the cloud. However, that also exposes the risk of redundant and consistent data.
- **Data security:** Data is present at multiple places; all places should have proper authentication and authorization. Data should adhere to the same security level irrespective of the cloud.
- **Performance:** Network latency and platform differences could severely impact database performance when accessed from a different cloud or platform.

## **Data protection challenges**

All public cloud providers provide encryption at rest and in transition. However, in multi-cloud deployment, data moves in and out of a cloud provider boundary, and this data movement across platforms becomes the most significant challenge in protecting data. Moreover, each cloud provider might not have the same data protection facilities available.

It is always recommended to encrypt data when it is in a cloud boundary and when the data is in transition from one platform to another.

Defining a custom encryption strategy across cloud providers with custom-managed encryption keys is crucial to maintaining consistency from a data protection perspective.

## **Increased architectural complexities**

Since there is no uniformity across cloud providers, it is vital to consider the cross-platform vibrations in architecture design. A few key challenges are as follows:

- Dependencies between applications deployed on different cloud and on-prem platforms.

- Involvement of different cloud providers/on-prem environments open scenarios where data has to travel across boundaries, which again brings unexpected latencies in the system.
- Reliance on a specific software and hardware version that might be present on one platform but not on another.
- Different licensing restrictions of software on different cloud providers need to be considered, while architecting across the cloud. Terms and conditions in the license can impact software usage across cloud platforms.

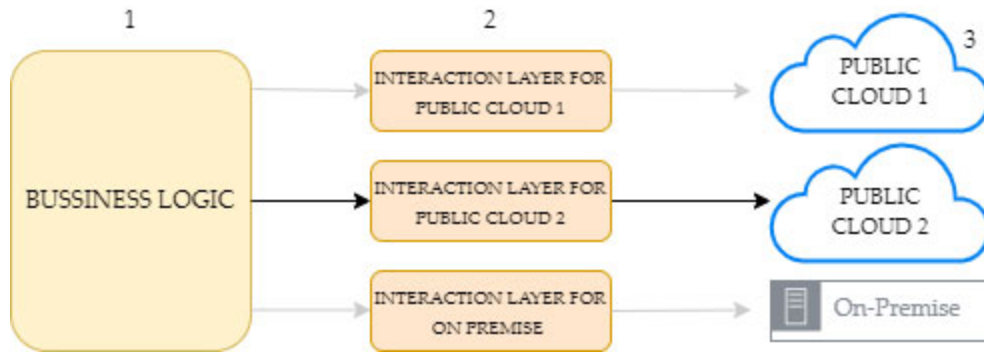
## **IaaS vs PaaS for multi-cloud and hybrid cloud deployments**

Developing an application for a multi-cloud environment needs application developers to think about the deltas in each cloud provider. For instance, consider a Blob store interaction in GCP vs. a Blob store interaction in Azure. It is advisable to have the code structured to separate business logic from infrastructure logic.

There are two strategies for a multi-cloud environment (as was shown in [Figure 4.1](#)) and two strategies for hybrid deployment (as was shown in [Figure 4.2](#)). Let us now look at the strategy of application development scenarios in both (redundant and distributed deployments).

## **Redundant deployment**

In redundant deployment, we deploy the complete application on multiple clouds or on-prem environments. This can be seen in [Figure 4.5](#):

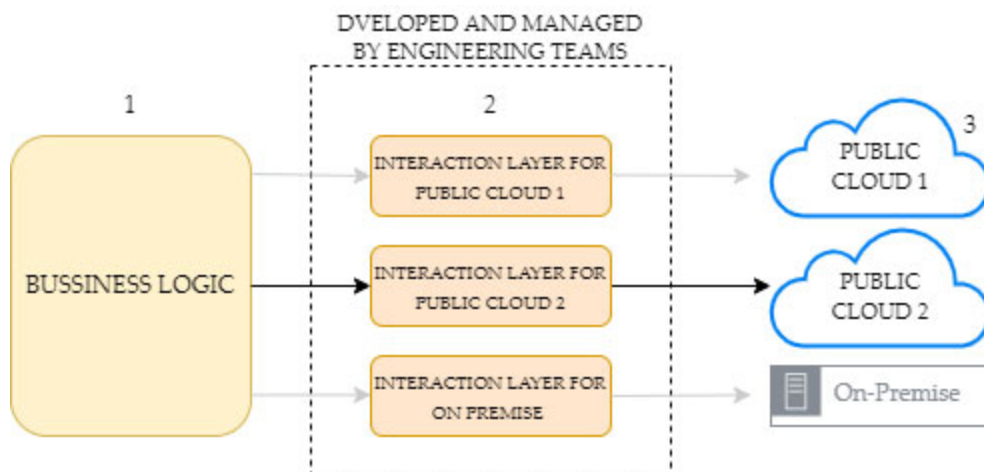


**Figure 4.5:** Redundant deployment application development

In [Figure 4.5](#), we have the business logic or the domain code (1), which interacts with cloud-specific interaction layers (2), to interact with multiple cloud providers (3).

Developing and maintaining the business logic (1) is part of the development team's responsibility. However, the interaction layer (2) responsibility could depend on the architecture.

Let us now see [Figure 4.6](#), where see the presence of the engineering teams is brought into the situation:

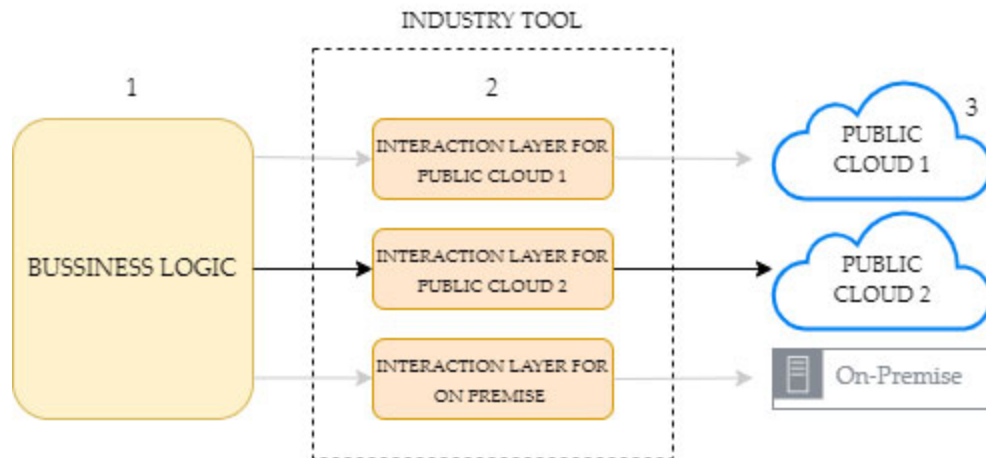


**Figure 4.6:** Redundant deployment application development

As in [Figure 4.4](#), here too the engineering teams can opt for developing their interaction layer and exposing all such interactions to business logic via custom-written interfaces. This interaction layer could be created outside of the scope

of one product, and could be used by all products as libraries.

However, there is another approach to this. The interaction layer is developed and managed by a separate team and made available in the industry as a tool. This can be seen in [Figure 4.7](#):



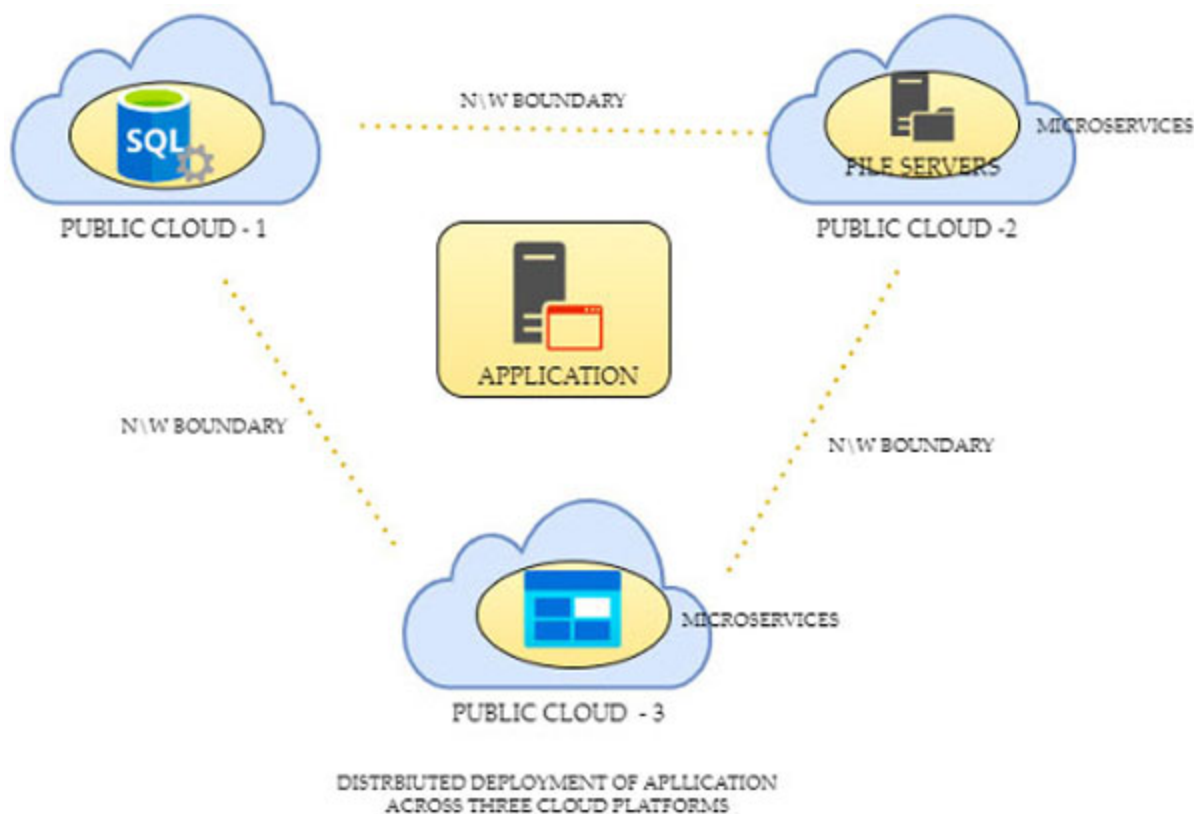
**Figure 4.7:** Redundant deployment application development

In this case, the applications are expected to know how to interact with the tool. The efforts of handling the multi-cloud environment becomes the responsibility of the tool. One such tool is *Red Hat's OpenShift Container* platform. If your application can be deployed as a container, the OpenShift Container Platform gives you operators to interact with. Red Shift manages the operator implementation, which has the responsibility to interact with different clouds. For example, an application can use a storage operator. The container platform manages storage operator implementation for each cloud provider. However, selecting one among multiple implementations depends on the installation parameters of the tool.

## [Distributed deployments](#)

In the case of distributed deployments (as was seen in [Figure 4.1](#)), the parts of the same application are deployed on multiple platforms. Hence, in this case, the primary concern for the engineering teams is how to communicate between 2 portions of applications deployed on two different platforms.

While there could be more than one way to achieve this, microservice architecture is commonly used in this situation, as seen in [Figure 4.8](#):



**Figure 4.8:** Distributed deployment application development

[Figure 4.8](#) is quite similar to [Figure 4.1](#), with just one addition: the application on different cloud providers is deployed as microservice and two applications on various cloud providers are interacting over HTTP protocol.

## IaaS vs PaaS

While developing applications for multi-cloud or hybrid environments, selecting the right tools upfront is crucial. Different cloud providers have different tools supported. Since the expectation is to have the same application code running across multiple cloud providers, selecting a tech stack that is easily manageable across the cloud makes sense.

IaaS solutions like VM and disks are the easiest to support hybrid and multi-cloud environments. However, the engineering team must manage the benefits of cloud-like elasticity.

Another option is to use PaaS, but due to different PaaS solutions, the engineering team has to make sure that the same business logic gets deployed on one technology on cloud one and a second technology on cloud 2. For example, microservices can be created using *App Engine* in GCP and *Service Fabric* in Azure. Specific PaaS services might not be available in an on-prem environment. Another example could be that of a database being used. Cloud providers offer different databases as PaaS; however, it might happen that a database available in one cloud is not available in other cloud. For example, Bigtable is managed HBase deployment available in GCP, but not in Azure.

While there will always be some differences, the most suitable solution cloud will be the one where these differences are minimal. A few such tools which can be used, are as follows.

## **Docker and Kubernetes**

Docker is a container solution, an isolated environment that contains applications, with all of them running on the same OS kernel with no relation to external OS. In short, an application packaged as docker can run in any place where docker installation is supported.

When we talk about scaling such docker deployments, we have orchestrator frameworks like Kubernetes, whose primary aim is to manage the complete life cycle of a docker container. Almost all public clouds support Kubernetes, and the inherent platform independence provided by docker, makes this a powerful solution for the microservices workloads.

These days, companies have started using multi-cloud Kubernetes, where one single Kubernetes cluster can span across multiple clouds.

## **Hadoop cluster**

Another everyday use case is that of a big data application (Hadoop, Spark, and Flink). Big data applications run on the Hadoop cluster, which again is made available by all public cloud providers; like AWS has *EMR*, GCP has *Dataproc*, and Azure has *Azure Databricks*.

## **OpenShift Container Platform**

This industry tool is available, which abstracts away the platform-specific differences at the installation level. Once installed, the application running inside remains the same. For example, once the OpenShift Container platform is installed on Azure and GCP, the same application can run on both environments.

## **Infrastructure as code**

Like application code, infrastructure can be written as code using tools like *Terraform* and *Ansible*. Rather than using cloud-specific APIs to spin up infrastructure, these tools can be used to set up similar infrastructures across clouds. These tools are well compatible with multiple public cloud providers.



## **Governance and way out**

By governance, we refer to an agreed-upon set of standards and policies based on assessments and analysis, to reduce risks. Teams owning applications agree to work on a joint expectation of tools, technologies, and processes in a multi-cloud environment. Governance also defines the other side of the story, that is, what steps to be followed in case of a breach of the governance model. For effective governance, it is imperative to do the following activities diligently.

## **Creating guidelines**

The very first task is to work upon creating and setting up enterprise guidelines which consists of standards and governance rules across all the aspects of enterprise such as business, technology, data, security, and others. The better the guidelines, the more aligned will be the behavior of teams, resulting in lesser surprises, while delivering the goals.

## **Effective communication**

It is vital to communicate the vision and strategy to all the stakeholders and get their commitments to stick to the process. It is critical to keep them updated with the goals, priorities, successes, and setbacks. Developing a mapping of high-impacted parties with the hybrid strategy, and supporting the parties informed transparently, are both essential for the plan's success. Additionally, any information relevant to stakeholders should be given on every possible channel to minimize the chances of missing out on a message.

Effective communication is vital because when driving even the smallest of changes across the organization, effective communication is essential to monitor the health of the

transformation. This ensures everybody is aligned and resynchronized in case a problem arises.

## **Effective planning**

Multi-cloud is a strategy that needs a lot of planning on multiple aspects as each organization's use case is different, and therefore, so is the multi-cloud approach. The better the plan, the lesser are the risks of failure. An effective plan includes the following aspects:

- **Prepare for complexity:** We might witness high complexity with the multi-cloud as we utilize multiple heterogeneous platforms. The complexity increases with the number and type of cloud provider platforms. It is essential to pull out metadata from each cloud provider to view a holistic image of cloud usage to make the right decision.
- **Practical cost considerations:** Cost is an essential factor, but it cannot be the only reason to use the cloud. Choosing the right vendor to deliver business value is vital.
- **Automation:** To have an effective strategy for multi cloud, it is essential to have automation. Multiple platforms with different underlying infrastructures expected to produce similar results are only possible if we have a capable automation strategy. Automation strategy should be for tasks related to application development, like the automated test, and infrastructural, like having CI/Cd pipelines to build and deploy.
- **Security is essential:** Planning for a consistent security strategy is crucial, as different cloud providers have different levels and types of security, and a consistent security policy will fill the gaps between the multiple platforms.

- **Plan for disaster recovery:** Individual cloud providers provide and support disaster recovery. But when it comes to a multi-cloud environment, a disaster recovery spanning across the environments needs to be created.

## Proper auditing

After implementing a multi-cloud deployment, we need to periodically review the strategy to ensure we are in line with our goals. Multi-cloud does not have a significant investment upfront, but we can track and adjust for practical cloud usage. For this, it is needed to set up a central team that can review and control the deployments both in cloud as well as on-prem.

The central team keeps an eye on the business needs, and we can adapt a multi-cloud strategy to adjust to new functionalities, additional applications, and other tech stack changes.

## Cloud agnostic automation - benefits and risks

Cloud agnostic means that the solution is not dependent on a specific cloud provider. Developing any workflows as cloud agnostic requires significant effort and time. The tools and infrastructure are created with features that make them deployable across multiple platforms. When the intent is to deploy the same application across multiple platforms, cloud-agnostic is the way forward.

Key advantages of using cloud agnostic approach are as follows:

- **Portability:** Applications are not tied to the platform and hence can easily be ported to a new platform.

- **Consistent performance:** The application's performance, once tuned, will work for most of the platforms.
- **Avoiding lock-in:** There is no vendor lock-in as applications can easily be moved to different platforms.

Disadvantages of the cloud agnostic approach are:

- **Misunderstandings:** People do not understand the cloud-agnostic approach correctly. They can think that cloud-agnostic code, once written, can work across all platforms; however, in practical scenarios, that is not the case.
- **Implementation barriers:** Cloud agnostic needs effort from developers, and hence it might not always be possible.
- **High cost:** There is high possibility of required extra effort, which leads to loss of time and more incurred cost. So, you have to balance of time and cost as per your organization road map.

## Conclusion

There are some powerful use cases for multi-cloud and hybrid models; however, a lot of effort and alignment is needed to do it effectively. Because there can be multiple strategies that could be used, it is crucial to analyze the use case and devise a common approach, which has to be followed by other teams. Strong governance is needed to control and minimize the risks that these deployment strategies bring.

This is the last chapter covering the general concepts around cloud scalability. In the next section (as well as a chapter), we will pick up GCP offerings and dive deep into the how and why of the scaling options available.

## **Points to remember**

- There are some essential needs to support multi cloud/hybrid cloud deployment, the most critical being regulatory and data sovereignty reasons and reducing vendor lockin.
- Multi-cloud and hybrid deployment bring in additional responsibilities for management and architectural challenges. It also means more work for managing environments.
- Looking into the IaaS vs. PaaS options for multi-cloud deployment is essential. Using more IaaS means more management, and using more PaaS means different environments for the same business applications.
- Systems following this strategy need strong governance to control infrastructure complexities.

## **Questions**

1. What is the difference between multi-cloud and hybrid cloud and possible use-cases?
2. What are some common use cases of multi cloud?
3. What are the pros and cons of using a multi-cloud strategy?

# CHAPTER 5

## Scaling Compute Engine

### Introduction

**Google Cloud Platform (GCP)** provides users the ability to create **Virtual Machines (VM)**. These virtual machine configurations could be either defined by Google or customized, based on configurations defined by the user. A group of virtual machines could be treated as one entity, and thus collectively called an instance group. The instance group enables features like autoscaling, high availability, cross-region deployments, and rolling updates, for all the virtual machines configured in the instance group. In modern workloads, having multiple spikes in workloads where we want processing results sooner than later, it is crucial to scale up quickly. However, due to differences in peak and off-peak hours, it is essential to scale down. If we do not accommodate both aspects of scaling, we might see unjustified costs.

Effective scaling strategy depends on the nature of the application. Hence, it is recommended to start with the best possible guess and fine-tune the scaling aspects in subsequent iterations. For example, scalability configuration for an application where the workload fluctuates fast, needs a different scaling strategy than a workload, where the fluctuation is not so frequent. This chapter will look into details on how to create virtual machines and talk in-depth about the concepts and implementation around scaling of virtual machines in Instance Groups, especially autoscaling.

# Structure

In this chapter, we will discuss the following topics:

- Interacting with GCP
- Introduction to instance groups
- Autoscaling groups of VMs
- Scaling
  - Autoscaling
  - Predictive scaling
- Scale- in controls
  - Maximum allowed reduction configurations
  - Trailing time configurations
- Developing and managing autoscalers
  - Scaling based on:
    - Cloud monitoring metrics
    - Scale based schedules
    - Predictive schedules
    - CPU utilization
    - Load balancing serving capacity
  - Creating autoscaling policy based on multiple signals
  - **Create, Read, Update, Delete (CRUD)** operations on autoscalers
- Autoscaling node groups
- Reserving resources for effective autoscaling
  - Single Project Zonal Reservations
  - Shared Projects Zonal Reservations
  - Consuming Reservations

- Load balancing
  - Adding instance group to load balancer
  - Configuring multi regional external load balancer
  - Cross regions load balancing

## Objectives

By the end of this chapter, we will learn how to create VMs and manage them as instance groups. We will also understand how to scale up and down the application seamlessly, based on user workloads, as well as, how to make sure that the application supports self-healing and high availability. We will then see the different autoscaling strategies available in the *Google Cloud Platform* and how we can use them effectively.

Lastly, we will talk about the complexities and solutions related to the regional scaling of virtual machines.

## Interacting with GCP

There are three ways to interact with the Google Cloud Platform.

### Using the console\UI portal

In this method, we can log in to the UI portal of Google Cloud console (**<https://console.cloud.google.com>**) with the user credentials, and perform actions on the portal.

### Using GCloud commands

This is the second way to interact with the Google Cloud Platform. GCloud Shell is like any available shell, with GCloud utilities installed. '*GCloud utilities*' are a set of shell commands developed and managed by the Google Cloud Platform team to perform actions. For GCloud utilities, you



have to install GCloud Shell on your local machine and do the setup. GCP portal also provides the facility to create a small gcloud Shell VM from UI.

This strategy is used by IT and system administrators to interact with GCP.

## REST APIs

We can also interact with the Google Cloud Platform by triggering REST APIs. This is used when we want to interact from within the application.

All the preceding three strategies provide identical capabilities to the user. Refer to the appendix section, for steps to step these up. We will use the GCloud commands for demonstration and explanation in the chapters.

## Introduction to instance groups

Google Cloud Platforms give the facility to create secure and customizable virtual machines running in Google Infrastructure. They provide pre-defined configuration machine types that can be used to create virtual machines quickly. These pre-defined configurations could be divided into four categories:

**General purpose:** As the name suggests, this class of machine is used for standard and cloud-native workloads. The machine type options available under this category are based on best price-performance, and give multiple flavors of vCores to Memory options. Applications like web applications, containerized microservices, web servers, and so on, are ideal candidates for this machine type.

**Compute optimized:** This class of machines consistently offers the highest performance per core, making it ideal for applications requiring high computations capabilities. Applications like multiplayer games, deep learning

workloads, and trading applications are ideal candidates for using this class of machines.

**Memory optimized:** This class provides the highest memory to vCore ratio and is ideal for applications aiming to store or hold data in memory. Typical applications which use this class of machines are distributed databases, real-time streaming analytics, and caches.

**Accelerator optimized machines:** This class is ideal for massively parallel computing workloads like machine learning (seismic analysis, fluid dynamics, speech recognition) and high-performance computing. This class is for workloads that need GPUs.

These instances run public images of Linux and Windows machines, or we can give out our private custom images to create a VM.

If your workload does not fit any of the given workloads, platforms provide the capability to define your vCore and memory configurations, which suit your workload the best and are also cost-optimized (such machines are known as custom machine types). We can create a machine as small as one vCore, to up to 96 vCore or any even number of vCPUs. The memory could be up to 8GB per vCPU.

The following GCloud command creates a custom VM:

```
gcloud compute instances create scaling-gcp-custom-vm\  
  --custom-cpu 4 \  
  --custom-memory 5 \  
  --zone=us-central1-c
```

The preceding command creates a custom VM with the name “scaling-gcp-custom-vm” with number of vCPUs as 4, and memory as 5 GB. The machine will be created in zone “us-central-1c”.

**Note: The options configured are not the exhaustive list of all the options. Please refer to official**

## documentations for complete list of options.

When a VM is created, some key points to remember are as follows:

- Each VM is created in a Google project. A Google project can have multiple VMs. When we create a VM, we can define the hardware or machine type, operating system, and storage locations.
- When a VM is created, there is always a small disk (boot persistent disk) attached, on which the operating system is running. If there is a need to attach more disks, it must be done explicitly.
- All network interfaces in the VM belong to the subnet of a unique VPC network.

GCP offers users to create spot VMs, which are lower in cost and are provided by Google because they are unutilized at that point in time. Once the need arises, GCP takes them back. Only in case of batch and fault tolerant jobs, can spot VMs be used.

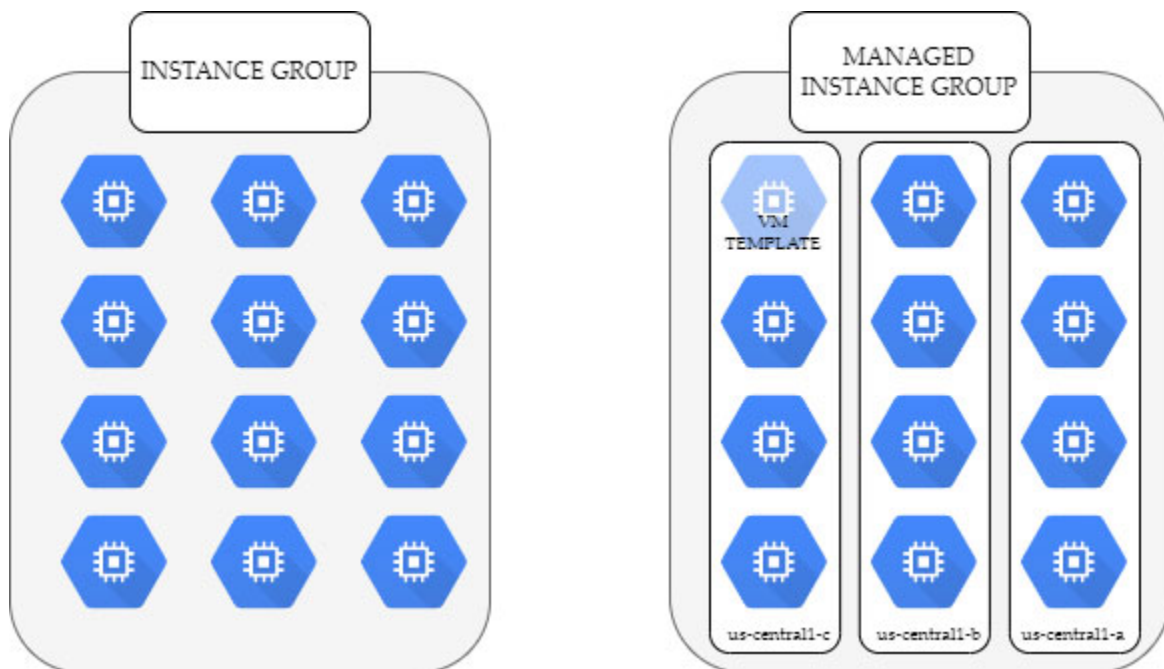
The following GCloud command creates custom VM using SPOT instances:

```
gcloud compute instances create scaling-gcp-custom-vm-spot \
  --custom-cpu 4 \
  --custom-memory 5 \
  --zone=us-central1-c --provisioning-model=SPOT
```

The preceding Gcloud command will create a customized VM “**scaling-gcp-custom-vm-spot**” with 4 vCore and 5 GB Memory in zone us-central1-c using spot instances.

With the preceding background on virtual machines, let us look into instance groups. An instance group is a group of virtual machines that can be managed as single entity. The primary aim of creating such an entity, is to enable the autoscaling of applications deployed on virtual machines, in

case of increase in load as well as to decrease the number of virtual machines in case of load going down. [Figure 5.1](#) features instance groups:



**Figure 5.1:** Instance groups

There are two types of groups:

## Managed Instance Group

In [Figure 5.1](#), in the right half of the image, all virtual machines in a **Managed Instance Group (MIG)** have the same hardware and operating system configurations. This is a group of identical virtual machines created using a single template.

Important features of Managed Instance Groups are as follows:

**Maintaining minimum number of instances:** Managed group makes sure that there is a minimum number of instances running for the application. In case a VM for an application fails, Managed Instance Group will create another instance in same or in different zone.

**Self-healing:** A health URL is configured for the VMs and in case the health URL does not respond within limits, a new VM instance is launched assuming that the VM is no more existent.

**Autoscaling:** We can increase or decrease the number of virtual machines based on number of users using the system.

**Load balancing:** We can create a load balancer which distributes user workloads across all the machines in a Managed Instance Group. This works well with autoscaling and auto healing configurations, as it needs new VM to be configured by load balancer and VMs to be rejected by load balancer.

**Regional groups:** We can create a Managed Instance Group as regional, that is, spread across multiple zones. In case of any zone going down, the number of instances is managed by creating VMs in either zones. This enables support for high availability.

**Rolling updates:** We can deploy new version of the application with 0 downtime. In rolling updates, we can deploy changes to instance groups one by one with load balancer, and therefore, not allowing traffic on one of the VMs where deployment is going on.

Managed Instance Group has 4 components, and because of these, the preceding features are also supported.

**Instance group manager:** Instance group manager is a component that controls VM location, its distribution across regions and makes sure that minimum and maximum number of VMs is ensured without fail. The main function of the group manager is to make sure that the group adheres to all the configurations defined.

**Instance template:** Instance template is a blueprint of virtual machines. It is a specification which defines configuration of the machine type, operating system, and

other specifications of machines like attached disks. We can configure multiple instance templates in a Managed Instance Group.

**Health check:** This is primarily used for auto healing. If the health URL fails to revert, a new VM is created for the VM where the health check was failed. It supports HTTP, HTTPS, SSL or TCP protocol. We can define custom interval and timeout for declaring a VM as failure.

**Autoscaling policy:** Policy defines the resizing metrics and minimum/maximum size for the group.

## Creating a Managed Instance Group

1. Before creating Managed Instance Group, we need to create an instance template.

```
gcloud compute instance-templates create scaling-gcp-  
instance-template \  
--machine-type=e2-standard-4 --image-family=debian-10 \  
--image-project=debian-cloud \  
--boot-disk-size=250GB
```

The preceding GCloud command will create an instance template with the name “**scaling-gcp-instance-template**” with machine type as e2-standard-4, image family debian-10 and attached boot disk size as 250 GB.

2. Once the instance group is created, we can use the same instance group name while creating the Managed Instance Group.

```
gcloud compute instance-groups managed create scaling-gcp-  
managed-instance-grp \  
--base-instance-name test --size 3 \  
--template scaling-gcp-instance-template
```

The preceding GCloud command will create a Managed Instance Group with the name “**scaling-gcp-managed-instance-grp**”, of size 3 (3 VM are created) using the instance template “**scaling-gcp-instance-template**”

created in step 1. The name of the 3 VMs will have the prefix as test.

Managed Instance Group are of two types:

## Stateless Managed Instance Group

Stateless Managed Instance Groups are those where the systems do not save the state of application. In case of any failures, the system creates everything from scratch. For example, if a VM crashes, no information of the crashed VM resides in the platform, and a completely new VM is created. It includes disks as well as IP addresses. One information which they preserve is the VM name, so that they can re-create the VM with the same name. Stateless MIGs are highly available and scalable, and provide features such as auto healing, autoscaling, recreation and rolling updates. Here are a few instances:

- Following is GCloud command for creating a Managed Instance Group:

```
gcloud beta compute instance-groups managed create
scaling-gcp-stateless-mig /
--project=scaling-gcp /
--base-instance-name=stateless-mig /
--size=1 /
--description=This\ is\ a\ stateless\ MIG.
--template=scaling-gcp-instance-template
--zones=us-central1-c,us-central1-f,us-central1-b
--target-distribution-shape=EVEN
```

The preceding command creates a Managed Instance Group '**scaling-gcp-stateless-mig**' with the following properties:

- **project=scaling-gcp**: This is GCP project, where we are creating the resources.

- **base-instance-name= stateless-mig**: The base name to use for the Compute Engine instances that will be created with the Managed Instance Group.
- **size=1**: Initial number of instances in the group.
- **description= ""**: Description of the Managed Instance Group.
- **template=scaling-gcp-instance-template:** Instance template name.
- **zones=us-central1-c,us-central1-f,us-central1-b** : Zones where virtual machines will be created.
- **target-distribution-shape=EVEN**: Virtual machines will be distributing over all the zones uniformly.
- The following command sets up autoscaling configuration on managed instance.  

```
gcloud beta compute instance-groups managed set-
autoscaling scaling-gcp-stateless-mig/
--project=scaling-gcp/
--region=us-central1 /
--cool-down-period=60 /
--max-num-replicas=6 /
--min-num-replicas=1 /
--mode=on /
--target-cpu-utilization=0.6
```

The preceding GCloud command updates the autoscaling policy for the MIG '**set-autoscaling scaling-gcp-stateless-mig**', setting minimum number of VMs to be 1, maximum 6. A virtual machine will be added every time the average CPU utilization goes preceding 60%.

## Stateful Managed Instance Group

We can configure a Managed Instance Group to be stateful, that can host stateful workloads by preserving the disk, IP addresses and metadata on the disk of the VM. Stateful



applications like databases, long running batch jobs and so on, where one transaction/processing depends on previous transaction/processing. When these applications break down, they need to start again, with the same data loaded as it was before the application went down.

A stateful Managed Instance Group maintains the unique state of each application on VM restart, auto healing, recreation and rolling updates.

Stateful MIGs support auto healing, multizone deployments and automated rolling updates, but they do not support autoscaling. Information preserved by stateful MIGs include instance names, persistent disks, instance-specific metadata, and IP addresses.

```
gcloud beta compute instance-groups managed create scaling-  
gcp-stateful-mig/  
--project=scaling-gcp/  
--base-instance-name=stateful-mig/  
--size=1 /  
--description=This\ is\ a\ stateful\ MIG /  
--template=scaling-gcp-instance-template /  
--zones=us-central1-c,us-central1-f,us-central1-b /  
--target-distribution-shape=EVEN /  
--instance-redistribution-type=NONE /  
--stateful-disk=device-name=persistent-disk-0,auto-  
delete=never
```

The preceding gcloud command creates a stateful Managed Instance Group - '**scaling-gcp-stateful-mig**'. The properties and values mentioned are similar to that mentioned in the creation of Managed Instance Group, except 2 new properties -stateful-disk and auto-delete.

**stateful-disk:** The configuration configures a disk **persistent-disk-0**, defined by the instance template, to be stateful.

**auto-delete** = never, that the disk will never be deleted in case of virtual machines going down.

## Unmanaged instance group

Unmanaged instance group is a group of VMs which are not same, that is, they have different hardware and configurations, as well as different images. Unmanaged instance group is required where we need different kind of VMs (heterogeneous group) in a group added or removed arbitrarily.

Unmanaged instance group does not provide features like autoscaling, auto healing, regional groups and rolling updates.

This is not the recommended way to create and manage workloads due to its unavailability of features.

Instance group create command lets us set the parameters as shown in [Table 5.1](#). When we trigger a create command, few properties are mandatory while few are optional. The Optional property has a pre-configured value; if you want to override it, you have to mention it.

Property Name	Description
size	This is the specification for number of virtual machines created at the time of creation of instance group. It is a mandatory property.
template	Using this property, you can specify the template of virtual machines which can be created in the instance group. It is a mandatory property.
base-instance-name	A string value specified here will become prefix of any virtual machine launched in the instance group. It is a mandatory property.
description	A managed group creator can give a small decryption of why, what and when of the Managed Instance Group.
initial-delay	It specifies the amount of time the virtual machine takes in initializing. No auto healing should happen during this time even if the VM looks unhealthy.
instance-redistribution-type	This is the instance distribution policy across zones. This is needed to have a uniform number for virtual

	<p>machines across zones. It is of 2 types:</p> <ul style="list-style-type: none"> <li>• <b>PROACTIVE:</b> The instances are redistributed across zones proactively.</li> <li>• <b>NONE:</b> Managed Instance Group does not redistribute instance across zones.</li> </ul>
stateful-disk	This property is used when there is a need to save the configurations across virtual machines. This property is demonstrated in the Stateful Managed Instance Group section.
target-distribution-type	<p>A regional MIG distributes its virtual machines across zones. The way distribution will happen depends on the value supplied for this property.</p> <ul style="list-style-type: none"> <li>• <b>EVEN:</b> When this property value is set to EVEN, Managed Instance Group creates and deletes virtual machines to maintain an even number for virtual machine across zones. This is recommended for high availability scenarios.</li> <li>• <b>BALANCED:</b> This value enables Managed Instance Group to acquire resources across zones where it is available, while distributing the VM as uniform as possible. This option is chosen for maintaining high availability for non-scalable applications, like batch jobs which do not need scalability.</li> <li>• <b>ANY:</b> This value tries to create virtual machines with the intention of utilizing reservations for the underutilized zones. This is recommended for batch jobs which do not need high availability.</li> </ul>
target-pool	A target pool is a pool of instances used for IP level load balancing.
region	Region for the virtual machines of the instance group.
Zone	Zone for the virtual machines. We need to either mention zone or region but not both.

**Table 5.1:** Properties for instance group create command

In the rest of the chapter, we are going to investigate the various aspects of Managed Instance Group and will dive deep into each aspect of autoscaling, healing, load balancing and regional configurations.

## **Autoscaling groups of VMs**

Managed Instance Group provides the capability to auto-scale, that is, scale-out and scale-in, based on the workload of the application. Autoscaling helps the application handle the workload gracefully during peak hours by scaling out (increasing the infrastructure), and scaling in during the non-peak hours to control cost.

When we configure autoscaling on Managed Instance Group, the autoscaler (a daemon) constantly monitors the load on the instance group and, based on the policy defined while creating the instance group, identifies the total number of VMs based on platform metrics from the previous 10 minutes. This duration of 10 minutes is the stabilization time, which will stop very frequent upscales and down scales.

The autoscaling policy governs the maximum and minimum sizes. Autoscaling provides features like scale-in, which throttles the downsizing of infrastructure, and predictive scaling, which scales up and down based on historical workloads.

You can create regional instance groups. As the name suggests, these instance groups are spread to multiple zones, and virtual machines created in the instance group are distributed uniformly across all zones. There are various strategies available, like the strategy of distributing VM evenly across zones, to placing the virtual machine in any possible zone. You can also configure certain fixed number of zones in a region where instance can spin up.

Regional Managed Instance Groups, by default, have proactive distribution enabled. Proactive distribution makes sure that you have an evenly distributed number of virtual machines in all zones. In case virtual machines are deleted in one zone, new virtual machines are spined up in the zone

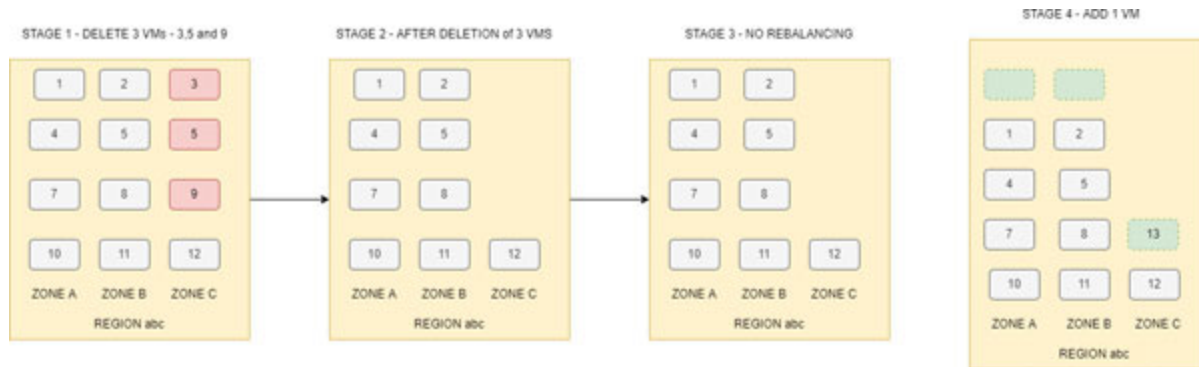
where the machines were deleted, and a few of the already running machines from other zones are deleted.

Consider the following [Figure 5.2](#), which has a regional Managed Instance Group created in region '**abc**' with VMs (numbered 1 to 12) uniformly distributed across all 3 zones (**A**, **B** and **C**). If 3 VM (stage 1) get deleted (**3**, **5** and **9**), in case of proactive scaling, Managed Instance Group deletes VM (stage 2) from other zones (VM 1 from **Zone A** and VM 2 from **Zone B**) and adds them to zone (1 and 2 to **Zone C**), to strike uniform number of VMs (stage 3) across all the regions. When another VM is added, it will be added in such a way that the number of VM will remain almost same across all zones (Stage 4). It could be added to any region in this situation as VMs are already uniformly distributed. The Green boxes without a number represent potential new virtual machines. Please refer to the following figure:



**Figure 5.2:** Proactive distribution

In case of non-proactive scaling in a similar situation as above, stage 3 varies. No rebalancing happens. However, in case of adding of one more VM, generally VM is added to the zone which has lower number of VM (**Zone C**). It could have been added to **Zone A** and **B** as well. This can be seen in [Figure 5.3](#):



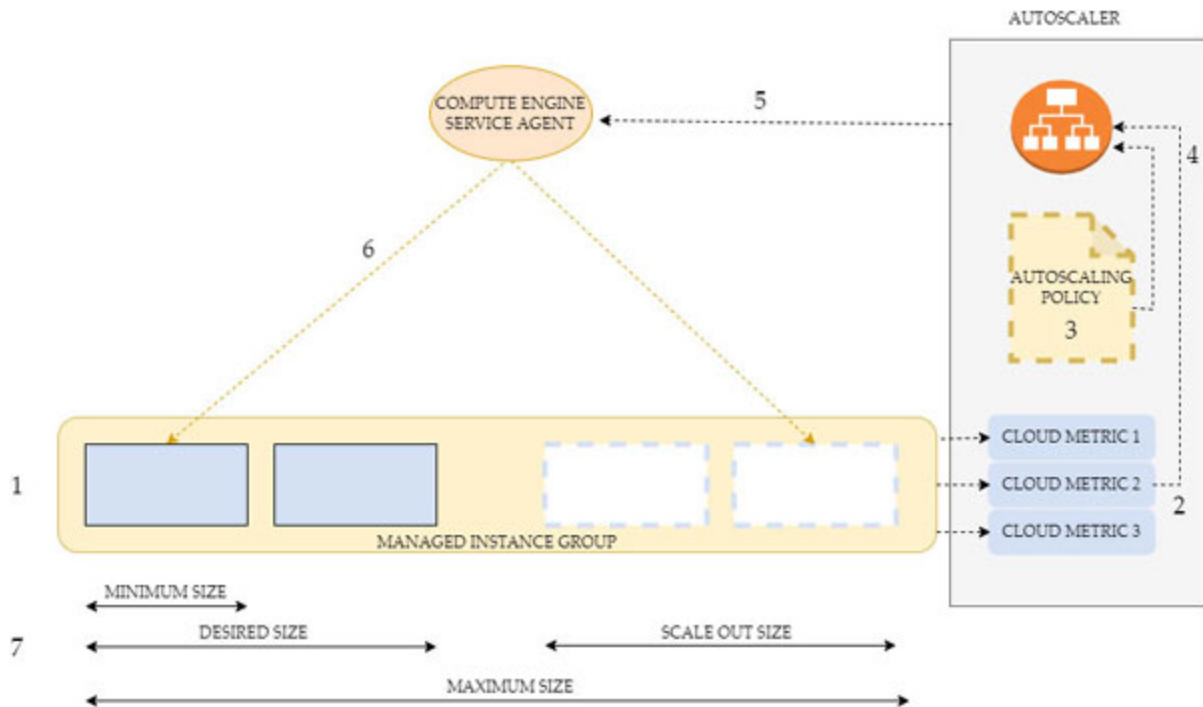
**Figure 5.3:** Non-proactive scaling

## Scaling

Predictive scaling and autoscaling are the most widely used strategies for scaling infrastructure on the cloud. This section will look into the scenarios where one takes an edge over the other and vice versa.

## Autoscaling

Managed Instance Group provides the capability to define autoscaling by adding and removing the virtual machines based on the load on the system. It helps in gracefully increasing the infrastructure when there is an increase in traffic and reduces infrastructure as the traffic goes down, making it the most effective cloud scaling strategy. In [Figure 5.4](#), we can see the complete autoscaling process:



**Figure 5.4:** Autoscaling

Follow the numerical labelling with the numerical explanations as follows.

1. It represents the Managed Instance Group. As per the figure, two virtual machines are running.
2. Cloud monitoring metrics have constantly been pulled and fed to the Autoscaler.
3. The autoscaling policy is defined by the user who created/updated the autoscaling policy for the MIG.
4. Autoscaler applies the autoscaling policy on each metric and calculates the maximum number of virtual machines which satisfy each metric.
5. Autoscaler instructs the *Compute Engine Service Agent* to increase or decrease the number of virtual machines.
6. Compute Engine service agent increases or decreases the number of virtual machines.
7. **Minimum size**: Defined in the autoscaling policy.

**Desired size:** Decided by autoscaler based on current load.

**Scale-out size:** The number of VMs that can be added.

**Maximum size:** Defined in the autoscaling policy.

Autoscaling works independently of auto-healing, that is, if the number of VMs is at the minimum configured value and one of the virtual machines fail the health check for auto-healing, we will witness the number of virtual machines going down below the autoscaling threshold of minimum replicas.

If we want to define autoscaling in a regional Managed Instance Group, you will have to ensure that the target instance group is set to EVEN. In the case of autoscaling policy with Autoscaling Mode as ON (increase and decrease Virtual machine), proactive distribution scaling has to be enabled, to ensure that we see an evenly distributed number of virtual machines across zones. In the case of just scale-out, enablement of proactive distribution is not required.

If you auto-scale a regional MIG, we could see a scenario where a new instance is added to a zone, due to the scaling metric reaching threshold in the zone. Immediately, see a virtual machine being deleted (same or different), as the overall regional utilization has not reached the threshold.

## **Predictive scaling**

In this mode, the autoscaler forecast future, loads based on historical data and scales out the number of virtual machines in Managed Instance Group well in advance, so that the application is infrastructurally ready when the load arrives.

The success of this strategy depends on two key aspects:



- How predictable are your daily and weekly workloads? If there are too many variations in the load, this strategy might result in oversized or under-provisioned infrastructure resulting in a breach of SLAs.
- This strategy becomes more valuable, especially for applications with considerable initialization time (cool down time). If the application boots uptime is more than 1 minute, the run time autoscaling might not be effective as there is a delay of more than a minute after trigger action is taken.

Without predictive scaling, Autoscaler takes actions based on the load in real-time; however, in the case of predictive scaling, historical as well as current load is taken into consideration. Additionally, Autoscaler also takes into consideration the variability of past trends, that is, the delta between prediction and actual need. Predictive scaling starts before the actual need by the amount of cooling period set by the user.

Predictive scaling works only based on CPU utilization. Metrics of cloud load balancing and cloud monitoring are not supported. In addition, at least three days of the history of workloads are needed only after predictive scaling starts.

All the predictions are based on weekly and daily load patterns. Annual and one-time load patterns are not supported. For such scenarios, scheduled scaling works the best. Moreover, any load pattern taking less than 10 minutes is not considered for predictions. For example, if your application is scaled up for just 5 minutes and then scaled down, this spike will not be considered while making predictions.

## **Scale-in controls**

Applications have different initialization times; few can initialize in seconds, and few might take time in minutes.

The application, which takes less time to initialize, can scale up and down in real-time. However, applications that take more than minutes cannot do the same.

Applications taking minutes to initialize, if left alone with just an autoscaling policy, might not be a good idea, as seen, based on real-time metrics. If there is a need to scale down and then immediately scale up again, then the application will not be able to handle such quick spikes in load.

To handle this situation, scale-in controls come in handy. Scale-in controls enabled throttled scale down of application. For example, even if the system can go down by 30% (based on real-time metrics), the application will be allowed to scale down by 10% (defined in configuration).

To enable scale-in controls, configure the two properties in the autoscaling policy, as explained.

## **Maximum Allowed Reduction**

The number of virtual machines that could be decommissioned from peak load in the trailing time window. You can define it in two ways – Number of VMs, and percent of VMs. It is essential to understand the nature of the application to set this up, as a high value might result in a large scale of the application (temporary under provision), and a small value might result in an application not scaling in quickly enough (temporary over-provisioning).

## **Trailing Time Window**

It is the time from the last scale in operation. Autoscaler analyses the activity since the previous scale is in-process, and decides on a time window rather than real-time. The only decision taken is whether to scale down or scale-up. If the decision comes out to be scaled down, the reduction of

virtual machines defined by *Maximum Allowed Reduction* will occur.

## **Autoscaling in Action**

Let us have a look at all the above-discussed concepts in practice. We will create a Managed Instances Group with all the above-discussed concepts in action.

Let us refer to the Google Cloud Platform portal to look into Autoscaling configuration options, as shown in [\*Figure 5.5\*](#):

### Auto-scaling

Use auto-scaling to automatically add and remove instances to the group for periods of high and low load. [Learn more](#)

Auto-scaling mode

On: add and remove instances to and from the group

1

Minimum number of instances \*

1

Maximum number of instances \*

8

2

1

To maximise availability, the minimum number of instances should be at least equal to the number of zones. Additional instances will be placed in different zones.

[Distributing instances using regional managed instance groups](#)

### Autoscaling metrics

Use metrics to help determine when to scale the group. [Learn more](#)

CPU utilisation: 60% (default)

Predictive auto-scaling is optimized for availability

3

ADD METRIC

### Autoscaling schedules

4

### Cool-down period

Specify how long it takes for your app to initialise from boot time until it is ready to serve. [Cool-down period](#)

Cool-down period \*

60

Seconds

5

### Scale-in controls

Set limits for scaling the group [Learn more](#)

☒ Enable scale-in controls

Don't scale in by more than \*

10

Unit

Percent of VMs

6

Over the course of

10 MIN

30 MIN

60 MIN

CUSTOM

**Figure 5.5:** Autoscaling MIG

Follow the numerical labelling of [Figure 5.5](#) with the numerically labelled explanation as follows:

1. **Autoscaling mode:** Autoscaling in MIG can be configured with various modes. Modes define the nature of scaling. The available modes are shown in [Table 5.2](#):

--	--

Autoscaling Mode	Description
<b>On:</b> Add and remove instances to and from the group	This is the scaling mode that allows scale up and down. This autoscaling mode is the most effective as the system scales up in case of high load (and thus ensures meeting SLA) and scales down as load decreases (to ensure cost-effectiveness) for applications with low cool down time.
<b>Scale Out:</b> Only Add instances to the group	This is the scale-out model, that is, the system will auto-scale when the load increases, but it will not scale down automatically.
<b>Off:</b> Do not auto scale	This mode (default) switches off autoscaling, that is, in case you need to scale up or scale down, you must take steps manually.

**Table 5.2:** Modes available in autoscaling

- 2. Minimum\Maximum number of instances:** The minimum number of instances is the lowest Number your MIG could shrink to, and the maximum is the maximum number of VM your MIG could scale up to.
- 3. Autoscaling metrics:** The metric whose value will trigger a scale up or down at a given point. For instance, in the given example, the metric is CPU utilization, and the value is 60%. When the average CPU utilization goes above 60%, a VM will be added automatically, and when the overall average CPU utilization goes below 60%, a VM is decommissioned. There are three categories of metrics that can be configured, as can be seen in [Table 5.3](#):

Autoscaling Metric	Description
CPU utilization	The expected value is a number between 0 to 100. This metric, set to 60%, means that when the average CPU utilization across VMs goes above 60%, MIG will add VM. On the contrary, when the average CPU utilization goes below 60%, a VM will be removed by MIG.
HTTP load balancing utilization	This autoscaling is based on an indication of a metric and its value. A VM will be added when the

	metric reaches a particular configured target value in the load balancer. We will look into this in-depth in the last section of this chapter.
Cloud monitoring metric	<p>Google Cloud monitoring captures various metrics out of the box. In addition, we can define our metrics as well. This scaling option could be configured on a per VM basis (scaling using per-instance metric), and for the whole group (scaling using per group metrics). For configuring this autoscaling policy, three things are essential to be understood:</p> <p><b>Metric identifier:</b> These are monitoring identifiers. We need to select one of them, or we can define a custom metric as well.</p> <p><b>Target utilization level:</b> Defines the threshold value for the cloud monitoring metric.</p> <p><b>Target type:</b> This defines how the autoscaler calculates the data.</p> <p><b>GAUGE:</b> In this, the autoscaler looks at the last few minutes, computes a value, and compares it with the utilization target to scale up or down.</p> <p>DELTA_PER_MINUTE: This autoscaler looks at the last minute to calculate the value of the metric.</p> <p>DELTA_PER_SECOND: This autoscaler looks at the last second to calculate the metric's value.</p>

**Table 5.3:** Categories of autoscaling metrics

This section also provides the option of selecting predictive scaling. *Predictive scaling* improves availability, by monitoring the daily and weekly load patterns, as well as scaling out proactively before the need arises.

4. **Autoscaling schedules:** The number of VMs is scaled up or down based on a fixed configuration in the Autoscaling schedule. In this example, we had not configured autoscaling schedules.
5. **Cool down periods:** It is the application initialization period. To configure a proper value, it is essential to understand the time taken by your application to initialize. While the application initializes, its usage data

might not represent normal conditions. Hence, autoscaler uses remarkable down period differently as per the need of scaling:

- **Scale-in:** For the scale in the decision, the autoscaler considers all the instances, even the instances in cooling down mode.
- **Scale-out:** For Scale-out decisions, the autoscaler ignores data from instances where the application is hosted in cool down mode.
- **Predictive mode:** If the predictive mode is set and the cool-down is defined, the scale-up of the application will happen before the actual serving. This before time is equal to the cool-down time defined. For example, if the predictive scaling defines the scaling of a VM at 11:00 AM and the cool-down is defined as 5 minutes, the VM will be added to MIG at 10:55 AM. From 10:55 to 11:00, the application installation will happen (cool-down), and at 11:00 AM, scaled-up infra is ready.

## 6. **Scale-in controls:**

This feature is especially needed when your application takes a long time to initialize, coupled with frequent load requirements. If you do not configure this value, the system will scale down, and in the very next moment, there is a need to scale up again; a scale-up of infra will happen, but effective usage can only start after application initialization is complete. This could have been avoided if the system was not allowed to scale down abruptly.

To set the scale in controls, we must set the the following properties.

- **Maximum Allowed Reduction:** The percent of the number of VM allowed decommissioning in one

trailing time window. The smaller the value, the more time to scale it will be.

- **Trailing Time Window:** It is the time window after which the autoscaler will attempt to scale down.

The following GCloud command for the preceding configuration creates a Managed Instance Group:

```
gcloud beta compute instance-groups managed create stateful-  
mig-autoscaling /  
--project=scaling-gcp /  
--base-instance-name=stateful-mig-autoscaling /  
--size=1 /  
--template=scaling-gcp-example-template /  
--zones=us-central1-c,us-central1-f,us-central1-b /  
--target-distribution-shape=EVEN
```

This command creates the autoscaling configuration:

```
gcloud beta compute instance-groups managed set-autoscaling  
stateful-mig-autoscaling /  
--project=scaling-gcp /  
--region=us-central1 /  
--cool-down-period=60 /  
--max-num-replicas=8 /  
--min-num-replicas=1 /  
--mode=on /  
--target-cpu-utilization=0.6 /  
--scale-in-control=max-scaled-in-replicas-percent=10,time-  
window=600
```

## [Developing and managing autoscalers](#)

In this section, we are going to dive deep into the autoscaler configurations.

To get the information about the Autoscaler configured for a Managed Instance Group, we can trigger the following



command on Google Cloud shell.

```
gcloud compute instance-groups managed describe <Managed-Instance_Group_Name>
```

**Example:** For the above created Managed Instance Group - **stateful-mig-autoscaling**

```
gcloud compute instance-groups managed describe stateful-mig-autoscaling
```

You can also update an Autoscaler: configuration:

```
gcloud compute instance-groups managed update-autoscaling  
<managed-instance_group-name> --max-num-replicas <max-replica>
```

It is now time to look into various Autoscaler configurations.

## Scaling Based on CPU utilization

Autoscaler looks at average CPU utilization across all virtual machines in this autoscaler configuration. If all virtual machines collectively have average CPU utilization above a certain threshold, a virtual machine is added to Managed Instance Group.

The best example to apply this strategy is a multithreaded application where the number of threads increases and decreases based on the data it is processing.

The following GCloud command creates CPU-based autoscaling:

```
gcloud compute instance-groups managed set-autoscaling  
example-MIG \  
  --max-num-replicas 20 \  
  --target-cpu-utilization 0.60 \  
  --cool-down-period 90
```

## Scaling based on load balancing serving capacity

When the backend applications behind an external load balancer is hosted on Managed Instance Group, external load balancer provides two balancing modes. The first mode is **UTILIZATION** and the other is **RATE**.

**UTILIZATION:** You can specify the maximum threshold for average utilization across instances in the instance group.

**RATE:** You can specify the threshold for the number of requests per second on an instance basis or per group basis. For regional MIGs, we cannot specify group-based **RATE**. For zonal, **RATE** on group is applicable.

When we align an autoscaler to an instance group backend of an external HTTP load balancer, the autoscaler makes sure to maintain a fraction of the load serving capacity.

For example, if we have a system which can handle 100 requests per second and we define the target utilization to be 75%, autoscaler will make sure that the utilization of backend instance should not go above 75%, that is, 75 requests per instance. The moment the number of requests goes above 75, a virtual machine is added.

```
gcloud compute instance-groups managed set-autoscaling
example-managed-instance-group \
  --max-num-replicas 20 \
  --target-load-balancing-utilization 0.6 \
  --cool-down-period 90
```

## **Scaling Based on cloud monitoring metrics**

Apart from the preceding two strategies (scaling based on CPU metric and load balancer utilization), another family of signals which could be used to define autoscaling strategy is cloud monitoring metric.

When we define autoscaling based on cloud monitoring metric, you can scale based on the following two metric

types:

- **Scale using the per instance metrics**, in which we analyze the metric data for each virtual machine indicating resource utilization. When using this kind of metrics, the system cannot scale below a size of 1 VM, because autoscaler needs metric from at least one VM. If you need metric of a group as a whole or there is a need to scale down VM to zero from time to time, you can configure per group metric.
- **Scale using per instance group metric**, where the group scales based on a metric whose value is determined by taking into consideration the complete instance group. All metrics do not qualify to be used here, for example a metric like `compute.googleapis.com/instance/cpu/reserved_cores` will not change based on usage and hence should not be used for autoscaling.

Apart from using the standard metrics, custom metrics can also be used. The process of creating custom metric is out of scope of this book. However, once you have the custom cloud monitoring metric created, it could be utilized for autoscaling similar to standard metrics.

## **Configuring auto scaling for per instance metric**

To create an autoscaler that uses cloud monitoring for metric per instance, you have to provide a metric identifier, the desired target utilization level as well as the utilization target type.

The target utilization level has to be a positive number which measures the value of the metric and takes a decision to either add or remove virtual machine. Target type

determines how the autoscaler processes data collected. The possible target values are as follows:

**GAUGE:** Autoscaler calculates the average value of data collected from past configured number of minutes.

**DELTA\_PER\_MINUTE:** The autoscaler processes data in last one minute and compares with target utilization.

**DELTA\_PER\_SECOND:** The autoscaler processes data per second and compares it with target utilization.

The following GCloud command is for autoscaling configuration:

```
gcloud compute instance-groups managed set-autoscaling
example-managed-instance-group \
  --custom-metric-utilization
metric=example.googleapis.com/path/to/metric,utilization-
target-type=GAUGE,utilization-target=10 \
  --max-num-replicas 20 \
  --cool-down-period 90 \
  --region us-west1
```

## [Configuring auto scaling for per group metric](#)

There are two strategies to define group metrics:

### **Instance assignment**

In case of instance assignment, you specify how much work you expect each VM to handle. Based on the work lined up, the number of virtual machines is added. For example, let us assume that you have a streaming application that listens to Pub/Sub messages. If you configure each VM to process 5 messages and there are total of 25 unread messages, the number of virtual machines will be scaled to 5.

The following GCloud command is used to achieve the preceding use case:

```
gcloud compute instance-groups managed set-autoscaling \
  our-instance-group \
  --zone=us-central1-a \
  --max-num-replicas=100 \
  --min-num-replicas=0 \
  --update-stackdriver-
metric=pubsub.googleapis.com/subscription/num_undelivered_me
ssages \
  --stackdriver-metric-filter="resource.type =
pubsub_subscription AND resource.labels.subscription_id =
our-subscription" \
  --stackdriver-metric-single-instance-assignment=15
```

## Utilization target

In this strategy, the autoscaling happens in order to maintain the utilization level. For example, assume you have an application and you had defined a custom metric of latency and configured autoscaling. The moment your latency metric starts increasing, virtual machine will be added. On the other hand, if the latency goes down, the VM will be deleted.

The following GCloud command demonstrates the utilization metric:

```
gcloud compute instance-groups managed set-autoscaling \
  our-instance-group \
  --zone=us-central1-a \
  --max-num-replicas=100 \
  --min-num-replicas=0 \
  --update-stackdriver-
metric=custom.googleapis.com/example_average_latency \
  --stackdriver-metric-filter "resource.type = global AND
metric.labels.group_name = our-instance-group" \
  --stackdriver-metric-utilization-target=100 \
```

```
--stackdriver-metric-utilization-target-type=delta-per-second
```

## Scaling based on schedules

This auto-scaling strategy lets you schedule virtual machine addition ahead of the anticipated load. We can design a required number of virtual machines for recurring load patterns and one-off events. Use of schedule scale is recommended when the application takes a long time to initialize, and you want to scale out in advance for the upcoming load. There are two limitations to this approach:

- You can only set 128 schedules per Managed Instance Group.
- The minimum duration for schedules is 5 mins.

The following GCloud command is used for creating a scaling schedule:

```
gcloud compute instance-groups managed set-autoscaling  
MIG_NAME \  
  [--min-num-replicas=MIN_NUM_REPLICAS] \  
  [--max-num-replicas=MAX_NUM_REPLICAS] \  
  [--set-schedule=SCHEDULE_NAME] \  
  [--schedule-cron="CRON_EXPRESSION"] \  
  [--schedule-duration-sec=DURATION] \  
  [--schedule-time-zone="TIME_ZONE"] \  
  [--schedule-min-required-replicas=MIN_REQ_REPLICAS] \  
  [--schedule-description="DESCRIPTION"] \  
  [--zone=ZONE | --region=REGION]
```

The preceding command has already been discussed multiple times in the previous section. In [Table 5.4](#), we will look into properties which are related to creating schedules:

Property	Description
set-schedule	SCHEDULE_NAME: The name of the scaling schedule.

schedule-cron	CRON_EXPRESSION: Start time and reoccurrence configuration represented as cron expression.
schedule-duration-sec	DURATION: Duration in seconds that this schedule is active.
schedule-time-zone	Time zone of the schedule start.
schedule-description	Description of the schedule.

**Table 5.4:** Properties related to creating schedules

The following command creates repeating schedule:

```
gcloud compute instance-groups managed update-autoscaling
example-mig \
  --min-num-replicas=0 \
  --max-num-replicas=30 \
  --set-schedule=scaling-gcp-schedule \
  --schedule-cron="30 8 * * Mon-Fri" \
  --schedule-duration-sec=30600 \
  --schedule-min-required-replicas=10 \
  --schedule-description="Have at least 10 VMs every Monday
through Friday from 8:30 AM to 5 PM UTC"
```

The preceding command creates a schedule with name **'scaling-gcp-schedule'**. It runs based on the cron **"30 8 \* \* Mon-Fri"**. The total duration of trigger is 30600 seconds and during this schedule run, the minimum number of replicas will be 10 virtual machines.

The following command creates a one-time schedule:

```
gcloud compute instance-groups managed update-autoscaling
example-mig \
  --set-schedule=example-onetime-schedule \
  --schedule-cron="0 0 30 1 * 2030" \
  --schedule-duration-sec=86400 \
  --schedule-time-zone="America/New_York" \
  --schedule-min-required-replicas=30 \
  --schedule-description="Schedule a minimum of 30 VMs all day
for January 30, 2030" \
  --zone=us-east1-b
```

In this example, the schedule is created which will create 30 virtual machines on each day of January.

The following command lists all the configured schedules:

```
gcloud compute instance-groups managed describe MIG_NAME \
  [--zone=ZONE | --region=REGION]
```

The preceding command is used to list all the schedules configured for an instance group.

The following command enables a schedule:

```
gcloud compute instance-groups managed update-autoscaling
MIG_NAME \
  --update-schedule=SCHEDULE_NAME \
  [--schedule-cron="CRON_EXPRESSION"] \
  [--schedule-duration-sec=DURATION] \
  [--schedule-time-zone="TIME_ZONE"] \
  [--schedule-min-required-replicas=MIN_REQ_REPLICAS] \
  [--schedule-description="DESCRIPTION"] \
  [--zone=ZONE | --region=REGION]
```

The preceding command can be used to update the autoscaling configuration of instance group with schedule.

The following command disables schedules:

```
gcloud compute instance-groups managed update-autoscaling
MIG_NAME \
  --disable-schedule=SCHEDULE_NAME \
  [--zone=ZONE | --region=REGION]
```

The preceding command is used to disable a schedule.

## **Scheduling based on prediction**

We can define autoscaling based on prediction with the following GCloud command. Prediction takes into consideration only the CPU utilization as a signal.

```
gcloud compute instance-groups managed set-autoscaling
scaling-gcp-stateless-mig \
  --cpu-utilization-predictive-method optimize-availability \
```



```
--target-cpu-utilization 0.75 \  
--max-num-replicas 20 \  
--cool-down-period 300
```

The preceding GCloud command set predictive autoscaling with target value of CPU utilization to be 75%.

## **Creating autoscaling policy based on multiple signals**

When we set up multiple signals for autoscaling, that is, autoscaling configurations on multiple signals – on CPU utilization and load balancer together – Autoscaler calculates the number of virtual machines needed, as per each signal and then chooses the maximum estimated value to be the new size of the Managed Instance Group infrastructure. AutoNation scaler accepts one signal per metric type, except for cloud monitoring metrics and schedules. In the cloud monitoring metric, we can select five signals and configure 128 schedules per Managed Instance Group.

In the following example, we are configuring 2 autoscaling signals, that is, CPU utilization and load balancing utilization and custom schedule:

```
gcloud compute instance-groups managed set-autoscaling  
scaling-gcp-stateless-mig\  
  --target-cpu-utilization=0.8 \  
  --target-load-balancing-utilization=0.6 \  
  --set-schedule=workday-capacity \  
  --schedule-cron="30 8 * * Mon-Fri" \  
  --schedule-duration-sec=30600 \  
  --schedule-min-required-replicas=10 \  
  --schedule-description="Have at least 10 VMs every Monday  
through Friday from 8:30 AM to 5 PM UTC" \  
  --min-num-replicas=1 \  
  --max-num-replicas=50
```

The preceding GCloud command is a working example that sets autoscaling when CPU utilization exceeds 80% or load balancing utilization exceeds 60%. Along with this, a scheduled scaling is configured as well.

## **CRUD operations on autoscalers**

In this section, we will go through few of the CRUD operations which we can perform to manage the Autoscaler configurations.

### **Describing an Autoscaler**

```
gcloud compute instance-groups managed describe  
INSTANCE_GROUP_NAME
```

The preceding GCloud command will return the complete configuration for the Autoscaler.

### **Updating a scalar**

```
gcloud compute instance-groups managed update-autoscaling  
INSTANCE_GROUP_NAME --max-num-replicas MAX_NUM
```

The preceding GCloud command updates the maximum number of replicas to **MAX\_NUM**.

### **Turning off a scalar**

```
gcloud compute instance-groups managed update-autoscaling  
INSTANCE_GROUP_NAME --mode NEW_MODE
```

The preceding command will change the scaling mode to **NEW\_MODE**. If the value of **NEW\_MODE** is set to OFF, the autoscaler will be turned off.

### **Deleting an autoscaler**

```
gcloud compute instance-groups managed stop-autoscaling  
INSTANCE_GROUP_NAME
```

The preceding GCloud command stops autoscaling and hence the Autoscaler configuration is completely lost.

## Autoscaling node groups

In use cases where you use sole-tenant-groups for your workloads, you can use the node group autoscaler to automatically manage the sizes of node group. You can specify the autoscaling at the time of creation or you can update it after creating the node group.

The following command enables the Autoscaler on node group:

```
gcloud compute sole-tenancy node-groups create group-name \
  --node-template template-name \
  --target-size size \
  --maintenance-policy maintenance-policy \
  --zone zone \
  --autoscaler-mode mode \
  --max-nodes max-nodes \
  --min-nodes min-nodes
```

The preceding command creates a node group with autoscaling policy already defined. There are 3 main properties related to autoscaling (in bold); rest of the properties are specific to node group creation, and these are explained in [Table 5.5](#):

Property	Description
autoscaler-mode	Mode for the autoscaler on this node group. It could have 3 different values: <ol style="list-style-type: none"><li>1. <b>off</b>: Disables an Autoscaler.</li><li>2. <b>on</b>: Enable scale in and out Autoscaler.</li><li>3. <b>only-scale-out</b>: Enables only scaling out.</li></ol>
max-nodes	Maximum size of the node group
min-nodes	Minimum size of the node group. Default is zero.

**Table 5.5:** Properties specific to node group creation

For example, refer to the following code:

```
gcloud compute sole-tenancy node-groups create scaling-gcp-  
node-pool \  
  --node-template scaling-gcp-node-pool-template \  
  --target-size 10 \  
  --maintenance-policy Default \  
  --zone us-central1-c \  
  --autoscaler-mode on \  
  --max-nodes 5 \  
  --min-nodes 1
```

The preceding command creates a node pool with name **'scaling-gcp-node-pool'**, using node template **'scaling-gcp-node-pool-template'** with maintenance policy as **Default** and target-size as **10**. Auto scale mode is set to ON, meaning that the node pool can scale up and down in the range of 1 to 5.

The following command updates node group with autoscaler:

```
gcloud compute sole-tenancy node-groups update name \  
  --autoscaler-mode mode \  
  --max-nodes max-nodes \  
  --min-nodes min-nodes
```

Here, the update command (in bold) updates an existing node group with the defined autoscaling policy defined in the command.

## **Reserving resources for effective auto scaling**

To ensure your workload gets the required resources in a zone, you can reserve them. When you reserve resources, billing starts even if you do not use them, and stops only when they are deleted.

Irrespective of whether you use the instance, GCP prevents the reserved resources from assigning it to other customers. Generally, when we agree to this kind of reservation, the cost associated is not the same as we have in the case of standard VMs. Since we had committed to using it, GCP has some offer discounts.

A virtual machine can consume reservations only when the following properties of reservations and virtual machine match.

- Project
- Zone
- Machine type
- Minimum CPU platform
- GPU type and count
- Local SSD type and count

You can control the virtual machines using reservations. There are two strategies to manage the acquisition of reserved resources:

- **Shared type**

This has two categories:

- **Single project reservation:** Reserved resources can only be used for virtual machines for one GCP project.
- **Shared reservations:** Reserved resources are shared between two or more GCP projects. Both projects can consume the reserved resources as per their auto-scaling needs.

- **Consumption type**

This two has two subcategories:

- **Automatic:** The Managed Instance Group automatically selects one reservation to consume, if multiple reservations are available.
- **Specific:** You can specify a certain reservation to use, for virtual machine creation.

If a virtual machine using the reserved resources is stopped, suspended, or deleted, the VM resources are no longer counted as the use of reservations. The resources are available for another virtual machine. If you delete a reservation that is being used by VMs, the VMs will keep on running, and the cost of those VM becomes that of general VMs.

## Single project zonal reservations

You can create reservations for compute engine zonal resources, that can only be used in a single project. A reservation assures obtaining capacity for resources.

The following GCloud command creates this type of reservation:

```
gcloud compute reservations create RESERVATION_NAME \
  --machine-type=MACHINE_TYPE \
  --min-cpu-platform MINIMUM_CPU_PLATFORM \
  --vm-count=NUMBER_OF_VMS \
  --
  accelerator=count=NUMBER_OF_ACCELERATORS,type=ACCELERATOR_TY
  PE \
  --local-ssd=size=375,interface=INTERFACE_1 \
  --local-ssd=size=375,interface=INTERFACE_2 \
  --zone=ZONE
  --project=PROJECT_ID
```

[Table 5.6](#) elaborates the options which can be specified in the preceding reservation create command:

Property	Description
----------	-------------

RESERVATION_NAME	The name of the reservation.
MACHINE_TYPE	A predefined or custom type machine.
MINIMUM_CPU_PLATFORM	Minimum number of CPUs to be used for each virtual machine.
NUMBER_OF_VMS	Number of reserved virtual machines.
NUMBER_OF_ACCELERATORS	The number of GPUs to add, per instance.
ACCELERATOR_TYPE	Type or class of GPU to be used.
INTERFACE_1 and INTERFACE_2:	The type of interface you want the local SSDs for each instance to use. Valid options are: <code>scsi</code> and <code>nvme</code> . Each local SSD is 375 GB.
ZONE	Zone for reservations
PROJECT_ID	Project id where you want resources to be reserved.

**Table 5.6:** Properties for reservation create command

You can specify, '**--require-specific-reservation**' flag to indicate that only VM instances, that explicitly target this reservation, can use it.

The following GCloud command is used to create reservations:

```
gcloud compute reservations create scaling-gcp-reservation \
  --machine-type=custom-8-10240 \
  --min-cpu-platform="Intel Haswell" \
  --vm-count=2 \
  --accelerator=count=2,type=nvidia-tesla-v100 \
  --local-ssd=size=375,interface=scsi \
  --require-specific-reservation \
  --zone=us-central1-a
```

The preceding command creates a reservation **scaling-gcp-reservation** with the specified configurations, like total virtual machine count is 2, zone is **us-central1-a**.

## [Shared project zonal reservations](#)

The concept remains the same as that of single project zonal reservations, with one difference being that multiple projects are configured with each reservation, which can consume it.

The following GCloud command is used to create shared project reservations:

```
gcloud compute reservations create RESERVATION_NAME \
  --machine-type=MACHINE_TYPE \
  --min-cpu-platform=MINIMUM_CPU_PLATFORM \
  --vm-count=NUMBER_OF_VMS \
  --
  accelerator=count=NUMBER_OF_ACCELERATORS,type=ACCELERATOR_TYPE \
  --local-ssd=size=375,interface=INTERFACE_1 \
  --local-ssd=size=375,interface=INTERFACE_2 \
  --zone=ZONE \
  --project=OWNER_PROJECT_ID \
  --share-setting=projects \
  --share-with=CONSUMER_PROJECT_IDS
```

Most of the options above are already explained in previous sections. Two new properties which are new and need discussion are explained in the following [Table 5.7](#):

Property	Description
OWNER_PROJECT_ID	The project ID where you want to create the shared reservation.
CONSUMER_PROJECT_IDS	List of project IDs who can utilize a reservation.

**Table 5.7:** Properties for shared project reservations

You must include the **--share-setting=projects** flag to share this reservation with other projects. Optionally, add the **--require-specific-reservation** flag to indicate that only VM instances that explicitly target this reservation can use it.

The following GCloud command creates a reservation:



```
gcloud compute reservations scaling-gcp-reservation-zonal \
  --machine-type=custom-8-10240 \
  --min-cpu-platform="Intel Haswell" \
  --vm-count=10 \
  --accelerator=count=2,type=nvidia-tesla-v100 \
  --local-ssd=size=375,interface=scsi \
  --zone=us-central1-c \
  --project=scaling-gcp \
  --share-setting=projects \
  --share-with=scaling-gcp-1,scaling-gcp-2 \
  --require-specific-reservation
```

The preceding command creates a reservation **scaling-gcp-reservation-zonal** which is created in project **scaling-gcp** and can be used in project **scaling-gcp-1** and **scaling-gcp-2**.

## Consuming reservations

In the previous two sections, we looked into how to create reservations. In the current section, we will investigate how to utilize those reservations:

### Consuming instances from any matching reservation

1. Create an open reservation called **scaling-gcp-reservation**.

```
gcloud compute reservations create scaling-gcp-reservation \
  --vm-count=2 \
  --machine-type=n2-standard-32 \
  --min-cpu-platform "Intel Cascade Lake" \
  --accelerator=count=2,type=nvidia-tesla-v100 \
  --local-ssd=size=375,interface=scsi \
  --zone=us-central1-b
```

2. The following command creates a virtual machine which is created using any open reservation, and that matches

the instance properties in **scaling-gcp-reservation**, including the virtual machine zone, virtual machine type (machine family, Memory and vCPUs), minimum and maximum CPU platform, GPU amount and type, and local SSD amount and interface:

```
gcloud compute instances create scaling-gcp-instance-1 \
  --machine-type=n2-standard-32 \
  --min-cpu-platform="Intel Cascade Lake" \
  --accelerator=count=2,type=nvidia-tesla-v100 \
  --local-ssd=size=375,interface=scsi \
  --zone=us-central1-a \
  --reservation-affinity=any
```

3. Create a reservation named **scaling-gcp-reservation-02** with the **--require-specific-reservation** flag. These reserved resources can be used only by instances that specifically target this reservation by name:

```
gcloud compute reservations create scaling-gcp-
reservation-02 \
  --machine-type=n2-standard-32 \
  --min-cpu-platform "Intel Cascade Lake" \
  --vm-count=10 \
  --zone=us-central1-a \
  --require-specific-reservation
```

4. Create a VM instance that targets **scaling-gcp-reservation-02** by name, by using the **--reservation-affinity** and **--reservation** flags.

Ensure that the instance's properties match the reservation's instance properties, including the zone, machine type (machine family, vCPUs, and memory), minimum CPU platform, GPU amount and type, and local SSD interface and size.

```
gcloud compute instances create instance-2 \
  --machine-type=n2-standard-32 \
  --min-cpu-platform "Intel Cascade Lake" \
  --zone=us-central1-a \
```

```
--reservation-affinity=specific \  
--reservation= scaling-gcp-reservation-02
```

## Consuming a specific shared reservation

1. Create a reservation named **scaling-gcp-reservation** with the **--require-specific-reservation** flag. These reserved resources can be used only by instances that specifically target this reservation by name.

```
gcloud compute reservations create scaling-gcp-reservation \  
\
```

```
--machine-type=n2-standard-32 \  
--min-cpu-platform "Intel Cascade Lake" \  
--vm-count=10 \  
--zone=us-central1-a \  
--project=scaling-gcp \  
--share-setting=projects \  
--share-with=scaling-gcp-1, scaling-gcp-2 \  
--require-specific-reservation
```

2. Create a VM instance that targets **scaling-gcp-reservation** by name, by using the **--reservation-affinity** and **--reservation** flags. To consume this reservation from any consumer projects that this reservation is shared with, you must also specify the project that created the reservation, **my-owner-project**.

Ensure that the instance's properties match the reservation's instance properties, including the zone, machine type (machine family, vCPUs, and memory), minimum CPU platform, GPU amount and type, and local SSD interface and size.

```
gcloud compute instances create scaling-gcp-instance-2 \  
--machine-type=n2-standard-32 \  
--min-cpu-platform "Intel Cascade Lake" \  
--zone=us-central1-a \  
--reservation-affinity=specific \  
--reservation=scaling-gcp-reservation-02
```

```
--reservation=projects/ scaling-gcp/reservations/  
scaling-gcp-reservation
```

## **Creating instances without consuming reservations**

If we want to create a virtual machine without using any of the reservations defined in the project, we can use the following command:

```
gcloud compute instances create scaling-gcp-instance-3 --  
reservation-affinity=none
```

## **Load balancing**

Google provides server-side load balancing, configuring which, we can distribute the incoming request to multiple virtual machine instances. Using load balancing you can:

- Scale your applications
- Handle high traffic
- Remove unhealthy virtual machines and re-join healthy virtual machines
- Route traffic to the closest regional virtual machine instance

GCP cloud load balancing is a managed offering from Google and hence is highly available. If an instance of cloud load balancing goes down, it is restarted again, all managed by GCP.

When you set up an autoscaler of a MIG, based on load balancing serving capacity, the autoscaler monitors the serving capacity of an instance group and adds/removes virtual machines. The serving capacity of an instance can be defined in the load balancers “backend service” and can be based on either request per second or utilization.

## **Adding instance group to load balancer**

Depending upon the type of load balancer, you can add a managed/unmanaged instance group to a target pool or backend service.

## **Aligning backend service with an MIG**

A backend service is the most widely used and necessary for most types of load balancing options of GCP. Backend service can have multiple quality and quantity of options, that can be configured. One such option is instance group. You can configure multiple instance groups in one backend service.

Backend service knows which instance it can use, and how much traffic each virtual machine can handle. Back-end services also run health check on the instances to identify the unhealthy virtual machines and not direct the traffic to those unhealthy machines.

We can add an instance group to a backend service using the add-backend command, as shown in the following code:

```
gcloud compute backend-services add-backend  
BACKEND_SERVICE_NAME --instance-group=INSTANCE_GROUP [--  
instance-group-region=INSTANCE_GROUP_REGION | --instance-  
group-zone=INSTANCE_GROUP_ZONE] \  
--balancing-mode=BALANCING_MODE
```

**BACKEND\_SERVICE\_NAME** is the name of the backend service with which we want to attach an instance group **INSTANCE\_GROUP**. Balancing mode defines the strategy to assess the backend if it can handle additional load.

## **Adding a Managed Instance Group to a target pool**

A target pool contains one or more virtual machine instances. A target pool is used for **Network Load Balancing (NLB)**, where the load balances forward the request to target pool, and one of the machines in the target pool processes the request. You can add Managed Instance Group to the target pool, so that when the instance is added or removed from instance group, the target pool is automatically updated. To add an existing Managed Instance Group to a target pool, follow these instructions. This causes all VM instances that are part of the Managed Instance Group, to be added to the target pool.

You can add a Managed Instance Group to a target pool using the `set-target-pools` command.

```
gcloud compute instance-groups managed set-target-pools  
INSTANCE_GROUP --target-pools TARGET_POOL
```

Here, a Managed Instance Group **INSTANCE\_GROUP** is added to the target pool **TARGET\_POOL**.

A Network Load Balancer (unlike HTTP(s) load balancer) is a pass-through load balancer. It does not proxy connections from clients. We use target pools (a selected group of virtual machines) to handle it. On the other hand, a HTTPs load balancer uses backend service to pass on the request to actual VM via URL mapping.

## **Configuring multi regional external load balancer**

For supporting multi regional external load balancer configuration, the idea is to have backend service backed up by individual regional Managed Instance Groups. When the client triggers a request, the regional setting directs the request to the regional subnet closer to request trigger, and the regional subnet calls backend service and back-end service calls regional managed instances. [\*Figure 5.6\*](#) features a multi-regional load balancing:



**Figure 5.6:** Multi regional load balancing

Look at the [Figure 5.4](#) labelled with numbers and follow the numerically labelled explanation as following.

1. A request is a trigger for some processing.
2. Forwarding rule redirects the request to the target HTTPS proxy.
3. Target proxy uses the rules set in URL map, to take a decision as to which backend service will be invoked.
4. Appropriate backend service is called.
5. The load balancer finds out which exact instance a request should be processed on. It is based on proximity to the client.

## **Cross regions load balancing**

We create one instance group per region and attach it to the backend service. Incoming requests are forwarded to the closest region. If the virtual machine in a region fails to serve the request, the request is sent to other instances in the same region or to a different region.

## **Conclusion**

GCP provides provision to create VM groups – managed and unmanaged. It is always advisable to select Managed Instance Group over unmanaged. With Managed Instance Group, comes a lot of vital features of mature scaling. Not

only can you apply multiple types (autoscaling and predictive) of scaling, but also configure them enough to support a wide variety of workloads. The autoscaler feature of Managed Instance Group provides autoscaling based on CPU utilizations, Load balancing serving capacity, Cloud monitoring metrics and schedules, making it apt to be used in production environments with variable loads.

## **Points to remember**

- GCP provides two kinds of instance groups - Managed and Unmanaged.
- GCP offers autoscaling and predictive scaling for instance groups.
- You can reserve instances which could be used while scaling for a certain managed instance.
- Platform also provides an elaborate list of configuration parameter to handle variety of workloads.

## **Questions**

1. Unmanaged group can have heterogenous instances added/removed manually from group. True or False?
2. Unmanaged group does not offer autoscaling. True or False?
3. MIG supports the deployment of containers to container-optimized OS, that includes docker, if the instance template used specifies a container image. Agree or disagree?
4. What are some key advantages of using Managed Instance Group over an unmanaged instance group?

## **Answers**



1. **True**
2. **True**
3. **Agree**

# CHAPTER 6

## Scaling Kubernetes Engine

### Introduction

Kubernetes is a well-known open-source container orchestration platform, available as managed service with almost all public cloud providers. In the GCP world, managed Kubernetes comes with the name **Google Kubernetes Engine (GKE)**. Similarly, in AWS, it comes with the name **Amazon Elastic Kubernetes Service (EKS)**. Cloud providers operating the Kubernetes platform, make Kubernetes adoption very easy for engineering and IT teams. One of the most famous workloads hosted on Kubernetes is the microservice architecture. However, not just microservices but big data and data science use cases also use the Kubernetes platform to orchestrate workloads.

Kubernetes is open source, and so you can download a copy of it and deploy it in your on-premises environment. The IT team must take care of the cluster's management and maintenance (scheduled/unscheduled). However, looking at the capability of Kubernetes to support so many use cases, all cloud providers have supported Kubernetes, which abstracted the maintenance and management aspects from the teams using it. Kubernetes has multiple in-built strategies to scale, including approaches based on predefined metrics within Kubernetes set up, as well as predefined external metrics, and custom metrics.

In this chapter, we will dive deep into the offerings of Google Kubernetes Engine and understand how we can plan to scale workloads on the GKE platform.

## **Structure**

In this chapter, we will discuss the following topics:

- Building and Packaging an Application on Kubernetes
  - Kubernetes Architecture
  - Building and Deploying a Web App
- Scaling an Application
  - Configuring Horizontal Pod scaling
  - Configuring Vertical Pod scaling
  - Configuring multi-dimensional Pod scaling
- Exponential Scaling of Fault Tolerant Workloads
  - Using Spot Pods
  - Using Spot VMs
  - Using Preemptible VMs
- Cluster autoscaler
  - Scaling limits
- Key considerations
  - Node pool configurations
  - Network policies for scale
  - Load balancing
  - Storage

## **Objectives**

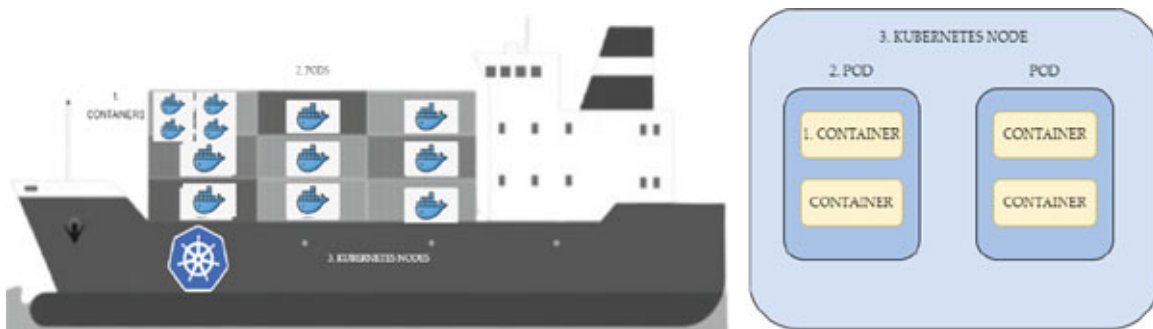
After studying this chapter, you should be able to learn how to deploy applications on Kubernetes and how to scale up the infrastructure manually and automatically. You will also get good insights into spot and preemptible instances and how to build and host applications to utilize spot infrastructures and scale exponentially cost-efficiently.

You will also learn how to scale not just the applications but the overall infrastructure, that is, Kubernetes cluster, automatically, and some key considerations which will make cluster and application autoscaling effective.

## **Building and packaging an application on Kubernetes**

Kubernetes manages the whole life cycle of a container – creation, running and deletion.

There is a famous analogy for Kubernetes: if Kubernetes is analogous to a ship, then Pods can be considered analogous to shipping containers/packing boxes. Consider [Figure 6.1](#) and examine the labelling on both the left and right images to draw the analogy:

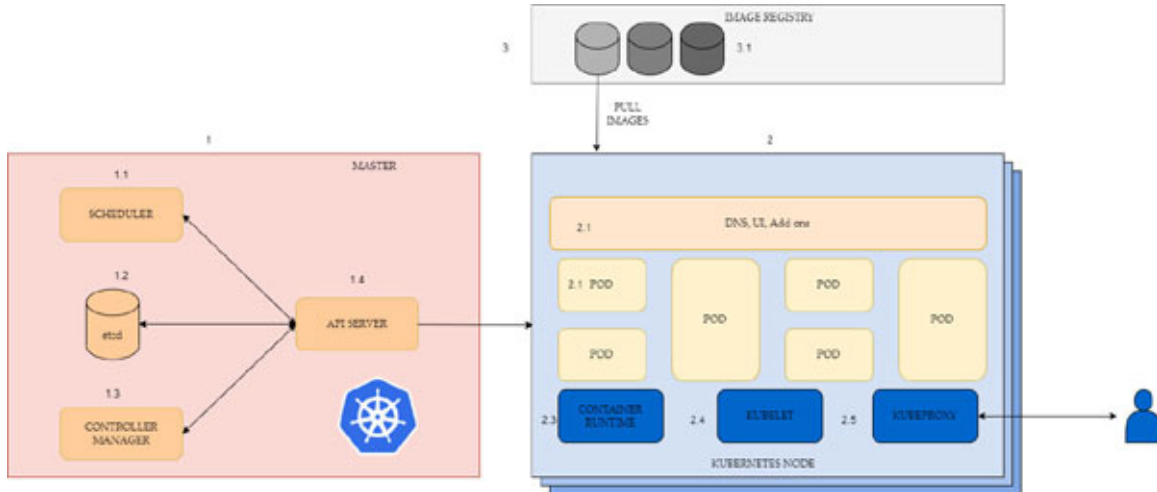


**Figure 6.1:** *Kubernetes and ship analogy*

In Kubernetes, there is a concept of a controller node (known as Kubernetes master) that manages multiple worker nodes. Worker nodes host numerous pods, and pods can hold various containers. Imagine your docker containers like packaging boxes (labelled as **1** in [Figure 6.1](#)); these packaging boxes are loaded in shipping containers analogous to pods in Kubernetes. A pod is a group of one or more containers (labelled as **2**). Just as a ship can have multiple shipping containers and manage the journey of all the shipping containers, a Kubernetes worker node can also have many pods managed by the Kubernetes master node (labelled as **3**).

# Kubernetes architecture

In this sub section, we will look into the high-level architecture of Kubernetes. Consider [Figure 6.2](#):



**Figure 6.2:** Kubernetes architecture

The corresponding descriptions for the numerically labelled sections are as follows:

1. **Master node:** The master node is the component that controls the orchestration of containers on the worker nodes. The complete life cycle of a container is managed and controlled by the master node. It contains the following components:
  - a. **Scheduler:** When there is a need to create a pod, the scheduler looks into the infra requirements and takes a call about where that pod can be placed. Along with that, it runs periodic health checks, to assess the health of the cluster.
  - b. **etcd:** etcd is the placeholder for the complete metadata of the cluster. The master node queries this data to retrieve parameters related to the state of the nodes, pods, and containers.
  - c. **Controller:** Controller gets the expected states of the nodes from the API Server. It then checks the current

state of the nodes it is expected to control, identifies if there are any deviations, and resolves them.

- d. **API Server:** This is the gateway to communication for worker nodes or other components in the cluster. It acts as the front end of the Kubernetes cluster.

2. **Worker node:** The worker node is responsible for running container workloads and additional components like container runtime, kubelet, and kube-proxy. Its purpose is to provide applications, computing, networking, and storage resources.

- a. Kubernetes provides additional services as *add-ons*; these are optional services. For example, the dashboard, which is deployed as a mini application in the worker node and well-integrated with Kubernetes components like kube-proxy.
- b. **Pods:** It is a group of one or more containers with shared storage, network, and compute power. It is the smallest deployable unit in Kubernetes.
- c. **Container runtime:** A container node must have a container run time environment. For example, the most widely used container runtime is Docker.
- d. **Kubelet:** Kubelet is a small application running on worker nodes and facilitates communication with the API server. It transmits the status of components running in the Kubernetes worker node and transmits information like status, infra consumption, and health status, and in return, takes commands from the API server and executes inside the worker machine.
- e. **Kube-proxy:** Kube-proxy is the network proxy and load balancer that enables the network to route requests to appropriate pods. It routes traffic to the correct pod, based on the coming requests' service name and port number.

3. **Image registry:** Image registry is a repository or a collation of repositories, where we store and access container images.

Kubernetes world is big and the preceding section by no means covers everything. However, it is enough to investigate the scalability aspects of workloads deployed. With the preceding basic knowledge of Kubernetes, it is time to deploy the first sample web application to see things running.

## **Building and deploying a web app**

This section will be a hands-on section, where you will deploy a sample web application with the preceding architecture in action. [Table 6.1](#) maps the components above, with the tool used in the example. For example, you will be using Docker as container runtime for this example. Please refer to the following table:

Components	Tools
Container Runtime	Docker
Image Registry	GCP Container Registry
Kubernetes	GCP managed Kubernetes.
Language and build tools	Java and Maven

**Table 6.1:** Components - technology Mapping for GKE

Refer to the code base (`scaling_kubernetes_engine`) shared with this chapter. At the root level, you will see a 10-stepword document `Deploy_Kubernetes_App.docx`. Follow the exercise step by step to deploy a basic Kubernetes based web application. Once done, we will be using the same app for further exercises.

1. Manual scaling of the application.  
`kubectl scale deployment hello-java --replicas=3`
2. Deployments are just the declaration states of components in a Kubernetes application. For example, the

deployment file mentions details like the number of replicas. When such a thing is mentioned, Kubernetes will make sure that it maintains the many numbers of replicas. Generally, in the industry, we use the YAML file to create the complete deployment specification.

In the preceding application, we can get the YAML file using the following command:

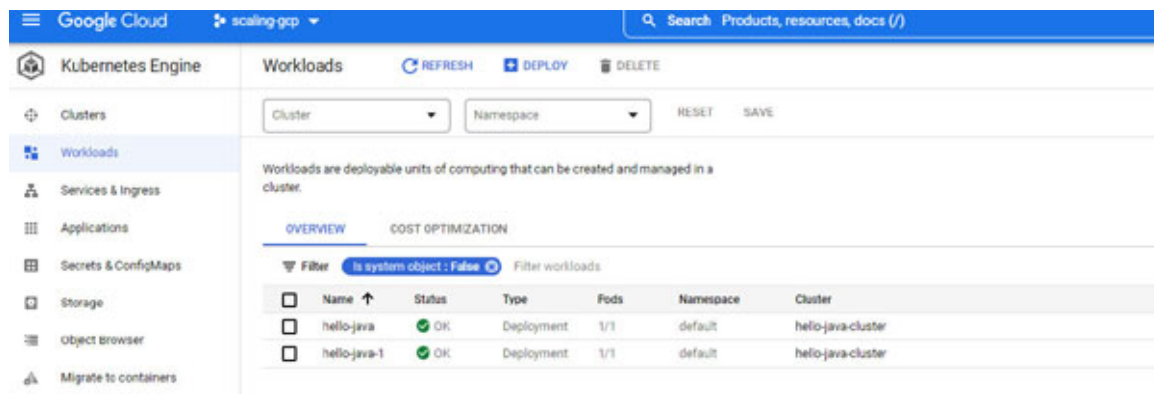
```
gcloud container clusters get-credentials hello-java-cluster
--zone us-central1-c --project scaling-353402 &&kubectl get
deployment hello-java -o yaml>>deployment.yaml
```

This command returns a YAML configurations and dumps them in the **deployment.yaml** file.

We will look into the constructs of YAML soon, but for now, let us change the name of the application. Modify the field **.metadata.name** to **hello-world-1** and redeploy the YAML configuration with the following command. The result of using this command is shown in [Figure 6.3](#):

```
kubectl apply -f deployment.yaml
```

Please refer to the following figure:



**Figure 6.3:** Kubernetes deployments

This will create one more workload in Kubernetes with name **hello-java-1**, which will use the same image used by **hello-java** workload.

3. You can delete a deployment using the following command:

```
kubectl delete deployment hello-java
```



4. Let us have a look of the Deployment YAML for the preceding application. Please note that Deployment YAML can become very complex depending on use case to use case:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-java
spec:
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: hello-java
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
      type: RollingUpdate
  template:
    metadata:
      labels:
        app: hello-java
    spec:
      containers:
        - image: gcr.io/<gcp-project-id>/hello-java:v1
      imagePullPolicy: IfNotPresent
      name: hello-java
      resources: {
```

[Table 6.2](#) provides descriptions of the various field names for Deployment YAML:

Field Names	Description
apiVersion	Version of Kubernetes API. The community releases new APIs and improvements periodically. This attribute refers to a release of APIs.

kind	Kubernetes has various kind of objects like deployment, horizontal pod scaling and vertical pod scaling.
.metadata	Data that helps uniquely identify the object, including a name string, UID, and optional namespace
.metadata.name	Name of the object.
.spec	Desired state of the Kubernetes object.
.spec.replicas	Number of replicas of the deployment.
.spec.strategy	This field describes the deployment strategy of a pod. Common strategies are recreate, rolling update, and so on.
.spec.template	Defines the configuration for different objects.
.spec.template.containers.image	Image of the container. This is normally pulled from image registries.
.spec.template.containers.resources	Infrastructure resources which will be configured for a pod.

**Table 6.2:** Deployment YAML for hello-java app

## Scaling an application

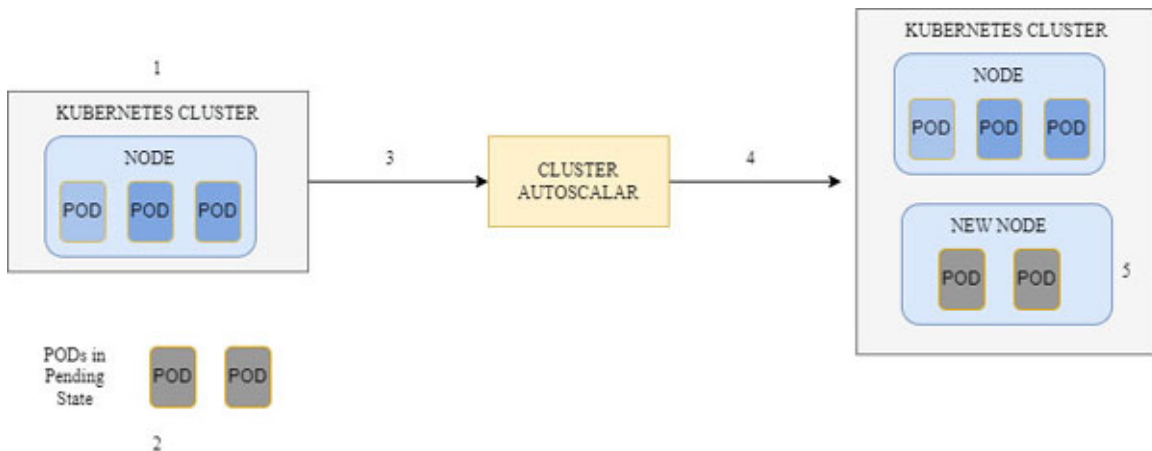
There are two levels of scaling in Kubernetes. One type of scaling is the Kubernetes cluster itself, that is, increasing the size of the Kubernetes cluster because of obvious needs for an increased number of deployments or no more infrastructures left for applications to scale up.

Another form of scaling is the scaling up of application. When the load increases/decreases in infrastructure, the deployed application should have the capability to increase/decrease the infrastructure for the application, either manually or automated.

In both scenarios, manual scaling can happen; however, you have options of autoscaling as well. Let us explore how and when autoscaling kicks in and what the relationship is between the preceding two scalings.

## Scale-up mechanism

The cluster autoscaler is the component that scales up the size of the cluster (increases nodes) when there are pods that are unable to launch due to a resource crunch. We can define the minimum and the maximum number of nodes in this policy. Consider [Figure 6.4](#):



**Figure 6.4:** Kubernetes Cluster Auto scalar

Follow the numerical labelling in the [Figure 6.4](#) with the explanation below:

1. It is the current Kubernetes cluster. Kubernetes cluster has just one node, and that node contains three pods. There is no space left to spin up another pod.
2. Two new pods need to be created. API server scans the system for unscheduled pods every 10 seconds (by default, governed by the flag `-scan-interval`). A pod is unschedulable when the Kubernetes scheduler cannot identify a node to place a new pod. Kubernetes cluster makes the Pod condition schedulable equal to false.
3. Cluster autoscaler requests for a new node from the node group. If there is not enough space for the new pod or the cluster, max size is not reached.
4. Based on the request from the autoscaler, a new provision starts. Autoscaler expects the new node to be up and running in 15 minutes (default, governed by `--max-node-`

provision-time). If the node does not come up in the allotted time, another attempt is made to create a node.

5. Once a new node is added, the scheduler schedules the pending pods on the new node.

In case any pods are still unscheduled, repeat the process from label 2 to label 5.

## **Scale-down mechanism**

Scaling down is needed when the workload on the application is low (vice versa of scaling up). Scaling down is more complex than scaling up, and following is a step-wise description of the complete scale-down process.

1. The stage during which the evaluation of the node starts for deletion (identified by the API server due to 10 seconds of scanning), begins when the utilization on the node goes below 50% (default). It only checks resource requests to identify consumption. The actual CPU and memory details at any point are not considered.
2. After this, the Kubernetes scheduling algorithm scans the pods on the qualified nodes and identifies if the pods running on under-utilized node should be shifted to other nodes.
3. If the node is not required for a period of 10 minutes, that is, for 10 minutes, the consumption is below 50%, the node gets deleted. Time is configurable. In addition, only one node is deleted at a time; multiple such deletes might result in many un-schedulable pods.
4. Nodes not used can be deleted in bulk—for example, ten unused nodes in one go.

The basic philosophy behind the analysis done, is to make sure that the states defined for the deployment do not get deferred. For example, all the minimum replicas should run at any given point, no matter what.

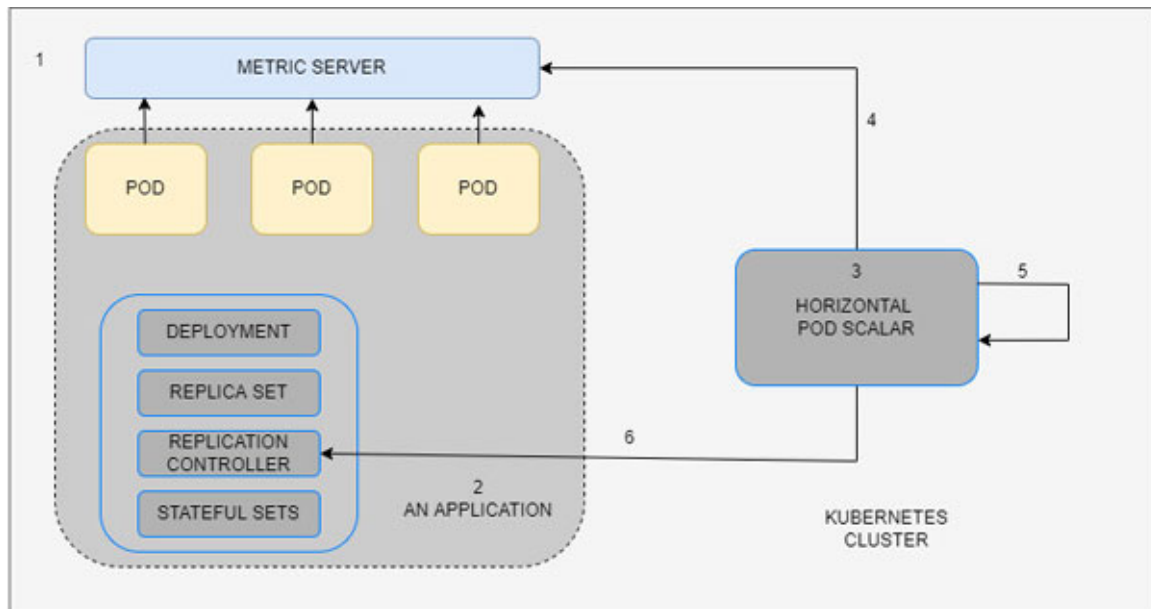
In the subsequent sections, we will explore the preceding two scaling strategies – Scaling infra for application, and scaling the complete cluster. Though you can technically configure one among the other two without restrictions, the real power of autoscaling inspires us to use both in combination.

## **Configuring horizontal pod scaling**

Horizontal pod scaling means increasing/decreasing the number of pods running for a deployment (application) in Kubernetes. Horizontal pod scalar changes the shape of the deployment by taking into consideration some metrics which fall under the following categories:

- Internal metrics from within a Kubernetes deployment, for example, CPU usage and memory usage in the deployment.
- Metrics external to the deployment on Kubernetes. For example, consider the application we deployed in the preceding sections. The application accepts HTTP requests and returns a response. You can define conditions such as increasing a pod for every 100 increments of concurrent requests.
- Custom metric reported by Kubernetes object in a cluster. For example, the rate of I/O per second by the application on the shared local disk.

Kubernetes implements a **Horizontal Pod Autoscaler (HPA)**, which runs in a control loop, meaning the Pod scaler runs periodically in a circle. By default, the interval is 15 seconds, set by setting flag `horizontal-pod-autoscaler-sync-period` in kube-controller-manager. The Horizontal Pod scaler checks the metrics and instructs the replication controller to increase the shape of the deployment. Consider the numerically labelled [Figure 6.5](#):



**Figure 6.5:** Horizontal Pod Scalar

Consider the figure's numerical labelling with the following numerically labelled points:

1. **Metric server:** Metric Server accumulates resource usage data across the whole cluster. It collects data from kubelet (running on each node) and makes it available in the API server via the K8s Metric APIs.
2. **Kubernetes application:** An application hosted on a Kubernetes cluster is also called deployment in Kubernetes. For example, we can consider the hello-java application. Each application has objects specified by the configuration.
  - a. **Deployment:** A Deployment represents an application running where we give specifications to specify the complete details. For example, the image to be used for the deployment should be `hello-java:v1` in the container registry.
  - b. **Replica Set:** A Replica Set tries to maintain a stable set of replica pods at any given time. For instance, if a pod goes down due to node failure, the replica set identifies that a new pod is needed to maintain the

current pod replica requirements, and it starts creating one on another suitable node.

c. **Replication controller:** As the name suggests, it controls the number of replicas at any given time for deployment. It takes the final decision to increase a pod and decrease a pod.

d. **Stateful sets:** This object maintains the stateful applications. It manages the scaling and deployment of pods and provides ordering and uniqueness guarantees.

3. **Horizontal Pod scaler:** A horizontal Pod scaler is an object that assesses the metrics in a Kubernetes cluster and informs the replica set to scale up or down.

4. Horizontal Pod scaler scans the metrics server for reading the metrics for which it is configured. This happens by default after every 15 seconds.

5. The horizontal Pod scaler calculates the pods needed, based on the metric defined for autoscaling.

6. Horizontal Pod scalar asks replica controller to create/delete number of pods as per latest calculations on current metrics.

## Metric threshold definition

You can define the threshold for a metric in two ways.

- **Absolute/Raw values:** You can specify absolute values as thresholds. For example, the threshold for CPU utilization can be set to 4 CPUs/mCPUs.
- **Percentage value:** You can also specify the threshold in percentages—for example, CPU utilization above 80%, scale-up.

The value is used if you specify an absolute value for a metric, either in CPU or memory. If you specify a percentage value for metric (CPU or memory), the HorizontalPodAutoscaler

calculates the average utilization value against the percentage of Pods CPU or memory utilization. The preceding strategy holds for custom and external metrics too.

## **Configuring multiple metrics**

If multiple metrics are configured for autoscaling, the horizontal Pod scaler calculates the number of pods per metric and then takes the most significant value for the pods required. This ensures that the applications adhere to all the scale-up policies. Assume a streaming application with consumer being deployed on Kubernetes application, is configured to scale on the number of unread messages in a Pub/Sub queue, as well as configured to scale, based on CPU consumption metrics. At the time of scaling, Autoscaler will calculate number of pods as per number of unread messages, and will calculate the number of pods based on CPU usage. Assume this calculation came out to be 10 in case of scaling based on unread Pub/Sub message and number of pods come out to be 5 according to CPU. System will be scaled to 10.

In case a few of the metrics are unavailable in the metrics server, the Horizontal Pod scalar will not scale down the application from the current state. However, if there is a need to scale up, that continues uninterrupted as per the available metrics.

## **Thrashing**

Thrashing is a situation where a workload requests for increased infrastructure, and until the infra is scaled up to process the load, the infra request decreases. Assume you have a REST API with configuration to scale up and down based on the number of incoming HTTP requests. If this response time is very less, and the number of request changes very fast, then you will see a situation where the infrastructure was scaled up due to high number of requests. When suddenly the number of requests is reduced, the system will try to



downscale. And in the very next instance again, there is a need to scale up. This scaling up and down takes some seconds and even before accomplishing it, the system tries to scale it back. Infrastructure will go in a loop of unnecessary scaling up and down. Auto scaler selects the most significant recommendation to tackle this problem, based on data from the previous 5 minutes.

## Issuing horizontal scaling requests

You can define horizontal pod scaling configurations and apply them on the application using below 3 ways:

- From the UI console
- Using the `kubectl apply` command
- Using the `kubectl autoscale`

The first option of using UI console, can be used for one of situations. However, it is definitely not the recommended way, as infrastructure creation like this cannot be automated. For the hands-on part, we will use option 2 and option 3 for our examples.

## Autoscaling on resource utilization

In this section, we will configure a horizontal pod scaler based on CPU utilization %age. CPU utilization is an internal metric (generated within) to a Kubernetes cluster. We will try to apply the autoscaling configuration- *"If the target CPU utilization goes above 80%, scale up"* to the `hello-java` application. Let us consider an autoscaling configuration for the following scenario:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: hello-java-autoscaler
spec:
  scaleTargetRef:
```

```
apiVersion: apps/v1
kind: Deployment
name: hello-java
minReplicas: 1
maxReplicas: 10
targetCPUUtilizationPercentage: 80
```

Key specification in YAML related to scaling are highlighted in BOLD. [Table 6.3](#) provides the explanations:

Field	Description
<code>.metadata.name</code>	Configured as <code>hello-java-autoscaler</code> , it is the name given to the Horizontal pod scalar.
<code>.spec.scaleTargetRef.name</code>	Configured as <code>hello-world</code> , it is the name of Kubernetes deployment.
<code>minReplicas</code>	Configured as 1, it is the minimum pods a deployment is going to maintain.
<code>maxReplicas</code>	Maximum number for pods.
<code>targetCPUUtilizationPercentage</code>	Autoscaling configuration. When CPU usage goes above 80%, it scales up.

**Table 6.3:** Describing YAML specification options

Create a separate YAML file for this configuration. For demonstration, we are giving the name of file as **autoscale-hello-java.yaml**.

1. Run the command to check if there is a HPA already configured. It returns all the HPA configured in the cluster:  
`kubectl get hpa`
2. Create a YAML file **autoscale-hello-java.yaml** and run the following command:  
`kubectl apply -f autoscale-hello-java.yaml`
3. Now when you run the following command:  
`kubectl get hpa`

You will see the HPA object as shown in following [Figure 6.6](#):

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
hello-java-autoscaler	Deployment/hello-java	<unknown>/80%	1	10	1	4m31s

**Figure 6.6:** HPA object at CLI

4. To see a more elaborate definition of configured Autoscaling, use the following command:

```
kubectl describe hpa hello-java-autoscaler
```

5. To delete the auto scaler configurations, use this given command:

```
kubectl delete hpa hello-java-autoscaler
```

Equivalent **kubectl auto scale** command:

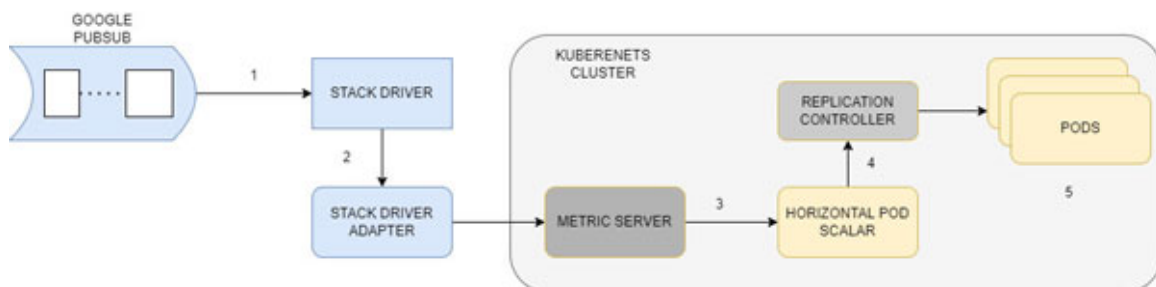
```
kubectl autoscale deployment hello-java --cpu-percent=80--min=1--max=10
```

Like the preceding autoscaling configuration, based on CPU, we can create auto scalers on memory usage as well.

## Autoscaling on external metric

You can set your horizontal pod scalar to scale, based on external parameters. By external parameter, it means the parameter resides outside of the running application. For this, we will take the example of scaling a Kubernetes deployment based on the number of unread records in Pub/Sub.

In the case of GKE, we can expose all the metrics available in GCP Cloud operation for Horizontal Pod scaler, to consume, using the *Stackdriver Adapter*. Stackdriver Adapter reads the logs generated in Cloud operations and fills up the metric server with details. Consider the numerically labelled [Figure 6.7](#) and its corresponding explanation:



**Figure 6.7:** Stackdriver Adapter Strategy

Explanation of the numerical labelling above:

1. Components like Google Pub/Sub expose metrics to the Stack driver.
2. Stackdriver Adapter reads these supported metrics and populates metadata in the metrics server.
3. The horizontal pod scalar reads these metrics from the metric server and calculates the number of pods needed.
4. The new number of pods is provided to the replication controller.
5. Replication controller, along with the deployment manager, scales up/down the number of pods.

In the following exercise, let us create the preceding picture technically one by one. There are 3 stages:

**Stage 1:** First of all, we need to have the following items up and running, to apply any HPA:

1. Enable the google Pub/Sub API.  

```
gcloud services enable cloudresourcemanager.googleapis.com
pubsub.googleapis.com
```
2. Create a Pub/Sub topic (**scaling-gcp-topic**) and a subscription (**scaling-gcp-topic-read**).  

```
gcloud pubsub topics create scaling-gcp-topic
gcloud pubsub subscriptions create scaling-gcp-topic-read --
topic=scaling-gcp-topic
```
3. Create a service account and give it the permission to read from the topic.  

```
gcloud iam service-accounts create scaling-gcp-pubsub-sa
gcloud projects add-iam-policy-binding $PROJECT_ID \
--member "scaling-gcp-pubsub-
sa@$PROJECT_ID.iam.gserviceaccount.com" \
--role "roles/pubsub.subscriber"
```

Make sure **PROJECT\_ID** variable is populated.

4. Since we want to use this preceding service account in an application in Kubernetes, we need to download and add

the service account to Kubernetes secrets.

```
gcloud iam service-accounts keys create key.json \  
  --iam-account scaling-gcp-pubsub-  
  sa@$PROJECT_ID.iam.gserviceaccount.com  
kubectl create secret generic pubsub-scale-key --from-  
file=key.json=./key.json
```

5. It is a simple subscriber application whose image resides in container registry. Download the **pubsub-hpa** folder. It contains artifacts for creating a subscriber application.

Docker build to create a local image.

```
docker build -t demo_hpa_external:1.0 .
```

Create a **tag**.

```
docker tag demo_hpa_external:1.0 gcr.io/$PROJECT_ID  
/demo_hpa_external:v1
```

Push the image to Container Registry.

```
docker push gcr.io/$PROJECT_ID /demo_hpa_external:v1
```

Subscriber code in **subscriber\_main.py**. **subscriber\_main.py** reads messages from Pub/Sub and prints the message on console.

6. Defining a Deployment YAML and applying to Kubernetes cluster.

```
kubectl apply -f pubsub-hpa.yaml
```

With the preceding step, a sample streaming subscriber application is running in Kubernetes listening to a Pub/Sub topic.

**Stage 2:** In stage 2, we must configure the Stackdriver Adapter.

1. Grant the user, the ability to create required authorization roles:

```
kubectl create clusterrolebinding cluster-admin-binding \  
  --clusterrole cluster-admin --user "$(gcloud config get-  
value account)"
```

2. Deploy the new resource model adapter on our cluster:

```
kubectl apply -f
https://raw.githubusercontent.com/GoogleCloudPlatform/k8s-
stackdriver/master/custom-metrics-stackdriver-
adapter/deploy/production/adapter_new_resource_model.yaml
```

The preceding step will create a deployment in Kubernetes cluster for stack driver adapter.

**Stage 3:** Create an autoscaling YAML configuration using the number of unread messages as the autoscaling parameter.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: pubsub
spec:
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - external:
        metric:
            name: pubsub.googleapis.com|subscription|num_undelivered_messages
            selector:
            matchLabels:
                resource.labels.subscription_id: scaling-gcp-topic-read
            target:
                type: AverageValue
  averageValue: 1
  type: External
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: pubsub
```

The highlighted section in the preceding YAML file, demonstrates autoscaling attributes. 'pubsub.googleapis.com|subscription|num\_undelivered\_messages' is the name of the metric pushed by Stackdriver Adapter into metric server.

## Autoscaling on custom metric

Autoscaling based on custom metric is like autoscaling based on external metrics. In case of external metrics, the different metrics were pushed to GCP logging out of the box. On the externally exposed metrics, you can define the auto scaler configuration.

Certain situations may arise when we want to define the autoscaling configurations on metrics, which are not available out of the box. Then, you can write code using the cloud monitoring APIs, to create custom metric and push appropriate values for the metrics in GCP logging (stackdriver).

Once these metrics are available, you can define autoscaling configurations like the external metric way.

Refer to the code available in the folder `custom-metric` where, `custom_metric.py` is a file that creates a metric and keeps pushing data every 60 seconds for the metric. The metric is specific to Kubernetes.

We will use this python file and create a docker image, push it to container registry and deploy on Kubernetes.

```
cd custom-metric
docker build -t demo_hpa_custom_metrics:1.0 .
docker tag demo_hpa_custom_metrics:1.0 gcr.io/<gcp-project-id>/demo_hpa_custom_hpa_metrics:v1
docker push gcr.io/<gcp-project-id>demo_hpa_custom_hpa_metrics:v1
```

The preceding command will deploy a custom-metric application. Moreover, the Stackdriver Adapter application must be up and running as it was in case of external metric.

You can now define a custom metric auto scaler. Refer to `custom.yaml` in the code base.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  .
spec:
  .
```

```

-type: Pods
pods:
  metric:
    name: my_metric
  target:
    type: AverageValue
    averageValue: 20

```

The highlighted section allows you to define scaling up on the metric, created by the application `my_metric`, when the average value (average across all pods) goes above 20. In that situation, a pod is added.

## Configuring vertical pod scaling

In Kubernetes Deployment YAML, you can mention the two critical values request and limit for memory and CPU. By setting a value for request, the cluster guarantees to allocate the mentioned number of resources. Limit defines the maximum amount a container infrastructure can scale up to, if the node has resources. Let us see these two aspects of the hello-java application we created before:

Original hello-java YAML	Updated hello-java YAML
<pre> apiVersion: apps/v1 kind: Deployment metadata:   name: hello-java spec:   .   .   template:   .   spec:   containers:   .     resources: {} </pre>	<pre> apiVersion: apps/v1 kind: Deployment metadata:   name: hello-java spec:   .   .   template:   .   spec:   containers:   .     resources: {}     limits:       cpu: "1"       memory: "200Mi"     requests:       cpu: 500m       memory: "100Mi" </pre>



---

The preceding deployment specification will request for 500m CPU and 100 Mi resources at the time of creation. Kubernetes will place the pod on a certain node, based on request for CPU and memory getting satisfied upfront. Maximum size consumption can go up to 1 CPU and 200Mi of memory. This will be allocated if there are resources free on the node.

Vertical Pod scalar proactively assess the containers and provides suggestions in resources. You can view these requests via cloud console, cloud monitoring and Cloud CLI. The requests can be applied manually to the containers. However, to enable vertical autoscaling, you must enable vertical pod scaling.

```
gcloud container clusters update CLUSTER_NAME --enable-vertical-pod-autoscaling -region us-central1-c
```

Once vertical pod scaling is enabled, like HPA, you have to create specifications for **Vertical Pod Autoscaling (VPA)**. Here is a sample VPA definition for the `hello-java` application:

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: hello-java-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: hello-java
  updatePolicy:
    updateMode: "Off"
```

Note the bold part - `updateMode : "Off"`. This implies that vertical scalar will only produce recommendations and will not apply automatically. `.targetRef` specifies details of deployment for which the autoscaling is defined.

Apply the following YAML:

```
kubectl apply -f vpa.yaml
```

Let the application run for some time and then try to get the recommendations produced by the VPA configuration:

```
kubectl get vpa hello-java-vpa --output yaml
```

Please refer to the following [Figure 6.8](#):

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"autoscaling.k8s.io/v1","kind":"VerticalPodAutoscaler","metadata":{"annotations":{},"name":"hello-java"},
      "spec":{"targetRef":{"kind":"Deployment","name":"hello-java"},"updatePolicy":{"updateMode":"Off"}}}
  creationTimestamp: "2022-06-20T10:14:09Z"
  generation: 433
  name: hello-java-vpa
  namespace: default
  resourceVersion: "727913"
  uid: 8b7fe8b6-d96c-4a6b-9015-66c29e503823
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: hello-java
  updatePolicy:
    updateMode: "Off"
status:
  conditions:
  - lastTransitionTime: "2022-06-20T10:23:28Z"
    status: "False"
    type: LowConfidence
  - lastTransitionTime: "2022-06-20T10:14:28Z"
    status: "True"
    type: RecommendationProvided
  recommendation:
    containerRecommendations:
    - containerName: hello-java
      lowerBound:
        cpu: 1m
        memory: "154140672"
      target:
        cpu: 3m
        memory: "154140672"
      uncappedTarget:
        cpu: 3m
        memory: "154140672"
      upperBound:
        cpu: 15m
        memory: "665845760"
```

**Figure 6.8:** Vertical POD Autoscaling Config on CLI

See the sample output from the last step. If you remember, we did not give anything related to request and limit for containers in the original Deployment YAML (`hello-java` YAML). However, VPA recommends the values.

We can enable auto scaling by giving the `updateMode: "Auto"`. It means that the VPA controller can evict a Pod, reassign the CPU and memory requests, and then create a new Pod.

## [Configuring multi-dimensional pod scaling](#)

Multi-dimensional scaling enables you to configure horizontal scaling as well as vertical scaling.

To maintain average CPU utilization, multi-dimensional pod scalar will increase the number of CPU in a pod (vertical scale) and add number of pods (horizontal scaling) simultaneously.

For the example, throughout the chapter that is, **hello-java** application, we can define the multi-dimensional pod scaling configuration as follows:

```
apiVersion: autoscaling.gke.io/v1beta1
kind: MultidimPodAutoscaler
metadata:
  name: hello-java-autoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: hello-java
  goals:
    metrics:
      - type: Resource
        resource:
          # Define the target CPU utilization request here
          name: cpu
          target:
            type: Utilization
          averageUtilization: 60
        constraints:
          global:
            minReplicas: 1
            maxReplicas: 5
        containerControlledResources: [ memory ]
        container:
          - name: '*'
            requests:
              minAllowed:
                memory: 1Gi
              maxAllowed:
                memory: 2Gi
            policy:
```

**updateMode: Auto**

Generally, we do not define HPA and VPA on CPU or memory together. We generally combine an external metric HPA with a VPA scaling.

## **Exponential scaling of fault tolerant workloads**

Fault-tolerant workloads are workloads whose results do not get affected by failures in the infrastructure.

For example, let us assume we have a processing request queued up in Pub/Sub topic. A subscriber application is deployed in Kubernetes, which picks up one message from the queue at a time and acknowledges the message when the processing completes. Now, even if the pods terminate, the processing for one message will fail and has to be restarted. Since the message is acknowledged for processing after processing completes, the same failed message will be processed by another pod.

Suppose we want to scale the same application and increase the number of pods based on the number of messages in the queue. Then the entire infrastructure deployed as a subscriber application depends on the number of unread messages in the queue.

For these and similar other situations, we can exponentially scale using spot Pods and VMs in a cost-effective manner. As mentioned earlier, there are two preconditions: Fault-tolerance and horizontally scalable application.

Spot VMs are **virtual machines (VM)** that Google offers at a 60-90% discount, with the condition that Compute engine can re-claim the virtual machines for other tasks. The compute engine stops (by default), or deletes Spot Instances, depending on some graceful shutdown instructions. Let us look at strategies to use SPOT and preemptible infrastructure in GKE.

## Using Spot Pods

We can reduce the cost of running a workload by deploying applications using the Spot Pods GKE clusters in autopilot mode. Spot pods are pods launched on spot VM in a GKE Autopilot cluster. Spot pods are much lower-priced than the regular pods, but they can be taken away anytime. Hence, a business-critical SLA-bound long-running application is not recommended to run on this type of pod.

A GKE cluster in autopilot mode manages and provisions the complete underlying infrastructure, including provisioning of node pools and nodes, giving the ease of management to teams. Let us look at what it takes to run this strategy. Follow the given steps:

1. To use this option, we have to make sure that the cluster is running in Autopilot mode.

```
gcloud container --project <GCP-PROJECT-ID> clusters create-  
auto<CLUSTER_NAME> --region "northamerica-northeast2"
```

When we created the cluster earlier, we used “*create*” to create standard clusters. However, if we use “*create-auto*”, it creates cluster in auto mode.

2. When we add the `cloud.google.com/gke-spot=true` label to our Deployment YAML, hello-java Deployment YAML will look like this:

```
.  
.
spec:
  containers:
  - name: hello-java
    image: gcr.io/infinite-zephyr-353609/hello-java:v1
  nodeSelector:
    cloud.google.com/gke-spot: "true"
  terminationGracePeriodSeconds: 25
```

3. Use the command `kubectl apply -f <deployment-spec>` to deploy the application.

Another way to achieve the same is via defining node affinity in Deployment YAML. By node affinity, we can place our pods on VM, which satisfies multiple criteria. Talking of this use case, we can define node affinity to use spot pods, by setting **requiredDuringSchedulingIgnoredDuringExecution** to use spot VMs.

Here is modified version of the preceding used yaml:

```
.
.
spec:
containers:
- name: hello-java
image: gcr.io/infinite-zephyr-353609/hello-java:v1
terminationGracePeriodSeconds: 25
affinity:
nodeAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
  - matchExpressions:
    - key: cloud.google.com/gke-spot
      operator: In
      values:
        - "true"
```

The preceding YAML enables applying the node affinity criteria, while scheduling of pods (time of creation). During scheduling of pods, create the pods only on the VMs where the key **"cloud.google.com/gke-spot"** is said to true.

## Using Spot VMs

Spot VMs are priced less by 60-90%, but can be taken back any time by GCP. Though this taking away of infrastructure can result in application failures, a fault-tolerant application can leverage the underlying reduced cost.

To accomplish this, we have to perform the following steps.

1. Create a node pool using spot instances.  
`gcloud beta container node-pools create POOL_NAME --spot`
2. Use the same yaml as defined in the first example, in step number 2, to create the deployment.

## Using preemptible VMs

Preemptible VMs are very similar to Spot VMs, with just one added condition: the preemptible virtual machines will work for only 24 hours. It is very similar to Spot VMs, and as such, using them in Kubernetes deployments is also similar.

As in the case of node affinity of spot VMs, this is a similar 2 step process (with an additional step), creating a cluster where preemptible nodes can participate.

1. Create a cluster with preemptible nodes.  
`gcloud container clusters create CLUSTER_NAME --preemptible`
2. This step is like node affinity example of spot VMs. There, we created a node pool using spot instances. Here, we will create node pool using preemptible instances.

```
gcloud container node-pools create POOL_NAME \
  --cluster=CLUSTER_NAME \
  --preemptible
```

3. Earlier, we mentioned `cloud.google.com/gke-spot: "true"` for spot pods. Here you have to mention `cloud.google.com/gke-preemptible: "true"`.

```
.
.
  spec:
    containers:
.
  nodeSelector:
    cloud.google.com/gke-preemptible: "true"
  terminationGracePeriodSeconds: 25
```

The highlighted section instructs deployment to use Virtual machines for which the key `cloud.google.com/gke-preemptible` is

set to true.

## **Cluster autoscaler**

Till now, we looked into vertical scaling, Horizontal Pod scaling, and Multidimensional scaling. All these scaling techniques were scaling the infra consumption at the application level. However, the overall cumulative infrastructure of the cluster was constant. We can also configure our cluster to auto-scale when the cumulative needs of all deployments in a cluster request for a larger cluster size, when cluster is scaled up with more virtual machines. Autopilot GKE cluster handles this automatically using a concept called node auto-provisioning.

Cluster auto scaler works at the node pool level. Node pool is a group of instances belonging to the managed instance group. For reading more about managed instance groups, refer the [Chapter 5, Scaling Compute Engine](#). Cluster auto scaler adds or deletes a VM in your node pool. It assesses that the pods cannot get scheduled by periodically scanning the usage across nodes. Thus, it tries to bring in more nodes to spin up the pod.

Similarly, if there are many nodes with less usage, cluster auto scaler tries to reschedule the pods spread across large machines, to a smaller number of machines and then removes the unused nodes. In the whole process of cluster autoscaling, there are chances of the pods getting rescheduled from one node to another, and this requires some time to delete a pod first and then spin it again. This results in transient disruption of the workload.

You can use multiple node pools in a Kubernetes cluster. If the node pools have the same virtual machine as part of the managed instance group, cluster auto scaler tries to scale up in keeping the consumption of VMs across node pools similar. This results in well spread-out cluster across multiple node pools. If the node pool belongs to different regions, it means that the cluster is well spread across the region too.



## Scaling limits

We can define the maximum and minimum number of nodes in a node pool. Cluster auto scaler will adhere to these boundaries. Let us look at the process to create an autoscaling cluster.

1. Create a cluster with autoscaling enabled.

```
gcloud container clusters create my-cluster --enable-  
autoscaling \  
--num-nodes 4 \  
--min-nodes 1 --max-nodes 4 \  
--zone us-central1-c
```

The preceding command creates a cluster of size 5. Node autoscaling is enabled. Scaler can reduce the size of each node pool to 1 and can expand each node pool to 5 nodes.

2. Another variation of the preceding command is as follows:

```
gcloud container clusters create my-cluster \  
--num-nodes 3 \  
--zone us-central1-a \  
--node-locations us-central1-a,us-central1-b,us-central1-f \  
--enable-autoscaling --min-nodes 1 --max-nodes 3
```

This command will spin up 3 nodes with each region (us-central1-a, us-central1-b, us-central1-f) having at least one node and a maximum of 3 nodes. The total maximum size of the cluster is 9 nodes.

3. You can add another autoscaling node pool to your cluster.

```
gcloud container node-pools create my-node-pool \  
--cluster my-cluster \  
--enable-autoscaling \  
--min-nodes 1 --max-nodes 2 \  
--zone us-central1-c
```

4. You can disable auto scalar on cluster.

```
gcloud container clusters update my-cluster \  
--no-enable-autoscaling \  
--node-pool= my-node-pool \  

```

```
--region= us-central1-c
```

## **Key considerations**

In this section, you will investigate some key aspects to be kept in mind which will enable effective auto-scaling in Kubernetes. These services are independent of Kubernetes, but Kubernetes's way of handling workloads expects these services to have a particular configured behaviour. A few important considerations are as follows.

## **Node pool configurations**

A node pool is a pool of nodes that have identical configurations. Managed instances back node pools. Default node pools will be used by all the deployments where you do not mention a node pool.

Kubernetes does not restrict the number and type of applications you can deploy; the nature of these applications could be different, and so is the need for infrastructure. For example, a few applications might need a specific node image or local disk, or minimum CPU. Such situations can be many, so it is advised to create a node pool by logically categorizing your infrastructure needs. For example, you can defiantly start with CPU-intensive and memory intensive pool for appropriate applications.

For a specific node pool, you can use a concept called node taints. When a workload is scheduled on the cluster, taints help applications run on a particular node pool. The significant advantage of using a node pool is controlled change and infrastructure upgrades. You can update and manage a node pool without affecting other node pools, and that gives a phase-wise upgrade of the infrastructure used by the cluster.

## **Network policies for scale**

When it comes to the way deployments in Kubernetes interacts with the outside world, there are some best practices to the egress and ingress roles. No service deployed in Kubernetes should allow incoming traffic from external IP. It should only allow the traffic from load balancer of ingress attached. If this is not done, it could result into situations where the scaling system is unable to contribute to processing the workload. In addition to this, services should only accept traffic on the protocol and port that is served by pods.

Kubernetes hosted services should only accept traffic from services (pods) that consume them, either in the same namespace or from another namespace. To write an ingress policy from a pod in another namespace, we need add a label to the namespace.

## **Load balancing**

It is recommended to create a *VPC-native* GKE cluster. Kubernetes, by default, utilizes static routes for the networking of pods, which require the control plane to maintain the routes to each node. However, this approach has obvious infrastructural bottlenecks.

In GKE, you can create clusters in VPC-native mode, which enables *container-native* load balancing that uses the NEG data model. This implies connectivity between pods without the overhead of maintaining routes, and hence no overhead of route scaling, resulting in evenly distributed traffic among the healthy backend endpoint groups.

## **Storage**

When it comes to storage, cluster auto scaler has limitations while using the local volumes. While upgrading or repairing a cluster, compute engine instance nodes are deleted, which also deletes all the data on the local SSDs. Local SSDs are not cleaned up when a node is deleted due to scale down.

Additionally, when pods request for ephemeral storage, cluster auto scaler does not support scaling up a node pool.

## **Conclusion**

Container workloads are one of the most widely used strategy to deploy business workloads. With enterprises moving to cloud with pay-as-you-go model, it is important to have right autoscaling strategies in place, to neither compromise with SLA expectations, nor spend too much as cloud cost. Kubernetes is a mature container orchestration platform with all providers providing the managed flavour of it. Autoscaling in Kubernetes has matured well to cater to current trends in workloads. Be it Microservices workloads, Big data workloads, AI/ML workloads or even monolithic workloads, Kubernetes has been a preferred choice to host the workloads.

## **Points to remember**

- GCP provides managed version of Kubernetes known as GKE.
- GKE is the container orchestration platform, which manages the complete lifecycle of a docker container.
- Kubernetes expects all configurations to be YAML based, and features like that of autoscaling, horizontal scaling, vertical scaling and so on, could be configured just by giving YAML configurations.
- GKE offers features such as Spot VM and Preemptible VMs, which could reduce the cost of your Kubernetes deployments.

## **Questions**

1. What are the main components of Kubernetes architecture?
2. What is a pod in Kubernetes?

3. What are the different services within Kubernetes?
4. What is the Ingress network, and how does it work?
5. What is the difference between a replica set and a replication controller?

# **CHAPTER 7**

## **Scaling VMware Engine**

### **Introduction**

GCP VMware Engine is an outcome of collaboration between GCP and VMware teams. This managed service enables teams to run the VMware platform in GCP, thus making it easier for enterprise customers to migrate their on-premise workloads to the cloud. This offering provides operational continuity, along with lowering infrastructural ownership. Multiple VMware-specific models like on-demand provisioning, pay-as-you-grow, and capacity optimization exist.

VMware engine runs natively on GCP bare metal infrastructure in GCP locations worldwide, with complete integration with other GCP services. GCP manages the entire infrastructure networking and other aspects, so that customers can consume VMware securely and efficiently. The offering includes *vCenter*, *vSphere*, *vSAN*, *HCX*, *NSX-T*, and all the corresponding tools making it completely compatible with your existing VMware tools. This compatibility enables teams to manage workloads seamlessly without disrupting the current security, data protection, auditing, and networking policies.

### **Structure**

In this chapter, we will discuss the following topics:

- VMware Engine
- Creating a private cloud

- Configuring autoscaling policies
  - Storage capacity optimization policy
  - CPU performance optimization policy
  - Memory performance optimization policy
  - CPU and memory performance optimization policy

## Objectives

After studying this chapter, you should be able to deploy VMware workloads in the Google Cloud Platform and understand the best practices around it. You will also understand the scenarios of deployments, which will eventually help you configure the right strategy for scaling up and down for VMware.

## VMware Engine

Enterprise applications are voluminous. Hence a big bang migration of all services to any private cloud is impossible. It is vital to have a mechanism to migrate applications piece by piece to the cloud, without affecting the business availability needs. VMware offering helps us achieve that, if the on-premises VMware load. Consider [Figure 7.1](#), which represents the most standard architectural footprint of the VMware engine:



**Figure 7.1:** VMware Engine

Follow the numerical labelling in the preceding figure with the numerical explanation as follows.

1. Label 1 represents a typical VMware workload, running in an on-premises environment.
2. VMware offering is available in GCP. GCP offers a one-to-one mapping of components available in an on-premises VMware tech stack and GCP-hosted VMware stack.
3. On-premises and GCP workloads interact using the cloud interconnect or VPN.
4. The VMware workload hosted on GCP has seamless integration with other GCP-managed offerings.

Let us now look into how to create a sample VMware engine application in GCP.

## **Creating a private cloud**

To create a private cloud, follow the given steps:

1. Access the Google Cloud VMware Engine portal.
2. On the **Resources** page, click **CREATE PRIVATE CLOUD**. A screen will open, similar to [Figure 7.2](#):



Home

Resources

Network

Activity

Account

Google Cloud VMware Engine

← Create Private Cloud ?

Private Cloud name \*

scaling-gcp-vmware-cluster

3

Location \*

europa-west2 > v-zone-a > VE Placement Group 1

4

Node type \*

ve1-standard-72

2x2.6 GHz, 36 Cores (72 HT), 768 GB RAM

19.2 TB Raw, 3.2 TB Cache (All-Flash)

5

☒ Multi Node

☐ Single Node

Node count \*

3

( 3 to 8 )

6

☒ Customize Cores

vSphere/vSAN subnets CIDR range \*

192.10.0.0

/

22

7

IP Range: 192.10.0.0 - 192.10.3.255

HCX Deployment Network CIDR range

192.12.0.0

/

26

8

IP Range: 192.12.0.0 - 192.12.0.63

Review and Create

Cancel

9

**Figure 7.2:** Create VMware private cloud

3. Give a name to your cluster.
4. Select a location for your private cloud.
5. Select the class of machines.

6. Select the number of machines for the private cloud. Generally, for production deployments, it is advised to create private cluster of at least 3 nodes. VMware Engine deletes private clouds that contain only 1 node after 60 days.

Optional: You can customize the number of available cores for each node.

7. Enter a CIDR range for the VMware management network.
8. Enter a CIDR range for the HCX deployment network, which is used for deploying HCX components.

**NOTE: Make sure that the CIDR range does not overlap with any of your on-premises or cloud subnets. The CIDR range must be /27 or higher.**

9. Select **Review and Create**.
10. Review the settings.
11. Click **create** to launch provisioning the private cloud.

Private cloud creation can take anywhere between 30 minutes up to 2 hours. After the provisioning is complete, you receive an email.

## **Configuring autoscaling policies**

Autoscaling policies automatically scale up or down your private cloud cluster. Autoscaling policies are based on memory, CPU, and storage utilization thresholds. GCP VMware monitors the workload based on metrics and automatically adds and removes nodes from the cluster. There is one restriction to VMware autoscaling: autoscaling a private cluster with one VM is impossible.

You can only configure the pre-configured autoscaling policies. There is no provisioning to create autoscaling on

custom metrics, as was the case with managed instance group and Kubernetes. Moreover, the threshold must withstand at least 30 minutes for the scaling action to start. For example, if you configured to auto-scale when memory goes above 70%, then a private cloud will only scale up if the spike in memory remains above 70% for at least 30 minutes.

CPU and memory incremental values are generally not interconnected. Hence, when the autoscaling is configured for both, policies that define scale-up take OR of both metrics, and while scaling down, AND of both metrics is taken into consideration.

[Table 7.1](#) features the pre-configured autoscaling policies available:

Autoscaling policy	Scale up\Add node	Scale down\Remove node
Storage capacity optimization policy	Storage consumed > 75%	Storage consumed < 40%
CPU performance optimization policy	CPU utilization > 70% OR Storage consumed > 75%	CPU utilization < 45% AND Storage consumed < 40%
Memory performance optimization policy	Memory utilization > 80% OR Storage consumed > 75%	Memory utilization < 45% AND Storage consumed < 40%
CPU and memory performance optimization policy	CPU utilization > 70% OR Memory utilization > 80% OR Storage consumed > 75%	CPU utilization < 45% AND Memory utilization < 45% AND Storage consumed < 40%

**Table 7.1:** Autoscaling strategies for VMWare workloads

Autoscaling configuration does not allow you to change the percentage value while configuring autoscaling. You can however, mention the number of nodes. [Figure 7.3](#) is a snapshot of the autoscaling configuration screen of GCP UI console:

Google Cloud VMware Engine

← Edit cluster Cluster

Autoscale policy \* 1

cpu-performance-optimization-policy

Policy details

Add Nodes 2

If CPU utilization > 70% for 30 minutes OR Storage consumed > 80% for 30 minutes add \*

1

Remove Node 3

If CPU utilization < 45% for 30 minutes AND Storage consumed < 40% for 30 minutes remove \*

1

Cool off period 4

Delay between add/remove of nodes (in minutes) \*

30 300

Node limits 5

Minimum number of nodes in the cluster \*

3

Maximum number of nodes in the cluster \*

4

Submit Cancel

**Figure 7.3:** Autoscaling VMware private cloud

Follow the numerical labelling in the preceding figure with the numerically labelled explanation as follows.

1. This section represents the strategies described above.
2. Specify the number of nodes to add with each threshold reached of scale up.
3. Specify the number of nodes to remove with each threshold reached of scale down.
4. Cool off period is the period for which the VMware engine will wait for a spike to normalize. If it does not,

then scaling up or down is triggered.

5. **Node limits:** Specify the minimum and maximum number of nodes to be allowed for the private cluster.

## Conclusion

Any enterprise that has VMware workloads in the on-prem environment and is willing to migrate to GCP, can use the VMware offering by the Google Cloud Platform. With VMware, workloads can be migrated to GCP without any changes to your application. Well-defined scale-up and down strategies make VMware more usable from a cost perspective.

In this chapter, we learnt how to Auto scale our VMware workloads. We also looked into how we can create a private VMware cloud and the different strategies which could be used to auto scale infrastructure.

## Points to remember

- VMware Engine eases the migration of your on prem VMware workloads to GCP.
- There is not much configuration in autoscaling configurations, as this platform is assumed to be like an on-prem setup, and hence similar scaling capability as that of an on prem application.
- The VMware workload hosted on GCP has seamless integration with other GCP-managed offerings.

## Questions

1. You can define scaling based on custom metrics in VMware Engine. True or False?
2. Can you change the threshold values of metrics (like CPU usage) defined by VMware engine?

## **Answers**

1. False. VMware do not allow you to scale on custom metrics.
2. No, we can only use preconfigured values.

# **CHAPTER 8**

## **Scaling App Engine**

### **Introduction**

Google App Engine is a serverless PaaS offering from GCP to host web applications. Serverless implies that there is no infrastructure on which the application is deployed when there is no usage, and when the need arises, GCP keeps on spinning new deployments for the applications. Currently, Google App Engine is a mature platform to support applications running on mobile or any web applications. You are required to write business logic in one of the supported languages (*Go, PHP, Java, Python, Node.js, .NET* and *Ruby*) and configure the complete behaviour using the configurations. GCP, in turn, takes care of creating and assigning infrastructure, maintaining it, and scaling up and down the applications.

There are some key advantages and disadvantages of using App Engine. The major advantage of using this strategy to write a Web app is the low number of configurations for scaling up and down. Penalties could be that an application deployed using App Engine is specific to run only on GCP.

Among multiple aspects of configuring and deploying applications effectively on the App Engine, this chapter will scratch strategies and ways to establish cost-effective scaling applications on the App Engine.

### **Structure**

In this chapter, we will discuss the following topics:

- App Engine under the hood
- Standard App Engine vs. Flex App Engine
- Standard App Engine
  - Configuring basic scaling
  - Configuring manual scaling
  - Configuring autoscaling scaling
- Flex App Engine
  - Configuring manual scaling
  - Configuring autoscaling scaling

## **Objectives**

After studying this chapter, you will be able to handle the scalability aspects of deployments on Google App Engine. You will be able to select the best class of machines for your applications, and define crucial scale-up and down parameters for applications, by looking at current and future ambitions of workload on applications. With the App Engine, there is no server to maintain; you can simply deploy an application, and you are all set for production-level aspects, taken care of by the Google Cloud Platform.

App Engine automatically scales based on incoming traffic, and aspects of applications like load balancing, authorization, connection with databases, caching, traffic splitting, logging, rollouts, and rollbacks, are natively supported and easily configurable based on needs. There are two modes of deployment - a standard environment and a flexible environment. The two environments provide developers with a lot of flexibility, with each type of deployment having strengths and weaknesses based on the nature of the applications. You will deep dive into standard vs. flexible in the next section, but for now, let us start with a simple Spring boot web application deployment in a



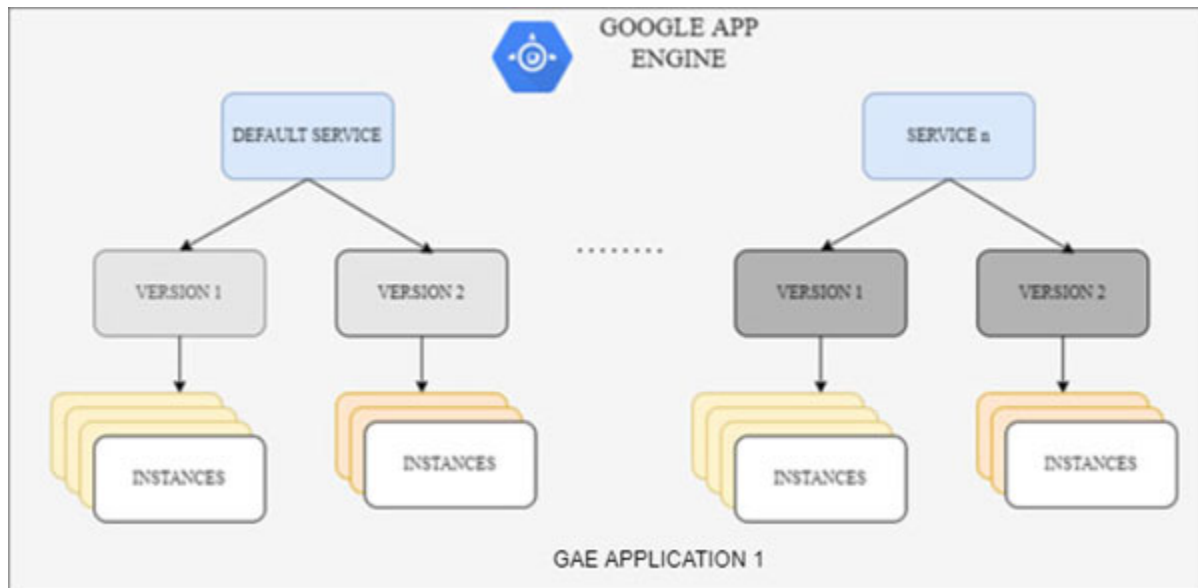
standard environment. The standard environment scales down to zero instances under no load, and it automatically scales up.

## **App Engine under the hood**

App Engine uses Google managed environment. There are 4 key organization terms in App Engine applications, and they are as follows:

- **Applications:** It is an abstraction that represents all your App Engine services. Each project has one application. The idea behind such a notion is that a project means an application/one use case. This application acts as a container for your deployment of business logic.
- **Services:** Like microservices, you can define multiple services in one App Engine application serving one specific purpose.
- **Versions:** These are point-in-time snapshots of services. App Engine supports multiple versions of your application at a time.
- **Instance:** The App Engine uses the notion of instances to define a chunk of infrastructure. A service is deployed on an instance and scaling of service under heavy load means increasing the number of instances serving the service.

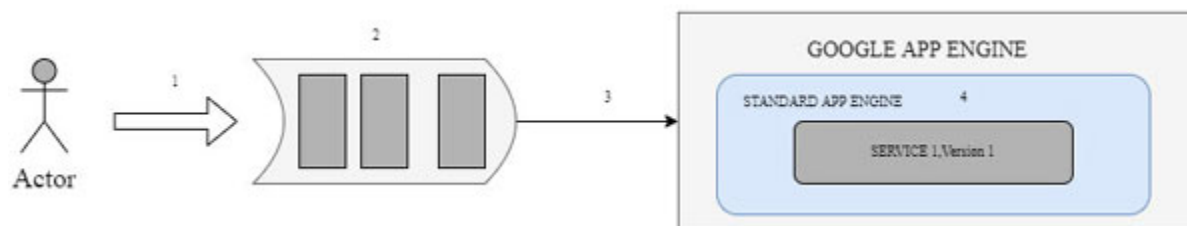
Consider the [\*Figure 8.1\*](#), which demonstrates relationship between the preceding components:



**Figure 8.1:** App Engine

In the preceding figure, there is just one App Engine application (**GAE APPLICATION 1**), and it has multiple services with multiple versions. Each version is running on multiple instances.

Let us now try and understand the lifecycle of a user request. Consider [Figure 8.2](#):



**Figure 8.2:** User request cycle

In the preceding figure, a user submits processing requests (1) to an App Engine service. These user requests are first queued (2), and this happens for each version of a service, that is, a queue is maintained for each version of a service. As per the completion of processing in the App Engine, requests are pulled (3) and processed one by one. Once the processing request is handed over to App Engine, the App Engine can execute the processing via GCP managed

environment (**4**), known as a standard App Engine (or in the user-managed environment, known as the Flex App Engine).

In this section, we will explore how to deploy a simple web application using the Google App Engine. App Engine apps are easy to create, easy to maintain, and easy to scale up and down, based on traffic and storage needs.

Refer to the code base shared with this chapter; the code base described is present in **folder gae-standard-app**. Perform the steps noted as follows, to have your first Google Standard App Engine up and running:

1. Take the code base and move to the folder **gae-standard-app**.
2. Open **pom.xml** and update the **GCP-PROJECT\_ID** variable with your project id.
3. Run the following command:

```
mvn -DskipTests package appengine:deploy
```

An App Engine application with the name default will be created. The version of the application can be modified in the pom.

4. Trigger the command given as follows, to browse through all the App Engine services:

```
gcloud app browse
```

This command will list all the App Engine services running in your current project. The list also contains the base URL which could be used to launch the web application.

Let us look more into App Engine specific constructs in the code base. In the **pom.xml** file refer to the plugin section:

```
<build>
  <plugins>
    .
    .
    .
  </plugin>
```

```
<groupId>com.google.cloud.tools</groupId>
<artifactId>appengine-maven-plugin</artifactId>
<version>2.2.0</version>
<configuration>
  <version>1</version>
  <projectId>GCP-PROJECT_ID</projectId>
</configuration>
</plugin>
</plugins>
</build>
```

The option mentioned in the deployment command (step 3) **appengine:deploy** is part of the plugin **app engine-maven-plugin**. This plugin deploys the application as an App Engine service. There are a couple of important parameters passed here:

- **version**: This is the version of the web application. If you want to support multiple versions of your web application, make sure to increment this counter.
- **projectId**: The parameter is the mention of the project id where you want to deploy the web application.

Refer to the App Engine service descriptor file at **src/main/appengine/app.yaml**. It has two properties mentioned:

- **runtime**: It is configured as Java 11. This is the specification for the runtime environment.
- **instance\_class**: Defines the type of machines where the service will be deployed. The value **F1**(default machine type) is specific to the machine type defined for Google App Engine.

The location at **src/main/appengine** in java-based applications holds special importance. This is the directory where you can define all the custom configurations by default, related to aspects of scaling, routing, and periodic (cron) tasks. In the case of other runtimes, for example, Python, the

concept remains the same, that is, deploy your business logic with a set of App Engine service descriptor files; the way deployment command is triggered could be different. In the case of Python, the following command works:

```
gcloud app deploy [CONFIGURATION_FILES]
```

**Example:** GCloud app deploy `app.yaml` `dos.yaml` `index.yaml`

[Table 8.1](#) describes the various app descriptor files and their purpose:

Descriptor file name	Purpose
app.yaml	Each service in your app has its own descriptor file. You can define all the app's setting in the app.yaml file. It contains information like the runtime (Java11,Python, and so on), as well as the class of machines and the latest version identifier.
appengine-web.xml	This file is needed when you are migrating an existing App Engine service from Java 8 to Java 11. This file specifies information about your app and helps to differentiate between the static files (images) and resource files within the app's WAR file.
cron.yaml	You can configure cron jobs, that is, jobs which run at specified intervals automatically.
dispatch.yaml	This configuration allows you to override the routing rules. You can use this file to send a request to a specific service, based on the path or hostname in the URL.
web.xml	This file is a description of how the URLs map to a specific servlet. This is specific to java-based run times. This file resides in app's WAR under WEB-INF/ directory.

**Table 8.1:** App Engine descriptor files

In the preceding example, we looked into the GCP App Engine standard environment. There is another environment known as flexible App Engine environment. The primary difference between both is that in flexible environment, the IT teams have more control on the infrastructure. For

example, rather than adhering to standard infrastructure provided by GCP, you can define a completely custom infrastructure. You will get more details about the deltas in both approaches in the next section. You can convert the Hello World Servlet App Engine service that we deployed in case of standard App Engine, into a flexible app environment.

Refer to the code in the folder `gae-flex` app. When you will compare the contents in two folders (`gae-standard-app` and `gae-flexible-app`), there are only two changes in the file.

1. Changes to the `pom.xml`. These changes are forced changes, as flexible environment only supports Java till version 8. So, you will have to lower the maven artefact version to support an older version of Java. Please note that App Engine specific property version as well as the Google project id remains as is.
2. Another change is the obvious one: modify the descriptor file `app.yaml`. In the case of flexible App Engine, you must define the flexible environment specific properties, which give you more control on the underlying infrastructure.

Apart from the preceding two changes, the rest remain same. For example, you can deploy the application using command:

```
mvn -DskipTests package appengine:deploy
```

## **Standard App Engine vs. flex App Engine**

App Engines services are best suited for web applications (HTTP based) with microservices architecture, especially if you have flavours for services that are good with standard controls and need a few services with more control. Aspects of networking and application scaling is managed by GCP. In

this section, you will investigate some critical differences in both approaches. Studying these differences will help you understand the need and to select the right App Engine environment. Consider the following [Table 8.2](#):

Standard App Engine	Flexible App Engine
In standard App Engine, the service (with a specific version) runs in sandboxes in one of the languages supported by the platform. You cannot select a programming language that is not supported by GCP.	Flexible app engine services deploy docker containers on Google-managed instances. You can use any of the programming languages which Docker supports. Since the number of languages supported in Docker is way ahead of the GCP standard environment, a flexible app engine service allows teams to make technology selections freely.
If the service is expected to need quick scale ups and downs, a standard app engine will be more suitable.	If you have a workload where it is all right for the services to scale up and down gradually, a flexible app engine would suit you more.
A standard environment is best suited for optimally using infrastructure from a cost perspective. Since the standard environment scales up and down quickly, the period of underutilized and overutilized infrastructure is minimal.	A flexible app engine takes time to perform scale up and down. In case of no load on a flexible environment, at least one instance will always be up and running, resulting in an ongoing cost.
WebSocket and background threads are not supported in the standard environment.	On a flexible engine, both are supported. A flexible app engine takes more time to boot up.

**Table 8.2:** Standard App Engine vs flex App Engine

It is recommended to do a comparative study of such features before opting for the correct type of environment. Selecting the type of environment as soon as possible in the development life cycle is beneficial. But if it is not possible, always start with a standard environment, since that is the GCP recommended approach, and many things get managed out of the box. As soon as your team has enough details to deploy an application in a flexible environment,

you can easily do that. As you can see in the previous section, the business logic did not need a change. If the basic software practices are followed, this change of environment is only related to descriptor files.

## **Standard App Engine**

In this and the section after, we will look into scaling aspects of standard and flexible App Engine workloads. There are three types of scaling supported in the standard environment of the App Engine:

- **Manual scaling**

In the case of manual scaling, the application keeps on running the configured number of instances, even if there are situations of underutilization or overutilization of infrastructure. This implies a consistent cost even when there is no system load.

- **Basic scaling**

In the case of basic scaling, the number of instances is added when the request made to service could not be fulfilled by the current quantity of infrastructure. Scaling down happens if the instance is kept idle for a designated time. Since this is pre-emptive scaling, the scaling up takes time. Hence, there are chances of your request waiting for a certain time needed for adding instances.

The App Engine waits for the instances to come up and be ready to serve the requests. Until that time, user requests are queued.

- **Automatic scaling**

Autoscaling is a case of proactive scaling; the App Engine scales up the environment when the thresholds are about to be reached. Since an instance is added before the actual need, it means a lower wait time for



requests. This is the default scaling configuration. When you configure autoscaling, each instance maintains a queue of incoming requests, and when this goes above a certain threshold, the App Engine automatically starts adding instances.

Before jumping onto the practical side of each of the preceding strategy, let us have a look at the class of instances supported by GCP for Standard App Engine workloads. [Table 8.3](#) describes the class of machines available in Standard App Engine:

S.No	Instance Class	Memory Limit	CPU Limit	Supported Scaling Types
1	F1 (default)	256 MB	600 MHz	Automatic
2	F2	512 MB	1.2 GHz	Automatic
3	F4	1024 MB	2.4 GHz	Automatic
4	F4_1G	2048 MB	2.4 GHz	Automatic
5	B1	256 MB	600 MHz	manual, basic
6	B2 (default)	512 MB	1.2 GHz	manual, basic
7	B4	1024 MB	2.4 GHz	manual, basic
8	B4_1G	2048 MB	2.4 GHz	manual, basic
9	B8	2048 MB	4.8 GHz	manual, basic

**Table 8.3:** Machine options for standard App Engine

- Manual and basic scaling share the instance class (S.no 5 to 9), whereas automatic scaling has its own separate class (S.no 1 to 4).
- The columns '**Memory limit**' and '**CPU limit**' define the upper limit to one instance.

Selecting a suitable class of machine for your workload is not straightforward. Doing multiple performance runs to understand the best classes, would be the best course of

action. However, you can start with some educated guesses. The following are a few scenarios and the selected type of machines:

- Applications with low memory and compute requirements should go with F1 and B1 machines. The idea is to scale a bare minimum infrastructure so that the effective SLAs can be met without over-provisioning to the best of our capabilities. However, in the case of basic scaling, consider that each spin-up of an instance takes time. You can compensate for that by being aggressive while defining the scaling thresholds.
- If your application is processing data, try and configure an infrastructure that is just enough to complete one request at a time. The strategy remains the same: to have a bare minimum over-provisioned infrastructure.
- If your application needs high compute power, and low memory, select the F4/F2 and B8 classes of machines. It depends on the situation, and any addition to computing will result in memory added. Similarly, if you have high memory requirements, select F4 and B8.

The following are the attributes which are utilized while configuring scaling of standard App Engine.

- **Configuring max instances:** This is the upper limit to the number of instances you allow your application to scale up to.
- **Selecting class of machines:** This selection of various classes of machines is supported by the Standard App Engine environment.
- **Selecting metric:** This is the metric whose value will determine whether to scale up or down. For example, one of App Engines' most common and widely used metrics is the average number of incoming requests.

- **Selecting threshold values:** This gives a threshold limit to the metric we defined. For example, scale up when the average number of incoming requests goes above 10.

## [Configuring basic scaling](#)

You can configure basic scaling by adding the following code fragment to your applications `app.yaml`.

```
basic_scaling:  
  max_instances: 11  
  idle_timeout: 10m
```

- **max\_instances:** Mandatory.
- **idle\_timeout:** Optional. The instance will be shut down after this duration of time, after receiving its last request. The default is 5 minutes (5m).

Refer to the `gae-standard-basic-scaling` folder and refer to `src/main/appengine/app.yaml`.

## [Configuring manual scaling](#)

Manual scaling can be configured by introducing the following code fragment in the `app.yaml` file.

```
manual_scaling:  
  instances: 5
```

Instances is a mandatory property, and this could only be applied if the class of machine is B1 or higher.

Refer to the `gae-standard-manual-scaling` folder and refer to `src/main/appengine/app.yaml`.

## [Configuring autoscaling scaling](#)

When you deploy an application in a standard App Engine environment and do not mention anything related to scaling, autoscaling is enabled by default with some default

behaviour. However, you can fine-tune that behaviour as per the need of your application. You can enable autoscaling on instances F1 or higher.

The code block as follows, when added to the `app.yaml` file, enables custom autoscaling for your application:

```
automatic_scaling:  
  target_cpu_utilization: 0.65  
  min_instances: 5  
  max_instances: 100  
  min_pending_latency: 30ms  
  max_pending_latency: automatic  
  max_concurrent_requests: 50
```

Refer to the **gae-standard-autoscaling** folder in the code base.

Let us have a look at all the possible autoscaling options available:

- **max\_instances:** This is an optional property ranging from 0 to 2147483647. A value of zero denotes disabling the autoscaling property. You can configure the maximum number of instances your application can spin up; a proper configuration is suggested to reduce noisy neighbourhood issues (one neighbour's noise affecting others) among applications. If you do not configure this value, one of your applications can spin up many instances, inversely affecting other applications.
- **min\_instances:** This is an optional property, and you can specify a value in the range 0 to 1000. This property signifies the number of instances the App Engine can scale down to, with a reduced load.
- **max\_idle\_instances:** This is an optional property with a value from 0 to 1000. This property defines the maximum number of idle instances which can be running for an application at any given point in time.

A high value will allow more instances to run even if the load does not need it. Whenever there is a sudden spike

in load, these already running idle instances are used to perform the processing. This results in a reduced impact of time taken by instances to heat up to serve meaningfully. However, a high value means a higher cost as well. A low value means more optimized for price, reducing the instances when the usage goes down. If there is a big spike in load again, many idle instances could get consumed, resulting in some load waiting for more instances to come up.

- **min\_idle\_instances:** The App Engine calculates the minimum number of instances by analyzing properties like **target\_cpu\_utilization** and **target\_throughput\_utilization**, and creates an extra number of defined instances on top of it. You will be charged for these many instances even if no workload runs on them. If you set a minimum number of idle instances, pending latency will affect your application's performance less.
- **target\_cpu\_utilization:** This is an optional property, and you can set a value in the range of 0.5 to 0.95. If not mentioned, the default value of .65 is applied by the Standard App Engine environment. A value of .7 means that when the average CPU utilization goes above 70%, new instances will be added, and traffic will be diverted to new instances. A high value for this parameter means laid-back scaling, resulting in lower performance and lower cost. A low value means more aggressive scaling, better performance, and high costs.
- **target\_throughput\_utilization** : This is an optional property and can accept a value between 0.5 to 0.95. If not mentioned explicitly, the App Engine configures this property to be .65. This property works in unison with another property, **max\_concurrent\_requests** (described next), and the App Engine environment intends to add more infrastructure when the **max\_concurrent\_requests**

multiple by `target_throughput_utilization` exceeds the value defined here.

- **max\_concurrent\_requests**: This is the number of concurrent requests the instances can handle, and any value above the value range will result in the scaling up of infrastructure. You can define a value in the range of 10 to 80. This property is used with `target_throughput_utilization`, and when the number of concurrent requests reaches a value equal to `max_concurrent_requests` times `target_throughput_utilization`, the scheduler tries to start a new instance.

It is recommended not to use a value below ten unless you intend to run a single-thread application. If the value is configured to be too high, an increased API latency might be witnessed.

- **max\_pending\_latency**: This is the maximum amount of time a request is submitted to the App Engine queue for processing. When a request is submitted to the App Engine, it is added to a queue. By default, the App Engine configures it to be 10 seconds, that is, if the processing of the request does not start in 10 seconds, the instance will be added. A higher value means a request will be allowed for a longer time in the queue. A lower value means aggressive scaling and will have obvious cost implications.
- **min\_pending\_latency**: This is the minimum time a request is definitely going to spend in the App Engine queue. The default value is 500 ms.

A request submitted for processing will wait in queue for at least the time configured in `min_pending_latency` and a maximum of time configured by `max_pending_latency`. The delta between these two values will be the time, which will give the App Engine a chance to start processing a

request on the existing infrastructure. If it is unable to get a slot after the `max_pending_latency` is achieved, a new instance will be created, and processing will be triggered. If your application uses automatic scaling, it will take approximately 15 minutes of idle time, for instances to start decommissioning them. To ensure you have at least one instance always up and running, set the property `min_idle_instances` to equal one.

## Flex App Engine

As discussed earlier in the chapter, Flexible App Engine provides more control to the engineering team to decide upon the type of infrastructure they want to use. You can add the following fragment in your `app.yaml` to override the default resource behaviour of flex app environment:

```
resources:
  cpu: 2
  memory_gb: 2.3
  disk_size_gb: 10
  volumes:
  - name: ramdisk1
    volume_type: tmpfs
    size_gb: 0.5
```

Refer to [Table 8.4](#) which describes the meaning of the key value pairs defined in the preceding yaml configuration code fragment. To see this in action, refer to the <path name>:

Configuration	Description	Default
Cpu	Using this property, you can configure the number of cores. It must be one, an even number between 2 and 32, or a multiple of 4 between 32 and 80.	1
memory_gb	RAM in GB. Use the formula: $\text{memory\_gb} = \text{cpu} * [1.0 - 6.5] - 0.4$	.6 GB

	to calculate the right value. Some space of RAM is used in housekeeping processes.	
disk_size_gb	Size in GB. The minimum is 10 GB, and the maximum is 10240 GB.	13 GB
Volumes	This complex object property defines value for the volumes attached to the Flex App Engine instances.	NA

**Table 8.4:** Configuration for flex app

As it was the case with Standard Application, flex application also supported two flavours of scaling.

## Configuring manual scaling

You can update your flexible App Engine deployment with increased number of instances as and when the need arises. Flexible App Engine will make sure that the configured number of instances runs even if when they are not needed. Also, in case more instances are needed under high load, the number of instances will not increase.

The yaml configuration fragment added as follows, can be added to the **app.yaml** to enable manual scaling.

```
manual_scaling:
  instances: 4
```

The preceding configuration will update the number of running instances to 4.

## Configuring autoscaling

Flex App Engine are docker images running on managed instances, and so one will observe similarity in the way autoscaling configuration is done.

Let us first look at the configuration options. We need to define the following configuration options:



- **Number of instances:** We need to define the maximum and minimum number of instances that we want our flex app to run on. The minimum instance should be at least equal to one, but it is recommended to use at least two. Like other autoscaling configurations, you can also set the maximum number of instances. By default, Flex services are limited to 20 instances, but you can increase or decrease that limit.
- **Selecting class of machines:** You have already seen how to configure infrastructure associated with an instance of flex App Engine. It is always recommended to create small machines, as scaling up and down happens by instance types; even a small scaling will add a large quantity of infra and vice versa, if our instances are big. However, if your App Engine is processing huge amount of data, set this value to be in accordance with business SLAs.
- **Selecting metric:** This is a property which specifies the scaling metric. You had already seen multiple examples, in case of Flex app two such metrics are supported:
  - **target\_concurrent\_requests:** When this value is specified, Flex environment calculates the total number of requests divided by the current running instances. If this value is greater than configured value, an instance is added.
  - **target\_cpu\_utilization:** When this value is configured, Flex environment calculates the average CPU utilization across all instances and checks if the average is greater than configured value. If so, instance is added.
- **Selecting threshold values:** Set the value for the preceding two metrics.

The following is the code fragment which can be added to the **app.yaml** to enable autoscaling for flex app service.

```
automatic_scaling:  
  min_num_instances: 1  
  max_num_instances: 4  
  cool_down_period_sec: 150  
  cpu_utilization:  
    target_utilization: 0.6
```

Refer to **app.yaml** file in folder **gae-flex-autoscaling-scaling\src\main\appengine** for using the preceding definition.

## **Conclusion**

Google App Engine is one of the most widely used PaaS offerings from the Google Cloud Platform. Standard app environment is the one that multiple enterprises especially use for their production workloads because of its powerful and cost-effective scaling. Since the GCP team manages this, engineers are expected to write and follow the guidelines. Google Cloud Platform also supports a Flex environment, where a lot more control lies with engineering teams.

Both versions of App Engine support manual and autoscaling. Configuring the Flex App Engine also increases efforts. All configurations are YAML based and can easily be implemented. It is always recommended to run performance tests to identify or fine-tune your scaling configurations.

## **Points to remember**

- Standard App Engine GCP is one of the most widely used PaaS offering. Every cloud provider has such an offering.
- App Engine is the most advertised way to build microservices on GCP because App Engines are

serverless, and it results in high throughput with low costs.

- App Engine has a very mature and robust yaml based configuration to define autoscaling.
- Offering like App Engine is the easiest to build for scalability, but since App Engine is not supported by any other cloud platforms, App Engine is not best suited for a business or user workflow which we intend to deploy across multiple cloud providers.

## **Questions**

1. App Engine is only suitable for short lived request response applications. True or False?
2. By default, scaling in Flex App Engine is set to autoscaling. True or False?
3. You can only write, and scale applications deployed in languages supported by Standard App Engine. True or False?
4. Can App Engine scale across multiple regions?

## **Answers**

1. False. There is no such limit. There are use cases in market where long running jobs have been hosted via App Engine.
2. False. By default, autoscaling is not present in Flex App Engine.
3. False. In this case, you can use Flex App Engine and define your own scale up and down configurations.
4. No, App Engine is regional. Apps are located within one region and GCP manages App Engine application availability redundantly across all zones in a region.

# **CHAPTER 9**

## **Scaling Google Cloud Function and Cloud Run**

### **Introduction**

In the previous chapter, we explored the App Engine – one of the PaaS offerings from Google. In the earlier chapters, we investigated scaling Kubernetes and computing VMs. While Kubernetes and compute VMs are not serverless, the App Engine was, up to a specific limit. This chapter will explore how to use the extreme serverless offerings from Google Cloud Platform – Cloud Runs and cloud functions.

Serverless architecture does not have any infra component running when there is no load on the system. The nitty-gritty of creating, hosting, and managing infrastructure is taken further away from the engineering teams, that is, their role in making infrastructure-related choices is further diminished. Engineering teams can configure high-level configuration properties related to the maximum and the minimum number of instances and memory allocation for each instance. The rest of the things are taken care of by the platform. As we become more serverless, there is an obvious advantage of reduced costs, but the control of applications is also moving away from engineering teams.

Thus, the more serverless we go, the stricter is the expectation of building the software as per guidelines. This can be a significant concern when avoiding vendor locking or supporting hybrid or multi-cloud deployments.

### **Structure**

In this chapter, we will discuss the following topics:

- Cloud Run
  - Nature of workloads
  - Infrastructural footprint
  - Autoscaling Container Instances
    - Configuring CPU Allocation
    - Configuring maximum concurrency
    - Configuring minimum and maximum Container Instances
- Cloud functions
  - Nature of work loads
  - Configuring memory
  - Configuring maximum and minimum instances
  - Addressing traffic spikes above max limits

## **Objectives**

After studying this chapter, you will be able to understand the key areas where cloud functions and runs are used, which will then help you understand the scaling needs. We will look into the infrastructure under the hood for hosting a Cloud Run and a cloud function, and will eventually look into the various strategies of scaling up and down the infrastructure, based on the need for applications.

## **Cloud Run**

Cloud Run is a Google-managed platform to run your containers. GCP will scale up and down your infrastructure based on configurations. Since it primarily allows for running containers, it brings with it the flexibility of containers, that is, tools and technologies running inside containers. If you look back to previous chapters, we had already discussed two other ways of hosting containers – one via Kubernetes engine and

two via the App Engine flex environment. The selection of the right hosting environment depends on the amount of control you want over the infrastructure. For example, if you are going to control networking and storage, want to set up observability, and support stateful applications, then GKE is the way forward. However, if such a level of flexibility and monitoring is not needed, a Cloud Run could be used.

Cloud Run is built from *Knative*, and it lets you choose to run your containers either in fully managed Cloud Run environment or on GKE cluster with Cloud Run on GKE.

Let us quickly look at deploying a Docker-based service on Google Cloud Run. Refer to the docker image “hello-java” created in [Chapter 6, Scaling Kubernetes](#). This image is already present in the container registry. In that chapter, we deployed the application on the GKE cluster. Here we will deploy that in a Cloud Run.

The following command is to be executed from Google Cloud SDK or cloud shell:

```
gcloud run deploy hello-java --image gcr.io/<project-id>/hello-  
java:v1
```

Here, `<project-id>` is the Google Cloud Platform project id. This will deploy the same image used in the earlier chapter, as a microservice in Cloud Run.

## **Nature of workloads**

Cloud Run is fully managed and is considered ideal for stateless container-based microservices that do not need features like colocation in pods (side cars) or node allocation and management. The microservices deployed are HTTP driven, and you ensure that Cloud Run handles infra, networking, scaling, provisioning, and managing servers out of the box.

The key features of Cloud Run are as follows:

- **Effortless deployments:** A container-based application could be easily deployed by triggering just a command.
- **Simplified developer experience:** All deployments are docker based; there is no other way of deploying workloads on Cloud Run.
- **Scalable:** A deployment in Cloud Run scales automatically based on the number of incoming requests.
- **Flexible:** All the tool and technology selection flexibility supported by docker, are by default, supported by Cloud Run as well.

Cloud Run scales up and down the deployment within seconds; hence the cost of deploying workloads is optimal. There are two flavours of workloads in Cloud Run - services and job. Cloud Run Services is used to run code that responds to web requests or events. For example, you can have a microservice which takes as input a Word file and converts that to PDF. Triggers for a Cloud Run services are Events and HTTP requests. Since Cloud Run is serverless, it is a fair expectation to have an auto scalable infrastructure for the microservices. Cloud Run job is used to run code that performs work, and quits when the work is done, unattended batch jobs. For example, database migration and daily report generation. Trigger from Cloud Run job could be a command from command line or cloud scheduler. The concept of scaling remains similar for both. We will use services throughout this chapter to demonstrate scaling in Cloud Run. Any application that can run in a Linux container and does not save data on a disk, can be handled by Cloud Run.

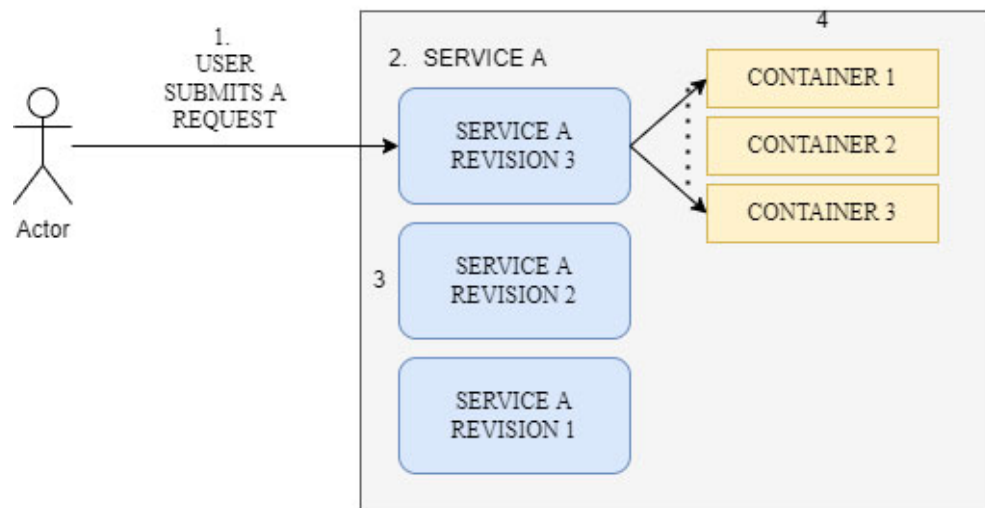
One might argue that a conversion of Word file to PDF could also be created as a job, or vice versa and that report generation could be exposed as a service. However, the time of execution is the deciding factor.

If your process can finish in seconds and respond back to the user, create a service on Cloud Run services and if it takes minutes to complete a work, create Cloud Run jobs. However,

there is a flip side as well. A cloud-run job can run up to a maximum of 60 mins. So, if some use case wants to run a job that runs more than 60 minutes, it can either look for options to split it into two or more, or may go ahead and choose options like Kubernetes for deployment.

## Infrastructural footprint

In this section, we will see the infrastructural mapping of a service deployed in Cloud Run. Consider [Figure 9.1](#) that represents a Cloud Run service. There could be a Cloud Run job as well, but it is very similar in terms of infrastructure. Please refer to the following figure:



**Figure 9.1:** Cloud Run infrastructural footprints

Refer to the numerical labelling in [Figure 9.1](#) with the corresponding numerical explanations as follows:

1. User submits a request for processing.
2. A service name “**SERVICE A**” is deployed. Current active version is 3 (revision).
3. It represents the fact that SERVICE A has been redeployed third time and previous 2 revisions have no infrastructural representation left now.



4. **SERVICE A Revision 3** is the current revision and is running on 3 instances of Cloud Run containers. These containers can be hosted in three possible ways: GCP managed Cloud Run, Cloud Run on GKE, and Cloud Run on Anthos.

## Autoscaling Container Instances

In Cloud Run, service deployment is automatically scaled (scaled up) to handle all the incoming requests. When a service does not receive traffic, the number of instances is scaled down and is scaled to zero. However, if needed, you can modify this setting to keep the service ready to serve the load and cut down on **Google Cloud Function (GCF)** cold boot time. Cold boot time is the time taken by new instances to spin up. Even if you configure the minimum instance value, there is always going to be a requirement of scaling up, which makes it important to have lower start up time for containers.

A major contributor to high start up time in case of Google Cloud Function instances are dependencies of the code. Lower the number of dependencies and lower is the linking requirement of dependencies, the better will be the start time. Moreover, if your code is using new dependencies all the time, those dependencies must be first brought to GCF cache. On the other side, if you can ensure that the application uses the same version of dependencies all the time, it is highly likely that you will get the version in the dependency cache of GCF.

A container remains active for 15 minutes without work. If no work is allocated, the container is deleted. A deletion might not happen only when the minimum number of instances is specified, and the current number of containers is already at a minimum.

## Configuring CPU allocation

The number of instances running in cloud deployment depends on the CPU utilization of existing instances when processing

requests. The target is to keep scheduled instances to a 60% CPU utilization.

By default, the CPU is allocated to Cloud Run instances during the request processing, container start up, and container shut down. You can modify this behaviour and allocate dedicated CPU slots for your deployments. This means that even when no requests are coming, CPU is allocated. Allocating the CPU is useful for applications that have short live background tasks and other asynchronous tasks.

Choosing the option to have the CPU allocated all the time is costlier than the other option of allocating the CPU when the request processing is needed. However, the latter has obvious initial warm-up delays. These delays could be reduced by carefully cutting down the mention of unused dependencies at the start, and by enabling lazy loading of modules wherever possible. If you choose to allocate CPU per request, you will be charged per request, that is, you will not incur any cost when no processing is happening. However, if you allocate CPU, there will be consistent costs incurred.

The following command configures the CPU to be allocated for a service:

```
gcloud run services update SERVICE--no-cpu-throttling
```

To set CPU allocation only during request processing, use the following command:

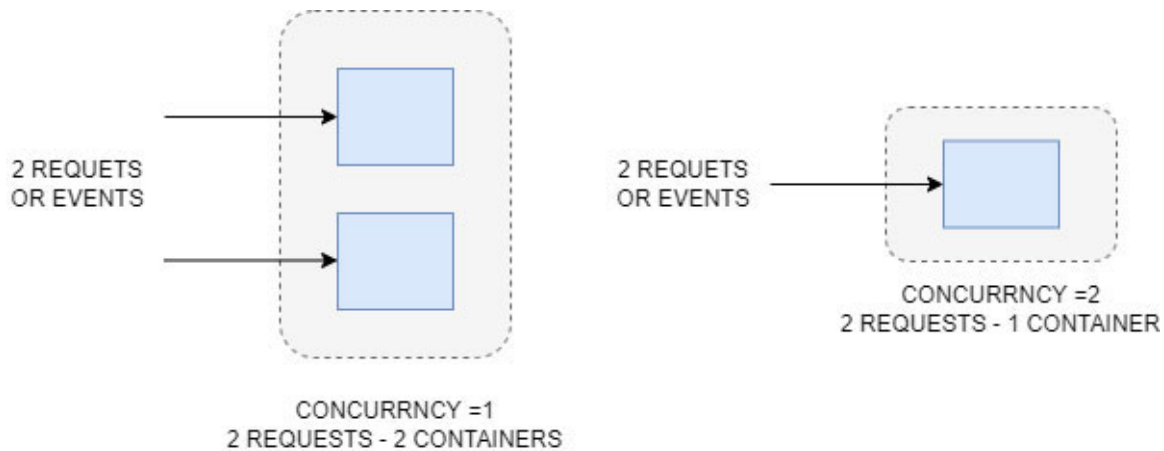
```
gcloud run services update SERVICE--cpu-throttling
```

**SERVICE** is the service for which we want or de-allocate the CPU.

## **Configuring maximum concurrency**

A single instance of Cloud Run container can serve up to 80 requests at the same time. You can increase this up to 1000. Although it is recommended to use the default value, we can modify it to fit our use case. For example, if your application cannot process the parallel request, you can set the concurrency to be equal to 1. Consider the following [\*Figure\*](#)

[9.2](#), which showcases 2 scenarios of handling 2 user request, one with concurrency as 1 and other with concurrency as 2:



**Figure 9.2:** Concurrency of cloud function

This number is just the maximum limit to the number of requests a single instance can handle. If during the actual execution, the CPU utilization became high and the maximum number of requests has still not been reached, the Cloud Run will not instantiate requests on the over-utilized instance.

You can use the following command to update the number of Max concurrent requests:

```
gcloud run services update <SERVICE-NAME>--concurrency  
<CONCURRENCY>
```

- **SERVICE-NAME** with the name of your service.
- **CONCURRENCY** with the maximum number of concurrent requests per container instance.

## **Configuring minimum and maximum Container Instances**

You can configure the maximum and minimum limit to the Cloud Run service deployment. When you specify maximum run, it is just a limit, and by no means a guarantee that Cloud Run will be able to spin up a configured number of max instances. It depends on other factors such as quota and how other applications behave. It is important to have the right

quota configured, which can handle current load as well as future loads.

However, features like retries in job and setting optimized parallelism value comes to rescue to handle such edge cases. There is also concept of affinity available, which enables a similar request to get handled, by the same container speeding up the process, which will then help speed up the execution time.

A minimum number of instances is a guarantee by Cloud Run, that these many instances will always be running at any given time.

The following is the command to configure the maximum and the minimum number of instances:

```
gcloud run services update SERVICE--max-instances MAX-VALUE  
gcloud run services update SERVICE--min-instances MIN-VALUE
```

- **SERVICE** is the name of the service deployed on Cloud Run, for which we intend to modify the maximum and minimum number of instances.
- **MAX-VALUE** is the maximum number of instances a Cloud Run service can scale up to.
- **MIN-VALUE** is the minimum number of instances a Cloud Run service will shrink to under no load.

## Cloud Functions

Cloud Functions are scalable **Function as A Service (FaaS)** offering from GCP. Cloud function allows you to follow the pay-as-you-go model, that is, pay only for the number of times you or your activities resulted in the execution of the function. This is a serverless execution environment for building and connecting services. Users write single-purpose functions attached to events emitted from cloud infrastructure or services using cloud functions. For example, you can write a function to process a file, the moment it is uploaded to a **Google Cloud Storage (GCS)** bucket.

## Nature of workloads

Cloud functions run in a fully managed, serverless environment where the complete ownership of infrastructure, networking, scaling, and security are taken care of by GCP. Each function runs in its separate environment and has no way to interact with other functions. The flexibility of using languages in case of Cloud Run is not available with Cloud functions, and you must write your function in GCP supported languages. A few prominent ones are Java 11, Python 3.7, Ruby 2.7, and so on.

Cloud functions are of two types:

- **HTTP functions:** HTTP functions are Cloud Functions that accept HTTP requests, perform processing, and revert with a response. These functions support all the request methods like GET, PUT, POST, DELETE, and OPTIONS. Functions are securely invoked via an auto-generated TLS certificate.
- **Event driven functions:** These functions listen to events, and when an event occurs, functions are triggered. Examples of events could be - new messages in the pub-sub or a new file uploaded to the storage bucket. This could be further subdivided into two categories - Background functions (functions written in Node.js, Python, Go and Java) and Cloud Event (functions written in .Net)

Let us have a quick look at how we can create a function. Follow the given steps:

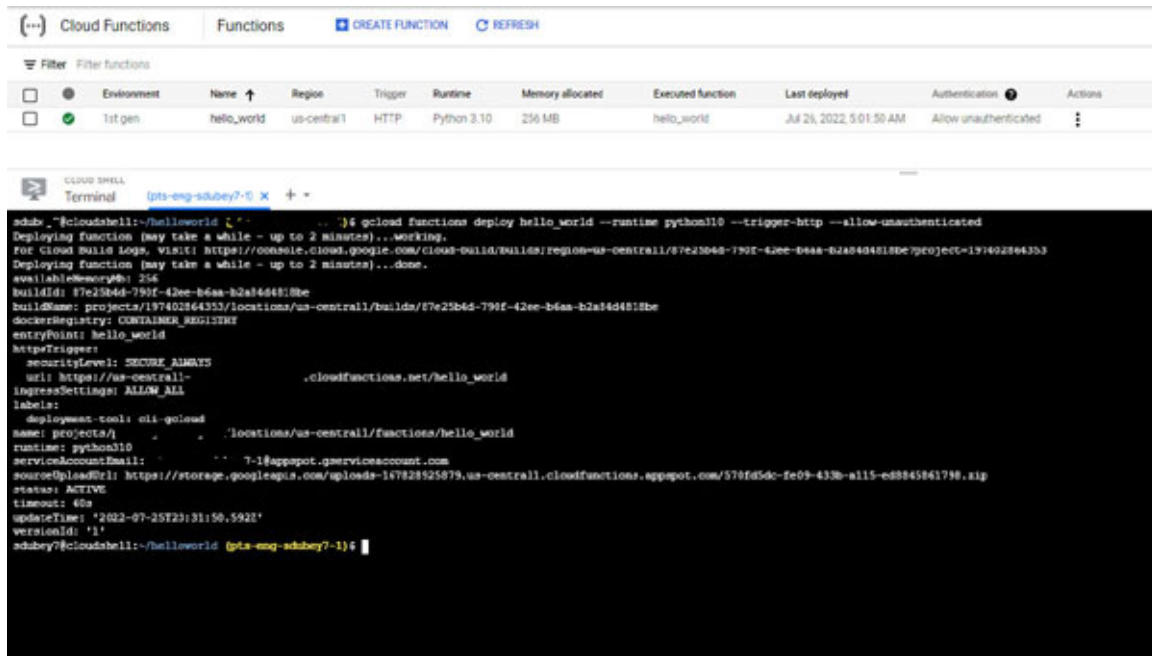
1. Refer to the code repo `scaling_cloud_function` attached with the chapter. This repo contains just one simple hello world function. Perform the following steps to deploy the hello world code as a function. Navigate to the directory where you have the `main.py` file.

```
cd scaling_cloud_function
```

2. Trigger the following command:

```
gcloud functions deploy hello_world --runtime python310 --  
trigger-http --allow-unauthenticated
```

3. After successful deployment of cloud function, the output will be as shown in [Figure 9.3](#):



**Figure 9.3:** Output after successful deployment of cloud function

4. Use the URL mentioned to trigger the function.

The critical point to note in the preceding cloud function sample deployment is that we only mentioned the business logic and did not mention anything related to infrastructure.

Until now, you would have started realizing that the complete infrastructure is abstracted away from the developers, and engineering teams have no control over the infrastructure. However, you can configure cloud functions-built configurations to control the behaviour. GCP will read these configurations and will enable infrastructure accordingly.

## Configuring memory

By default, a memory of 256 MB is allotted to each function. However, you can modify it as per the need of your application. The amount of memory you allocate corresponds

to an amount of CPU allocated for your function. Consider [Table 9.1](#) for the memory and CPU correlation details:

Memory	CPU
128MB	.083 vCPU
256MB	.167 vCPU
512MB	.333 vCPU
1024MB	.583 vCPU
2048MB	1 vCPU
4096MB	2 vCPU
8192MB	2 vCPU

**Table 9.1:** Memory - CPU ratio

The command to configure the memory for cloud function is as follows:

```
gcloud functions deploy FUNCTION --memory=MEMORY_SPEC
```

In the preceding command, **FUNCTION** is the name of the function (**hello-world** in our preceding example), and **MEMORY\_SPEC** is one of the values in the first column of [Table 9.1](#).

## [Configuring maximum and minimum instances](#)

You can configure the maximum and the minimum number of Cloud function instances you want the platform to run. Cloud function scales up by creating more instances of cloud functions, and each can execute one request at a time. A sudden spike in load might result in hundreds of cloud functions.

Scaling up is mostly beneficial. However, there could be a few situations where you will want to restrict the number of cloud functions running. For example, if your cloud function interacts with databases, letting a high number of cloud functions can create issues.

Refer to the following command to simultaneously configure the maximum allowed number of cloud functions.

```
gcloud functions deploy FUNCTION--max-instances  
MAX_INSTANCE_LIMIT
```

Here, **FUNCTION** is the function's name, and **MAX\_INSTANCE\_LIMIT** is the value of the maximum allowed number of cloud function instances running at a time.

You can delete the max instance configuration by using the following command:

```
gcloud functions deploy FUNCTION--clear-max-instances
```

You can set the minimum number of running cloud function instances under zero load. Even if there is no load, the minimum configured number of cloud functions will continue to run. This is done to reduce latencies in serving the upcoming requests. A minimum value of zero will result in the cloud function getting launched, and then serving the request. The launch time can also be reduced. This is beneficial in the case where the frequency of cloud function execution is high.

The following command sets the minimum number of Cloud function instances:

```
gcloud functions deploy FUNCTION--min-instances  
MIN_INSTANCE_LIMIT
```

You can also delete this configuration by using the following command:

```
gcloud functions deploy FUNCTION_NAME--clear-min-instances
```

Here, **FUNCTION** is the function's name, and **MIN\_INSTANCE\_LIMIT** is the minimum number of instances value.

## **Addressing traffic spikes above max limits**

It is recommended to set the maximum number of cloud function limits. Otherwise, a cloud function can scale up to very high numbers. If your cloud function is interacting with a nonscaling component (databases connections), the system could not work as expected and will be limited by the nonscaling piece or can even choke in the worst case.



Mostly, when the number of requests increases, the number of cloud function instances are added to cater to it. However, if the maximum number of instance limit is set, you can witness a scenario where there are insufficient instances to cater to incoming load. In this scenario, the cloud function tries to serve a new inbound request for up to 30 seconds:

- If an instance is free in the meanwhile, processing of a new request will start.
- If no instance is free, the request will fail.

In some cases, especially with sudden traffic spikes, the number of cloud functions can go above the maximum configured value, and if your use case cannot tolerate this, configure the max value to be lower than the maximum your use case can tolerate.

## **Conclusion**

Google Cloud Run and cloud functions are two offerings from GCP that belong to the serverless category. Cloud Run allows you to deploy docker containers without worrying about networking, security, and scale infrastructure requirements. On the other hand, Cloud Functions allow users to write single-purpose functions attached to events emitted from cloud infrastructure or services.

These two serverless options are cost-optimized, that is, you only pay for how much you use. However, such ideal conditions of complete serverless, come with known issues of some latencies at the start of processing requests. The underlying infrastructure is wholly abstracted from the user and hence, the user can only configure some allowed properties for the infrastructure to behave in a specific manner. Cloud Run and cloud functions are propriety GCP products, and any workload hosted on them might need substantial work if you intend to deploy them to other cloud providers.

## **Points to remember**

- Cloud Run and cloud functions are serverless propriety offerings from GCP.
- You can package an application as a docker container and deploy it on Cloud Run with just one command, without really bothering about the underlying infrastructure. You do not need to worry about scalability and day-to-day management of infrastructure.
- You can write single-purpose functions attached to events emitted from cloud infrastructure or services, and deploy them as cloud functions, as was the case with Cloud Run. There is no need to worry about scaling and maintaining infrastructure.
- As a user, you have minimal options to tweak the infrastructure per your need. You can only mention some properties to configure the behaviour, but essential maintenance lies with GCP.

## **Questions**

1. You want to migrate an application written in C++ to GCP, and want to use one of the serverless offerings from GCP. It is accepted to have some latencies at the start of each process. What technology option will you suggest?
2. You have wanted to write an ingestion utility and for that you are uploading files to GCS. There is a requirement to check the format of each file the moment they are uploaded. Format check has checks like schema checks and mandatory column checks. What will you suggest using – Cloud Run or cloud functions?
3. You have a data processing user journey, which runs for approximately 2 hours. You want to host the preceding user journey using Cloud Run. What will be your key concern areas?

# CHAPTER 10

## Configuring Bigtable for Scale

### Introduction

Google Cloud offers multiple managed databases. GCP offers *Cloud SQL* and *Spanner* as relational databases, Cloud Datastore as a document database, and Bigtable as a columnar database. Each of these different types of database solutions is optimized for handling certain kinds of use cases. Bigtable is used to manage large-scale structured data. It expects data to be modelled as key-value pairs, and Bigtable internally maintains a sorted partitioned map of these key-value pairs.

The Bigtable is a managed version of Apache HBase (an open sources columnar database), with some minor differences. HBase is a column-oriented non-relational database management system that runs on top of the **Hadoop Distributed File System (HDFS)**. One of the primary initial uses of Bigtable at Google was to store a web search index, and it has gone on to become one of the leading technologies, backing other systems such as Cloud Datastore and megastore.

In this chapter, we will see more details on which kind of use cases could be handled, and that will help us appreciate why scaling is exceptionally critical, especially for Bigtable as a database solution.

### Structure

In this chapter, we will discuss the following topics:

- Nature of data and its handling
- Bigtable infrastructural footprint
- Scaling Bigtable options
  - Autoscaling
    - When to Autoscale
    - Autoscaling triggers
  - Manual node allocation
  - Programmatically Autoscaling
- Limitations of Autoscaling

## **Objectives**

After studying this chapter, you will be able to understand the need and ways to scale up and down the GCP Bigtable infrastructure. We will not only investigate the infrastructure scaling, but will also get insights into a few aspects of the obstacles that come up while scaling up, and learn how to resolve them. The reader will also gain insights into the limitations of autoscaling.

## **Nature of data and its handling**

The first use case for why Google started using Bigtable was for the handling of web search indexes. Web search indexes are always supposed to be on, and the data query must be extremely fast. To solve the said requirements, Bigtable compromises on a few of the commonly used features of modern databases. The key here is the performance of queries over large datasets and infrastructure, which can scale up to store massive data. The properties of data that Bigtable handles are as follows.

## **Voluminous datasets**

Datasets handled, for example, web search index data, are expected to be of the order of petabytes. Such a large volume of data requires multiple machines to store it, and data is distributed across them, since just one machine cannot hold all. When you have such a large number of machines handling the data, the machines must use commodity hardware. High-end machines will be difficult to procure, and even if procurement is not an issue, high-end machines have very high costs. It is essential to have multiple copies of data shared across multiple machines. This is because if the server fails, a portion of the data will become unavailable.

## **High throughput**

Storing a massive volume of data is one thing, but the real business value will be compromised if you cannot query them fast enough. Each query needs to return the result quickly (in milliseconds), no matter how much data resides in the system.

## **Fast writes**

Irrespective of the already existing data and the number of read queries, a use case of a web search index also requires fast writes. Although the SLA requirements for these write requests are not as high as read/search queries, it could not be too long, as it will result in piling up of write requests. Also, a single write request might take time, but the average time taken by a write request is over a period matter.

## **Versioning changes**

Data stored is expected to change rapidly over time, bringing in the obvious requirement of storing and retrieving

the older data states without affecting the performance. This could be handled by including a timestamp in the key. However, a data store handling such aspects makes the database client reading versioned data simple and light.

## Strong consistency

By strong consistency, it is meant that anyone querying the datastore will never see stale data. If the state of the same data retrieved by two queries is different, it could potentially lead to errors in your results.

## Atomic writes

No two clients can modify the same data at a single point in time. If allowed, it could lead to dirty read issues, and hence the updates can result in inconsistencies in data.

## Selection of data

With the way queries are constructed, it is highly likely that the query will need specific columns or a collection of columns of a row. In the world of columnar databases, it is achieved by creating the column families of columns that are expected to be retrieved together.

Examples of real-world datasets for which people around the world had used Bigtable are as follows:

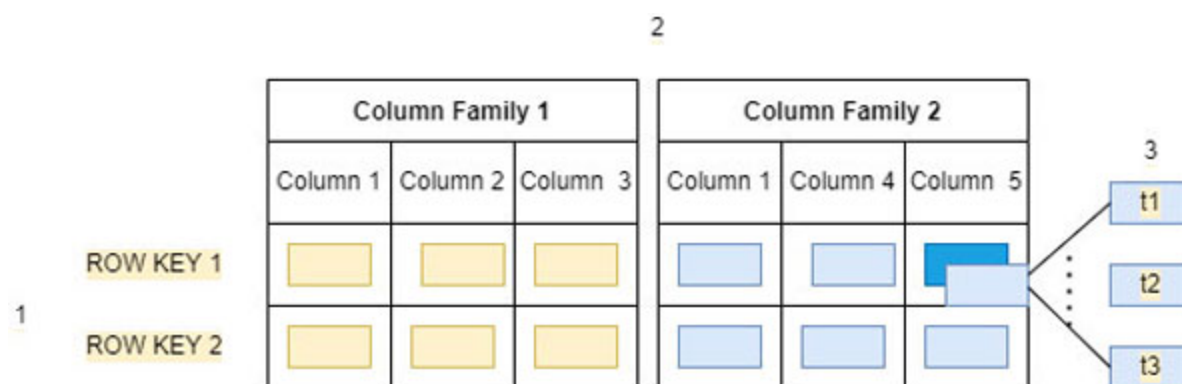
- **Time-series data**, such as storage and compute stats across multiple servers in a use case.
- **Marketing data**, such as data related to customer affinity for product.
- **Financial data**, stock market data and transaction data.
- **Internet of Things data**, data related to various sensors.

In the next section, you will see how Bigtable handles the preceding needs and the nature of the data. The complete data modelling is out of the scope of this book. However, a basic understanding will help you understand the aspects of scaling.

## Bigtable infrastructural footprint

Before analysing the architecture of Bigtable, it is crucial to understand the storage model of Bigtable.

Bigtable stores data as massive sorted partitioned maps, served via multiple nodes. The map's key represents the query parameters, and the value represents the value of the outcome of queried data. Consider [Figure 10.1](#):



**Figure 10.1:** Columnar data model

[Figure 10.1](#) features a table comprised of multiple rows (1) and multiple columns (**Column 1**, **Column 2**, **Column 3**, **Column 4** and **Column 5**) like a **Relational Database Management System (RDBMS)**. Each row is identified by a row key (1), like the primary key in RDBMS. But dissimilar to RDBMS, columns related to each other are grouped as column families (2). A single column can be used in multiple column families. For example, **Column 1** has been used in both the column family.

A row and column intersection cell can have multiple values. Each value is categorized by a timestamp (3).

With the preceding fundamental understanding of the data model, it is time to dive deep into the architecture of Bigtable. As stated before, Bigtable is a managed version of Apache HBase. The architecture looks like HBase, but GCP manages it slightly differently. These differences are primarily in the components used; for example, HBase uses **Hadoop file system (HDFS)**, whereas Google Bigtable uses Colossus, a Google-managed files system. Similarly, there are a few other differences that add value to Google Bigtable when compared to traditional HBase deployment. The advantages of Bigtable over HBase are as follows:

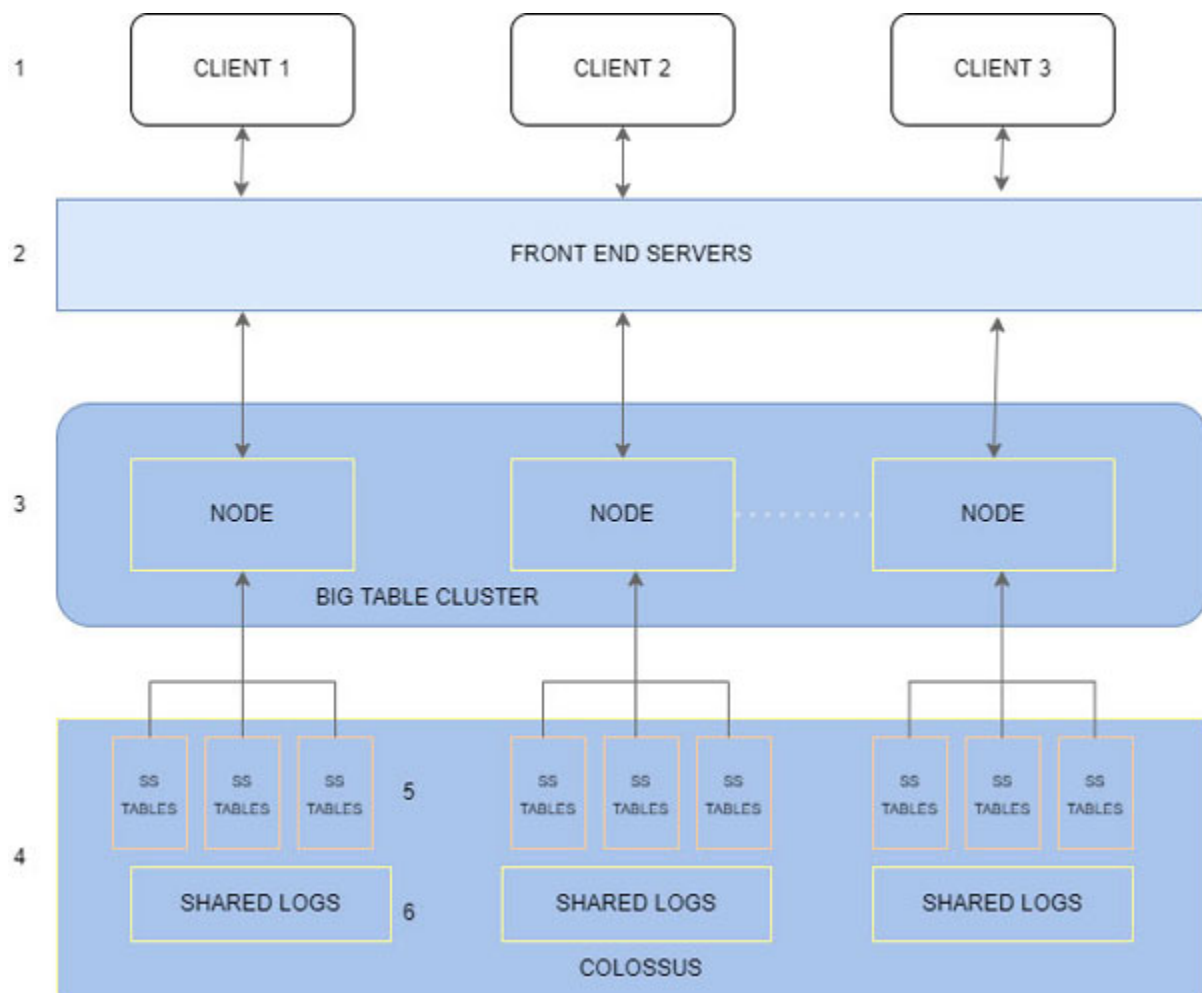
**Highly scalable:** It is challenging to scale HBase beyond a certain point; however, in Bigtable, the capacity increases linearly by adding nodes without restrictions.

**Simplified administration:** HBase does need a lot of administrative tasks, for which sometimes even downtime happens. With Bigtable, those admin tasks get simplified.

**Resizing without downtime:** It was impossible to add nodes without downtime, as new nodes needed repartitioning to be triggered again, and the HBase cluster was not available during that time. However, there are no such constraints when it comes to Bigtable.

Consider [Figure 10.2](#), which showcases the architecture of Google Bigtable:





**Figure 10.2:** Bigtable architecture

Clients **(1)** can submit user requests (read and write) to a pool of front-end servers **(2)**. These front-end servers direct the read/write requests to a cluster of nodes (Bigtable cluster) **(3)**. Scaling up and downscaling of Bigtable will primarily aim to increase or decrease the number of nodes in the cluster. A Bigtable table is sharded into chunks called tablets, and these tablets are saved in format SS Tables **(5)** on Google's File System Colossus **(4)**. Apart from writing data to SS tables, the data is stored in a shared log as soon as they are acknowledged by Bigtable, providing increased reliability **(6)**.

To create Bigtable, you must create instances. In these instances, you have to create a Bigtable cluster. Each

Bigtable cluster contains nodes; these nodes are the components that perform the read and write operations.

## **Scaling Bigtable options**

As seen in other chapters, Bigtable also supports autoscaling and manual scaling. In the case of autoscaling, GCP monitors a metric, and based on a threshold value, GCP automatically adds or removes infrastructure. Bigtable supports a new type of scaling, that is, Programmatic scaling. In programmatic scaling, we can use other metrics not supported out of the box, to be used for scaling up and down. None of these scaling requires a downtime of the Bigtable cluster.

### **Autoscaling triggers**

Bigtable identifies the number of nodes at any time based on the following 4 dimensions:

- CPU utilization target
- Storage utilization target
- Minimum number of nodes
- Maximum number of nodes

Each scaling dimension in Bigtable individually calculates the number of nodes, and Bigtable finally scales up to the highest calculated number. For example, if the current set up of Bigtable has five nodes, and as per target CPU utilization, the number of nodes required is 6, and as per storage, the number of nodes needed is 8, Bigtable will scale the cluster to 8.

As the number of nodes in the cluster increases, Bigtable automatically balances the storage in the cluster. During this time, the read and write requests continue as before,

without disruptions. The balancing is needed to ensure that the traffic is evenly distributed across nodes.

If your cluster has already reached the maximum number of configured nodes value, and the load on the system continues to increase, then the queries will run slow or might even start failing in the worst cases. Removal of a node from the cluster happens slowly when compared to scale up. This is done to reduce the latency of queries. It might happen that all nodes are serving query requests.

Let us have a closer look at the preceding mentioned four dimensions as well as the logic to determine a logical value for your application.

## **Autoscaling**

When you intend to create a Bigtable cluster, the first thing you must do is create instances. Once the instances are created, you can create your Bigtable cluster. While creating a Bigtable cluster, you get options to either select manual scaling or autoscaling. Consider [\*Figure 10.3\*](#), which is a snapshot of the GCP UI console:

## ← Create an instance

Scaling mode and configurations can be changed at any time.

### ☐ Manual allocation

Set your node count for fixed costs and compute resources.

Starti

### ☒ Autoscaling

Let Bigtable automatically add and remove nodes.

Minimum \*

1

Nodes

Maximum \*

3

Nodes

Bigtable adjusts the number of nodes within your minimum and maximum range. Because your cluster needs enough nodes to meet both your CPU utilization target and the fixed storage utilization target, Bigtable uses the number of nodes that accommodates the higher target.

The maximum node configuration cannot exceed 10x the minimum.

CPU utilization target \*

60

%

Target can range from 10%-80%

Your CPU utilization target depends on your workload requirements. For example, when autoscaling is enabled we recommend 60% for low latency workloads. [Learn more.](#)

Storage utilization target \*

2.5

TB per node (50%)

Target can range from 2.5 TB to 5 TB per node

For latency sensitive applications, we recommend configuring a storage utilization target that is lower than 60% of the per-node limit.

[Contact us](#) if you need to increase your maximum node quota.

✓ SHOW ENCRYPTION OPTIONS

**Figure 10.3:** Bigtable Autoscaling

As you can see in [Figure 10.3](#), you can select the scaling type while creating the cluster.

## When to Autoscale

Autoscaling brings multiple benefits:

- **Cloud Costs:** When a Bigtable cluster is configured with Autoscaling, Bigtable automatically reduces the number of nodes whenever possible. This helps in optimal utilization of infrastructure and hence more justified cloud costs. It is important because the primary use case that Bigtable is trying to solve, is the handling of data of the petabyte scale. Therefore, even a tiny cost improvement could eventually become a substantial amount.
- **Performance:** Bigtable automatically adds nodes when needed. Hence, when the load increases (number of reads and writes), Bigtable will scale up and provide the necessary infrastructure for serving the requests. Bigtable keeps checking the target CPU utilizations and storage requirements, and when stats go above the threshold, a scale-up happens.
- **Automation:** Autoscaling reduces the management of a cluster and its operations. If a cluster is configured for manual scaling, an admin person must update the number of nodes every time there is a need for faster queries.

Though Autoscaling is helpful in scenarios where there is a steady increase or decrease of the load on the system, in case the workload is busy, or there are very frequent up and down of infrastructure needed, Autoscaling might not be able to cater to frequency. The reason behind this is that in Bigtable, every time a new node is added, balancing data across nodes takes place and this is not a milli-second activity.

## **CPU utilization target**

CPU utilization is the percent usage of cluster CPU capacity. You can configure any value between 10% to 80%. Let us say you had configured the value to be 50%. When the CPU utilization goes above 50%, Bigtable will add more nodes to bring CPU utilization below 50%. Similarly, if the CPU usage is low, Bigtable will reduce the number of nodes.

The correct value of this parameter depends and varies from use case to use case. The decision of the right value depends on the throughput and latency needs of your application. For example, in the case of a real-time application, latency matters. On the other hand, in the case of batch processing applications, throughput matters the most. Generally, Bigtable offers optimal latency when the CPU load on the cluster is 70%. However, for latency-sensitive applications, you could set CPU utilization to be 50%. The lower the percent of CPU mentioned, the more is the number of nodes and higher the cost.

## **Storage utilization target**

Storage utilization is the number of terabytes consumed on the nodes before Bigtable starts scaling. This should be configured so that Bigtable is always ready to handle the fluctuations in the amount of data. The capacity limit for nodes is 5 TB per node for SSD storage and 16 TB per node for HDD storage.

For latency-sensitive applications, configure this value to be low, that is, approximately 60% utilization. Thus, if SSD is configured, value should be 3 TB. Similarly, in case of HDD storage utilization target becomes 9.6 terabytes.

## **Minimum number of nodes**

This property sets the minimum number of nodes a Bigtable can scale down to. The value must be greater than zero and cannot be less than 10% of the maximum number of nodes.

You can intend to keep this value to as low as a minimum, since that will imply the lowest consumption of nodes when the load is less. However, the actual state of the number of nodes is greatly affected by the preceding two properties of storage and CPU.

You can think of having a higher value for this property in the following example situations:

1. Because of the restriction that minimum nodes cannot be less than 10% of maximum nodes, if your use case needs a high maximum number of nodes for peak loads, you will have to configure an increased number of minimum nodes as well.
2. If your use case needs spiky read and write on Bigtable, Bigtable will scale up. However, scale up comes with a balancing process, which can take a few minutes. In this situation, you can take a conservative approach and have a higher minimum number of nodes to accommodate a sudden high load of reading and write on Bigtable.

## **Maximum number of nodes**

The maximum number of nodes is the value you want your Bigtable cluster nodes to scale up to. It has a value greater than 0, can never be less than the minimum configured nodes, and have a maximum value equal to 10 times the minimum number of nodes configured. To configure this value, identify the maximum amount of data you plan to store on Bigtable.

Suppose you want to configure SSD for saving data. The default value of storage utilization for SSD is 2.5 TB. If your requirements need 20 TB to be saved, divide 20 by 2.5, which equals to 8. It would be best if you chose a value in multiples of 8. A higher multiple will tackle the latency sensitiveness of your use case.

You can enable or disable autoscaling on an already created Bigtable instance. You can also update the dimensions, that is, CPU utilization target, minimum number of nodes, and maximum number of nodes for a cluster. Let us have a look at the GCloud commands to accomplish the operations.

- The following GCloud command enables autoscaling on your Bigtable instance:

```
gcloud bigtable clusters update CLUSTER_ID \
  --instance=INSTANCE_ID \
  --autoscaling-max-nodes=MAX_NODES \
  --autoscaling-min-nodes=MIN_NODES \
  --autoscaling-cpu-target=CPU_TARGET \
  --autoscaling-storage-target=STORAGE_TARGET
```

- The following command disables autoscaling:

```
gcloud bigtable clusters update CLUSTER_ID \
  --instance=INSTANCE_ID \
  --num-nodes=NUM_NODES --disable-autoscaling
```

- The following command updates the autoscaling configuration:

```
gcloud bigtable clusters update CLUSTER_ID \
  --instance=INSTANCE_ID \
  [--autoscaling-max-nodes=AUTOSCALING_MAX_NODES] \
  [--autoscaling-min-nodes=AUTOSCALING_MIN_NODES] \
  [--autoscaling-cpu-target=AUTOSCALING_CPU_TARGET] \
  [--autoscaling-storage-
target=AUTOSCALING_STORAGE_TARGET]
```

In the three commands shared above, **CLUSTER\_ID** is the cluster of your Bigtable, which is a permanent identifier for your cluster. **INSTANCE\_ID** is the identifier for your instances. **MAX\_NODES** and **MIN\_NODES** are the maximum and the minimum number of nodes for the Bigtable instance. **CPU\_TARGET** is the percent CPU utilization target for your Bigtable instance and can have a value between 1 to 80%. **STORAGE\_TARGET** is the storage utilization target in GiB per node.



## Manual node allocation

You can also choose an option to scale up and down manually. The decision of when to scale up or down depends on someone manually assessing the correct cluster size. Calculating the numbers should be like strategies defined in the autoscaling section. Refer to the following code:

```
gcloud bigtable clusters update CLUSTER_ID \  
  --instance=INSTANCE_ID \  
  --num-nodes=NUM_NODES
```

**NUM\_NODES** is the number of nodes you want your Bigtable to scale up/down up to.

## Programmatically Autoscaling

You can also scale up and down the Bigtable cluster programmatically. Bigtable pushes many metrics to Monitoring APIs, and you can write a code to read those metrics and scale up or down the Bigtable instance. A complete list of metrics can be accessed here.

**[https://cloud.google.com/monitoring/api/metrics\\_gcp](https://cloud.google.com/monitoring/api/metrics_gcp)**  
**- gcp-bigtable**

Consider one of the metrics: **bigtable.googleapis.com/server/latencies**. It gives the server versus the latency distribution. Another critical metric **bigtable.googleapis.com/cluster/cpu\_load** represents the CPU's cluster load.

Refer to the folder Bigtable in the repo attached with the project. It contains a java class **BigtableScalar.java**, demonstrating this scaling based on metric **bigtable.googleapis.com/cluster/cpu\_load**.

The whole idea demonstrated is as follows:

1. Fetch the Monitoring metrics for the last 10 minutes using the google-cloud-monitoring library and filter the values with filter **metric.type=**

`bigtable.googleapis.com/cluster/cpu_load`. This will return the values for all the nodes in the Bigtable instance, and we will pick and choose the largest value of CU utilization percent.

2. Check if the value of CPU utilization percent is above 70%, and then add one node to the cluster. If the CPU percent is lower than 50%, remove one node from the cluster. These operations are performed by using the library - **Bigtable-client-core**.
3. Keep running this process in a runnable thread.

## **Limitations of Autoscaling**

You looked into how to define autoscaling of the Bigtable cluster. However, there are limitations associated with the process, as discussed in the following points:

- **Delays and latency due to load balancing**

When you add or remove the number of nodes, Bigtable tries to rebalance the data across the nodes, which can go up to 20 minutes before seeing a performance improvement. If the workloads involve short bursts, autoscaling will not affect the performance immediately. However, because of this long delay, the sudden burst of load activity will be over, until Bigtable is ready infrastructurally to handle the load.

To tackle this, in the case of autoscaling, you can keep the system configured with the excess minimum number of nodes. With programmatic and manual scaling, it is vital to identify when there will be a high load and increase the number of nodes proactively so that system is ready when the actual need arises.

- **Schema design issues**

If the schema is not well defined, it could lead to multiple read and write requests simultaneously on the

same node. This will result in no performance improvements even if the Bigtable cluster is scaled, because the requests are distributed across the newly added infrastructure.

## **Conclusion**

Bigtable is a managed HBase datastore, backed by Google's file system Colossus, which handles and manages columnar data store use cases. GCP abstracts the complete infrastructure management and admin tasks from the user, and gives a more uncomplicated and painless way to handle huge amounts of petabytes of data. Google Bigtable supports multiple ways to scale the number of nodes in the cluster. These scaling strategies do not need downtime but rather, rebalancing the data phase requires some time before the actual benefits of scaling can be leveraged.

## **Points to remember**

- Bigtable is a Google Managed HBase setup that could be easily set up and scaled.
- Although GCP improves most of the pain points of HBase in terms of management and scaling of infrastructure, due to its inherent nature, the scale strategies are not reflected in real-time.
- It is essential to define the schema of the data well enough. Otherwise, the benefits of scaling could not be leveraged due to the hot spotting of user queries.
- Bigtable supports autoscaling and programmatic scaling. It is always recommended to have your infrastructure already scaled up even if it is not needed, especially if your application is latency sensitive. The impact of adding infrastructure takes time to show performance improvements.

## **Questions**

1. Bigtable saves data in the form of key-value pairs. The maximum size of the value is 10 MB. True or False?
2. What could be the minimum number of nodes for a Bigtable setup whose maximum value is configured as 100 nodes?
3. Bigtable scaling is real-time. You scale up the infrastructure, and the impact is immediate improvement of queries. True or False?
4. Scaling up or down needs downtime in Bigtable. True or False?

## **Answers**

1. True. The maximum size of the value you can save in Bigtable is 10 MB.
2. Minimum number of nodes cannot be less than 10% of maximum number of nodes, Hence, for maximum node 100, you must set a value of 10 or higher for Minimum nodes.
3. Scaling is not real time. It can take up to 20 minutes to leverage true benefit of scaling due to the rebalancing phase.
4. No. Bigtable does not require downtime for scaling.

## CHAPTER 11

# Configuring Cloud Spanner for Scale

## Introduction

**Google Cloud Spanner** is Google's mission-critical scalable relational database service, that is fully managed globally. It supports traditional relational database semantics like schemas, ACID transactions, and SQL constructs. Google Spanner supports automatic and synchronous replication across regions of availability.

Before Spanner, no solution in the market could support all the preceding features without downtime. For its use cases like AdWords and Google Play, Google developed cloud Spanner and has used Spanner successfully to handle such high volumes of data for over five years. These use cases (AdWords and Google Play) need to support millions of users throughout the day, without being down for even a minute, as that would imply a loss of revenue. Spanner is horizontally scalable to hundreds of machines, without the user needing any downtime (Spanner is 99.999% up). Hence, it has become one of the most opted for choices, to handle relational data in GCP.

## Structure

In this chapter, we will discuss the following topics:

- Nature of workload
- Cloud spanner infrastructural footprint
- Manual scaling
- Autoscaling using Autoscaler
  - Autoscaler architecture
  - Autoscaler deployment topology
- Scaling strategies as per load
  - Stepwise scale up
  - Linear scale up
  - Direct scale up

## Objectives

After studying this chapter, you will be able to understand the type of workloads Spanner is used for and how Spanner scores above the rest of the competing technologies. You will also look into the infrastructural footprint of Spanner and see what happens under the hood when we increase or decrease the number of nodes. After that, we will dive deep into how to scale – manual and auto-scaling and finally, learn of the scaling up strategies used in the industry like stepwise scale-up and linear scale-up.

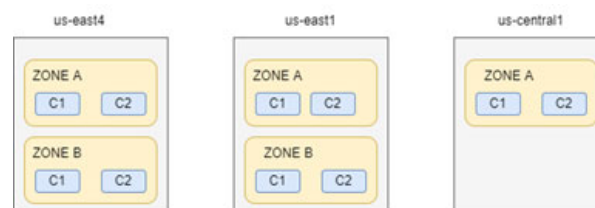
## **Nature of workload**

Cloud Spanner is a fully managed, globally scalable, and mission-critical database service. Google started building this service for a few use cases of ad server and *Google Play*. Spanner can be used for scalable online transaction processing databases, where you cannot afford your system to be down.

Claims at various portals say Google Spanner has handled 2 billion requests per second at its peak, making it best suited for most of the applications within Google, and outside as well. The Spanner has reportedly been used to hand massive datasets in banks, gaming industries, retail industries, and eCommerce. Three words are key for selecting this database: Relational datasets, Google managed, and horizontally scalable without downtimes. All the types of the preceding use cases mentioned, need these aspects in place so that they can work 24\*7 without loss of revenue, due to infrastructure operations.

You not only get the RDBMS semantics, but Spanner also provides a key feature of horizontal scalability where the number of instances can scale to huge numbers without any degradation in the performance of queries. You need to define the size of the infrastructure, and Spanner automatically distributes your data across the infrastructure equally, resulting in better query processing.

Data saved in Spanner is safe, as Spanner provides automated replication across regions. What this means is that, if you deal with data that you do not want to lose under any circumstances, Spanner could be used. Spanner automatically creates multiple replicas across different geographical locations. For multi-region configurations, there will be 5 replicas of created and distributed across 3 regions by Cloud Spanner. Two regions will hold two copies of data and the third region will have one of the copies (fifth one), also known as witness replica. Consider the following [Figure 11.1](#), which showcases an example of data 3 regions and 5 copies:



**Figure 11.1:** Multi regional Spanner

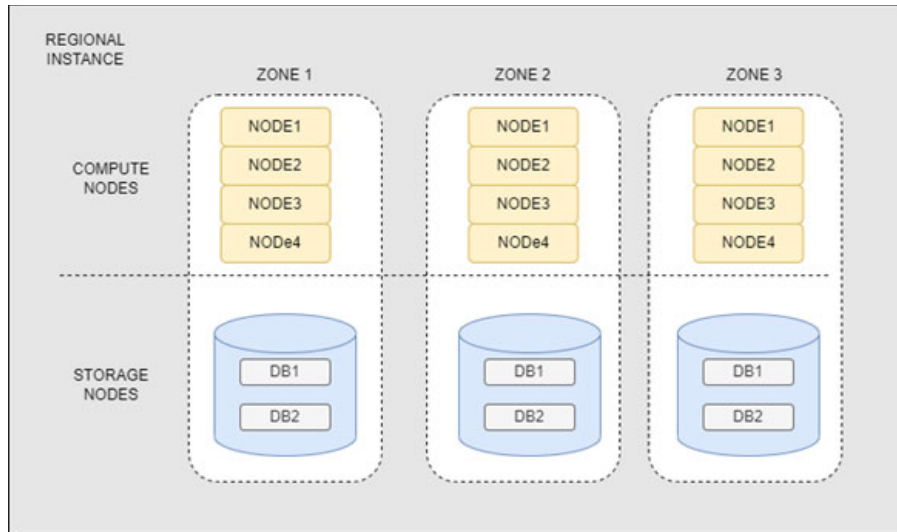
It has 2 replicas each in **us-east4** and **us-east1** regions and a witness replica in **us-central1** region. The database C1 has 2 replicas in different zone in **us-east1** and **us-east4**, and one last witness chunk in one zone in us-central1.

The witness replica (**us-central1**) does not store the complete copy of the data. Their responsibility is to participate in the voting to commit a write. Consider the scenario where the database needs to survive a regional failure and the database has application traffic from two regions. If you do not have the third region, then you will not be able to construct a majority quorum that could survive the loss of either region. Typically, the witness replica region will be close to one of the two read-write regions, such that a majority quorum of the witness and the close by region, can be used to quickly commit the writes.

Until now, you would have sensed that Spanner is for supporting a massive volume of relational data. One of the most critical aspects of the Relational Data model, is finding support for **Atomicity, Consistency, Isolation, and Durability (ACID)** properties. Spanner supports all four aspects of ACID properties well. It promotes robust consistency options, which keeps your data in sync across multiple geographical locations. Spanner point-in-time-recovery enables protection against accidental deletion or writes. For example, due to any operation if the database gets corrupted or deleted, with **Point-In-Time Recovery (PITR)**, it could rollback to any point of time in last 1 hour. You can extend one-hour up to 7 days by configuring the property `version_retention_period`. PITR allows you to recover a portion of database as well as entire databases. PITR results in increased storage utilization, increased CPU utilization and increased time to perform schema updates.

## **Cloud Spanner infrastructural footprint**

In Cloud Spanner, the data is saved on Google-backed distributed file system *Colossus*, and the read and write of data happens via nodes. Consider [Figure 11.2](#), which represents a regional instance of Google Cloud Spanner:



**Figure 11.2:** Spanner infrastructure footprints

In the preceding figure, the whole setup represents a Spanner Instance of a regional type. You can see the nodes spread across three zones. Each zone has three nodes represented as Node1, Node 2, Node 3, and Node 4. The Spanner instances contain two databases, DB1 and DB2, replicated across regional zones.

A node is a measure of computing in Spanner, which is meant to serve the read and write queries. Node servers do not store data. When we talk about scaling Spanner, we are talking about adding more nodes to the Spanner instances. Nodes are responsible for reading and writing data in their zone. It stores the data in the Google-backed distributed file system Colossus. This is key to scaling Spanner instances because the data is not linked to a node. If there is a failure in the node or the whole zone fails, the responsibility of serving data is assigned to a new set of nodes, and hence no downtime happens in Spanner.

Colossus's storage solution has its own support for the high availability of data and is not tied to any zone. Replicating data under the hood results in no data loss even when the whole region goes down. Data serving might get impacted, but data loss will not happen. Spanner partitions the rows of a database into multiple splits; a split holds a range of contiguous rows, where the rows are ordered by primary key. These splits are stored and replicated using Paxos algorithm, where one replica set is elected as a leader. All the writes happen via the leader replica while any replica which is configured as read-write or read-only can serve the read requests.

Spanner supports 3 kinds of replication: read-write, read-only and witness replicas (refer to [Figure 11.1](#)). Single region Spanner deployment has all replicas marked as read-write. Apart from witness replicas, all the replicas store data. However, witness replica does not store data - it only participates in leader election.



The following [Table 11.1](#) summarizes the preceding 3 replica types:

Replica type	Can vote	Can become leader	Can serve reads
Read-write	Yes	Yes	Yes
Read-only	No	No	Yes
Witness	Yes	No	No

**Table 11.1:** Replica types

## **Manual scaling**

You can manually scale up and down your Cloud Spanner instance after creating the instances. While we do not have any restrictions on scale up, however, while scaling down, there could be situations when scaling down is not allowed. Two such situations are as follows:

- You cannot store more than 4 TB of data per node (1000 processing units). If scaling down results in this quantity of data expanding (more than 4 TB per node), then the operation of scaling down is not permitted.
- Based on the usage patterns, Spanner has created multiple splits, and Spanner is unable to merge the splits in case of scaling down.

For the second scenario, there are various ways of scaling down, so that the Spanner gets the best chance of merging splits. One such strategy is to not scale down in one go but instead scale down progressively until you reach the most optimal scale-down number. Trigger a scale down, let the system perform merge, and trigger another scale down once complete. The wait between those two scales actions could be of the order of weeks.

Another critical aspect to keep in mind while scaling down is that, you have to monitor the latencies of queries via cloud monitoring and not let the CPU usage go above 65% in the case of regional instances, and 45% for each region in case of multi region deployment. Query times are expected to increase with a decrease in the number of nodes.

When not performing queries, Cloud Spanner performs background work such as optimizing splits and protecting data. The compute capacity in Spanner is a dedicated resource, and even when we are not running queries, the above-mentioned background tasks continue to run.

You can update the number of nodes in three ways:

- **UI console:** You can edit the instance and give a new value for the number of nodes or processing units. This can be seen in [Figure 11.3](#), where the user will have to fill up the highlighted section with appropriate values and then save:

← Edit instance

Instance ID: instance1  
Configuration: europe-west1

To move this instance to a different regional or multi-region configuration, [contact Google](#).

**Update name**  
You can rename your instance at any time. While the ID is permanent, the name is for display purposes only.

Instance name \*: instance1  
Name must be 4-30 characters long

**Allocate compute capacity**  
Your compute capacity determines the amount of data throughput, queries per second (QPS), and storage limits in your instance. One node equals 1,000 processing units. Affects billing.

Unit \*: Nodes  
Quantity \*: 1  
Enter an integer of 1 or greater

✓ COMPUTE CAPACITY GUIDANCE

SAVE CANCEL

**Figure 11.3:** Spanner scaling UI console

- Another way is via *Gcloud command*:

```
gcloud spanner instances update INSTANCE_ID --nodes=NUMBER_OF_NODES
```

In the preceding command, **INSTANCE\_ID** is the unique identifier for the Spanner instance and **NUMBER\_OF\_NODES** is the value for the number of nodes you want the Spanner to scale up or down to.

- The third way to perform modification of number of nodes, is via client libraries, that are available in 8 programming languages.

## **Autoscaling using Autoscaler**

When we want to scale up and down the number of processing units of cloud Spanner based on the workload, we can look up Autoscaler. Autoscale is a companion tool that is not available as managed service but is open source and can be configured and used easily for tackling loads on cloud Spanner. Note that not all performance problems could be solved by increasing the number of nodes in Spanner. For example, the hot spotting of user requests could not be improved no matter how many nodes we scale our system. Autoscale does not support such issues. Using Autoscaler, you can only add or remove nodes based on some metric of computing and storage.

Autoscale keeps monitoring your instances and automatically adds or removes nodes or processing units to ensure that CPU consumption and storage of data per node remains within recommended limits. CPU utilization recommendations are not to let your CPU consumption go above 65% in the case of regional deployments and 45% for each region in the case of multi-region deployment. Similarly, for storage, the recommendation is to set the

threshold to 75% of the maximum storage per node, multiplied by the number of nodes.

## **Autoscalar Architecture**

The auto scale deployment consists of Cloud Scheduler, two Pub/Sub topics, two cloud functions, and a Fire store. The utilization metrics for computing and storage of Spanner instances are taken via the Cloud Monitoring API by Autoscalar, and then scaling up and down happens. Let us have a look at each of the components.

### **Cloud scheduler**

The Cloud scheduler runs at a configured frequency to read the metrics coming via Cloud Monitoring API. It also compares the received values with the configured thresholds. Cloud scheduler can perform actions concurrently for one Spanner instance or multiple Spanner instances.

### **Poller cloud function**

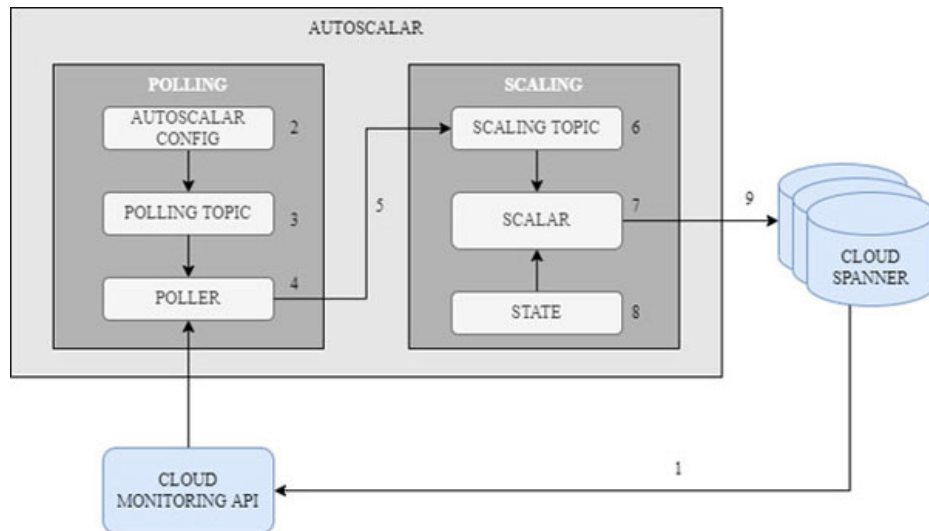
Poller cloud function collects and processes time series metrics data from all configured instances of Cloud Spanner. Poller preprocesses the data coming from Cloud Monitoring API and identifies the most relevant data points that will help the Cloud function take action.

### **Scaler Cloud Function**

The Scaler function processes the data points pulled by the Poller Cloud function and then calculates the number of nodes/processing units to be configured for the Spanner instance. It compares the metric value with the threshold values to determine whether to scale up or down the infrastructure.

### **End to end working**

In this section, you will investigate how different components operate, to establish an act of Spanner instance scaling end to end. Consider [\*Figure 11.4\*](#):



**Figure 11.4:** Autoscaler working

Follow the numerical labelling in the figure with the corresponding explanation as follows:

1. Cloud Spanner instance continuously pushes metrics to Cloud Monitoring APIs. For complete details of metrics exposed, visit the page - <https://cloud.google.com/spanner/docs/monitoring-cloud>
2. You must configure the schedule, time, and frequency of your autoscaling setup in the cloud scheduler. Cloud scheduler pushes a JSON message with Autoscaler configuration for one of the Spanner instances into the Polling Topic of Pub/Sub.
3. Polling Topic is the Pub/Sub topic that receives the JSON message from the scheduler. Each JSON message is an instruction for Poller to perform some action related to one of the instances.
4. Poller is a cloud function that gets triggered when a new message arrives on the topic. This function reads the metrics from the cloud monitoring API in a time window.
5. Poller pushes one message for each Spanner instance into the Scaling Pub/Sub topic. This message contains the metrics and configuration parameters related to a specific Spanner instance.
6. The Scalar topic is the Pub/Sub topic, where each message pushed by Poller will result in a trigger of the scalar function. A scalar function performs the actual scaling of the Spanner instance.
7. Scalar gets triggered once for each message in the Scalar topic. The scalar function receives the current state (from the scalar topic) of the metrics, compares them against the configured threshold values, and comes up with a new value for the number of nodes.
8. The scalar function pulls the time when the instance was last scaled from the fire store and compares it with the current time to identify whether

scaling up or down is allowed based on cool-down periods.

9. If there is no conflict with the cool-down period, Scalar performs the scaling action on the Cloud Spanner instance.

In the next section, we will discuss the deployment topologies of Autoscaler. Deployment topology simply implies the team that will manage the components of Autoscaler deployment. Let us consider that we have two teams, the Engineering team that writes business logic and uses a Spanner instance, and the IT team that manages infrastructure. If it was decided that everything will be handled by IT team, then the deployment topology will be different when both teams hold the responsibility of partially managing the components.

## **Autoscaler deployment topology**

There are three deployment topologies used by teams that use Spanner. This decision depends on how many of the IT/infra team and engineering teams have decided to bear the responsibilities of managing the Autoscaler deployment, since it is not available out of the box. Ideally speaking, one Autoscaler deployment for all your Spanner needs is the way forward. Most preferred among all three strategies is the distributed one, as it shares management responsibility in the central team and individual engineering teams, and provides the benefit of having different Autoscaler configurations for each Spanner instance. This brings us to the three possible strategies, as explored in the following.

### **Deployment of Autoscaler per project**

This is a recommended topology for teams that want to manage separate Autoscaler configuration and infrastructure. This is a good starting point to introduce the Autoscaler use case in the project. The advantage of this approach is that each team has freedom while selecting Autoscaler parameters. The disadvantage is the management overhead of maintaining individual Autoscalars.

### **Centralized deployment topology**

Autoscale is deployed in one project. This Autoscaler can scale the Spanner instances in different projects. This deployment is suited for a team managing the configuration and infrastructure of several Autoscalers in a central place.

The advantage of this approach is that teams do not need to manage the Autoscaler. However, the teams have to adhere to one centralized configuration of Autoscaler.

### **Distributed deployment**

All the components of the Autoscaler remain in the same project, except the cloud scheduler. This deployment is a hybrid deployment where teams managing Spanner do not want to maintain the Autoscaler, but want to have their Autoscaler configurations.

This intends to be mid-way between the preceding two approaches. The advantage is that teams can have different configurations of Autoscaler as teams manage some parts, while the rest of the portion resides with the centralized team.

Deployment of all three is out of the scope of the book. The deployment steps are straightforward and are present in the GitHub repository of Autoscaler. Consider [table 11.2](#), for looking at installation/set up details of each strategy:

Deployment Strategy	Set Up Link
Per Project Deployment	<a href="https://github.com/cloudspannerecosystem/autoscaler/blob/master/terraform/per-project/README.md#before-you-begin">https://github.com/cloudspannerecosystem/autoscaler/blob/master/terraform/per-project/README.md#before-you-begin</a>
Centralized Deployment	<a href="https://github.com/cloudspannerecosystem/autoscaler/blob/master/terraform/per-project/README.md#before-you-begin">https://github.com/cloudspannerecosystem/autoscaler/blob/master/terraform/per-project/README.md#before-you-begin</a>
Distributed Deployment	<a href="https://github.com/cloudspannerecosystem/autoscaler/blob/master/terraform/per-project/README.md#before-you-begin">https://github.com/cloudspannerecosystem/autoscaler/blob/master/terraform/per-project/README.md#before-you-begin</a>

**Table 11.2:** Installation guides

After the finalization and deployment of Autoscaler, you are expected to configure its parameters. To do so, follow these steps:

1. Go to the Cloud Scheduler console page.
2. Select the checkbox next to the name of the job created by the Autoscaler deployment, and tick on poll-main-instance-metrics.
3. Click on **Edit** on the top bar.
4. Modify the Autoscaler parameters shown in the job payload. The following code is an example of the same:

```
[
  {
    "projectId": "scaling-gcp",
    "instanceId": "instance-1",
    "scalerPubSubTopic": "projects/scaling-gcp/topics/spanner-scaling",
    "units": "NODES",
    "minSize": 2,
    "maxSize": 8
```

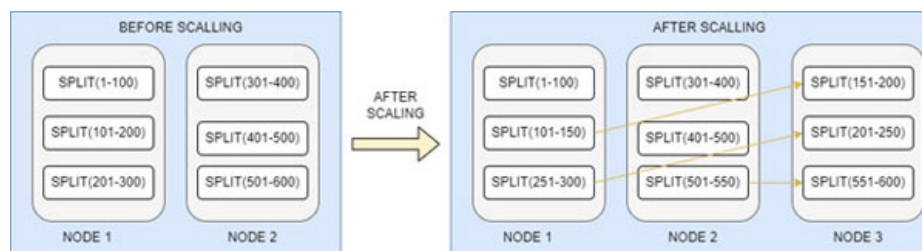
```

}, {
  "projectId": "scaling-gcp",
  "instanceId": "instance-2",
  "scalerPubSubTopic": "projects/scaling-gcp/topics/spanner-scaling",
  "units": "PROCESSING_UNITS",
  "minSize": 500,
  "maxSize": 3000,
  "scalingMethod": "DIRECT"
}
]

```

## Scaling strategies as per load

You looked into the technicalities of scaling the infrastructure based on two scaling strategies: manual and auto. This section will explore what happens when a new node is added and how Spanner redistributes the responsibility of serving the load. This rebalancing of responsibilities takes time, and due to this, scaling down is not a real-time activity, that is, the moment you scale up, you can start leveraging the benefits of scale in queries. It takes some time to become effective. Consider [Figure 11.5](#), to understand the rebalancing of responsibility phase:



**Figure 11.5:** Rebalancing Serving

In [Figure 11.5](#), the **BEFORE SCALING** phase represents two nodes in the Spanner instance, where each node is responsible for serving specific rows. Node 1 serves rows between 1 and 300, and node 2 serves rows between 301 and 600.

When the number of nodes is increased by one more (Node 3), Spanner identifies the nodes under high workload and distributes the responsibility to another node. See the **AFTER SCALING** section of the figure. Rows 151-200 and 201 to 250 from Node 1 are moved to be served on Node 3. Similarly, a portion of rows from Node 2 is also shifted.

To support these scenarios, there are three strategies based on which you should scale the number of nodes.

## Stepwise scale up

Consider the situation of a use case where the application has small high frequency spikes in workloads. Since we know scaling in Spanner takes some time to show impact, till the time we spike for the small load, the load decreases. To tackle this, increasing the number of nodes is recommended instead of trying to decrease the node for these frequent small spikes.

In this strategy, when the threshold is crossed, this method scales up or down nodes/processing units, using a fixed configurable number. For example, increasing the number of nodes by two will result in adding two nodes.

## **Linear scale up**

You can select a linear scale up and down strategy for patterns where load grows gradually and no frequent ups and down are seen. The number of nodes to be added depends on the load on the system. If two nodes are needed to tackle the workload, two nodes will be added. This requirement of two nodes might be 3 or 4 in the next scale-up cycle.

To calculate the new size, the following formula is used:

$$newSize = currentSize * currentUtilization / utilizationThreshold$$

## **Direct scale up**

This strategy could be applied in workloads, where we know the workload pattern. For example, if, over the past month, the number of nodes required at 11 AM was ten and at 1.30 was 12, a similar workload will also be assumed for today. Scale up to 10 nodes will start at 10:50 AM, assuming an increase in nodes will take 10 minutes to show impact. Similarly, at 11:20 AM, scale up to 12 nodes will happen. This is known as Direct scale-up because the workload is increased on experience rather than metrics of the Spanner Instance.

## **Conclusion**

For handling a massive amount of relational data, consider using Spanner. There are ways in the industry to achieve it with other technologies as well, but the Spanner is Google managed, highly available (99.999% availability), distributed, and provides features like ACID transactions, strong consistency, and replication.

Spanner is battle tested for production workloads, and with mature autoscaling in place, it is one of the best tools to handle mission-critical relational workloads.

## **Points to remember**

- Spanner is built and tested to handle critical relational workloads in your enterprise.



- The Spanner has mature scaling strategies – manual and auto – that are widely used in the industry.
- Scaling in Spanner takes time to show its true impact, as there is a rebalance phase associated with each scale up and down.
- Spanner supports autoscaling using an open-source Autoscaler, which requires deployment and management.
- There are multiple strategies recommended to deploy Autoscaler. For example, deploy one Autoscaler for each Spanner instance, and deploy on Autoscaler for all instances across various projects.

## **Multiple choice questions**

1. Which of the following is not valid for Google Spanner?
  - a. Spanner is horizontally scalable.
  - b. Spanner ensures 99.999% availability.
  - c. Spanner scaling is real-time. You add infrastructure, and it immediately improves the system.
  - d. Autoscaling of Spanner is supported out of the box by GCP.
2. What are the qualities of datasets that result in selecting Google Cloud Spanners as the database for an application?
3. Google Spanner uses which storage solution to store data?

## **Answers**

1. c and d are not true. Scaling nodes take some time to show impact. Autoscaling is not supported by default. You have to install and configure Autoscaler.
2. Highly scalable, huge, relational datasets support millions of concurrent queries.
3. Google distributed file system – Colossus.

# CHAPTER 12

## Scaling Google Composer 2

### Introduction

**Apache Airflow** is a tool for authoring, scheduling, and monitoring data pipelines programmatically. It was created at Airbnb in the year 2014 and was bought by Apache software foundation in March 2016. In the year 2019, Airflow was announced to be the Top-Level Apache Project and is now considered a leading orchestration tool by public cloud providers. Google Composer is managed to offer Airflow (both 1 and 2), where the complete management of Airflow setup is abstracted from the IT teams, and the responsibility of managing them is taken over by GCP. Like GCP Composer, AWS offers Amazon Managed Workflows with Apache Airflow. The key reason why Airflow has gained so much recognition is its proven functionality for data pipelining, extensible framework, scalability, and large vibrant community.

### Structure

In this chapter, we will discuss the following topics:

- Introduction to Composer
- Options for horizontal scaling
  - Adjusting minimum and maximum number of workers
  - Adjusting number of schedulers
- Options for vertical scaling

- Adjusting worker, scheduler, and web server scale and performance parameters
- Adjusting environment size
- Composer Autoscaling
  - Role of Airflow worker set controller
  - Factors affecting environment scaling
  - Composer Autoscalars
- Optimizing the Airflow environment
- Observe the environment

## Objectives

After studying this chapter, you will have a good understanding of Airflow architecture and its infrastructural footprint on GCP. You will also be able to understand the concepts of how Airflow enables vertical and horizontal scaling. We will then learn about the factors that affect scaling, and the different horizontal scaling levels available. In the end, we will investigate how to optimize the Airflow environment for high throughput in cost-optimal manner.

## Introduction to Composer

Airflow lets you build and execute workflows. A workflow is represented as a **Directed Acyclic Graph (DAG)**, that contains individual pieces of work known as tasks and the dependencies among tasks to create an execution plan also known as DAG.

A DAG defines a task, specifies the ordering/dependencies among tasks, and runs retries. Consider [\*Figure 12.1\*](#) which represents a very simple DAG that prints log statements:



**Figure 12.1:** Sample DAG

The following code is the DAG definition of the preceding DAG:

```
1 from datetime import datetime
2 from airflow import DAG
3 from airflow.operators.dummy_operator import DummyOperator
4 from airflow.operators.python_operator import PythonOperator
5 from airflow.operators.bash_operator import BashOperator
6
7 def print_function():
8     return 'Upload the data.'
9
10 dag = DAG('Scaling_GCP_DAG', description='Hello World DAG',
11     schedule_interval='0 12 * * *', start_date=datetime(2017,
12     3, 20), catchup=False)
13
14 extract_demo_operator =
15     BashOperator(task_id='Extract_Data', bash_command="echo
16     'Perform extract of data.'", dag=dag)
17
18 filter_demo_operator = BashOperator(task_id='Filter_Data',
19     bash_command="echo 'Perform filtering of data'", dag=dag)
20
21 upload_demo_operator = PythonOperator(task_id='Upload',
22     python_callable=print_function, dag=dag)
23
24 extract_demo_operator>>filter_demo_operator>>upload_demo_operato
```

In the code of the preceding sample DAG python file:

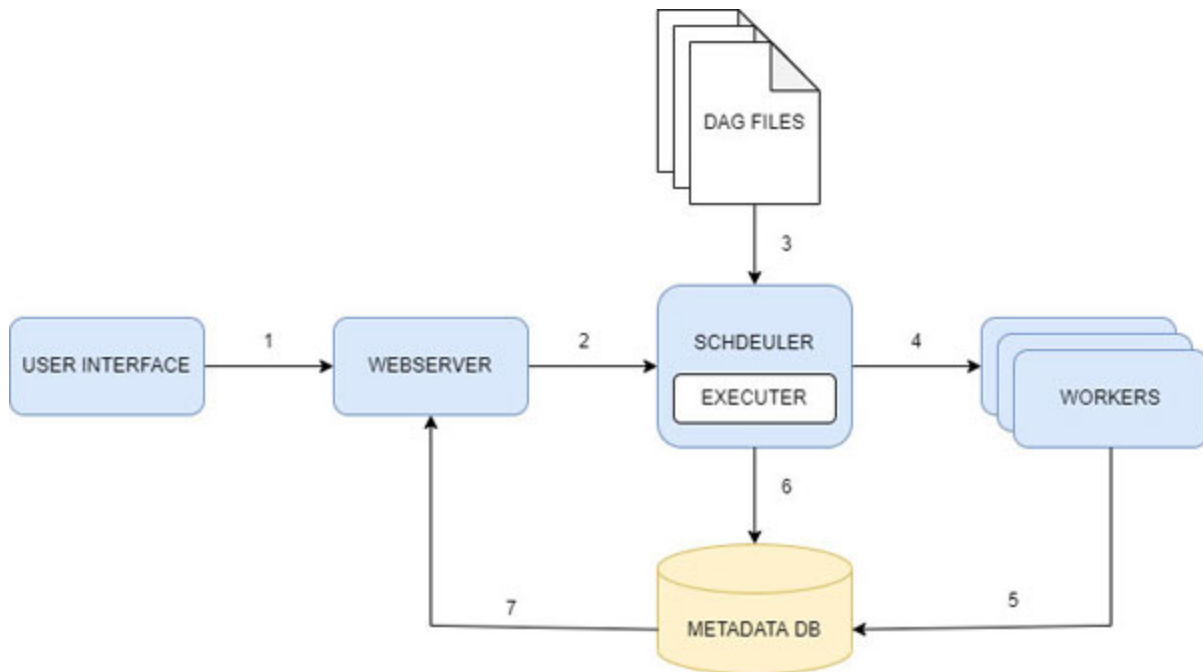
- Line numbers 1 to 5 is import statements of modules.

- Line numbers 7 and 8 is a simple function written in python and called from a python task.
- Line number 10 and 11 represents a DAG object. Here you can define DAG level properties like name, frequency, and start date.
- Line numbers 13 to 15 represent the various tasks in the DAG.
- Line number 17 represents the order of execution of tasks.

Airflow installations consist of the following components:

- A scheduler is a which register/de-registers a DAG and triggers execution of tasks.
- DAGs folder contains DAG definition files; this folder is periodically ready by the scheduler, and the DAG is either added, removed, or updated based on the files in the DAG folder.
- An executor manages a running task, and can push and manage the work on worker nodes.
- A web server component, which provides the REST API to interact with Airflow. A *metadata database* is used by the preceding components like webserver, scheduler, and executer, to maintain state.

Consider [Figure 12.2](#), which shows the preceding components in action:



**Figure 12.2:** Airflow architecture

Follow the numerical labelling of the preceding architecture with the following explanation. A user interface interacts with the web server to populate the user interface for the end-user. A user can submit a:

1. User request for DAG execution.
2. The web server submits the request to execute a DAG to the scheduler. The executor is a component inside the scheduler, that takes care of managing the workers who are going to perform the task.
3. The scheduler keeps scanning the DAGs directory to register, modify or delete a DAG. When a user uploads a python file representing a DAG in the DAG folder, the scheduler scans the folder and gets the DAG registered to the UI, by putting an entry in metadata DB.
4. The request for processing is launched on worker nodes.
5. Worker nodes keep updating the metadata database with the latest happenings on worker nodes.

6. The scheduler also logs the complete activity happening, to the metadata database.
7. Users can get all the happening on the scheduler and worker via a query by the web server on the metadata database.

Google Composer is managed offering on GCP. Let us see how each Airflow architectural component map to GCP offerings. Refer to [Table 12.1](#):

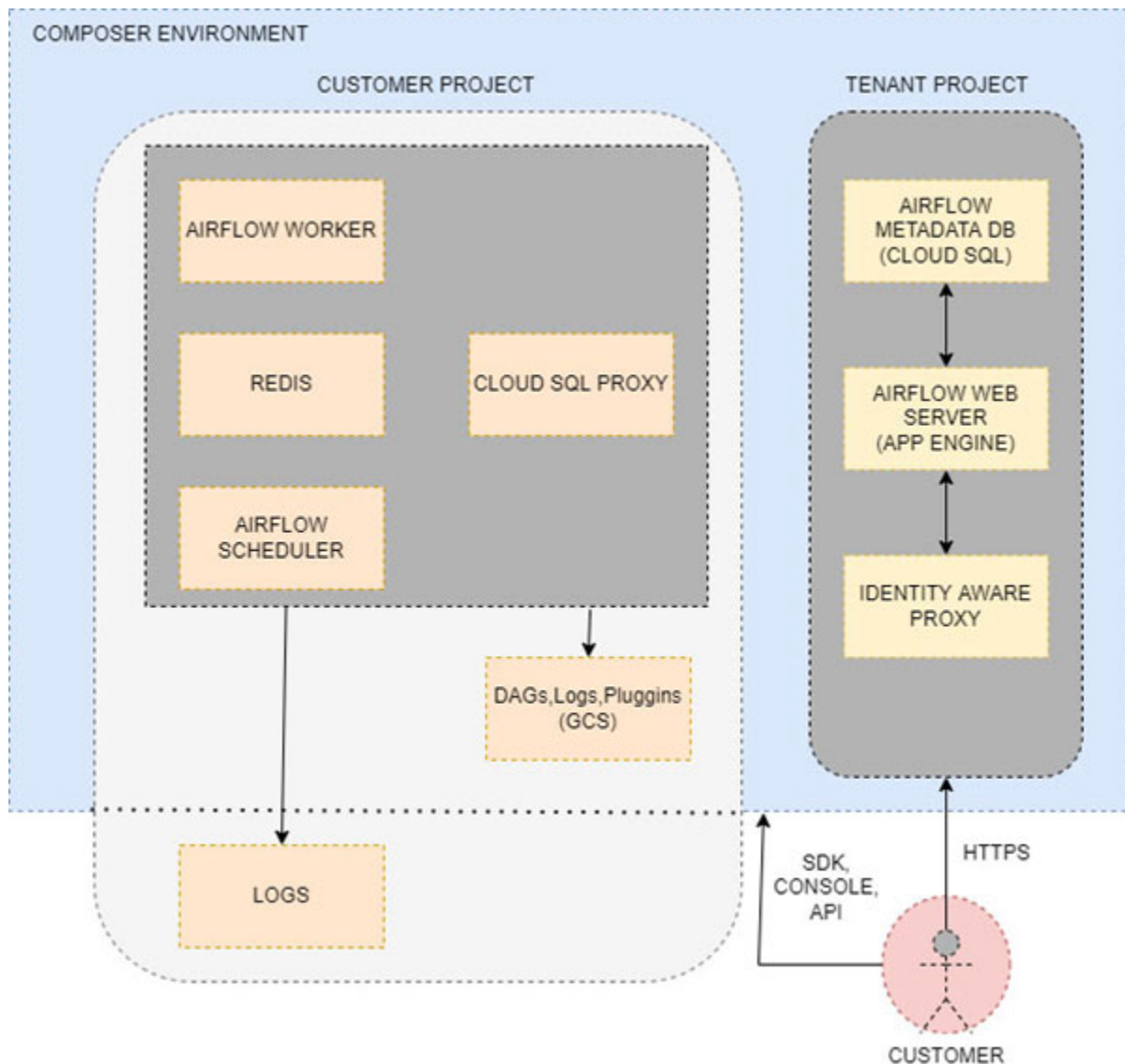
Airflow Components	GCP components
Webserver and Airflow UI	App Engine flex.
Scheduler	Pods on Kubernetes cluster
Worker	Pods on Kubernetes cluster
Metadata DB	Cloud SQL
DAG directory	GCS bucket. The GCS bucket is mounted in Airflow using GCS FUSE.

**Table 12.1:** Airflow components to GCP components mapping

If you carefully observe, all the technology under GCP components has well-defined scaling features. When the workload increases in a Composer environment, there is need to increase the number of workers. Since Composer uses Kubernetes to deploy Airflow, scaling means creating more worker Pods in the Kubernetes cluster.

Composer distributes the preceding environment resources in GCP managed tenant project and customer project. Customer project is a Google Cloud Project where your environment is created. You can create multiple Composer environment in single project. Tenant project is a Google managed (out of your control) project, which provides access control and data security for your environment. Each Composer environment will have its own tenant project.

Consider the following [Figure 12.3](#), which displays environment architecture of Composer when a public IP is exposed to end user. There are other strategies as well, for example Composer with private IP, where Composer environment architecture varies slightly.



**Figure 12.3:** Airflow environment architecture on GCP

As you can see in the preceding figure, all the components of Composer, such as Airflow workers, Redis queue, Airflow scheduler, Logs and so on, reside within the customer project. Tenant project just holds the components, which



ensures safe access to Composer resources via Identity Aware proxy.

## **Options for horizontal scaling**

In this section, you will investigate the various manual scaling options provided by the Composer. You can scale a Composer environment horizontally, by just adding a greater number of components, as well as vertically, by increasing the size of already hosted components. In Horizontal scaling of the Composer environment, the following options are available.

## **Adjusting minimum and maximum number of workers**

You can configure the maximum and the minimum number of worker nodes for the Composer environment. Under no circumstances, will the Composer environment breach the min-max configurations.

There are multiple ways of modifying and applying these configurations, such as GCP console, REST APIs, and GCloud. In our case, we will look into GCloud options.

The command to configure the maximum and the minimum number of worker nodes (worker Pods) on a Composer cluster, is given as follows:

```
gcloud composer environments update scaling-gcp-composer \
  --location us-central1 \
  --min-workers 5 \
  --max-workers 10
```

The preceding command will update Composer environment (**scaling-gcp-composer**) to have a minimum of 5 worker nodes and a maximum of 10 worker nodes.

## **Adjusting number of schedulers**

The Composer environment can run more than one scheduler at a time. In Composer 1, this was not the case, and you can only configure one scheduler, which was the major bottleneck in scaling up a Composer environment. Configuring multiple schedulers to distribute and balance the load between multiple schedulers results in better performance and reliability. Moreover, if one scheduler goes down, the Composer continues to work.

Just increasing the number of schedulers might not result in better performance. A single scheduler might provide better performance in situations where the extra scheduler has not been used, and just consumes resources. The scheduler performance depends on the number of DAGs, the number of workers, and the number of tasks that run in the environment. It is recommended to start with two, as just setting to one, might result in a single point of failure. You can configure the number of schedulers when needed, by using the following command:

```
gcloud composer environments update scaling-gcp-composer \
  --location us-central1 \
  --scheduler-count 2
```

The preceding cloud command will update the number of schedulers to 2 in the Composer environment **scaling-gcp-composer**.

## **Options for vertical scaling**

In the preceding section, you looked into horizontal scaling, that is, increasing the number of workers and schedulers, without modifying the class of infrastructure. However, Composer gives you the facility to modify the existing class of infrastructure as well.

## Adjusting worker, scheduler, web server scale and performance parameters

You can tune attributes (CPU, memory, and storage) related to infrastructure components of worker, scheduler and web server.

Use the following GCloud command to modify the parameters:

```
gcloud composer environments update ENVIRONMENT_NAME \
--location LOCATION \
--scheduler-cpu SCHEDULER_CPU \
--scheduler-memory SCHEDULER_MEMORY \
--scheduler-storage SCHEDULER_STORAGE \
--web-server-cpu WEB_SERVER_CPU \
--web-server-memory WEB_SERVER_MEMORY \
--web-server-storage WEB_SERVER_STORAGE \
--worker-cpu WORKER_CPU \
--worker-memory WORKER_MEMORY \
--worker-storage WORKER_STORAGE
```

An example of the preceding command is as follows:

```
gcloud composer environments update scaling-gcp-composer \
--location us-central1 \
--scheduler-cpu 0.5 \
--scheduler-memory 2.5GB \
--scheduler-storage 2GB \
--web-server-cpu 1 \
--web-server-memory 2.5GB \
--web-server-storage 2GB \
--worker-cpu 1 \
--worker-memory 2GB \
--worker-storage 2GB
```

In the preceding command, the value of disk and memory needs to be a number along with unit. You can visit the

monitoring page of your Composer environment to see if any of the preceding parameters needs modification, and tune the attributes accordingly.

## **Adjusting environment size**

In Composer, we have the concept of environment, which is nothing but a standard collection of infrastructure resources, pre-created by the GCP team. Composer contains multiple infra components and these Environments are a good starting point to start your Composer deployments. Composer came up with an option to select an environment at the time of creation, as well as environments that can be updated from one to another, as per need. Over the period of time, the environment drifts from these standard default configurations. As each use case vary in nature of workloads, engineering teams monitoring dashboard of Composer fine tune the property values of the attributes.

The following command cloud is to be used to update the Environment type of a Composer environment:

```
gcloud composer environments update scaling-gcp-composer \  
  --location us-central1 \  
  --environment-size medium
```

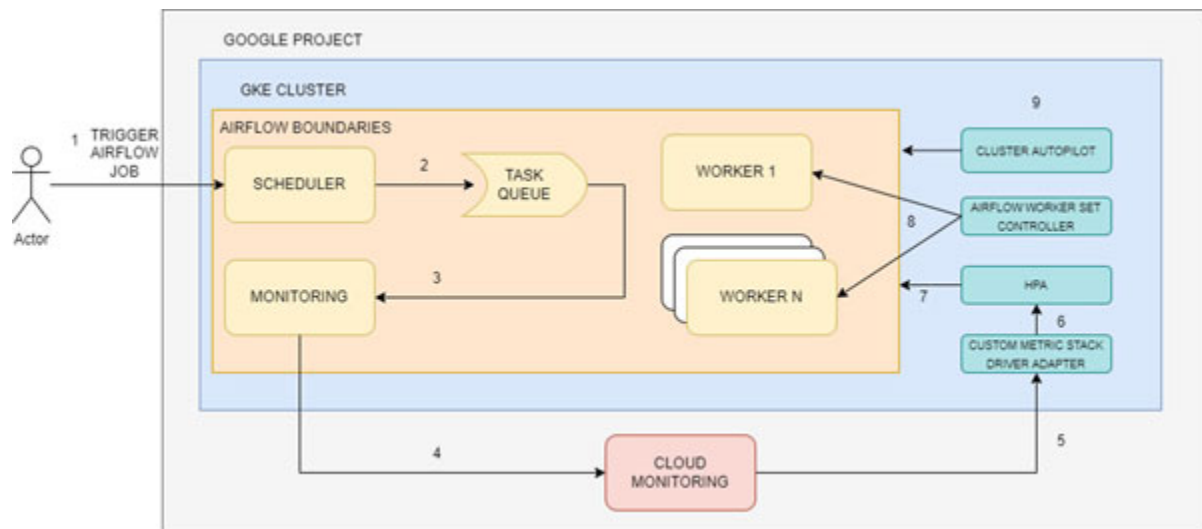
## **Composer Autoscaling**

In Composer 2, some of the biggest benefits are the efficient autoscaling features, where the infrastructure scales up/down based on executed DAGs and tasks.

- The Composer environment automatically increases the number of worker nodes in case it experiences a heavy load.
- The Composer identifies if the workers are not being used, and if so, scales them down.

- You can configure the minimum and the maximum number of workers, Composer scales up/down the workers within defined limits.

For appreciating the factors affecting autoscaling, it is vital to understand the autoscaling architecture of Airflow. Consider [Figure 12.4](#), which demonstrates the autoscaling architecture of Composer:



**Figure 12.4:** Auto scaling Composer

In the preceding figure, the grey box represents the Google cloud project, where the whole Composer is deployed. The blue box represents the boundaries of the Kubernetes cluster on which Airflow components (orange box) and supporting autoscaling components (green box) are deployed. The red box at the bottom represents the cloud monitoring API that lies within the GCP cluster.

Follow the preceding numerical labelling with their respective explanation as follows:

1. A client (a user triggering a job from a UI console or a system component triggering a request via REST API) submits an Airflow job to the Airflow scheduler.
2. The scheduler will add the job submission request to a task queue.

3. The monitoring component of Airflow reads the number of jobs (size of the queue) in the task queue, that are still waiting to be processed.
4. Airflow monitoring then exports and pushes the scale factor metrics (`composer.googleapis.com/environment/worker/scale_factor_target`) to the Cloud Monitoring API.
5. The metric in Cloud monitoring is read via the Stack Driver Adapter.
6. The stack driver Adapter provides the metric to the Horizontal Pod scalar, and the horizontal Pod scalar calculates the number of worker Pods based on the scale factor metric.
7. HPA instructs the Kubernetes cluster to adhere to the number of worker node configurations.
8. In Kubernetes, you have the concept of the controller, which maintains the state of the cluster based on configurations. In the last step, the Horizontal Pods scalar sets the new number of workers in the cluster. The default controller is overridden in the case of the Composer by Airflow Worker Set Controller, and it ensures that the number of worker nodes adheres to the instruction by HPA.

Default controllers do not take into consideration if a task is running on a Pod before removing them. In such a case of scale down, removing a worker where a task is running will be harmful. We will do a deeper dive into these scenarios in the next section on *“Role of Airflow worker Set”*.

9. Kubernetes clusters are created in autopilot mode, hence the increase in the number of virtual machines in the Kubernetes cluster will be taken care of by the Cluster Autopilot component.

## **Role of Airflow worker set controller**

In the case of scaling up, new workers are added, and this means that there is no disruption for any of the running tasks. However, at the time of scaling down, there could be a few tasks which are being read from the task queue and processing has started. In middle of processing, if the worker nodes are removed, the affected tasks will have to start again, resulting into more time being taken to process. There could also be a few tasks which might take lot of time on adhoc basis, for example, a task triggers a REST API or interacts with a database, and due to some reason if REST API or DB retrieval is slow, your task will take more time than normal. The Scale Factor metric does not consider the long execution time of tasks, and hence scaling down that is just based on the scaling factor. might result in a few tasks starving. Because the worker nodes have already been occupied by the tasks, if we remove a worker where tasks are already running, you will have to start from scratch.

By default, Kubernetes components, in case of scaling down, do not look inside the happenings inside Pods. To tackle this scale-down condition, instead of the default deployment controller, the Composer introduced a custom deployment controller – Airflow Worker Set Controller.

Kubernetes controllers are processes that hold the responsibility to maintain the state of the Kubernetes cluster. For example, if there are 4 Pods in the cluster and based on workload, and 3 Pods are needed, the controller will make sure that one Pod is deleted.

Airflow Worker Set Controller before downscaling, connects with the metadata DB of the Airflow, to identify which are those workers where no tasks are running and removes only those workers. Because of this, there could be a situation where the number of worker running are high than what the

autoscaling Airflow parameters intend to run. However, all the workers will be executing some tasks.

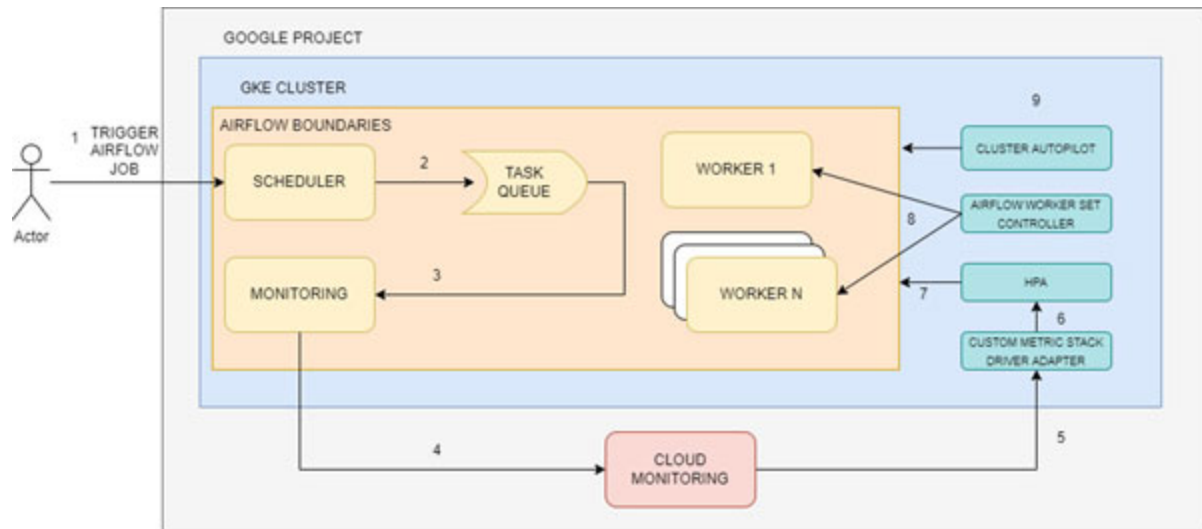
## **Factors affecting Composer autoscaling**

The number of workers is identified, based on the scaling factor target metric and this factor is calculated based on the following factors:

- Current number of workers
- Number of tasks in the queue not assigned to any worker
- Number of idle workers
- `celery.worker.concurrency` Airflow configuration

Among the aforementioned factors, `celery.worker.concurrency` is not discussed till now. Consider [\*Figure 12.5\*](#), which shows the high-level working of a worker node:





**Figure 12.5:** Worker machine

An Airflow worker can launch multiple celery processes (3 in the preceding diagram), configured by setting the property `celery.worker.concurrency`, where each celery process picks up a task and triggers tasks. A trigger task will read the DAG file from metadata DB and then perform actual processing.

## Composer Autoscalars

In the whole architecture of Composer autoscaling, three Autoscalars are involved. These are as follows:

- **Horizontal Pod Autoscaler (HPA)**
- Cluster Autoscaler

- Node auto-provisioning

In this section, we will see the role played by each one of them in detail. For the in-depth analysis of these 3 scalars, refer to [Chapter 6, Scaling Kubernetes](#).

## **Horizontal Pod scalar**

Horizontal Pods scalar instructs the Kubernetes cluster, on which the Composer setup is running, to scale up or down to a certain number of workers. Once the signal reaches the Airflow Worker Set Controller, the system is scaled up/down accordingly.

## **Cluster Autoscaler**

Composer 2 deploys components on the Kubernetes cluster and the Kubernetes cluster has a configured Cluster Autoscaler in place. Cluster Autoscaler automatically resizes the number of nodes in a node pool, based on workload. The only requirement is to set the maximum and the minimum number of nodes; the rest is automatic.

In the case of autoscaling, resources are deleted or moved when autoscaling a cluster. The Composer is designed to have rerun associated with tasks, meaning in case of those disruptions, the task will be automatically recreated on a different worker.

## **Node auto provisioning**

**Node auto provisioning** automatically manages a set of node pools on the user's behalf. Without node auto provisioning, GKE starts new nodes only from user-created node pools. With node auto-provisioning, new node pools are created and deleted automatically.

With Autopilot clusters, you do not need to worry about provisioning nodes or managing node pools. This is because

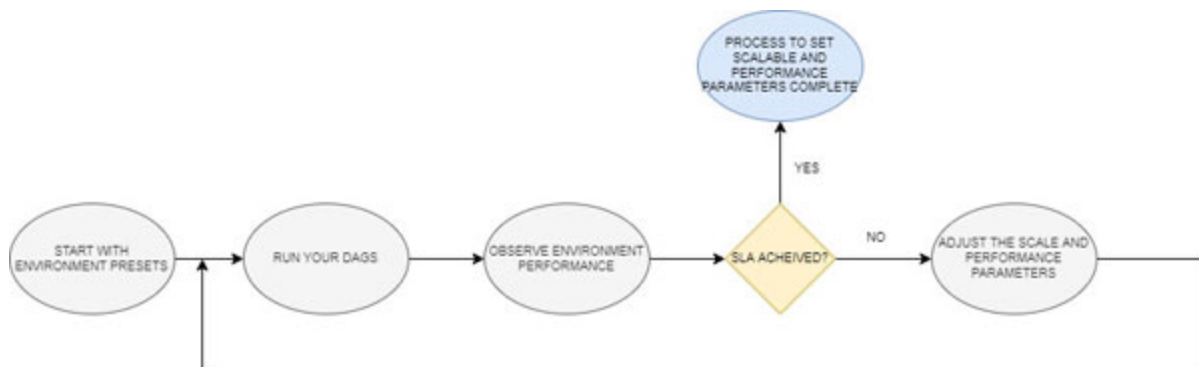
node pools are *automatically* provisioned through node auto-provisioning, and are automatically scaled to meet the requirements of your workloads.

Cloud Composer configures these Autoscaler in the environment's cluster. This automatically scales the number of nodes in the cluster, the machine type, and the number of workers.

## Optimizing the Airflow environment

Along with autoscaling, you can also control the performance and scale parameters of the environment by configuring the right memory, CPU, and disk for web servers, workers, and schedulers. If such components (web servers, workers, and scheduler) are under-scaled, the overall scale-up capacity of the Composer environment will be compromised. If over-provisioned, they will incur high costs even when not in use.

There is no silver bullet to achieving a perfect number at the start. Generally, engineering teams follow the given set of steps to finalize a value. Consider [Figure 12.6](#):



**Figure 12.6:** Optimizing Airflow process cycle

## Start with environment pre-set

There are three environment pre-sets made available by the Composer – small, medium, and large. The guidelines to

start with one of them are also given by GCP. [Table 12.2](#) can be used as a good starting point:

Recommended pre-set	Number of DAGs	Max concurrent DAGs	Max concurrent tasks
Small	50	15	18
Medium	250	60	100
Large	1000	250	400

**Table 12.2:** *Composer pre-set configuration capabilities*

These recommendations are taken from official GCP documentation.

## **Run your DAGs**

Upload all your DAGs to the DAGs directory (a GCS location). This directory is mounted as a local directory inside the Composer environment using GCS fuse.

Try to replicate the execution of DAGs very similar to your actual real-world scenario.

## **Observe the environment**

GCP Composer has a comprehensive monitoring page that gives in-depth insights into various attributes.

### **Monitoring the scheduler CPU and memory**

Observe the dashboard in the scheduler section of the monitoring page. Look at the two graphs available – Total schedulers CPU usage and total schedulers memory usage.

- If the CPU usage is consistently below 25-30%, reduce the number of schedulers or reduce the CPU of the scheduler.

- On the contrary, if the CPU usage is consistently above 80%, you can increase the number and CPU of the scheduler.

In the case of high CPU usage of the scheduler, you can also think of increasing the DAG file parsing interval and increasing the DAG directory listing interval. The CPU cycles used in parsing and registering the DAG are reduced and hence, the scheduler will have more CPU cycles to perform processing.

### **Monitoring total parse time of DAGs**

When a new DAG is uploaded to Airflow, it must be first parsed and registered by Airflow/Composer before use can schedule it.

- If the time taken to parse the DAG is very high, it means that the scheduler is busy doing tasks. This could be improved by increasing the number of schedulers and CPU on a scheduler.
- It is important to keep the total artifacts to the lowest minimum in the Airflow DAGs directory. For example, keep only the DAGs and not the jars that might be used in an Airflow job. If you have a use case of jar, put it in a different directory.

### **Monitoring worker Pod evictions**

Generally, Pod eviction happens in the case where the resources in the Pods have hit the upper limit. For example, the major issue behind Pod eviction is out of memory error. You can interpret this by looking at the monitoring graph worker Pod evictions.

In the preceding situation, it is recommended to either increase the memory available to each worker or reduce worker concurrency. By reducing worker concurrency, a

smaller number of tasks will launch at each worker, and worker resources will be divided among fewer tasks, resulting in more infrastructure allocated per task.

### **Monitoring active workers**

Observe the graph, number of active workers, and number of tasks in the queue.

If the environment is reaching the max number of workers and the number of tasks in the queue is continuously high, then you can increase the number of maximum number of worker nodes.

- If there are long inter tasks scheduling delays, and the Composer is not scaling up, it implies that there is some setting which is not letting the Composer scale up.
- These settings could be low worker concurrency, low DAG concurrency, and low value for maximum active runs per DAG.

### **Monitoring workers CPU and memory usage**

Observe graphs for the CPU and memory usage by Airflow workers: Total workers CPU usage and total workers memory usage.

- Increase the worker memory if their worker memory consumption is consistently high or the number of Pods is evicting at a high rate. On the contrary, if the consumption is very low, it decreases the memory.
- If the worker CPU usage is high, increase the number of worker nodes and reduce the number of tasks that can run concurrently on each worker. This will enable more CPU to get allocated to each task.

### **Monitoring running and queued tasks**

Observe the graph for the number of running tasks and the number of queued tasks. If the number of queued tasks is not coming down, it can be reduced by:

- Increasing the value for the maximum number of workers. Spin up more workers so that more tasks could be pulled from the queue.
- Another factor would be to look into, if and only if workers appear to be free, is to increase the worker concurrency. It means that the existing workers can host more, but they are unable to do the property, and increasing the value will result in better worker utilization.

### **Monitoring the database CPU and memory usage**

Observe graphs for the CPU and memory usage by the Airflow database: Database CPU usage and database memory usage.

- If database CPU usage is above 80%, scale-up is needed.
- In the database CPU usage is below 20%, scale down is needed.

You can scale up and down by increasing the environment size, that is, from small to medium or medium to large.

### **Monitoring the task scheduling latency**

If the inter tasks latency has increased to unacceptable limits (10 seconds or more), it simply means that either the scheduler is overloaded or is not allowed to process fast. Another reason could be that there are not enough workers available to execute the scheduled jobs. This situation could be handled by:

- Increasing scheduler CPU and memory.
- Increasing worker concurrency, increasing DAG concurrency, or increasing max active runs per DAG.
- Increasing the maximum number of workers.

## **Monitoring web server CPU and memory**

If the Airflow UI performance or the response time of Airflow REST APIs has increased, it is due to load on the webserver. You can increase the number of CPU and memory of the web server.

## **Commands to perform the preceding changes**

- The following command modifies the number of schedulers:  

```
gcloud composer environments update scaling-gcp-composer \
  --scheduler-count=2
```
- The following command modifies the CPU and memory of Schedulers:  

```
gcloud composer environments update scaling-gcp-composer \
  --scheduler-cpu=0.5 \
  --scheduler-memory=3.75
```
- The following command modifies the maximum number of workers:  

```
gcloud composer environments update scaling-gcp-composer \
  --max-workers=6
```
- The following command modifies the worker CPU and memory:  

```
gcloud composer environments update scaling-gcp-composer \
  --worker-memory=3.75 \
  --worker-cpu=2
```



- The following command modifies the Web server configurations:

```
gcloud composer environments update scaling-gcp-composer \
  --web-server-cpu=2 \
  --web-server-memory=3.75
```

- The following command modifies the environment type:

```
gcloud composer environments update scaling-gcp-composer \
  --environment-size=medium
```

- The following command modifies the Airflow configuration variables:

```
gcloud composer environments update scaling-gcp-composer \
  --location us-central1 \
  --update-airflow-configs=key1-value1,key2-value2
```

The preceding command (text in bold) accepts key-value pair, where the key is Airflow configuration keywords, and the values are custom configurations. [Table 12.3](#) lists such keys and their meaning:

Airflow Property	Purpose
<code>min_file_process_interval</code>	Minimum time taken for parsing each file. A low value means that the scheduler will have to complete this in less time, and hence there is pressure on the scheduler. Minimum value is 30 seconds.
<code>dag_dir_list_interval</code>	Time elapsed between two cycles of scanning the DAG directory to identify new and modified DAGs. This is down by scheduler. Hence, lowering the value will result in reducing load on scheduler. Minimum value is 30 seconds.
<code>worker_concurrency</code>	Number of celery processes per Airflow worker. The default value is equal to <code>12 * worker_CPU</code> for your environment.
<code>max_active_tasks_per_dag</code>	DAG concurrency defines the maximum number of task instances allowed to run concurrently in each DAG.
<code>max_active_runs_per_dag</code>	This attribute defines the maximum number of active DAG runs per DAG.

**Table 12.3:** Key Airflow environment properties

## **Conclusion**

Google Composer is a managed Airflow offering from GCP. Composer support Airflow 2.x is known as Composer 2. Composer exhibits a mature scaling framework to scale up and down without impacting or disrupting the already running tasks. The Composer comes up with a pre-configured environment, which acts as a good starting point to start scaling. Engineering teams are expected to do performance runs to fine-tune Composer properties. Vertical scaling is also possible in Composers.

## **Points to remember**

- GCP Composer is managed offering of Airflow 2.x.
- You can spin up a Composer environment with a set of values that could be updated at any point in time.
- Kubernetes being the underlying infrastructure of Composer provides a robust autoscaling mechanism, by use of Autoscaler like Horizontal Pod scaling, Cluster auto-scaling, and node auto-provisioning.
- There are a few components that are not horizontally scalable, and hence fine-tuning becomes important to leverage the true autoscaling power of Composer. These are the scheduler, web server, and metadata database. These components do not auto-scale, so you have to provide them with the maximum autoscaling in mind for a use case.

## **Questions**

1. When the setup of the Composer was done, the inter-task time was very less. Over the period, it has

drastically increased. What could be the possible reasons?

2. During scaling down the number of worker Pods, deletion of Pods which is running a job can happen and that can lead to a re-run of the whole task from scratch. Yes or No?
3. What will you suggest doing, if the Composer cluster is not overloaded but there are a lot of unprocessed tasks in the queue?

## **Answers**

1. This could happen due to the scheduler being occupied with something else. One common reason for this behavior is an increase in the number of DAG files over the period.
2. No, it is not possible. Airflow Worker Set Controller does not allow a worker running a task to get killed.
3. Refer to the section “Monitor running and queued tasks” under the heading “Optimizing the Airflow environment”.

# CHAPTER 13

## Scaling Google Dataproc

### Introduction

Google Dataproc is a managed offering from **Google Cloud Platform (GCP)**, to run Hadoop and Spark jobs. In 2003, Google released a white paper on the Google File System. In the year 2004, they released another paper on processing framework *MapReduce*. These papers resulted in the release of Hadoop in the year 2006. Hadoop solved the problem of processing huge volumes of data using a cluster of machines (commodity hardware). Hadoop was later improved by Spark with its first release in the year 2014. Though there were differences in the way data is handled inside Hadoop and Spark, both were compatible to run on a **Yet Another Resource Negotiator (YARN)** cluster.

Hadoop and Spark were market leaders in the Big Data processing industry and that has resulted in almost all data-driven enterprises using them. When these enterprises started their cloud journey, it was important for a cloud provider to support Hadoop and Spark jobs on the cloud as well. That is why all public cloud providers have support for running these workloads. AWS has *Elastic MapReduce*, Azure has *Databricks* and GCP has *Dataproc*, to enable the running of Hadoop and Spark jobs on the cloud.

In the on-premises versions of Bigdata (Hadoop and Spark jobs), the infrastructure was fixed, which mostly resulted in either your pipelines starving for infra or having excessive infra. However, the overall cost of maintaining the cluster was the same. In the cloud pay-as-you-go model, it becomes important to use optimal quantity of infrastructure. Till now, whenever we tried to achieve it, we always talked about Autoscaling of infrastructure. In this chapter as well, you will see how to scale up and down your Dataproc cluster, to achieve justified cost and high throughput execution of Bigdata workflows.

## **Structure**

In this chapter, we will discuss the following topics:

- Introduction to Dataproc
- Manual scaling
  - Using the GCloud command-line tool in the GCloud CLI
  - Edit the cluster configuration in the Google Cloud Console
  - Use the REST API
- Auto scaling
  - Autoscaling deep dive
  - Introducing Autoscaling policies API
  - CRUD on Autoscaling policies
  - Applying Autoscaling policies to Dataproc cluster
- Limitations of scale
- Graceful decommissioning of clusters
- Using preemptible VMs to scale

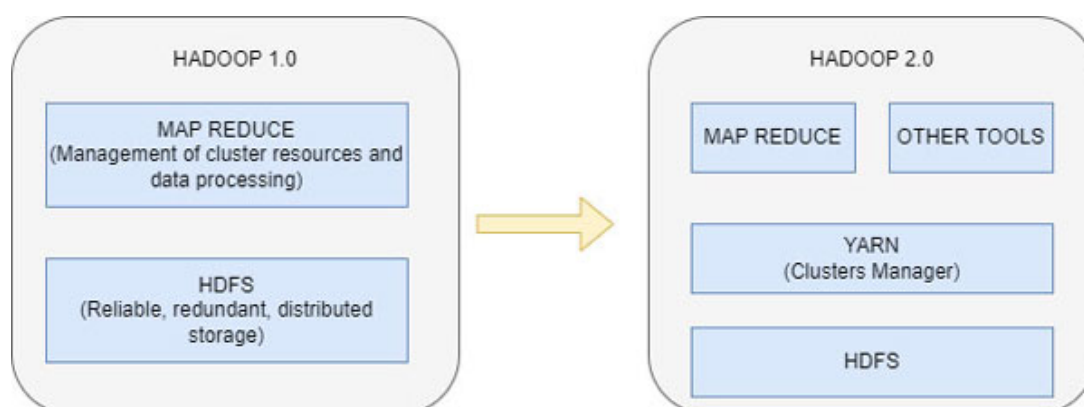
## **Objectives**

After studying this chapter, you will understand how you can scale your Hadoop/Spark/Dataproc cluster to meet the changing needs of the workload. You will see the manual approach first and eventually dive deep into auto scaling concepts of Dataproc. You will learn about the various precautions to be taken, for graceful commission and de-commissioning of nodes in the Dataproc cluster. In the last section, you will see ways to reduce costs while scaling up by using preemptible virtual machines as secondary virtual machines.

## **Introduction to Dataproc**

Hadoop jobs run on a cluster of commodity hardware machines, where the fault-tolerant workloads produce reliable results, that is, even if a node in your cluster goes down, the Hadoop framework makes sure that the failed tasks are re-executed, and its results

are incorporated into the result. While Hadoop processing capabilities saw a lot of improvements over the years, so was the case when it comes to improvement in the Hadoop jobs run on clusters. When it all started, Hadoop was using a cluster manager Hadoop v1, which advanced over years to Hadoop 2.0 or YARN architecture. Consider [Figure 13.1](#):

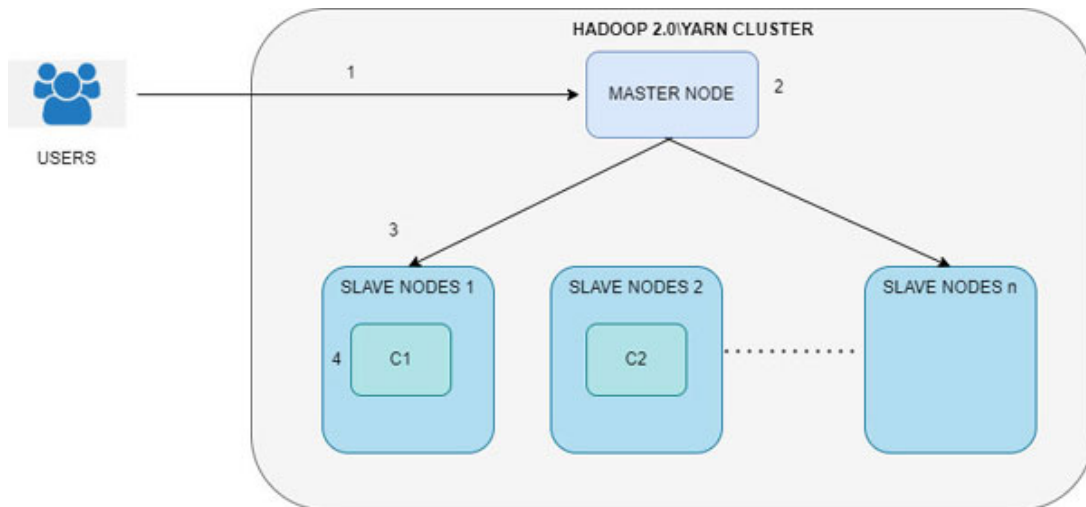


**Figure 13.1:** Hadoop 1.0 vs Hadoop 2.0

In Hadoop 1.0 architecture, the responsibility to manage the cluster and run the data processing job was tied. However, in Hadoop 2.0, this tight coupling was broken down and the complete cluster management responsibility was taken away by the YARN cluster manager. While in Hadoop 1.0, a user could only write Hadoop jobs and run them on a cluster. However, in the case of Hadoop 2.0, the YARN cluster manager supported not just MapReduce, but also other applications.

Apache Spark became one of the most famous tools, that not only provided similar capabilities but also several other benefits on top of Hadoop, such as in-memory processing and better capability to handle a wide variety of use cases, such as that of machine learning and graph processing.

Let us have a very high-level overview of what a cluster means in these cases. Consider [Figure 13.2](#):



**Figure 13.2:** Job submission on a cluster

Follow the numerical labelling with the explanation as follows.

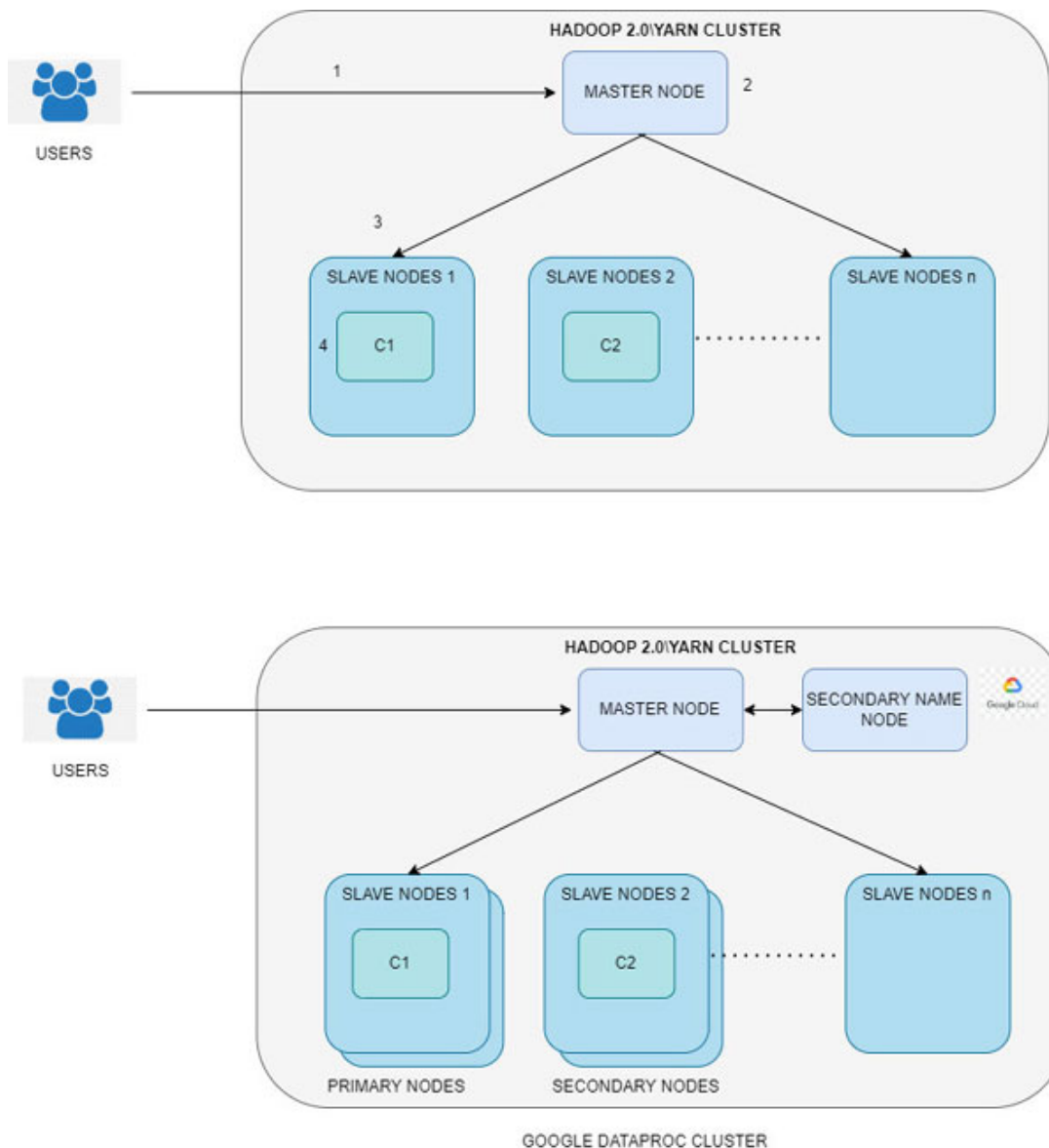
1. A user/client submits a data processing request to a Hadoop 2.0/YARN cluster.
2. The YARN cluster follows the master-slave architecture. The master manages the complete execution of a data processing job and makes sure that the data is processed reliably on slave nodes. It does not perform any data processing itself.
3. Slave nodes are the ones that pull the data from disks and perform actual data processing. These slave nodes are also known as worker nodes and the effective processing capability depends on the number of worker nodes configured in a cluster.
4. When slave jobs are asked to perform processing, slave nodes launch multiple containers inside, to process a part of the data. In the preceding case, two containers C1 and C2 are launched on two different machines, which are part of the same data processing job.

The preceding picture, in reality, is very complex. However, for understanding scaling, knowing this much is enough. In the preceding setup, Master nodes are big machines that can handle the management of multiple jobs running on the worker nodes. Master nodes are the single point of failure in the YARN architecture and to tackle this, in production deployments you deploy two name nodes to support high availability in case active

name node goes down, the secondary name node becomes active to handle job runs. Scaling a cluster means the capability to add or remove the worker nodes to a cluster.

Dataproc is managed to offer from GCP, which primarily provides users with the Hadoop 2.0/YARN cluster. The complete management related to creating, maintaining, and managing the cluster is abstracted away from the IT teams, and GCP makes sure that everything is running in high availability mode. Scaling of GCP Dataproc means scaling up and down the worker nodes, which, if done manually, is known as manual scaling, and if done in an automated fashion, is known as Autoscaling. Consider [Figure 13.3](#), which is simply [Figure 13.2](#) superimposed with GCP components to represent GCP Dataproc:





**Figure 13.3:** Dataproc cluster

GCP goes further ahead and optimizes the preceding architecture and creates worker nodes of two different types.

- **Primary worker node:** Primary worker nodes are standard compute engine VMs, on which data can be stored (Hadoop file system). The data stored here is replicated with a replication factor of 2, and backed up on GCS as well. It is not advised to apply Autoscaling on these nodes, as scaling up is non-disruptive, but scaling down could be because of data stored. Data must be moved to other nodes before de-commissioning a node.

- **Secondary worker node:** Secondary worker nodes are Managed instance groups with no storage availability. Their prime aim is to scale up and down fast, adding or removing the compute capability on a cluster.

A YARN cluster generates a wide variety of metrics, which could be analyzed to assess whether to scale a cluster or not. A complete list of metrics can be seen here:

<https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/Metrics.html>

In GCP, these metrics are available in Cloud Monitoring. Autoscaling uses one such metric to decide scaling, and that is YARN memory. It might sound like a limitation, as one might argue that scaling needs could be on basis of cores too. Dataproc however, currently supports YARN memory metric scaling only.

Let us now go ahead and create a Dataproc cluster and launch our first Spark job. You can create a Dataproc cluster from the UI console, or by triggering the following GCloud command.

```
gcloud dataproc clusters create scaling-gcp-demo-cluster --region us-central1 --zone us-central1-f --master-machine-type n1-standard-4 --master-boot-disk-size 500 --num-workers 2 --worker-machine-type n1-standard-4 --worker-boot-disk-size 500 --image-version 2.0-debian10 -project scaling-gcp
```

The preceding command will create a cluster named '**scaling-gcp-demo-cluster**' in region **us-central1**, with master and worker machine type as **n1-standard-4** in the project **scaling-gcp**.

There are some pre-created examples present on Dataproc nodes. One such use case is the counting of words. Understanding Spark APIs is out of the scope of the book. But once you have a written Spark job, packaged as a jar, and copied to one of the nodes in the Dataproc cluster, you can submit the following command to trigger a Spark job.

```
gcloud dataproc jobs submit spark --cluster=scaling-gcp-demo-cluster \
  --region=us-central1 \
  --jars=file:///usr/lib/spark/examples/jars/spark-examples.jar \
  --class=org.apache.spark.examples.JavaWordCount \
  -- gs://gcp-scaling-dataproc/input/
```

The preceding command submits a Spark job on the cluster 'scaling-gcp-demo-cluster' in us-central1 region using the main class 'org.apache.spark.examples.JavaWordCount'. It is further packaged in JAR 'spark-examples.jar' reading input from the GCS location 'gs://gcp-scaling-dataproc/input/\*'.

## Manual scaling

You can scale up and down your Dataproc cluster by increasing or decreasing the number of primary and secondary worker nodes. However, you cannot use a new machine type while scaling up. Because of the way Dataproc jobs are structured, it does not matter how many the number of nodes is, if the cumulative compute and memory per cluster level are as per the need of workflows. In other words, if your workload needs 10 vCPUs and 100 GB RAM, it does not matter if you made the infra available in 2 nodes of 5 vCPU and 50 GB RAM or 5 nodes of 2 vCPUs and 20 GB RAM. There could be a few situations like that of a Data science workflow where this might get violated, especially if you are performing iterative processing, which involves reading data in memory and then the data being available across multiple nodes for processing.

If you need to change the machine type, there is no other way apart from creating a new cluster and then migrating all your jobs to the new cluster.

Apart from the obvious need to increase and decrease the computing power of your cluster, another reason why we need to scale up is so that we can increase the number of nodes to expand available Hadoop Distributed File System storage. Generally, while scaling up, increasing infrastructure has no downside to already running jobs. However, when downscaling, it is vital to have at least the infrastructure running, so that currently running jobs does not starve due to infrastructure. Job execution will become slow and might even freeze in the worst cases.

There are multiple ways provided by the GCP to perform the activity. Three such ways are as follows:

- 1. Use the GCloud command-line tool in the GCloud CLI**

You can use the following GCloud command to increase/decrease the number of nodes (primary and secondary) in the cluster.

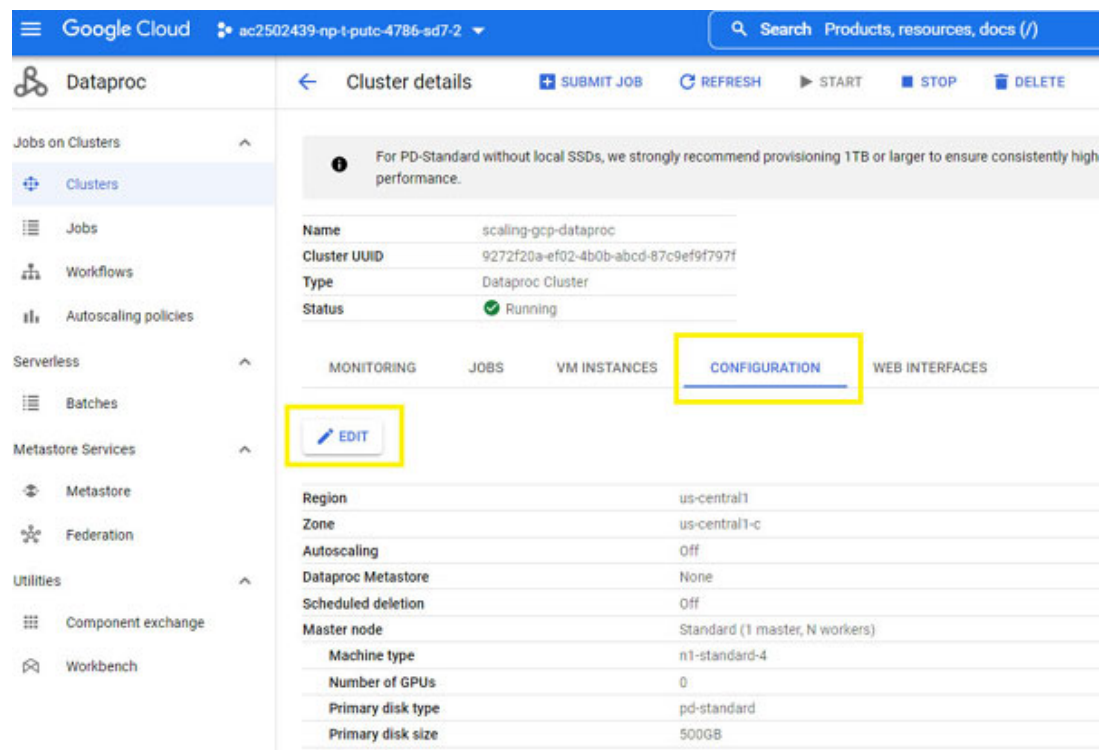
```
gcloud dataproc clusters update cluster-name \
  --region=region \
  [--num-workers and/or --num-secondary-workers]=new-number-of-workers
```

Assuming that we have a cluster named **scale-gcp-dataproc-cluster** and want to configure the new primary node to be 5 and secondary nodes to be 2, the preceding command will become:

```
gcloud dataproc clusters update scale-gcp-dataproc-cluster \
  --region=us-central1 \
  --num-workers = 5 \
  --num-secondary-workers=2
```

## 2. Edit the cluster configuration in the Google Cloud Console

You can scale up/down the Dataproc cluster by going to **Cluster details** page and clicking on the **Edit** option on **Configuration**, as shown in [Figure 13.4](#):



The screenshot shows the Google Cloud Console interface for a Dataproc cluster. The left sidebar contains navigation links for Jobs on Clusters, Clusters, Jobs, Workflows, Autoscaling policies, Serverless, Batches, Metastore Services, and Utilities. The main content area displays the 'Cluster details' page for a cluster named 'scaling-gcp-dataproc'. The 'CONFIGURATION' tab is selected, and the 'EDIT' button is highlighted with a yellow box. The configuration details are as follows:

Property	Value
Name	scaling-gcp-dataproc
Cluster UUID	9272f20a-ef02-4b0b-abcd-87c9ef9f797f
Type	Dataproc Cluster
Status	Running
Region	us-central1
Zone	us-central1-c
Autoscaling	Off
Dataproc Metastore	None
Scheduled deletion	Off
Master node	Standard (1 master, N workers)
Machine type	n1-standard-4
Number of GPUs	0
Primary disk type	pd-standard
Primary disk size	500GB

**Figure 13.4:** Manual scaling

This will open a pane where you can configure the number of primary and secondary nodes, as shown in [Figure 13.5](#):

Editing cluster

Worker nodes \*  
5

Secondary worker nodes \*  
2

Labels

Key 1 goog-dataproc-cluster-name	Value 1 scaling-gcp-dataproc
Key 2 goog-dataproc-cluster-uuid	Value 2 9272f20a-ef02-4b0b-abcd-87c9
Key 3 goog-dataproc-location	Value 3 us-central1

+ ADD LABEL

☐ Use graceful decommissioning ?

SAVE CANCEL EQUIVALENT REST

**Figure 13.5:** Manual scaling attributes

### 3. Use the REST API

You can also perform the preceding option, that is, setting number of primary worker nodes to 5 and secondary worker nodes to 2 using REST API.

PATCH

`https://dataproc.googleapis.com/v1/projects/<PROJECT_ID>/regions/us-central1/clusters/<CLUSTER_NAME>?`

`updateMask=config.worker_config.num_instances,config.secondary_worker_config.num_instances`

```
{  
  "config": {
```

```

    "workerConfig": {
      "numInstances": 5
    },
    "secondaryWorkerConfig": {
      "numInstances": 2
    }
  },
  "clusterName": "scaling-gcp-dataproc"
}

```

## Auto scaling

Identifying the right number of machines for a workload manually is difficult, and thus we have the concept of auto-scaling. In Dataproc as well, you can configure auto-scaling, such that the infrastructure will scale up and down based on workload. These actions of scaling up and down happens by monitoring system metrics and then eventually Dataproc Autoscaler ensures to maintain the value of the metrics in configured threshold ranges.

In Dataproc, you can define Autoscaling policies. It defines the complete scaling plan for a cluster, that is, max-min nodes, frequency, and aggressiveness to provide fine-grained control over cluster resources throughout the cluster lifetime.

The key considerations before enabling Autoscaling are as follows:

- A data processing job reads from a source and writes to a destination. If we allow the application to scale, thanks to Autoscaling, it will result in a lot of connections getting created to source and input. Thus, Autoscaling should be enabled only when the source and destination also scale appropriately to support connections.
- A Dataproc cluster generally takes the same time for creation and Autoscaling. Thus, if you intend to scale down the Dataproc cluster to minimum nodes when there is no job running, it is always recommended to shut it down. Do not use Autoscaling to shrink the Dataproc cluster to the bare minimum number of nodes.
- Not all features available in Spark and Hadoop are auto-scalable. For example, Spark Structured Streaming is not

compatible with the Autoscaling of the Dataproc cluster.

- It is not recommended to use HDFS scaling as a reason to enable Autoscaling. HDFS is only hosted on primary nodes and the number of primary worker nodes should be sufficient to store HDFS data. A decommissioning of nodes in case of it holding HDFS data, can be time consuming. The HDFS blocks stored on the nodes should be first migrated to other nodes, and then the nodes should be decommissioned.

## **Autoscaling deep dive**

Let us have a look into how Autoscaling works in a Dataproc cluster. Dataproc checks Hadoop YARN metrics at a frequency known as the cooldown period. On each cycle of this check, the number of worker nodes scaled up or down is calculated. This attribute contributing to the calculation is contributed to Autoscaling policy configurations. Autoscaling for Dataproc cluster takes place in the following steps:

1. On each cool-down cycle, Autoscaling identifies pending, available, allocated, and reserved memory. The multiple calculations of such past values are averaged to identify the needed number of worker nodes.

$$\text{Estimated Number of Workers} = \frac{\text{AVG (Pending Memory - Available Memory)}}{\text{Memory Per Node}}$$

Here, Pending Memory is a signal that there are queued-up tasks that are waiting to be completed, and that the delay is due to infrastructure unavailability. Available Memory is a signal that the cluster has extra resources available. A negative value here means that your cluster is over-provisioned, and that the autoscaler needs to de-commission the number of nodes by - the current number of nodes commissioned minus the number of worker nodes calculated above. In case the preceding number comes out to be positive, which represents the scale-up condition, it will be vice versa.

2. Triggering the commissioning or de-commissioning of the number of worker nodes, calculated by the preceding

formulae is aggressive. To control this aggression, you have two configurations:

### **Scaleup factor**

When there is a need to scale up the worker nodes by a few worker nodes (calculated above), the following formula is used to identify the actual workers.

Actual Number of workers =  $\text{ROUND\_UP}(\text{Estimated Number of Workers} * \text{scaleup factor})$

For example, if the formula in Step 1 calculates the estimated number of nodes to scale up as 5, and you had configured a scale-up factor of .5, then the cluster will be scaled by ( $\text{ROUND\_UP}(0.5*5)=\text{ROUND\_UP}(2.5)=3$  ) by 3 worker nodes.

### **Scaledown factor**

As is the case with scale up, you have the configuration of scale down as well, which works very similar to scale up. In case of scale down, the actual number of worker nodes to decommission is calculated by the following formula:

Actual Number of workers =  $\text{ROUND\_DOWN}(\text{Estimated Number of Workers} * \text{scale down factor})$

For example, if the formula in Step 1 calculates the estimated number of nodes to scale up as 5, and you had configured a scale down factor of .5, then the cluster will be scaled down by ( $\text{ROUND\_DOWN}(0.5*5)=\text{ROUND\_DOWN}(2.5)=2$  ) 2 worker nodes.

3. Using the scale up and scale down factor in step 2, one can know the mechanism to handle the aggressiveness of scaling. However, Autoscaling is not triggered in every cycle. In this section, we will look into how to control the frequency of updates. Frequency is controlled by two factors: **scaleUpMinWorkerFraction** and **scaleDownMinWorkerFraction**. Both the parameters act as a threshold to determine if the scaling will autoscale the cluster.

The rules associated with the factors are as follows:

if (actual workers >  $\text{scaleUpMinWorkerFraction} * \text{current cluster size}$ ) then scale up

if (actual workers >  $\text{scaleDownMinWorkerFraction} * \text{current cluster size}$ ) then scale down



4. If the preceding calculations result in the scaling up or down decision, Autoscaling policies define the scale up and down the lower limit as `minInstances`, and upper limit as `maxInstances` of `workerConfig` and `secondaryWorkerConfig`. There is a policy called `weight`, which is the ratio of primary to secondary workers, and it is used to determine how to spill the workers in primary and secondary worker instance groups.

The result of these set of calculations leads to the final Autoscaling change that is to be applied on the cluster for the scaling period.

## **Introducing Autoscaling Policies API**

For interacting with GCP Dataproc, there are three interfaces available: GCP client libraries, gRPC and REST APIs. Client libraries are the ones supported by the community and are available in multiple programming languages. If you do not have a ready-made library present, you can write one using gRPC. However, there is an easier way of handling the second case, and that is by using REST APIs. Almost all programming languages support REST APIs. In this section, you will get a good insight into the Dataproc Autoscaling API and what each attribute means.

In the Autoscaling APIs, the resource is `AutoscalingPolicy` and following that, we have various methods defined on the resource. The resource `AutoscalingPolicy` describes the configuration which Dataproc expects, and methods are just a way to pass these configurations to the Dataproc cluster.

An Autoscaling resource can be defined as a YAML file and applied to the Dataproc cluster by the following command: `gcloud dataproc Autoscaling-policies import`. Another way is to represent the Autoscaling Resource as JSON, and apply that to the Dataproc cluster via REST APIs.

### **Autoscaling\_policy resource**

This resource describes an Autoscaling policy for Dataproc cluster autoscaler. Refer to the following code:

#### **JSON Representation**

```
{  
  "id": string,
```

```

    "name": string,
    "workerConfig": {
      object (InstanceGroupAutoscalingPolicyConfig)
    },
    "secondaryWorkerConfig": {
      object (InstanceGroupAutoscalingPolicyConfig)
    },
    "basicAlgorithm": {
      object (BasicAutoscalingAlgorithm)
    }
  }
}

```

[Table 13.1](#) describes what each and every field in the preceding JSON representation means:

Fields	Description
id	Required: Unique identifier for Autoscaling policy.
name	<p>This is the name of Autoscaling policy resource.</p> <ul style="list-style-type: none"> <li>For <code>projects.regions.autoscalingPolicies</code>, the resource name of the policy has the following format: <code>projects/{projectId}/regions/{region}/autoscalingPolicies/{policy_id}</code></li> <li>For <code>projects.locations.autoscalingPolicies</code>, the resource name of the policy has the following format: <code>projects/{projectId}/locations/{location}/autoscalingPolicies/{policy_id}</code></li> </ul>
workerConfig	<p>Required. It describes Autoscaling configurations for primary worker nodes.</p> <p>It is an object of type - <code>InstanceGroupAutoscalingPolicyConfig</code> (Described as follows)</p>
secondaryWorkerConfig	<p>Optional. It describes Autoscaling configurations for secondary worker nodes.</p> <p>It is an object of type - <code>InstanceGroupAutoscalingPolicyConfig</code> (Described as follows)</p>
basicAlgorithm	<p>It is an object of type <code>BasicAutoscalingAlgorithm</code>. This object has a field "yarnConfig" using which the Autoscaling configurations like <code>scaleup</code>, <code>scaleDown</code> and so on is configured.</p>

**Table 13.1:** Autoscaling policy resource descriptions

## [BasicAutoscalingAlgorithm Resource](#)

The configurations for **BasicAutoscalingAlgorithm** resource are as follows:

**JSON Representation**

```
{
  "cooldownPeriod": string,
  "yarnConfig": {
    object (BasicYarnAutoscalingConfig)
  }
}
```

[Table 13.2](#) describes the preceding JSON resource for **BasicAutoscalingAlgorithm**:

Fields	Description
cooldownPeriod	This is an optional field of type string, and represents time between two scaling events. Scaling period starts after the completion of update operation.  By default, the value is 2 minutes. However, a lower value means workload changes will more quickly affect the cluster size, which might be completely unnecessary.  Bounds: [2m, 1d]. Default: 2m.
yarnConfig	This is an optional field of object type BasicYarnAutoscalingConfig (described as follows). This defines the Autoscaling config for your YARN cluster.

**Table 13.2:** BasicAutoscalingAlgorithm Descriptions

**BasicYarnAutoscalingConfig Resource**

The configurations for **BasicYarnAutoscalingConfig** resource are as follows:

**JSON representation**

```
{
  "gracefulDecommissionTimeout": string,
  "scaleUpFactor": number,
  "scaleDownFactor": number,
  "scaleUpMinWorkerFraction": number,
  "scaleDownMinWorkerFraction": number
}
```

[Table 13.3](#) describes the preceding JSON resource for **BasicYarnAutoscalingConfig**:

--	--

Fields	Description
gracefulDecommissionTimeout	This is a required field of type string. It configures the timeout for YARN graceful decommissioning of node managers. It is only applicable to decommissioning of nodes, and it specifies the duration to wait for the jobs to complete on a node before forcefully removing nodes. The value range is in between 0 seconds to one day.
scaleUpFactor	<p>This is a required field of type number. Refer to the explanation of <code>scaleUpFactor</code> in the previous section “Autoscaling deep dive”(Point 2 with the heading Scaleup Factor). It could take up any value in the range of 0 to 1. A scaleup factor or 1 represents aggressive scaling, resulting in no pending memory remaining after the update. A scale factor of 0 means no scaling up for a cluster.</p> <p>Generally, a good starting value for this attribute is .05 for Spark and Hadoop jobs with dynamic allocation enables. For Tez jobs and Spark jobs with a fixed executor count, we should use the attribute value to be 1.</p>
scaleDownFactor	<p>This is a required field of type number. Refer to the explanation of <code>scaleDownFactor</code> in the previous section “Autoscaling deep dive” (Point 2 with the heading Scaledown Factor). It could take up any value in the range of 0 to 1. A scaleup factor or 1 represents aggressive downscaling, resulting in no memory remaining after the update. A scale factor of 0 disables removing of worker nodes.</p> <p>Set this value to 0.0 to avoid scaling down the cluster, for example, on ephemeral clusters or for a single-job cluster.</p>
scaleUpMinWorkerFraction	This is an optional field of type number. Refer to the explanation in the previous section “Autoscaling deep dive”(Point 3). A worker cluster of 20 worker nodes and a threshold of .2 means Autoscaler must recommend at least 4 worker nodes to be added. A value of 0 implies Autoscaler will scale up on the recommended change. The default value is 0 and can take any value between 0 and 1.
scaleDownMinWorkerFraction	This is an optional field of type number. Refer to the explanation in the previous section “Autoscaling deep dive” (Point 3). A worker cluster of 20 worker nodes and a threshold of .2 means Autoscaler must recommend at least 4 worker nodes to be de-commissioned. A value of 0 implies Autoscaler will scale down on the recommended change. The default value is 0 and can take any value between 0 and 1.

**Table 13.3:** BasicYarnAutoscalingConfig descriptions

## [InstanceGroupAutoscalingPolicyConfig Resource](#)

The configurations for `InstanceGroupAutoscalingPolicyConfig` resources are as follows:

**JSON Representation**

```
{
  "minInstances": integer,
  "maxInstances": integer,
  "weight": integer
}
```

[Table 13.4](#) describes the preceding JSON resource for `InstanceGroupAutoscaling PolicyConfig`:

Fields	Description
minInstances	This is an optional field of type integer. It configures the minimum number of worker instances. Default value is 0.
maxInstances	This is a required field of type integer in the case of primary worker nodes and if secondary worker nodes are configured with minInstances, then it becomes mandatory for secondary worker nodes as well. It configures the maximum number of worker instances.
weight	<p>This is an optional field of type number. This represents the ratio of primary worker nodes vs. secondary worker nodes. For example, if this value for primary worker nodes is set to 2 and the value is set as 1 for secondary, then for each primary worker node, Autoscaler will try to ensure 1 secondary node.</p> <p>The cluster might not be able to reach the desired configuration if it starts contradicting min/max configurations.</p> <p>If no value is set on any instance group (primary or secondary), then the Autoscaler configures an equal value for both, that is, 1 primary worker node and 1 secondary worker node. If any of the groups is not set, it will be assumed to be 0. For example, if weight is only set for primary, a value of 0 will be assumed for secondary worker nodes and this implies that the cluster will use primary workers only and no secondary workers.</p>

**Table 13.4:** *InstanceGroupAutoscalingPolicyConfig* descriptions

You can define actions on the resources mentioned above. This action will apply the configurations on the Dataproc cluster. [Table 13.5](#) describes these various actions which could be taken:

Actions	Description

Create	Create a new Autoscaling policy.
Delete	Deletes an Autoscaling policy.
Get	Retrieves an Autoscaling policy.
getIamPolicy	Gets the access policy for a resource.
List	Lists all the Autoscaling policies in the project.
setIamPolicy	Sets the access policy on the specified resource.
testIamPermission	Returns all the permission on a resource.
update	Updates the Autoscaling policy.

**Table 13.5:** Autoscaling Actions

In the subsequent sections, we will see how to apply the preceding Autoscaling resources (configurations) on a Dataproc cluster, using the actions defined in [Table 13.5](#).

## **CRUD on Autoscaling policies**

You can perform all CRUD operations on Autoscaling policies, using the resource format as JSON and action as REST API. Another option is to use YAML representation as a resource and GCloud commands as a way to take action. The third approach is via the GCP console. For showcasing the different CRUD operations of Autoscaling policies, we will use the second option, that is, YAML and GCloud to see how these concepts look in action.

Here is an Autoscaling policy YAML file, which contains all the attributes:

```
workerConfig:
  minInstances: 1
  maxInstances: 10
  weight: 1
secondaryWorkerConfig:
  minInstances: 0
  maxInstances: 100
  weight: 1
basicAlgorithm:
  cooldownPeriod: 2m
yarnConfig:
  scaleUpFactor: 0.05
```

```
scaleDownFactor: 1.0
scaleUpMinWorkerFraction: 0.0
scaleDownMinWorkerFraction: 0.0
gracefulDecommissionTimeout: 1h
```

Save the preceding configuration as **autoscaling-policy.yaml** and use the following command to create an Autoscaling policy named **scaling-gcp-autoscaling-policy** in **us-central1** region.

```
gcloud dataproc autoscaling-policies import scaling-gcp-autoscaling-
policy \
  --source=autoscaling-policy.yaml \
  --region=us-central1
```

Once the preceding command executes, check whether the Autoscaling policy is created or not.

```
gcloud dataproc autoscaling-policies list --region=us-central1
```

For fetching the details of a particular policy, use the following command:

```
gcloud dataproc autoscaling-policies describe scaling-gcp-
autoscaling-policy --region us-central1
```

Here, '**scaling-gcp-autoscaling-policy**' is the policy id.

To delete an Autoscaling policy, use the delete action:

```
gcloud dataproc autoscaling-policies delete scaling-gcp-autoscaling-
policy --region us-central1
```

In all the commands, '**scaling-gcp-autoscaling-policy**' is the policy id and **us-centra1** is the policy region.

## [Applying Autoscaling policies to Dataproc cluster](#)

Autoscaling policies can be applied to a Dataproc cluster in two ways:

1. At the time of creation of new cluster.
2. Updating the Autoscaling policy on an already existing cluster.

For applying an Autoscaling policy at the time of creation of new cluster, use the following command:

```
gcloud dataproc clusters create scaling-gcp-dataproc-cluster \
  --autoscaling-policy=scaling-gcp-autoscaling-policy \
```

```
--region=us-central1
```

The preceding command will create a Dataproc cluster 'scaling-gcp-dataproc-cluster' with Autoscaling policy 'scaling-gcp-autoscaling-policy' in region us-central1.

For applying an Autoscaling policy to an already existing cluster, use the following command.

```
gcloud dataproc clusters update scaling-gcp-cluster \  
  --autoscaling-policy=scaling-gcp-autoscaling-policy --region=us-  
  central1
```

The preceding command will enable the policy 'scaling-gcp-autoscaling-policy' on an existing cluster 'scaling-gcp-cluster'.

For disabling the Autoscaling policy on a cluster, use the following command:

```
gcloud dataproc clusters update scaling-gcp-cluster --disable-  
  autoscaling \  
  --region=us-central1
```

**Note: Your Autoscaling policies and cluster policies can never contradict. However, if they do, the action resulted in contradiction will not run successfully. For example, if a cluster has minimum size 2 created and you try to enable Autoscaling policy with minimum of 10 nodes, the console will show error.**

## Limitations of scale

Till now, we looked into all the good things about scaling. However, there are also a few restrictions/operations that are not allowed, when it comes to scaling a Dataproc cluster. In this section, let us deep dive into a few key ones:

- The machine types of a cluster should be consistent. You cannot configure an Autoscaling policy that adds machines of different types. If you have a machine type of n1-standard1, and you want to change it to any other class, you have to make sure that all jobs finish on the earlier cluster and only then can you create a new cluster with updated machine types.



- GCP provides a platform where a YARN cluster is set up. So even when GCP does not put a restriction, if the YARN cluster has a restriction, it automatically becomes a restriction of the platform as well. For example, a restriction on YARN is that it cannot exceed more than 10k nodes. Hence, even if GCP has no such condition since YARN is hosted on GCP, Dataproc has the same restriction too.
- Like the preceding point, there could be a restriction on GCP as well, which can further restrict scaling. Make sure that you have enough quotas defined for CPU, RAM, Disk, external IP, QPS limitations, and so on. Ensure that they are aligned with your scaling expectations.
- There is restriction on how much memory you can attach to a virtual machine in GCP. This restriction inherently becomes restriction for Dataproc node as well. Maximum limit of memory you can attach is 64 GB, and so, design your applications accordingly.
- While it looks very fascinating that a cluster can auto-scale and are the number of worker nodes for a job, pay attention to the input source and output source capabilities. There could be hundreds of such threads trying to connect to the sources, when the sources are not scaling when needed. This puts restrictions on the throughput of the job even when it has scaled.

## **Graceful decommissioning of clusters**

Decommissioning removes worker nodes from the clusters. When you downscale a cluster, the work in progress on the worker node may stop before completion (forceful decommission). There is also the concept of graceful decommissioning, where the cluster waits for all the tasks to finish on a worker node and only when every task is complete, the nodes are taken away. This feature is available in Dataproc version 1.2 or later, and is based on the graceful decommissioning feature of YARN. For more details visit the link:

**<https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/GracefulDecommission.html>**

By default, graceful decommissioning is disabled in the cluster, and you can enable it by mentioning `'gracefulDecommissionTimeout'` attribute in the YARN config section of Autoscaling YAML. The timeout value can be configured as zero, which is the default, that is, forceful decommission or duration in seconds, minutes, and hours. The maximum value for this period is 1 day.

As a best practice, a graceful decommissioning value timeout should be set to a value longer than the longest job running on the cluster. If you had configured the `'gracefulDecommissionTimeout'` time configured as 1 hour and your cluster has jobs that take more than an hour, consider moving them to a different cluster with a larger time configured for the property.

## **Using preemptible VMs to scale**

In GCP Dataproc, there are two kinds of worker nodes – primary and secondary. Primary and secondary worker nodes perform the same compute operations. However, secondary worker nodes do not store data. The only function of secondary nodes is to perform computations. While both should have the same machine type, the attached persistent disk can vary. By default, the persistent disk configuration in secondary nodes is just 100 GB.

There could be two kinds of secondary worker nodes – preemptible and non-preemptible. All nodes in the secondary worker should be of the same type, whether preemptible or non-preemptible. The default is preemptible, meaning, they can be claimed or removed from the cluster if GCP needs it for other tasks. Such removals can impact job stability, but you can still use them to reduce the cost associated with the machine. YARN workloads are designed with the presumption that worker nodes will fail, as the original document uses the term “commodity hardware” while describing nodes in a Spark cluster. By nature, if a task fails, or a node is decommissioned, the whole job is not marked as failed. Rather, the task running on the node is re-performed and results are aggregated. The number of times a that task is retried, is configurable.

However, you would not want to ideally have the number of preemptible worker nodes to be greater than 50% of your total

worker nodes. Since failures are involved, configure a higher number of retries for the tasks.

Preemptible machines come at one-third of the cost of the non-preemptible machines, and that is the biggest advantage in selecting preemptible machine types. However, the cost generally does not reduce by one-third. A task that failed due to decommissioning of a preemptible worker node, must redo the tasks on another worker (primary or secondary), and hence it needs infra for rescheduling.

## **Conclusion**

Google Dataproc is managed to host a YARN cluster, which allows you to run Hadoop and spark jobs. The complete responsibility of creating and managing the YARN cluster is done by GCP, and the Engineering teams are expected to work more on creating business values. Because Hadoop and Spark workloads are the most famous and widely used frameworks to process huge amounts of data, all cloud providers managed the YARN cluster. On GCP, you can manually as well as automatically scale the Dataproc cluster.

Dataproc allows configuring Autoscaling by defining Autoscaling policies, along with CRUD operations, which can be applied at the time of the creation of the Dataproc cluster or can be added afterward as well on an old Dataproc cluster. Commissioning and decommissioning of nodes to the cluster is not real-time, and hence due diligence must be taken while performing the activities. Generally commissioning does not have an impact on processing; they only have cost implications. However, de-commissioning can affect processing. Thus, defining graceful decommissioning is highly recommended.

## **Points to remember**

- Google Dataproc is managed service for the YARN cluster, enabling the deployment of Hadoop and Spark jobs.
- You can scale a Dataproc cluster manually, as well as in an automated fashion, by creating and applying Autoscaling policies.

- Autoscaling policies provide provisions to define minimum and maximum nodes for both primary and secondary worker nodes, along with parameters to define scaling strategies for Autoscaling.
- Autoscaling policies also provides a mechanism to configure `gracefulDecommissionTimeout` attribute to make sure a worker node is decommissioned when all the assigned work is complete on a worker node.
- Dataproc secondary worker node provides a mechanism to create preemptible worker nodes, which results in cost-effective scaling.

## Questions

1. Can we configure preemptible and non-preemptible machines as secondary worker node?
2. Can we change the type of worker virtual machines while scaling up?
3. Can we apply one Autoscaling policy on multiple Dataproc clusters?
4. What are the ways to configure Autoscaling on a Dataproc cluster?

## Answers

1. No, you can either set preemptible or non-preemptible, but not both.
2. No, machine type has to be always same.
3. Yes, one Autoscaling policy can be applied to multiple clusters.
4. Multiple ways - Google Client Libraries, gRPC to write custom library, REST API, GCloud command and UI console.

# **CHAPTER 14**

## **Scaling Google Dataflow**

### **Introduction**

Google Cloud Platform Dataflow is a platform on which you can execute Apache Beam (an open-source distributed computing framework) pipelines. Apache Beam provides constructs to chunk and distributes datasets across multiple machines, processes them parallelly, and aggregates the result. An execution platform like Dataflow supports these constructs infrastructurally and facilitates actual execution over a cluster of machines. A beam pipeline is supported by multiple other runners such as *Spark Runner*, *Apache Flink*, *Apache Samza*, *Apache Nemo*, *Hazzlecast Jet*, and so on.

If you see closely, when we talk about scaling up such a distributed system, there are two main areas of concern. One is Horizontal scaling, that is, increasing the number of virtual machines/threads, that are reading and processing a chunk of data. A smaller chunk of data supplemented with a thread to process each small chunk, speeds up the data processing. The more optimal is the data chunk and parallel threads availability, the better is the throughput of the system.

The second concern is to shuffle the data at one machine, and to aggregate the results of processing that took place on multiple machines. To effectively scale a system, you will have to take care of the preceding two concerns, and once that is done, the time taken to process the data will become a function of infrastructure, that is, the moment SLAs are breached, scale up will be done. When we talk about scaling the Apache Beam pipeline running on Dataflow runner,

scaling essentially means how Dataflow handles the aspects of multiple worker nodes, related optimizations, and shuffling of data.

## **Structure**

In this chapter, we will discuss the following topics:

- Introduction to Dataflow
- Dataflow autotuning
  - Horizontal autoscaling
    - Scaling Dataflow for batch jobs
    - Scaling Dataflow for streaming jobs
  - Vertical auto scaling
  - Dynamic work rebalancing
- Autoscaling algorithms
- Scaling and Dataflow Prime right fitting
- Limiting max nodes
- Scaling the persistent disk
- Optimizing data shuffle using Dataflow shuffle

## **Objectives**

After studying this chapter, you will understand the basics of Dataflow, that is, concepts and infrastructural footprints of a Dataflow job. This understanding will eventually help you in appreciating the concepts behind scaling (horizontal and vertical) of the data flow job. Dataflow supports both streaming and batch workloads written in Apache Beam. This chapter also covers the strategies to optimize stages like the shuffle phase, and to effectively auto-scale to hundreds of machines. This will help reduce the chokes in the system by effectively managing the movement of data across the network.

## **Introduction to Dataflow**

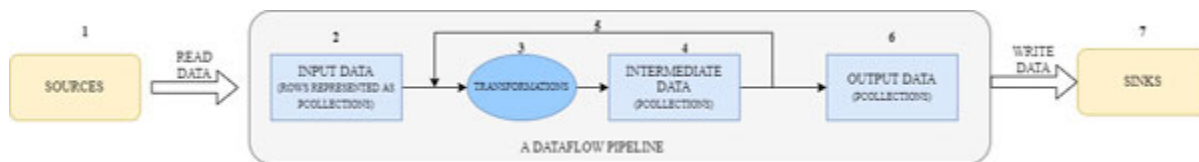
As discussed, Dataflow is just a platform (one option among many supported platforms) on which you can execute Apache Beam pipelines. Management of these execution platforms bring along with it, a lot of management and maintenance tasks for the IT teams. Those who had ever worked on on-premises Hadoop cluster will know the cumbersome day to day activity involved in managing a cluster for distributed computing. Nodes are expected to go up and down, network is expected to face glitches and many other runs time issues. However, the application is expected to process huge volumes of data with business SLAs. The expectation from any other processing framework is also similar. Obviously, greater the number of glitches, slower will be the time taken in processing.

GCP Dataflow abstracts away the complete cluster management dynamics from the IT teams. The engineering team can then concentrate on the business logic using Beam constructs, without bothering about anything related to the running of the logic, involving huge datasets on a cluster of machines. The only protocol Engineers must adhere to, is following Beam constructs while writing the business logic. The granular responsibilities and reliability aspects will be taken care of by GCP Dataflow. Dataflow supports serverless architecture, meaning that there is no infrastructure involved when you are not processing any data. Hence, it becomes a very cost optimal solution to handle and process Bigdata.

In Apache beam, there are not many differences between writing a processing logic for batch, versus writing logic for streaming. Dataflow too offers a similar ease between two approaches. You can pretty much manage the batch job almost in the same manner as streaming job.

## Apache Beam pipelines

Before you jump into analyzing a small word count problem using Dataflow, let us have a look at some key concepts and terminologies involved with an Apache Beam pipeline code. You are going to investigate a high-level container (a pipeline), data flowing through the pipeline (PCollections), and manipulation of data (using transforms). Consider [Figure 14.1](#):



**Figure 14.1:** A Dataflow Pipeline

Refer to the serialized points as follows:

1. **Sources:** A beam pipeline can read data from a wide variety of sources. Different runners support different options. For example, Spark runner supports Hadoop file system as a source. Similarly, Dataflow supports a wide variety of connectors to connect to sources supported in GCP. The most famous ones are Cloud Storage, Big Query, and Pub-Sub.
2. **Input Data:** When the data stored in sources are read via Beam connectors in memory, it is known as input data. Beam represents the in-memory data in the form of *PCollections*. *PCollections* are data structures on which you can easily apply transformations.
3. **Transformations:** Transformations are first-class functions that contain the business logic to process the data.
4. **Intermediate Data** - After applying a business logic (transformation) on a row of data (an item in *PCollections*), the output is a *PCollections*, again known as intermediate data.



5. The intermediate data could again be used as input to a transformation.
6. **Output Data:** If all the business logic is represented and applied to a series of transformations, and we got the result, it is known as Output data.
7. **Sinks:** The output data must persist in a data storage solution. As was the case with sources, different platforms provide different sinks. A pipeline running on Dataflow can push the out-put data to a wide variety of well-supported sinks such as Cloud Storage, Big Query, Big Table, and so on.

For information on how to author a pipeline in Apache beam, please refer to the official documentation:

<https://beam.apache.org/documentation>

Not all runners (Dataflow and spark), provide support for all the constructs of Beam. For identifying the different capabilities and their comparative analysis, visit the given link:

<https://beam.apache.org/documentation/runners/capability-matrix/>

Those who are new to Beam and had never written a pipeline, please go through the official “*Hello World*” example of distributed computing, that is, Word Count on the beam website.

## **Wordcount Dataflow job**

Once you are ready with code in Apache beam, you have to compile that code using the Dataflow libraries and then provide the infrastructure level parameters for execution.

For example, you can execute the preceding generic beam pipeline code to a Dataflow code, using the following command:

```
mvn compile exec:java -
Dexec.mainClass=org.apache.beam.examples.WordCount \
-Dexec.args="--runner=DataflowRunner --
gcpTempLocation=gs://GCS_BUCKET/tmp \
--project=YOUR_PROJECT --region=GCE_REGION \
--inputFile=gs://apache-beam-samples/hakespeare/* --
output=gs://YOUR_GCS_BUCKET/counts" \
-Pdataflow-runner
```

If you view the preceding command, the section in bold represents properties related to the execution of the beam code on Dataflow. Since we configured the Dataflow-runner, these bold sections are properties that configure the Dataflow runner. Each runner has its own set of properties. For example, the preceding command in case of spark runner now becomes:

```
mvn compile exec:java -
Dexec.mainClass=org.apache.beam.examples.WordCount \
-Dexec.args="--runner=SparkRunner --inputFile=pom.xml --
output=counts" -Pspark-runner
```

So, if we summarize running an Apache beam code on the Dataflow platform, it involves the following steps:

1. Build the beam code with Dataflow libraries.
2. Mention the runner as a Dataflow runner.
3. Provide the properties to Dataflow runner options specific to the platform.

**NOTE: Once the responsibility of executing the code is taken by the Dataflow runner, the complete responsibility to spin up machines, attach persistent disks, and so on, becomes the responsibility of Dataflow. Any optimization on the Dataflow platform will be automatically enabled under the hood.**

In the subsequent sections of the chapter, you will observe instantiating Dataflow runner class with proper attributes. This instantiation will configure the autoscaling behavior of the Dataflow job. Autoscaling, class of machines, and optimizations are all just configurations mentioned in the Dataflow Runner class.

When a pipeline is submitted to the Dataflow platform, the Dataflow platform creates an execution plan. If the pipeline is valid, a successful plan is created. This plan at first is unoptimized and the Dataflow after successfully analyzing the structure of the pipeline and creating an execution graph, performs optimizations under the hood. Two such optimizations are as follows.

### **Fusion optimization**

In this stage, Dataflow fuses multiple steps or transformations of your pipeline, into one step. It results in not materializing all the intermediate Pcollections in the pipeline, since materializing each PCollections could be costly in terms of memory and processing overhead.

### **Combine optimizations**

An operation like `GroupByKey`, `CoGroupByKey`, and `Combine`, involved a data shuffle, resulting in bringing the data with the same key to one worker. This optimization tries to combine and aggregate data locally on a worker node before starting the data shuffle.

Along with optimizations, the Dataflow service offers some advancements and customizations in different problematic areas. By problematic, it means that these areas were probably the ones where the jobs did not have an efficient strategy, which Dataflow did now. Thus, the result is a very efficient mechanism to handle complex areas in distributed pipelines.

## Dataflow Shuffle service

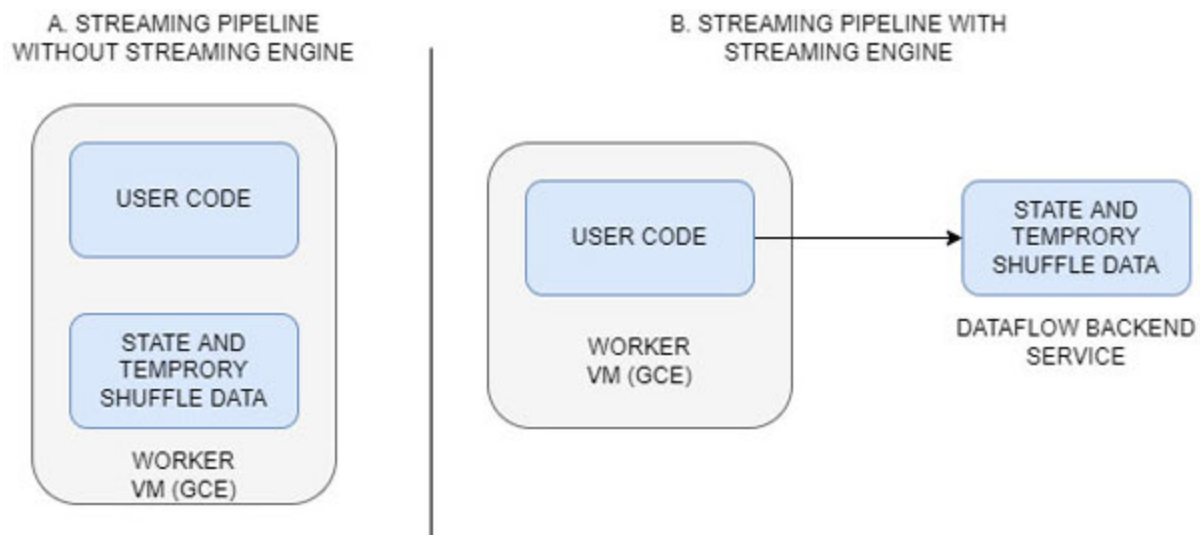
Operations like **GroupByKey**, and **CoGroupByKey** combine involved data shuffle, resulting in bringing the data with the same key to one worker. Before shuffle service, this responsibility was handed by worker nodes, which used to consume CPU cores, memory, and persistent disk to accomplish this. Dataflow service is a managed service by Dataflow, which moves the shuffle operation out of the worker VMs to a Dataflow managed backend service.

To activate this feature in your pipeline, mention the following parameter while triggering a job:

```
--experiments=shuffle_mode=service
```

## Dataflow streaming engine

Streaming applications need to shuffle data and store time window states. Cloud Dataflow historically performed these operations on worker virtual machines and used persistent disks to state states and store temporary data during the shuffle. Consider [Figure 14.2](#):



**Figure 14.2:** Streaming pipeline with and without Streaming Engine

This figure represents the scenario where the streaming engine is not used and hence the responsibility of

maintaining states and shuffling data lies with the worker node. The responsibility to maintain states and temporary data is handed over to a backend service by Dataflow, known as streaming engine. The streaming engine moves the operation of shuffle to a Dataflow backend service, leading to the following improvements:

- Reduced usage of resources on worker VMs. Streaming works best with small VMs (`n1-standard-2` or `n1-standard-4`) and does not need big persistent disk space.
- Easily adaptable to spikes in data. It supports granular scaling as worker node CPU consumption will not be spiked due to shuffle stages.

## **Dataflow Prime**

Dataflow is a data processing platform for your Apache Beam code. The Dataflow platform had some shortcomings mainly due to cohesive computation and state maintenance of job states. Dataflow Prime is an improvement over Dataflow, which segregates the data processing infrastructure (compute) and state-separated architecture. It also has features that improve efficiency, productivity, and scalability. Dataflow Prime brings to the table the following benefits:

- Vertical scaling of worker memory
- Right fitting of infrastructure
- Better recommendations of configuration parameters
- Smart visualizations of pipelines

The features *Job Visualizer*, *Smart Recommendations*, and *Data Pipelines* are also supported for non-Dataflow Prime jobs.

## **Configuring infrastructure**

The preceding Dataflow job used the default values for worker infrastructural components. As the engineering team, it is vital to understand the nature of your application and configure them on per need basis. Among multiple such components, a few key Dataflow worker properties which could be optimized while deploying the pipeline are as follows.

## **Disk size**

By default, the disk size for batch jobs is 250 GB and for streaming jobs, it is 400 GB. If your use case is not writing any data to disk and mostly processing the data in memory, this disk space is wasted. The recommendation is to run the job by progressively reducing the disk space to reach an optimal level.

You can do this by either setting the value in pipeline options or by just mentioning the following fragment in the Dataflow job trigger command.

```
--diskSizeGb=30
```

## **Machine type**

The default machine that a Dataflow job uses is *n1* machine type, and these machine types generally suffice all types of purpose. However, in case of need for a powerful CPU or high RAM, it is recommended to select the right machine type for your job to optimally run the workflow. You can use the default machines provided by GCP or can even mention a custom machine type. In custom machine type, you can use accelerators as well.

You can easily configure the machine type either as pipeline options in code, or by simply passing the following fragment while deploying a Dataflow pipeline:

```
--workerMachineType=custom-8-7424
```

The preceding fragment will spin up VMs with 8 cores and 7424 MB RAM.

## Disabling public IPs

By default, Dataflow service not only assigns private IP but public IP as well, to your worker VMs that are used in Dataflow. Generally, assigning public IPs is not needed for worker VMs. You can switch it off by passing the following fragment while creating a Dataflow template:

```
--usePublicIps=false
```

## Selecting right regions

It is recommended to run your Dataflow worker nodes in the same region where your source and sink reside. This reduces the cross-region movement of data, which not only reduces cost but also reduces the time taken to bring data into the memory of the Dataflow job.

Set the region value while deploying a Dataflow job.

```
--region=europe-west1
```

The preceding command sets the region value to **Europe-west1**.

## Dataflow job lifecycle

When a Dataflow pipeline is triggered, Dataflow generates an execution graph (an execution plan), which includes a complete plan of how each transformation and other processing functions will run. For example, a business logic not written in the **DoFn** object will run on the machine that runs the pipeline. However, a business logic written inside the **DoFn** object, is distributed across worker nodes. This is just one example of multiple other phases that a Dataflow job has to go through, before actually starting the execution. These phases are the Dataflow pipeline lifecycle stages. We are going to discuss four such stages.

## **Distribution and parallelization**

Dataflow Service distributes and parallelizes the processing logic among the workers. Anything processing logic written in a **ParDo** transform, will result in the distribution of processing on data across multiple worker nodes. If the processing of any of the distributed processing fails, Dataflow makes sure that it reruns the failed part.

## **Execution graph**

Dataflow creates a directed acyclic graph of steps that depicts the pipeline. This is the complete execution plan of a pipeline running on multiple machines. The graph is known as the **pipeline execution graph**.

## **Combining optimizations**

Google Dataflow observes the complete pipeline and tries to merge multiple steps into one step. This not only reduces the time of execution, but also reduces the compute, memory, and network requirements for running a job. When constructs like **CoGroupByKey** and **GroupByKey** are involved, Dataflow tries to locally combine the data before distributing them across worker nodes for further processing.

## **Fusion optimization**

Once the complete execution plan is ready, Dataflow might modify the graph to execute optimizations. Such transformations include combining multiple transformations into one step and hence resulting in a more optimized pipeline execution. This brings us to the fact that with Fusion optimizations, it does not matter in which order the instructions had been passed to a Dataflow job. The optimization phase will ultimately result in an optimized run of the pipeline.



## **Dataflow autotuning**

Dataflow service contains multiple autotuning features which can optimize a running Dataflow job on the fly. In this section, you will look into the three types (horizontal, vertical, and dynamic work rebalancing) of autoscaling provided in the Dataflow.

### **Horizontal autoscaling**

Horizontal scaling, when enabled automatically, chooses an appropriate number of worker machines to run a job. Let us assume that there are 3 phases in your Dataflow job. In such a case, the number of worker machines for each phase could be different, and this automatic addition of machines per phase is taken by Dataflow. There could be a few phases in the pipeline which are computationally intensive when compared to other phases and in those situations, Dataflow can automatically spin up more machines for a phase. Once the load decreases, these machines are automatically teared down.

Horizontal scaling is enabled by default on all streaming and batch jobs. You can disable it by using the following flag while triggering a pipeline:

```
--autoscalingAlgorithm=NONE
```

If you disable autoscaling, the number of workers assigned will be 3, by default. In case you want to modify the number of workers, specify the following fragment while triggering a pipeline:

```
--numWorkers = 5
```

When horizontal scaling is enabled, the user has no control over the number of worker nodes assigned to each of the phases. However, since the number of worker machines can become very high in number, it is always recommended to give an upper limit to the number of worker machines that

can be scaled up. For batch jobs, the default value for max worker machines is 1000 and for streaming jobs, the max number of machines is 100.

Users can override these default configurations by using the following fragment in the trigger pipeline command:

```
--maxNumWorkers = MAX_NUMBER_WORKER
```

**MAX\_NUMBER\_WORKER** is the number of worker machines that the user wants to allow the Dataflow job to scale up to.

Dataflow job scales are based on the amount of parallelism in your pipeline. Parallelism of the pipeline is the total number of threads that process data most efficiently at any given time.

## **Scaling Dataflow for batch jobs**

In the case of batch workloads, Dataflow tries to identify the number of worker machines for each stage. The number of worker machines depends on the amount of data being processed via stage as well as the stage throughput. Based on the two parameters, Dataflow evaluates the number of worker machines needed to process data and scales up/down accordingly.

The relationship between the amount of work and the number of workers shows a sub-linear relationship. Meaning if there are two data jobs, one with x load and the other with 2x load, then the number of worker machines in case of 2x will not be 2 times of job with x load.

A Dataflow job scales up or down based on the following conditions:

- Worker VMs will be released/torn down if the average CPU usage is below 5%.
- The amount of parallelism that can be achieved depends on the type of input data (zero parallelisms in

case of non-splitable formats of input), as well as the design of the pipeline.

## **Scaling Dataflow for streaming jobs**

In the case of a streaming pipeline with horizontal scaling enabled, Dataflow adapts to workload changes by scaling up/down the number of worker machines per phase. Another aspect that contributes to scaling up/down is resource utilization. Dataflow calculates a backlog time. Backlog time is calculated based on the number of bytes left to process from the input source and the throughput of the system. A pipeline is called backlogged when this backlog time remains above 15 seconds. Based on the value of backlog time, the Dataflow service decides to scale up and down. The conditions for the same are as follows:

### **Scale-up**

If a streaming pipeline remains backlogged, with average CPU utilization of more than 20%, for more than a couple of minutes, Dataflow scales up. Dataflow will clear the backlog after 150 seconds of scaling up, assuming throughput remains constant in the whole process.

### **Scale-down**

If the backlog is lower than 10 seconds and the average CPU utilization is less than 75% for approximately 2 minutes, the Dataflow scales down. After the scale-down activity, there is an average of 75% CPU utilization for workers.

### **No scaling**

No backlog and average CPU utilization of 75% or greater, results in no scale down. On the contrary, the CPU usage of 20% or less with no backlog, means no scale up.

## Predictive scaling

The streaming engine uses a timer backlog to enable predictive autoscaling. Streams of data mean small discrete datasets corresponding to a time window. These discrete periods are known as windows. It could happen that the streaming engine is falling short in terms of processing, that is, let us say the expectation was to finish processing 100 items in a window, but 10 got missed out. In such a case, the 10 missed out ones will become part of the new window, which is expected to process 10 (from the last window) and 100 (an additional record) in next window. Hence Dataflow can increase the number of worker instances to increase, to cater to 110 records.

## Horizontal scaling of streaming pipeline

### Autoscaling

For streaming jobs using the streaming engine, autoscaling is enabled by default. For jobs not using streaming engine, you can enable autoscaling by specifying the following execution parameter while triggering pipeline run:

```
--autoscalingAlgorithm=THROUGHPUT_BASED  
--maxNumWorkers=10
```

The preceding execution parameter will assign the autoscaling algorithm to be **THROUGHPUT\_BASED**, with a maximum number of workers to be equal to 10. Make sure while updating the pipeline with an increased maximum number of nodes, and do not forget to mention the algorithm again. Otherwise, it will default to no autoscaling.

**NOTE: For streaming jobs not using a streaming pipeline, the minimum number of nodes is 1/15th of the maximum number of nodes.**

## Manual scaling

You can configure a streaming pipeline to scale up and down manually. For this, you have to configure two parameters:

- Set `--maxNumWorkers` which is equal to the number of worker nodes needed to handle the peak load of your application.
- Set `--numWorkers` to configure the number of worker nodes and pipeline to be started with.

**NOTE: You need to be very careful while selecting the value for `maxNumWorker`. `maxNumWorker` configures the maximum number of worker nodes that a Dataflow job could scale up to. However, this value also configures the number of persistent disks. Each worker node needs one persistent disk for execution. When the system scales up or down, the persistent disk remains as it is, meaning you will be paying for the persistent disk even when you are not using it. A low value of `maxNumWorker` means a lower number of persistent disks. A Dataflow worker node can scale max up to the number of persistent disks, as one disk is needed per worker.**

## Vertical auto scaling

Vertical autoscaling allows Dataflow to dynamically scale up or down the memory configured to worker nodes. This helps the job to cater to errors such as out-of-memory errors and to improve efficiency as Dataflow does log of in-memory processing. Dataflow monitors the pipeline and identifies worker nodes choking due to lack of memory or exceeding available memory, and then replaces that worker node with a new worker node.

To leverage this feature, you must enable Dataflow Prime. Dataflow Prime enables automated and optimized resource management, reduced operational cost and improved monitoring capabilities.

The steps which will enable this feature for your pipeline are as follows:

- Enable the Cloud Autoscaling API. Dataprime uses Cloud autoscaling API for automatic scaling of memory.
- Enable Prime in your pipeline options. You can do this programmatically or by just passing the parameter while triggering a Dataflow job.

```
--DataflowServiceOptions=enable_prime
```

**Note: No code changes are needed in the Dataflow code, which was previously running without this feature.**

Once the preceding two steps are down, it is time to enable vertical autoscaling on memory. This could be done by including the following key-value pair in the Dataflow job trigger command:

```
--experiments=enable_vertical_memory_autoscaling
```

Vertical scaling takes place by replacing existing workers with bigger better new workers. It is recommended to use a custom container to improve latency arising due to transfer of load, from old to the new worker. For details to configure custom images in Dataflow, please read- [\*\*https://cloud.google.com/Dataflow/docs/guides/using-custom-containers\*\*](https://cloud.google.com/Dataflow/docs/guides/using-custom-containers)

A few points to be kept in mind, are as follows:

- Only streaming jobs can be vertically scaled.
- Only the memory attached to the worker nodes can be scaled up vertically.

- By default, the memory range can lie in between 6 GiB to 16 GiB. When using GPUs, the memory range is 12 GiB to 24 GiB. These upper and lower limits can be changed by giving resource hints.

## **Dynamic work rebalancing**

This feature allows Dataflow jobs to dynamically re-partition work based on run time situations. These conditions could be:

1. Unbalanced work assignments.
2. Workers taking longer than normal to finish.
3. Workers finish faster than expected.

Dataflow automatically identifies these situations and dynamically reassigns work to underused workers to decrease the overall time of completion.

Dynamic load balancing process of work happens when the pipeline is processing some data in parallel – reading data from a source and creating Pcollections, or while working on the result of some intermediate data shuffle operation (**GroupByKey**). In case a lot of steps in your Dataflow job are fused, there will be fewer Pcollections in the job, and dynamic load balancing will be restricted to the number of elements in the materialized PCollections. To apply this dynamic load balancing to particular PCollections in the pipeline, you can prevent fusion to ensure dynamic parallelism.

Dynamic load balancing cannot be applied at the record level, that is, if one of your data records is taking time to process, it will still delay the overall job. Dataflow will not try to rebalance the processing associated with a record.

## **Autoscaling algorithms**

In Dataflow, there are three autoscaling modes available. A few of these are also used in the preceding sections. These autoscaling modes are as follows:

## **NONE**

By specifying the autoscaling algorithm as NONE, you are telling the Dataflow pipeline to disable scaling. In other words, you are telling the Dataflow engine to not auto-scale the worker pool. Your pipeline will only have a fixed number of workers running all the time, resulting in over-utilized or under-utilized infrastructure often. By default, the number of workers is 3; use the parameter 'numWorker' to explicitly set it to a specific value.

## **BASIC**

In this strategy, the number of workers becomes equal to the `maxNumWorkers`, until the job completes. This is deprecated now.

## **THROUGHPUT\_BASED**

This strategy when specified, is a signal to the Dataflow job to enable autoscaling. Autoscaling means you start with a certain number of worker nodes, although you can scale up to a certain higher number under peak loads. The strategy auto-scales the worker pool based on throughput (up to `maxNumWorker`).

The definition of throughput is different as per different types of pipelines. For example, a streaming pipeline is based on the number of new incoming records, plus the number of records left out in the last processing, combined with the current capability of processing, to identify the right number of worker nodes. In the case of a batch pipeline, it depends on average CPU utilization.



Be it streaming or batch, the intention is to utilize the assigned resources to the max possible level and reduce the time of processing.

## Scaling and Dataflow Prime right fitting

Dataflow provides the capability to support horizontal as well as vertical scaling. Both flavours of scaling are applied to worker nodes. With Dataprime, you do not need to specify the number, size, and shape of your worker nodes. Dataprime Right fitting feature uses resource hints to customize worker nodes of a pipeline. You can apply this to the complete pipeline, as well as customize individual steps of the pipeline.

The facility to apply infra and scaling configurations at the pipeline step level increases flexibility and capability, in addition to benefitting by potential cost savings. You can easily apply a higher grade of computing for compute-intensive steps and a better memory configuration for memory-intensive steps.

For a Dataflow job to resource hints, it is vital to use Apache Beam 2.30.0 or later. The following resource hints are available:

- `min_ram= "numberOfGB"`

This is the minimum GB of RAM to allocate to workers. Dataflow Prime uses this configuration, when allocating value to new workers (horizontal scaling) or existing workers (vertical scaling). This is the memory Dataflow and it is going to apply per node and not per CPU. For example, you can configure `min_ram=20GB`, such that the Dataflow will set the worker node memory to be 20 GB distributed across all the vCPUs.

- `accelerator="type:type;count:number;configuration-options"`

This property expects value in format of the GPU type, number of GPUs, and GPU configuration options to use, separated by a semi colon. This user-supplied configuration helps the user to set the base value of infrastructure, to compute the power of each worker node. When a scaling up happens, the computing power will increase in a discreet amount of these values.

You can configure the resource using the following parameters while triggering a pipeline:

```
--resource_hints=min_ram=numberGB \  
--resource_hints=accelerator="type:type;count:number;install-  
nvidia-driver"
```

You can configure them programmatically as well.

## Limiting max nodes

By default, machine type for batch job is **n1-standard-1**; for streaming engine enabled job, it is **n1-standard-2** and default machine type of non-streaming engine jobs are **n1-standard-4**. Hence, while using default machine types, a Dataflow service can scale up to maximum of 4000 cores per job. If your use case needs a larger machine, you can select a larger machine type.

To allocate additional memory, you can opt for custom machine type with extended memory. For example, **custom-4-15360-ext** is an n1 machine type with 4 CPUs and 15 GB of memory. Dataflow identifies the number of worker threads per worker VM, by looking at the number of cores in a VM. A memory intensive workload can use a custom VM with lower cores and high memory assigned, per CPU leading high memory, per worker thread.

Dataflow billing is done by a number of vCPUs and GB of memory in workers. Billing is independent of the machine type. However, a single Dataflow job generally never suffices to all business use cases. When you have multiple

Dataflow jobs, it is always advisable to configure a maximum limit per Dataflow job, or else one Dataflow job can really start consuming vCPU at a level that the other Dataflow jobs start starving for infrastructure. As a best practice, make sure that you have sufficient quota in your project, enabled to successfully tackle cumulative max scaling of all your workloads, because that is the worst-case scenario when each application scales up to maximum.

## **Scaling the persistent disk**

Dataflow job attaches at least one persistent disk per instance. The default values are as follows:

- The default disk size for batch jobs is 250 GB and for streaming pipelines, it is 400 GB.
- The disk size for Dataflow shuffle batch pipelines is 25 GB.
- Disk size for streaming engine streaming pipelines is 30 GB.

When you decide upon the configuration of autoscaling of your Dataflow jobs, do take into consideration the minimum/default needs of each of your Dataflow jobs. You need to make sure that there is a sufficient quota of persistent disks available for the Dataflow jobs to execute.

## **Optimizing data shuffle using Dataflow shuffle**

Dataflow transforms like `GroupByKey`, `CoGroupByKey` and `Combine` require a lot of data to be shuffled, so that data of the key comes together for aggregation. This data shuffle historically has been the choking point of all distributed applications if not done correctly. Dataflow's Shuffle operation partitions and groups data by key in an efficient,

fault-tolerant, and scalable manner. This is available only for batch pipelines and it moves the shuffle operation out of the worker VM and into the Dataflow service backend.

The use of Dataflow shuffle service facilitates the following aspects:

- Reduction in CPU and memory, and persistent disk usage on worker VMs.
- Faster execution time.
- Efficient horizontal scaling as virtual machines are no longer holding shuffle data and can be scaled down easily.

For the majority of workloads, a job running using a data flow shuffle is expected to be executing faster and incurring lower costs, versus where the data shuffle is happening on the worker VMs. There is a charge associated with using the data shuffle service, but by using the service, the overall cost is not expected to be different.

Dataflow shuffle service is also quota bound. When there is an increase in the number of jobs using the Dataflow service, make sure that you have sufficient shuffle slots available to prevent starving for resources. In regions like **Europe-west1** and **us-east1** there are 160 slots available, which could easily shuffle 100 TB of data concurrently.

## **Conclusion**

Dataflow is the serverless (and most cost-optimized) way to handle your big data use case, which requires the processing of TB and PB of data. When we talk about scaling Dataflow, we are only talking about passing proper values to configurations of the Dataflow runner. Dataflow along with its custom optimizations (optimizations not provided by other runners), tries to make Dataflow efficient and resilient. Moreover, Dataflow has features like *Fusion*, *Streaming*, and

*Dataflow Shuffle*, all optimized to handle the workload with huge datasets.

Any of the scaling changes related to Dataflow optimization can generally be passed as arguments, while triggering the pipeline. Hence, no business logic change is expected while optimizing the execution environment.

## **Points to remember**

- Dataflow is a serverless runner platform (an execution platform) to Apache Beam pipelines managed by GCP.
- Dataflow provides multiple optimizations to tackle big data use case problematic areas, making it an extremely resilient and efficient system to host workloads.
- Dataflow jobs can scale to a very high number of worker nodes, and such high numbers involve a lot of costs if not configured well for scaling.
- Dataflow provides good source and sinks options for various GCP services like pub-sub, big query, cloud storage, and so on.

## **Questions**

1. Which of the following is NOT true about Dataflow pipelines?
  - a. Dataflow pipelines are tied to Dataflow, and cannot be run on any other runner.
  - b. Dataflow pipelines can consume data from other Google Cloud services.
  - c. Dataflow pipelines can be programmed in Java, Python and Go.
  - d. Dataflow pipelines use a unified programming model, and can thus work both with streaming and

batch data sources.

2. You cannot configure a machine with custom compute power for your Dataflow workers. True or False?
3. Batch jobs can undergo vertical scaling under exceptional circumstances of memory consumption going above 90%. True or False?
4. You need to make code changes to configure the autoscaling options to you pipelines. True or False?

## **Answers**

1. a. Dataflow pipelines are Apache beam pipelines, and Dataflow is just a runner. So, the same pipeline could be executed on other platforms like Spark.
2. False. You can configure compute and memory assigned to each worker node.
3. False. Vertical scaling is only for streaming jobs.
4. False. You can do it programmatically, however, best practice is to send parameters while triggering a job.

# CHAPTER 15

## Site Reliability Engineering

### Introduction

The term **Site Reliability Engineering (SRE)**, conceptualized by Google Engineer *Ben Treynor* in 2003, aims to increase the reliability of the sites and services that a team offers. According to Treynor's definition, "*SRE is what happens when you ask a software engineer to design an operations team.*"

At the core, an SRE team is a group of software engineers who build and implement software to maintain the reliability of system/services.

Consider an example of a service: an online banking system that ideally has hundreds of financial transactions happening every second. If this service goes down frequently, it is considered a loss of business for the bank. A new role Site Reliability Engineer is introduced in software industry, whose main task is to make sure that the reliability of the system is maintained without an impact on the velocity of new features rolling out.

In a traditional system, there were two parties. One was the developer, whose aim was to build new features at a fast pace, and the other was the Operator, whose intent was to deploy the new features without compromising the stability of the system. Because of the different intents, there was always a conflict between the two teams. To resolve this, a new role of DevOps was introduced. While DevOps culture did talk about deploying code fast and in high frequencies, reliability and stability were not generally given importance. In the DevOps team, you will not hear about a persona whose responsibility is to ensure system reliability. This is where the SRE came in.

At this point, you need to think about which actions can make a system/services unreliable. The major reason behind this is the change in platform, change in services themselves (new features),

infrastructure changes, and so on. The solution to this could be that no changes be made or limiting the number of changes to the system. However, that limits the business. Instead, we want to write more code and develop new features quickly, making the application better and thus, increasing business value. SRE tries to automate the process of evaluation of the effects the changes will have. Since this automation has no discussions with other teams and no manual checks, it makes releasing the changes fast and safe. In this chapter, you will look into how to build and deploy the automation aspects of assessing the effectiveness of changes.

## **Structure**

In this chapter, we will discuss the following topics:

- Introduction to SRE process
- Defining SLO, SLI and SLAs
- Service monitoring using Google Cloud Monitoring
- Selecting metrics for SLIs
  - Out of the box platform metrics
  - Log based metrics
  - Key SLIs
- Setting SLO
- Tracking error budgets
- Creating alerts
- Probes and uptime checks
- Aggregating logs to set up cloud monitoring dashboard
- Responsibilities of SRE
  - Incident management
  - Playbook maintenance
  - Drills
- Automating SRE actions

## **Objectives**



After studying this chapter, you will learn and understand the basics of Site Reliability Engineering in general. You will learn in detail how to build and host the SRE automation stack on the Google Cloud Platform. You will investigate the ‘*how*’ of aspects like selecting metrics, setting objectives, tracking, alerting, and so on. In the last section, we will discuss more about the responsibilities owned by SRE teams.

The chapter does not intend to make you an expert in SRE practices, rather, this will make you an expert in handling the infrastructure part of SRE. The right value allocation to metrics, the frequencies of the run, and other aspects defining the reliability of the system is out of the scope of this chapter. We will also look into the aspects of scaling the stack as well as how the tech stack behaves when the applications scale.

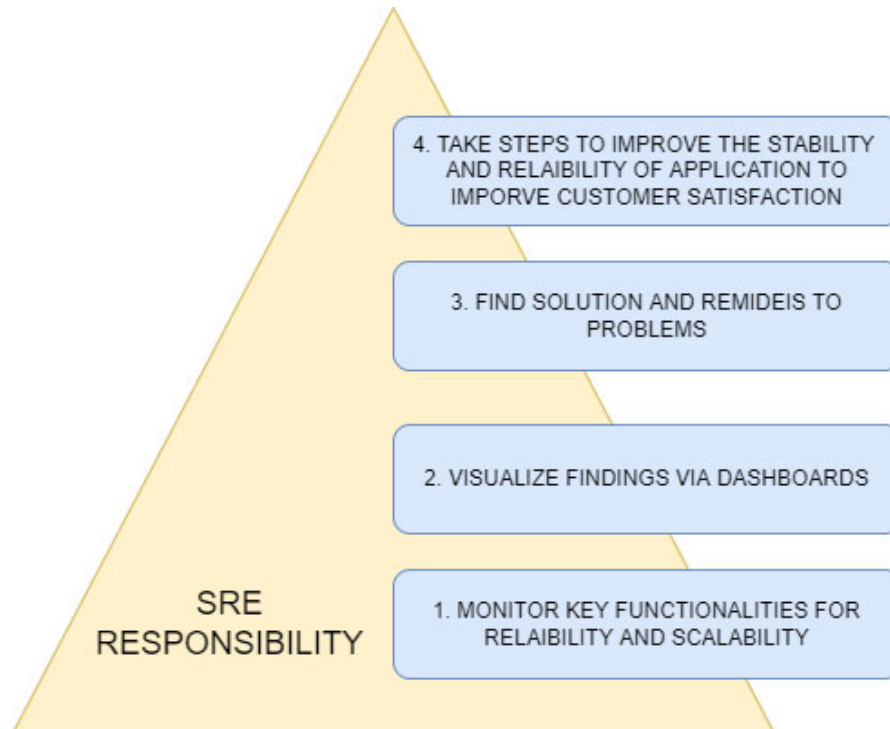
## **Introduction to SRE process**

The SRE team’s primary task is to concentrate on engineering new products and services by automating and scaling applications and production environments. Even though SRE concentrates on operational stability, it helps reduce the friction of handovers from product development teams. When it comes to SRE, there are a few core principles that each SRE team follows:

- You cannot get away from failures; they are bound to happen. However, you can learn from them.
- Automate wherever it is possible, to minimize manual works/reworks.
- The SRE team and the engineering team work together to find the issues that lead to system breaks. No blaming is involved. Both parties are equally responsible for the proper running of the system.
- While a lot of effort could be spent on making the service more reliable, it is important to put in the right amount of effort that satisfies the end user of the service. Efforts saved could be used elsewhere.

## **Defining a typical SRE process**

There is no definition of what falls in the SRE process and what does not. However, looking at various use cases, a typical SRE process is more of a set of the best practices. Consider the [Figure 15.1](#), which shows a very high-level responsibility for SRE:



**Figure 15.1:** SRE responsibility

These best practices include the following:

- The SRE process involves monitoring data using some monitoring tool like *Datadog*, *Prometheus*, *Nagios*, and so on, to collect information from the system on how it is performing. Automated alerts are used when an abnormality is detected.
- When something goes wrong, SRE teams manage and apply the backup plan to deal with the issue and restore the appropriate state. This backup plan is known as **runbooks**.
- Performing retrospectives from incidents, so that the SRE team learns and matures into better handlers of such issues in the future.
- By looking at an incident, the SRE team intends to identify the impact on customers with SLIs and SLOs. This helps them to frame responses about such mishaps in the best possible way.

- SRE teams can easily facilitate and help in deciding how fast the development team can move with releasing new features. They utilize the error budgets to make the best decision for the customers.

The preceding practices help the teams in releasing newly developed software on production without impacting customers. Each SRE team matures with time, learning from the type, frequency, and cause of any issue, resulting in better handling of unwanted situations for your clients.

## Defining SLO, SLI and SLAs

SRE does automated evaluation of the reliability, by bringing in a change in the system/services. In this section, we will learn about the three most common terminology used in SRE worlds.

### Service Level Objectives (SLO)

Availability is key to successful system/services. A system that is not available cannot do what it is intended for, and by default, will fail. Availability in terms of SRE means, service/system performing the intended tasks. We define a numerical percentage, which is the threshold target for availability. Under no circumstances, can the service be below this amount of availability. This threshold is known as **Service Level Objective (SLO)**. All non-functional requirements are defined with the intent of at least maintaining or improving the SLO for the service.

It might look like it is best to keep the SLO at 100%, and thus we all aim for that. However, to do so, there is a generally very high cost associated. Not just the cost, but even the technical complexity increases exponentially as we inch closer to 100%. The rule of thumb is to define the lowest possible value that you can get away with, for each service. Many teams implement periodic downtime to prevent the service from being overly available. This downtime of services can be beneficial in identifying the inappropriate use of services by other services.

The value of this metric is usually decided in collaboration with product owners and with the engineering team in advance. This

results in lesser confusion and conflicts in expectations in the future.

## **Service Level Indicators (SLI)**

We need to make sure we understand availability and we should have clear numerical indicators for defining availability. How we do that is not just by defining service level objectives, but **Service Level Indicators (SLI)**. SLIs are more often metric over time. Example metrics could be to request latency, batch throughput in case of batch systems, and failures per request. These metrics are aggregated over time, and we typically apply a function like a percentile or median. That is how we learn whether a single number is good or bad. An example of a good Service Level Indicator could be is the 99<sup>th</sup> percent latency, of the latency of request over the last 5 minutes is 300 milliseconds. Another example could be that the ratio of error requests to the total number of requests over the last 5 minutes, is less than 1%.

These numbers aggregated over a longer time, let us say a year, will tell for how much time the application was down. If the total downtime comes out to be less than 9 hours over a year, it corresponds to 99.9 percent availability.

## **Service Level Agreement (SLA)**

This is a commitment between the service and the customer, about how reliable the service/system is going to be for end users. It is promised that the availability SLO will not go below this number or else there will be a penalty of some kind.

Because of the penalties involved, the availability SLO in **Service Level Agreement (SLA)** is a looser objective in comparison to internal SLO. The teams should feel 100% confident in maintaining this availability. You may have an agreement with a client that the service will be available 99% of the time each month. You can configure an internal SLO to be 99.9% and configure alerts when this number is breached.

The calculation is as follows:

- **SLA with clients:** 99% availability – 7.31 hours of acceptable downtime per month.
- **SLO:** 99.9% availability – 43.83 minutes of acceptable downtime per month.
- **Safety buffer:** 6.58 hours.

No time could be enough in case the service faces a major disruption. However, the bucket of 6+ hours above between the internal and external objective provides peace of mind when you deploy.

An **error budget** is the maximum amount of time that a technical system can fail without contractual consequences. In the preceding example, the time of 43.83 minutes which is the acceptable downtime per month, is also known as the error budget.

## **Service monitoring using Google Cloud Monitoring**

In the subsequent sections, you will see all the SRE philosophies discussed above, as actual technical components and how they look in **Google Cloud Platform (GCP)**. You will also see a few of the concepts getting extended more as per the needs. This section will give a quick insight into Google Cloud Monitoring. This hands-on exercise will give you insights into how you can use cloud monitoring in your projects. Google acquired StackDriver in the year 2014 and since then, the whole suite of operations is coming out very well. Cloud monitoring is built in the cloud console.

For this exercise, follow the given steps:

1. Create a small virtual machine with the name **quickstart**.
2. Install a web server.
3. Install a monitoring agent.
4. Create uptime check dashboards and extend for other metrics.
5. Enhancing the basic monitoring configuration.
6. Simulate load and see patterns on pre-configured dashboards.

## 7. Creating an alerting policy.

Following are the in-depth details of aforementioned steps:

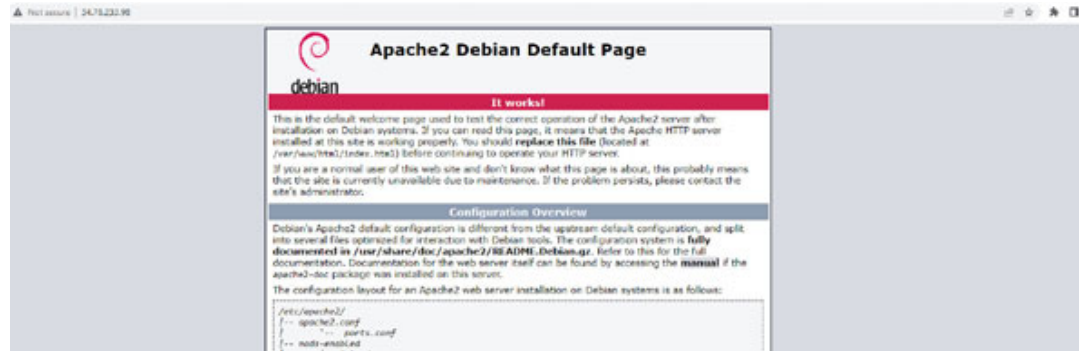
### 1. Creating a virtual machine.

- a. Go to the GCP console | **Compute Engine** | **Instance**. Click **New Instance**.
- b. Now fill in the fields for your instance. In the **Name** field, enter **quickstart-monitoring**.
- c. In the Machine type field, select the smallest possible configuration. This is just a demo application and the performance of a machine is not what we are looking for.
- d. Ensure that the Boot disk is configured for Debian GNU/Linux.
- e. In the firewall field, select both Allow HTTP traffic and Allow HTTPS traffic.
- f. Leave the default values for the rest of the fields.
- g. If doing it from the UI is not possible, use GCloud command "**gcloud compute instances create**" to create the virtual machine.

### 2. Install an Apache Web server in the virtual machine.

- a. **Secure Socket Shell (SSH)** into the virtual machine and to install an Apache2 HTTP Server, run the following command:  

```
sudo apt-get install apache2 php7.0
```
- b. Open the browser and open the URL **http://EXTERNAL\_IP**. **EXTERNAL\_IP** is the external IP of the virtual machine. This will open a page similar to [\*Figure 15.2\*](#):



**Figure 15.2:** Apache2 Debian Default Page

### 3. Install the google monitoring agent.

- a. Run these two commands to download and update monitoring agent.

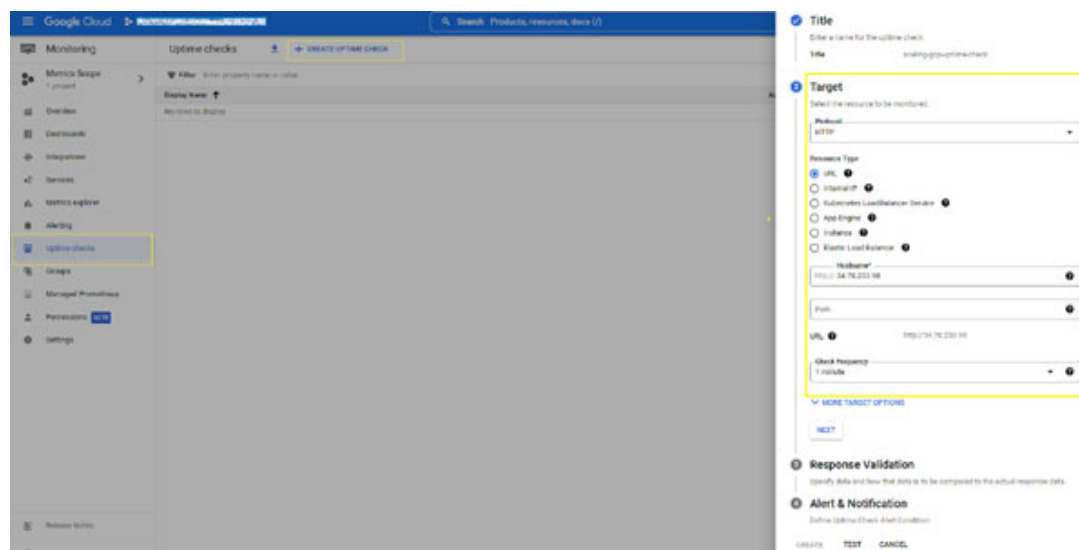
```
curl -sS0 https://dl.google.com/cloudagents/add-google-cloud-ops-agent-repo.sh
```

```
sudo bash add-google-cloud-ops-agent-repo.sh --also-install
```

You will get a message: “google-cloud-ops-agent installation succeeded.”

4. With the environments set up done, you can now **configure cloud monitoring**. For this, let us first **configure uptime check**, that will monitor web service and will inform if it goes down.

- a. Go to the Cloud Monitoring | Uptime checks | Create Uptime checks. Create a new one, as shown in [Figure 15.3](#):



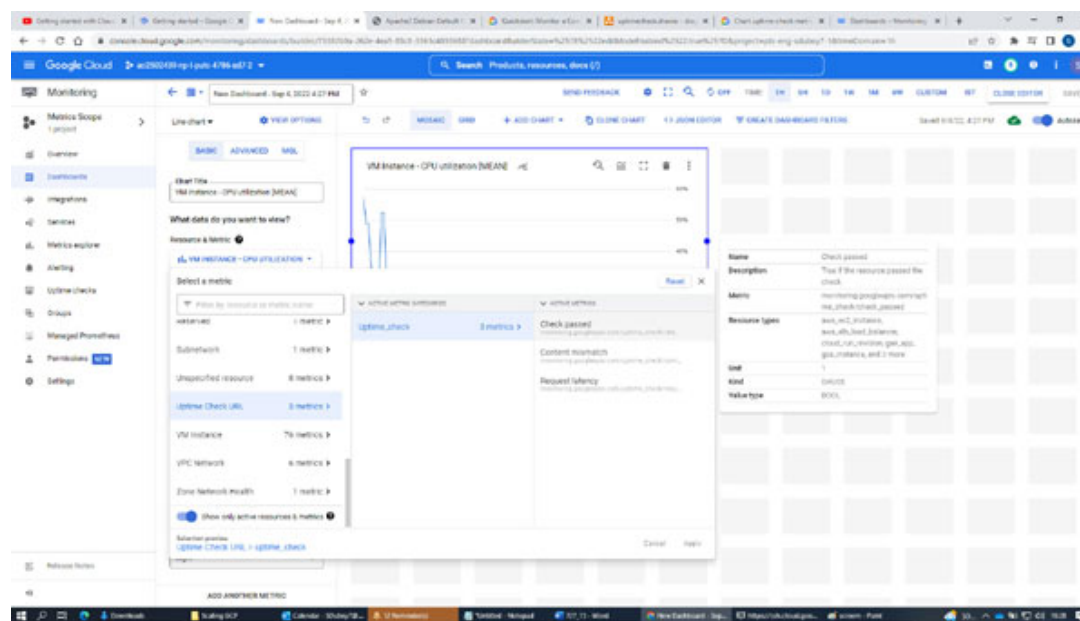


**Figure 15.3:** Uptime checks

In the preceding [Figure 15.3](#), look at the highlighted sections. In the Target type of the uptime check, mention the external IP of your virtual machine. Keep the rest of configuration as it is. Click on **Create**.

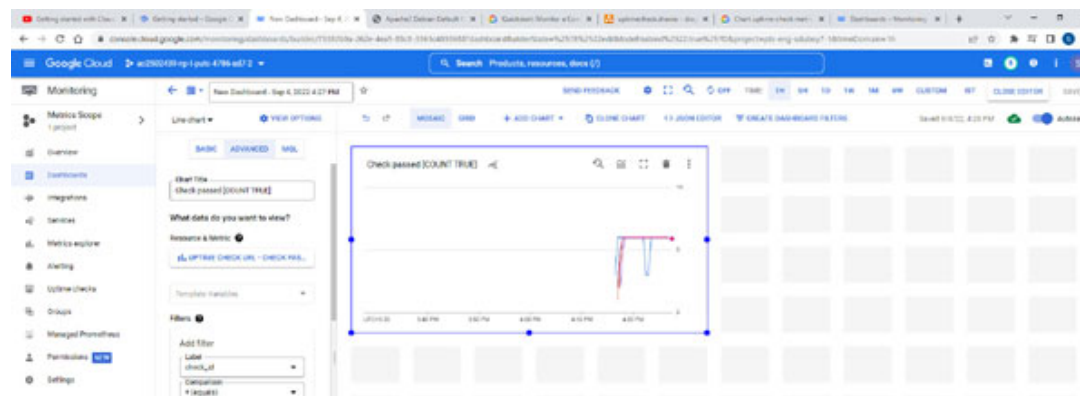
Now let us **configure dashboards** for the uptime check stats. To do the same, follow the given steps:

**STEP 1:** Go to **Dashboards** | Select the Metric as **Uptime Check URL** | **Uptime\_check** | **Check Passed**, as shown in the [Figure 15.4](#):



**Figure 15.4:** Configuring uptime

**STEP 2:** Add the filter. **Label** = **check\_id**, comparison as **=** and value equal to the uptime check which we created, as shown in the following [Figure 15.5](#):

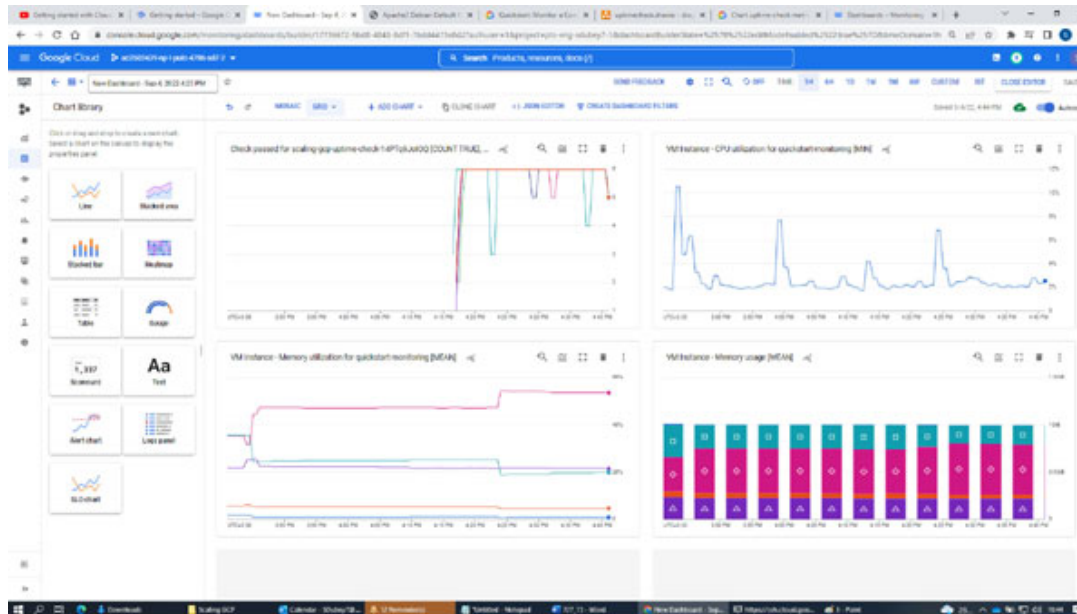






**Figure 15.5:** Configuring uptime

**STEP 3:** You can repeat step number 2 any number of times and various available metrics, out of the box. In our case, we further extended the dashboard to track the following metrics for the virtual machine (**quickstart-vm**) – minimum CPU utilization, mean memory utilization and mean memory usage. Refer to [Figure 15.6](#):



**Figure 15.6:** Configuring uptime

## 5. Enhancing the basic monitoring configuration.

For the purpose of enhancing the basic monitoring configuration, we were using the Ops agent with the following configuration to update the web server. More information can be seen at [\*\*https://cloud.google.com/logging/docs/agent/ops-agent/third-party/apache\*\*](https://cloud.google.com/logging/docs/agent/ops-agent/third-party/apache). Please refer to the following code:

```
logging:
  receivers:
    apache_access:
      type: apache_access
    apache_error:
      type: apache_error
  service:
    pipelines:
      apache:
        receivers:
          - apache_access
          - apache_error
metrics:
  receivers:
    apache:
      type: apache
  service:
    pipelines:
      apache:
        receivers:
          - apache
```

Edit the file `/etc/google-cloud-ops-agent/config.yaml` with the preceding configurations and restart the web server.

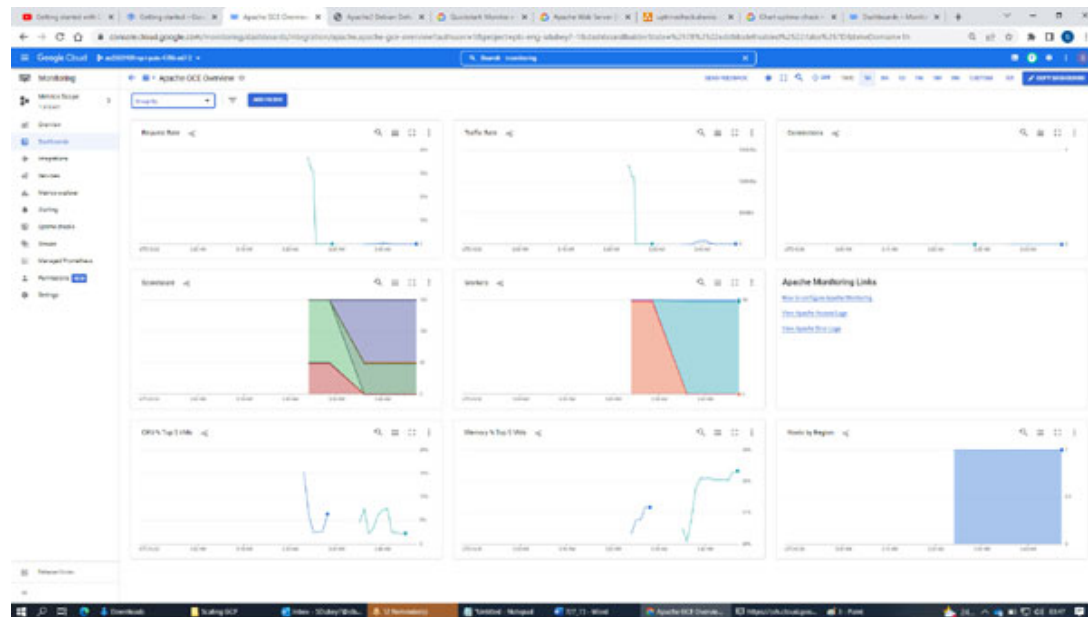
```
sudo service google-cloud-ops-agent restart
```

## **6. Simulate load and see patterns on pre-configured dashboards.**

Simulate the load by doing SSH in the virtual machine and generating a load by using the following curl command.

```
timeout 180 bash -c -- 'while true; do curl localhost; sleep 2 ; done'
```

Go to **Monitoring | Dashboard | Apache GCE Overview**, and you will see multiple charts in the dashboard, showcasing details of the virtual machine running the Apache Webserver, as shown in [Figure 15.7](#):

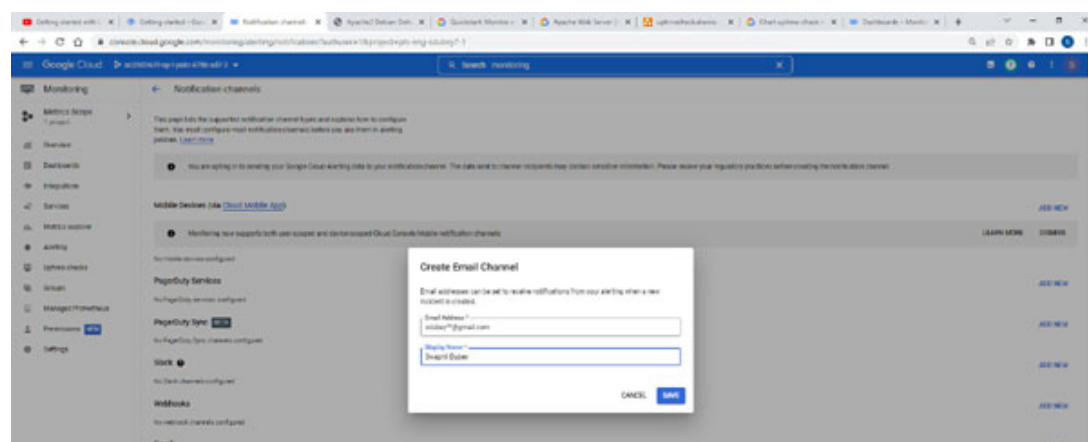


**Figure 15.7:** Monitoring dashboard

## 7. Creating an alerting policy.

Let us first create a notification channel. Whenever the conditions are met, an alert will be generated. For this, you can use various options such as Slack, Email, SMS, Pager Duty options and so on.

In this case, as an example, let us go ahead with email. Go to **Monitoring | Alerting | Edit Notifications** channel, as shown in [Figure 15.8](#):

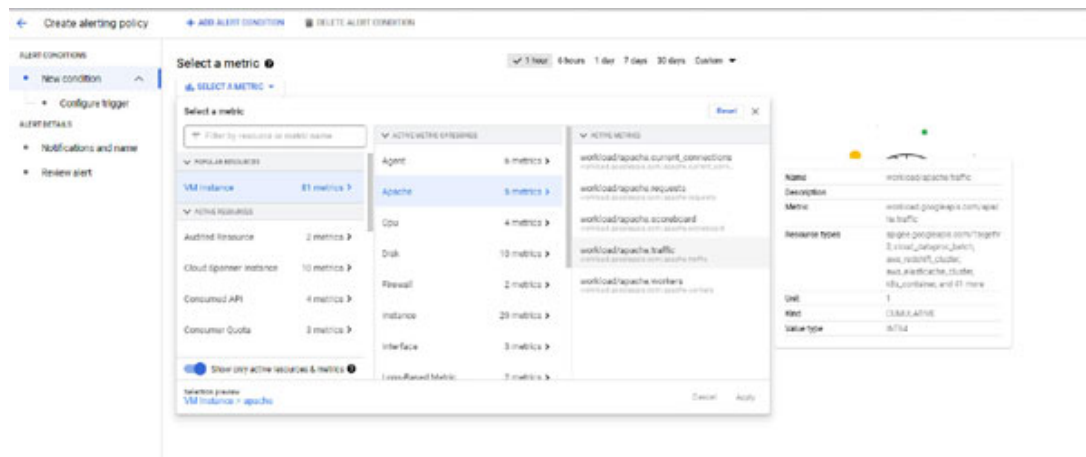




**Figure 15.8:** Alerting channel

As the last step of the exercise, let us go ahead and configure the alerting. Go to **Monitoring | Alerting**, and create a policy.

**STEP 1:** Select a metric | VM instance | Active Metric Categories List | Apache | Workload/apache.traffic, as shown in [Figure 15.9](#):



**Figure 15.9:** Selecting metrics

**STEP 2:** In the **Transform data** pane, set rolling window as 2 minutes and rolling window function as rate.

**STEP 3:** In the **Configure alert trigger** pane, configure Alert Trigger as Any time series violations, threshold position as Above threshold and Threshold value as 1000.

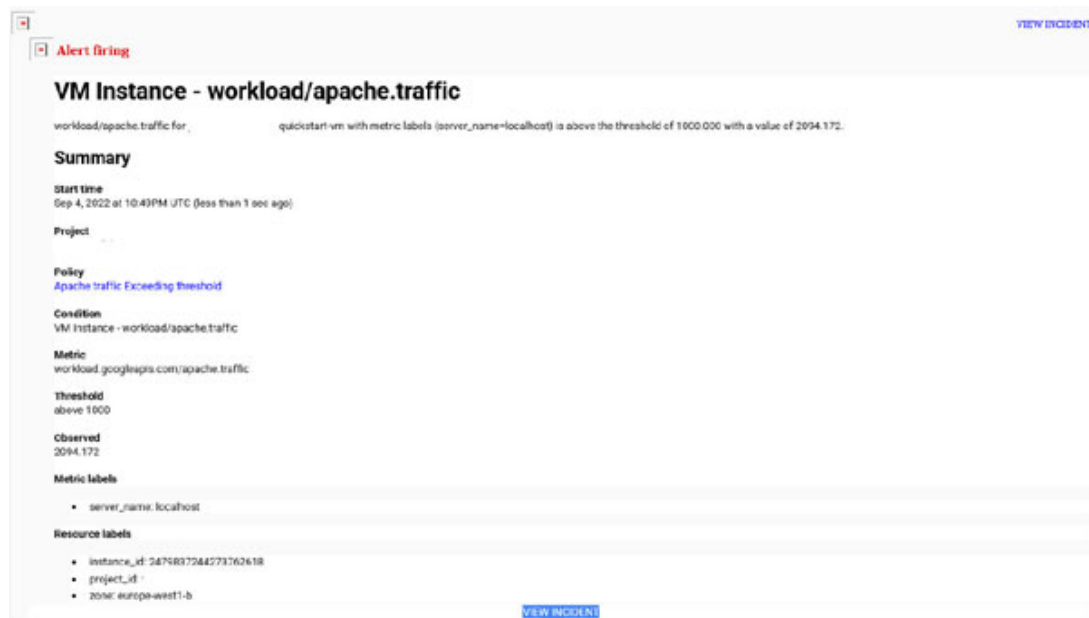
**STEP 4:** In the **Configure notifications and finalize alert** section, configure the notification channel with notification as email, incident close duration as 10 minutes and the name of alert policy as Apache traffic Exceeding threshold and create the policy.

Refer to the following [Figure 15.10](#):



**Figure 15.10:** Configuring alert

Regenerate the load again. After the load trigger completes, an alert mail is generated, as shown in [Figure 15.11](#):



**Figure 15.11:** Firing alert

In the preceding exercise, we investigated the dashboarding and alerting capabilities of Google cloud monitoring. The same approach can be replicated to other workloads, where the application can emit metrics and these metrics can be grouped

over a period. The aggregated value can be plotted in a dashboard, and we can generate alerts when the aggregated value reached thresholds.

## Selecting metrics for SLIs

In this section, we will look into the various Service Level Indicator metrics that you ideally should track for an application. Service monitoring and the SLO API enables us to manage your services the way Google manages its services. The steps involved in this activity are as follows:

1. Identifying metrics that can be used for service-level indicators.
2. Using the SLI to set service level objectives for the SLI values.
3. Identifying risk by using the error budget implied by the SLO, to handle risk in your application.

Let us discuss the first point in more detail. SLI is a quantifiable measure of service reliability. There are two kinds of SLIs – SLIs available out of the box from GCP, and custom SLIs. Let us see how you can leverage both approaches.

## Using the out of box SLI metrics

SLIs are represented as ratios or percentages of good events to the total events. This helps us normalize various measures of availability and set reliability targets in a uniform way across the services. Generally, SLIs fall into two categories: request based and window based.

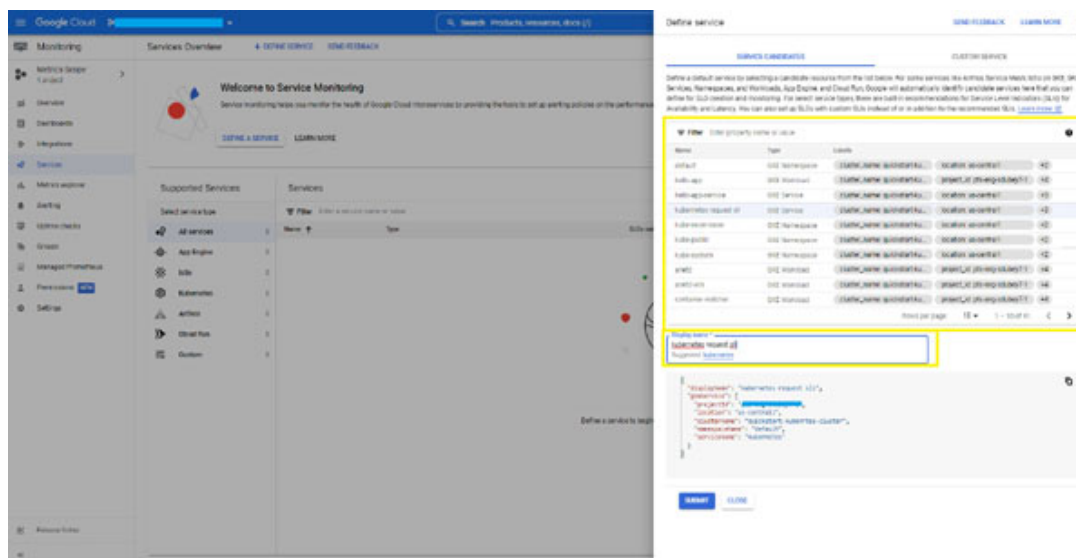
A **request based SLI** is the measure of good requests to the total requests. For example, the number of HTTP requests was successful to the total request made in each time window.

**Window-based SLI** measures the fraction of time intervals during which, service meets the threshold of reliability over a given time window. Now there could be scenarios where we need to define altogether new metrics for the SLIs. However, here we will discuss the metrics made available by GCP out of the box. We will see an example of Kubernetes workload. Kubernetes emits metrics that could be used as SLIs. These metrics available are available out of

the box, that is, the moment to configure a Kubernetes cluster, cloud monitoring will have the metrics available.

As an example, consider a hello world app deployed on Kubernetes (refer to [Chapter 6, Scaling Kubernetes](#)). We will use a very simple curl command to generate some loads continuously for 5 minutes.

Go to **Monitoring | Services | Define service**, as shown in [Figure 15.12](#):



**Figure 15.12:** Out of Box SLI metrics

The moment you click on **Define service**, a pane opens with all possible metrics in your project. Select an appropriate filter (Kubernetes GKE service in this case), and click **Submit**. Immediately a service is created with dashboards.

The preceding operation from UI console can also be accomplished by REST APIs.

```
{
  "name": "projects/197402864353/services/ist:<PROJECT_ID>-location-
us-central1-quickstart-kubenrtes-cluster-default-kubernetes",
  «displayName»: "kubernetes request sli",
  «telemetry»: {
    «resourceName»:
    «//container.googleapis.com/projects/<PROJECT_ID>/locations/us-
central1/clusters/quickstart-kubenrtes-
cluster/k8s/namespaces/default/services/kubernetes"
  },
  «gkeService»: {
```

```

«projectId»: "<PROJECT_ID>",
«location»: «us-central1»,
«clusterName»: "quickstart-kubernetes-cluster",
«namespaceName»: "default",
«serviceName»: "kubernetes"
}
}

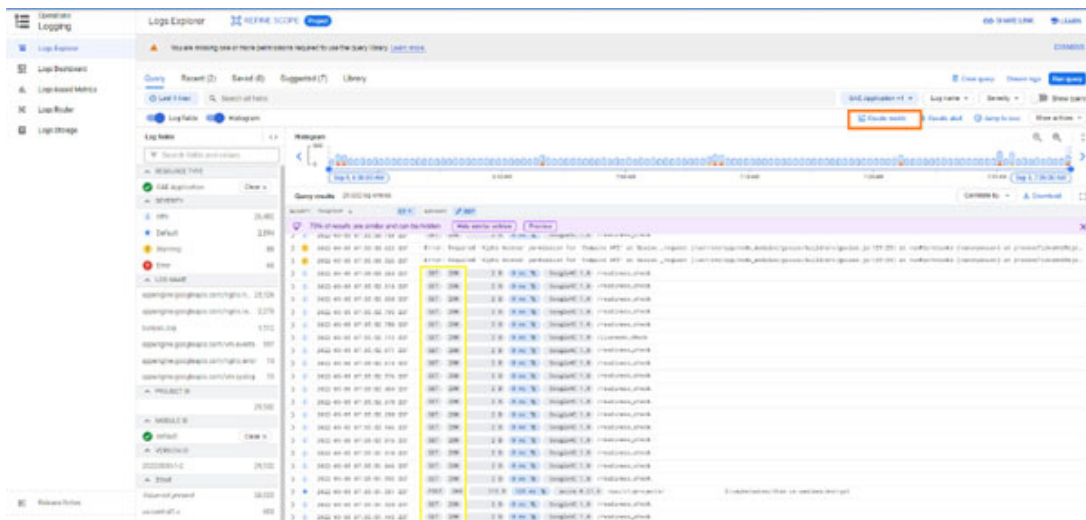
```

We have seen how to use platform metrics. Now in the next section, we will dive deep into log-based metrics or custom metrics.

## Log based metrics

These are the cases when none of the out of box metrics work for us. In such a scenario, we can use the log-based metrics. This could be done by using cloud logging, cloud monitoring and service monitoring. In the following exercise, you will see how to create a metric for a REST API.

Consider the following cloud logging screen of a REST API, shown in [Figure 15.13](#). The API returns various http status codes of 2xx and 4xx with the obvious meanings (highlighted in yellow). Please refer to the following figure:



**Figure 15.13:** Out of Box SLI metrics

You can create a metric by clicking the option on the right top (highlighted in orange). This will open a pane with the option to



start metric creation, as shown in [Figure 15.14](#). Select the metric type and assign a name to the metric. Please refer to the following figure:

Operations  
Logging

Create log-based metric

Configure the settings below to define and create a log-based metric.

**Metric Type**

☒ Counter  
Counts the number of log entries matching a given filter.  
[Learn more](#)

☐ Distribution  
Counts numeric data from log entries matching a given filter.  
[Learn more](#)

**Details**

**Log entry count**  
Log entry count  
http.request.status

**Description**  
Enter a description for this metric (optional).

**Unit**  
The units of measurement that apply to this metric (for example, bytes or seconds). For counter metrics, enter the units in step 1. For distribution metrics, you can optionally enter units, such as s, but not [count/second](#).

**Filter selection**  
Define your log-based metric.  
[Preview filter](#)

**Build filter**  
Press shift to see auto-suggested filters.

```
1 resource.type = "http_request"
2 resource.labels.status != 200
```

**Figure 15.14: Log based metric**

Assign a label name according to your wish, and then a label type and the field name. Consider the following [Figure 15.15](#). The entries in the Label section counts the number of http request status. Please refer to the following figure:

Operations  
Logging

Create log-based metric

**Description**  
Enter a description for this metric (optional).

**Unit**  
The units of measurement that apply to this metric (for example, bytes or seconds). For counter metrics, enter the units in step 1. For distribution metrics, you can optionally enter units, such as s, but not [count/second](#).

**Filter selection**  
Define your log-based metric.  
[Preview filter](#)

**Build filter**  
Press shift to see auto-suggested filters.

```
1 resource.type = "http_request"
2 resource.labels.status != 200
```

**Labels**  
Labels allow log-based metrics to count multiple time series. [Learn more](#)

**Label name**  
http.request.status

**Label type**  
COUNTER

**Field name**  
http.request.status

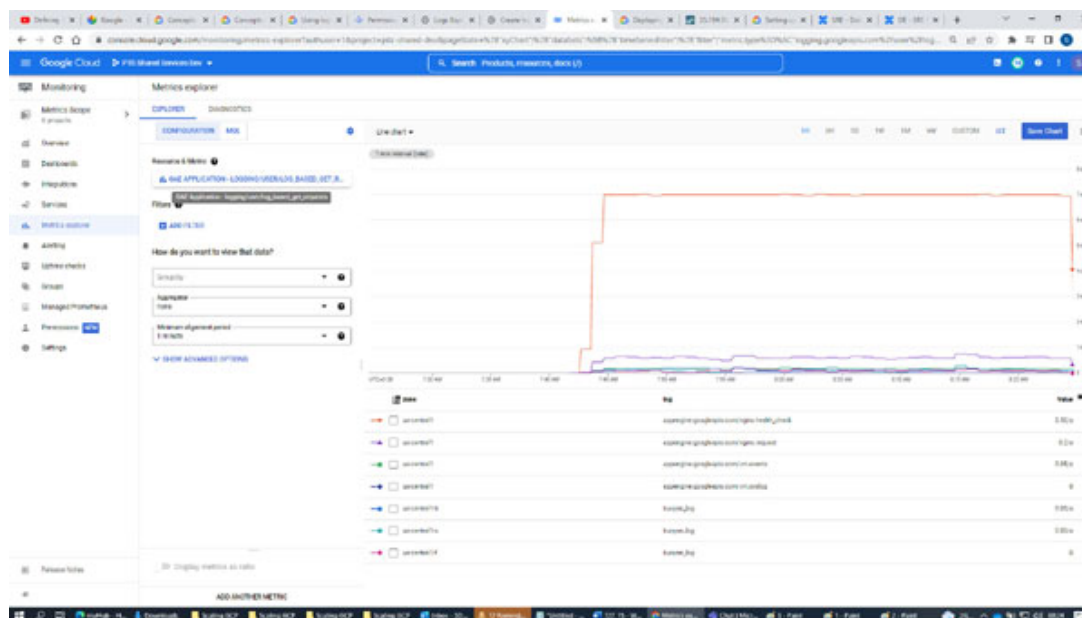
**Regular expressions**  
A regular expression to filter log entries that match the filter. [Preview](#)

**DONE** **CANCEL**

**CREATE METRIC** **LINKS**

**Figure 15.15: Log Based Metric**

Let the system run for few minutes and once the metrics are available in the metric dashboard, you can create the dashboard in the **Monitoring | Metric Explorer** tab, as shown in [Figure 15.16](#):



**Figure 15.16:** Log based metric dashboard

After the preceding step, we have to now create the step of making the SLIs.

## Key SLIs

Though the number and type of metric depends on your use case, as there is no limit to how many metrics/SLI there can be, there are three major categories of such Service Level Indicators that are most widely used in industry. These are, availability, latency, and correctness, and they are discussed as follows:

- **Availability**

An extremely important SLI is availability, which is the fraction of time the service was unusable. It can also be described as the ratio of the number of successful responses, to the total number of responses. Although various enterprises aim for high value for this SLI, achieving 100% is near impossible, and is expressed in terms of “nines” in availability percentage. For example, the availability of 99.9% and 99.99% can be referred to as three and four nines.

- **Latency**

Most use cases consider request latencies, that is, how long it takes to return with a response as key SLI. Sometimes client-side latency is more often the metric that impacts the end user. Nonetheless, sometimes it is only possible to measure latency, as the non-measurable part is out of the boundary of SLI calculation. For example, network slowness, or API gateway latencies.

- **Correctness**

Correctness is the measure of how correctly your application is behaving within a given time. You must trigger a synthetic workload at a periodic interval and check the outcome of the processing of the workload with configured result expectations.

- **Error Rate**

Error rate is defined as the ratio of requests that errored out to the total number of requests.

- **Data Freshness**

In case of an update of data to a system, how quickly is the update available across all queries? For example, if a customer does a hotel booking and opts for a room, how quickly is this information available to others so that they do not book the same room again?

There are a few more SLIs such as throughput and durability, which are self-explanatory. Although we can measure lots of such metrics, it would be best if we do not have more than a handful of SLIs to measure. It is best to stick to 4 or 5 SLIs that directly relate to customer satisfaction. Good SLI ties with user experience, that is, a low SLI will represent low customer satisfaction and a high value represents more satisfied customers. If an SLI fails to achieve that, there is no point in capturing and measuring that SLI.

## **Setting SLO**

SLOs are the targeted SLI indicator values that the engineering team intends to achieve. The time window due to the difference in the numbers of SLA and SLO depends on the confidence of the

The given steps follow the complete SLO creation process:

- Point numbers 1, 2, and 3 are points where the whole engineering team collaborates with Business Team to identify the numbers. In this section, you will investigate how you will perform step 4 on the examples which we had already seen in the chapter. For this, we will use the log-based metrics example. Following are the steps to configure an SLI and then an SLO.

To configure an SLI, follow the given steps:

- 
- The screenshot shows the Google Cloud Service Monitoring console. On the left, the 'Services' menu item is highlighted with a yellow box. In the main content area, the 'Define service' dialog is open. The 'Service name' field is highlighted with a yellow box. Below it, the 'Create a new service' button is also highlighted with a yellow box. The dialog includes a 'Service description' field and a 'Create' button.

**Figure 15.17: Creating SLI**

2. The first thing that we must select is a metric. The metrics available out of the platform has pre-configured metrics of availability and latency. However, since we are using custom metrics, there is an option to configure your metric using the other option. The second thing is configuration of weather; this metric is going to be count based or window-based, as shown in [Figure 15.18](#):

The screenshot shows the 'Create a Service Level Objective (SLO)' interface. On the left is a sidebar with four steps: 1. Set your service-level indicator (SLI), 2. Define SLI details, 3. Set your service-level objective (SLO), and 4. Review and save. Step 1 is currently active. The main content area is titled 'Set your SLI' and includes a sub-header 'Service details' with a table showing 'NAME' as 'Custom Log Based SLI', 'TYPE' as 'Custom', and 'LABELS' as 'project\_id: 927110120163'. Below this is a message box stating that default availability and latency metrics are not available for Custom services. The 'Choose a metric' section has three radio buttons: 'Availability', 'Latency', and 'Other (advanced)', with 'Other (advanced)' selected. The 'Request-based or windows-based?' section has two radio buttons: 'Request-based' and 'Windows-based (advanced)', with 'Request-based' selected. At the bottom are 'CONTINUE' and 'CANCEL' buttons.

NAME	TYPE	LABELS
Custom Log Based SLI	Custom	project_id: 927110120163

**Choose a metric**

☐ Availability  
Measures how available your service was to users. You'll get a metric related to how many requests were successful within a time period that you define.

☐ Latency  
Measures how quickly your service responded to users. You'll get a metric related to how many responses were faster than a threshold that you define.

☒ Other (advanced)  
Configure your own metrics to measure the performance of your service.

**Request-based or windows-based?**  
The method of evaluation you choose will affect how compliance is measured.

☒ Request-based  
Counts individual events. This lets you know how well your service performed over the entire compliance period, no matter how the load was distributed.

☐ Windows-based (advanced)  
Counts "good minutes" versus "bad minutes" according to criteria you define. This lets you measure performance in terms of time, regardless of how load is distributed.

**CONTINUE** **CANCEL**

**Figure 15.18: Creating SLI**

3. For selecting the metric for SLI, choose the 'log\_based\_get\_request' metric as seen in [Figure 15.19](#):

The screenshot shows the 'Create a Service Level Objective (SLO)' interface, Step 2: Define SLI details. The sidebar shows Step 2 is active. The main content area is titled 'Define SLI details' and includes a sub-header 'Performance Metric'. Below this is a table with columns 'Metric name', 'Metric kind', and 'Value type'. The 'Metric name' is 'logging.googleapis.com/user/log\_based\_get\_requests', 'Metric kind' is 'Delta', and 'Value type' is 'Int64'. There is a 'Learn more' link and a 'CONTINUE' button at the bottom.

Metric name	Metric kind	Value type
logging.googleapis.com/user/log_based_get_requests	Delta	Int64

**CONTINUE**

4 Review and save  
Review details and name your SLO



Figure 15.19: Creating SLI

## Creating SLO

To configure a SLO, follow the given steps:

1. First you have to define the aggregation window of the SLI, as shown in [Figure 15.20](#):

The screenshot shows the 'Create a Service Level Objective (SLO)' form. The sidebar on the left has four steps: 'Set your service-level indicator (SLI)', 'Define SLI details', 'Set your service-level objective (SLO)' (which is highlighted), and 'Review and save'. The main area is titled 'Set your SLO' and has a description: 'Define a time period for compliance and set a goal for "good service." This is called a service level objective (SLO)'. It has two sections: 'Compliance period' with 'Period type' set to 'Calendar' and 'Period length' set to 'Calendar week', and 'Performance goal' with a 'Goal' input field set to '50%'. At the bottom is a 'Preview' section with a line chart showing compliance over time.



**Figure 15.20:** *Creating SLO*

In the preceding [Figure 15.20](#), you can see that we had used period type as calendar, meaning absolute days. There is another option to select a rolling window as well. Next, we have defined period length, that is, in what time frame we will club the numbers for percentage calculation. Finally, we define the goal for SLO. Here we had given it as 50%. In the real world use cases, it is much higher than 50%.

2. You can review the configurations done above in the json format, and can download this json file for automation purposes, that is, creating similar set up in a new environment via pipelines. Refer to [Figure 15.21](#):

Create a Service Level Objective (SLO)

[SEND FEEDBACK](#)
[LEARN MORE](#)

✓

Set your service-level indicator (SLI)

Choose the aspect of service health for which you want to set a performance goal

✓

Define SLI details

Specify more details for the metric you've chosen

✓

Set your service-level objective (SLO)

Set targets for how well your service should perform

4

Review and save

Review details and name your SLO

Review and save

SLO details

Display name

50% - Good/Total Ratio - Calendar week

SLO JSON preview

Based on current parameters, this JSON can be used with the SLO API. [Learn more](#)

Copy JSON

```

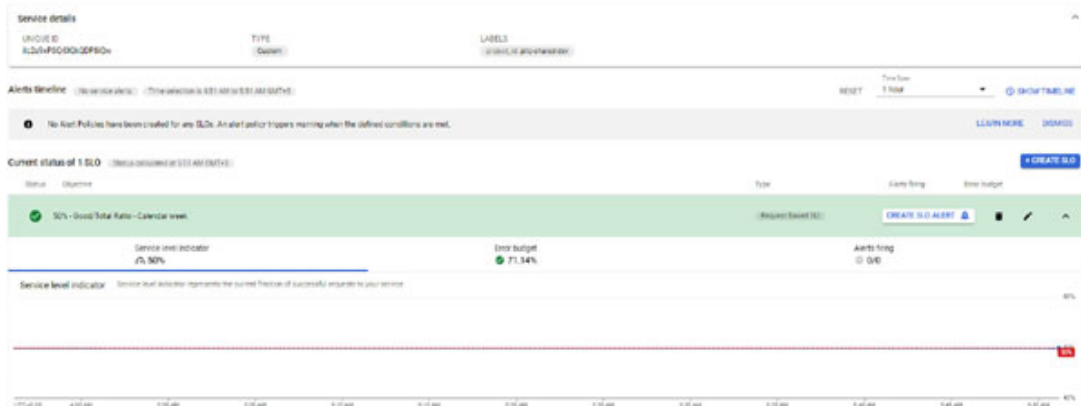
1 {
2   "displayName": "50% - Good/Total Ratio - Calendar week",
3   "goal": 0.5,
4   "calendarPeriod": "WEEK",
5   "serviceLevelIndicator": {
6     "requestBased": {
7       "goodServiceFilter": "metric.type=logging.googleapis.com/user/log_based_get_requests",
8       "resource.type=gae_app",
9       "badServiceFilter": "metric.type=logging.googleapis.com/user/log_based_get_requests",
10      "resource.type=gae_app"
11     }
12   }
13 }
```

CREATE SLO

CANCEL

**Figure 15.21:** Finalize and save SLO

Once you click **create SLO**, the SLO will be created. As you can see, various options of error budget, Number of fired alerts, option for creating and alert appears, as shown in the following [Figure 15.22](#):



**Figure 15.22:** Final SLO screen

## Tracking error budgets

In a software system, there are a lot of metrics that can be tracked, including infrastructure monitoring, service monitoring,



and so on. The art of tracking lies in the fact of identifying the key SLIs and setting appropriate SLOs, that is, tracking the metrics, which really affect end-user journeys. For example, tracking CPU usage as an SLI might not be a good idea; however, still tracking them for the internal infrastructure team to tackle the need for additional CPU, might be crucial for debugging purposes. Tracking too many metrics will result in a noisy system, where there are high chances that the risky problems may get missed simply because of the number of metrics that you are tracking.

There are multiple ways to track, creating dashboards for metrics, creating automatic alerting, and so on. In the case of SRE, you are working on metrics that affect customer satisfaction. Thus, any metric which can affect that becomes important and we want to have sufficient bandwidth allowed by the system, for us to work upon, before an issue starts impacting the customers.

The SLI, SLO, and SLA combination results in the definition of error budgets. It is never ideal to utilize our complete error budget. Meaning, if the error budget allows the service to be down for 7 hours (99.9% SLI and 99% SLA) in a month, it makes sense to distribute these seven hours across the month, that is,  $7/30 = .23$  hours of the service being down every day. If the service is burning the error budgets at more than .23 hours a day, there are high chances of missing the SLA.

This rate becomes an important dimension to track and generate alerts if the rate increases. It is of prime importance to investigate the failures happening as early as possible, so that the SLAs are not breached. There might be a situation when the resolution might need multiple teams to react, and hence, it is key to start working on these deviations early. As an SRE, the following aspects should be very clearly defined for each SLA committed to the customer:

- A very well-created incident response plan and rehearsals to recover from it. For example, if a Blob store location going down results in your service going down, it is important to have a well-defined plan for how this will be tackled. Not only plan, it is also equally vital to perform that exercise at regular intervals for the team to feel confident in the plan. These regular activities are called SRE drills.

- It is important to trigger alerts at the right time. For example, there is no point in alerting when a 100% budget is exhausted. Rather, alerts at 25%,50%,75%, and 90% could help the teams to plan the resolution well and smoothly.
- As far as possible, try to automate the actions. For example, if the service is behaving slow in certain regions, create a few instances in the backup region to make the process fast.

Rather than tracking everything, tracking the service level indicators which determine meeting SLO and hence SLA, is the key to success.

## Creating alerts

You can generate alerts based on the exhaustion of your error budget on SLOs. In this section, you will see how to generate alerts based on different percentages (25%, 50%, and 75%) of exhaustion of error budgets.

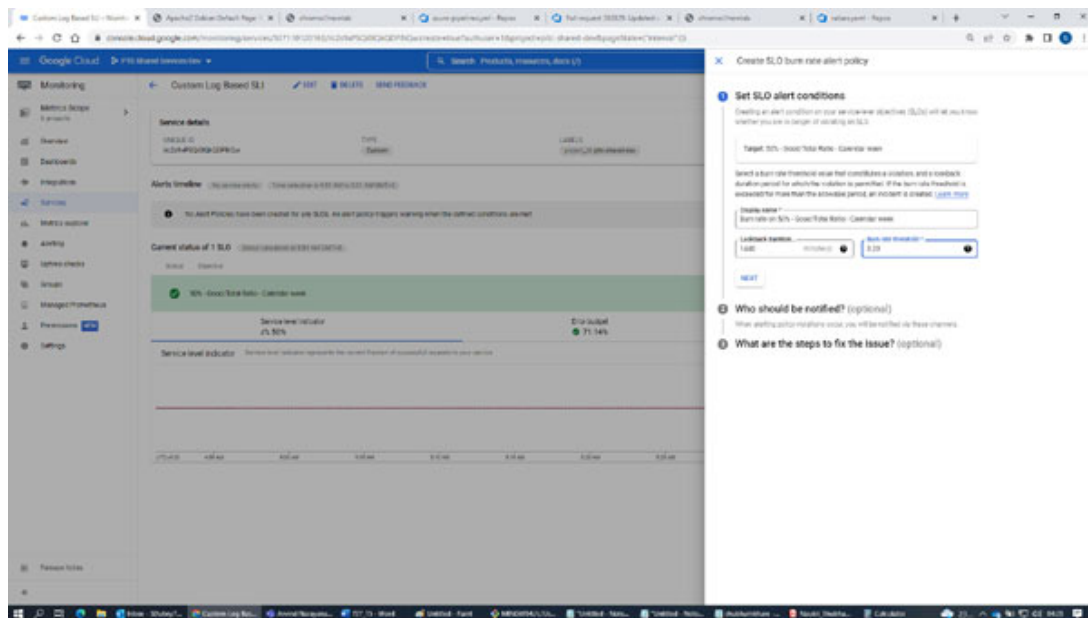
While configuring the alerts, you need to set a **lookback duration**, which is a time window for which we are going to track the error budget. Smaller lookback durations (fast burn scenario) result in faster detection of issues, but with a caveat that the error rate over the course of the day may result in over-alerting by the system. Longer duration (slow burn) if not alerted may result in exhaustion of the error budget before the end of the compliance period.

The second parameter is the **Burn Rate threshold**, which is the percentage of the percent budget burned. If the burn rate exceeds the time window of Lookback duration, an incident is generated. A good starting point for a fast-burn threshold policy is 10x the baseline with a short (1 or 2 hour) lookback period. A good starting point for a slow-burn threshold is 2x the baseline with a 24-hour lookback period.

An example configuration could be a situation where the Lookback duration is 24 hours (that is, 1440 minutes) and burn rate threshold is 3.33. That is, in 24 hours, if the burn rate is above 3.33 percent, you burnt more than what was expected in a 30-day month. This alert makes sense, since if this continues, the chance of breaching SLA is high. Along with the preceding mandatory

parameters, you can set the optional parameter of the alert notification channel, that is, how to inform the concerned team about the incident.

You can also go ahead and optionally mention the steps which could be taken when this kind of incident happens. This could be as simple as just telling to inform a team XYZ about the incident, to instruct some technical action to the SRE, as shown in [Figure 15.23](#):



**Figure 15.23:** SLO alert screen

One good practice is to configure different levels of alerting mechanism. For example, the preceding daily alert on per day basis (fast burn) could be suited for the engineering team, but for management, it will too much. Hence, you should ideally set up an alert over larger lookback (slow burn) duration.

## **Probes and uptime checks**

Testing specifies acceptable behaviour of an application with known data, while monitoring specifies acceptable behaviour in the condition of unknown user data. It might seem that the major risks of both known and unknown are covered by testing and monitoring defined for the system. Unfortunately, the risk is more complicated. The known bad request should error out, and known

good requests should work. Implementing both as part of integration tests is a good idea, as you can run the same tests again and again when at the time of each release.

However, it makes sense to think about setting up such checks as monitoring probes. It might seem that it is over-engineering to do so and therefore, pointless to deploy such monitoring because the exact check has been applied as integration tests. However, there are good reasons to think like that, and they are as follows:

1. The release test most probably has wrapped the server with the front end and fake back end.
2. The tests wrapped the release binary with a load balancing frontend and separate scalable backend.
3. Front end and backends have different release cycles, and there are high chances of these cycles being different.

Therefore, monitoring probes is a configuration that was not tested. Probes should ideally never fail, but if they do, it means that either the back end or front end is not consistent in release and production environments. One can argue that there are more sophisticated ways that provide better system observability. However, probes are a lightweight way to determine what is broken, by providing insights into if the service is reachable or not. Probes give only two answers – available or not available. This simple answer does not provide the overall health of the system, but it does provide critical first-level insight – reachable or non-reachable.

For any service involving request/response, reachability is a prime prerequisite. Probes are tailor-made for this. For a reactive queue-based service, the same rules do not hold. Services that are not exposed to the client's throughput and other system indicators are a better indicator of system health.

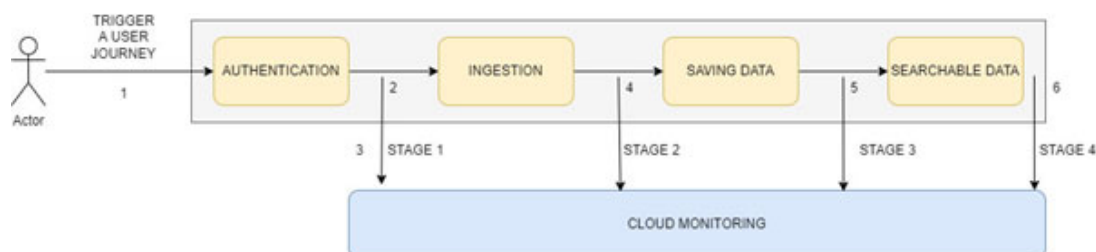
Generally, in the real world, it is not one service but rather, a user journey involving multiple services working together, which makes sense for clients. Hence, rather than probing one service, a probe at the complete user journey makes more sense. For example, let us say we have a user journey, where we ingest some data. Then, an example user journey could be as follows:

- User is authenticated.
- Ingestion process starts.
- Records are saved properly.
- Records are searchable and available for other systems.

Rather than triggering probes for individual services, it is wise to trigger a probe for complete flow. You can configure synthetic data ingestion, which has a dataset that matches all your probable datasets expected to be ingested by this journey. This collection of data sets will be used by the periodic trigger of the workflow (probes) and the pass or fail of each of the preceding steps will be checked. The automated trigger can happen via a scheduled job on the orchestrator or cron job existing inside Kubernetes or even by tools such as cloud probe. You can have an additional check of how much time will this probe take, that is, (end time - start time). If this is below a threshold, it means your application is scaling well and in a predictable manner.

Triggering probes explicitly might not be a need in case of situations where the user journey has explicitly been used by end users. When the users are using the data, you can track the aspects of your metrics on live data, and hence there is no need to put an extra burden on the platform.

Consider the pictorial representation of the preceding user journey as shown in [Figure 15.24](#):



**Figure 15.24:** User journey

The complete user journey is broken into 4 stages, as shown in the preceding figure. After completion of each stage, an entry is made in cloud logging, as “stage:”1”, which means that in the user journey, stage 1 is completed successfully. You can go ahead and create a metric on this log entry, the way we did for HTTP status

codes in the example under the section *Log based metrics*. On the metrics, you can also create SLO and assign error budgets.

In the preceding example, the customer satisfaction will be met only when we get stage 4 in the same count as stage 1, that is, the entire user journey is getting completed successfully. If that is not happening, you can easily count which stage has a problem and try to fix it.

## **Aggregating logs to set up cloud monitoring dashboard**

In this section, you will not see something new. Rather, we will discuss how we can create customized metrics and customized dashboards. The GCP monitoring suite provides ways to aggregate metrics based on various mathematical functions like mean, average, and so on.

For instance, let us consider that you do have a use case to apply a mathematical function that is not supported out of the box. Another situation could be that you want to combine two different metrics together and see the contribution of both your SLO and SLA contributions. Assume that you have a scenario, where you want to add two metrics of a virtual machine instance and create dashboards for them. For combining the metrics, you can use **Monitoring Query Language (MQL)**.

Here is an example MQL:

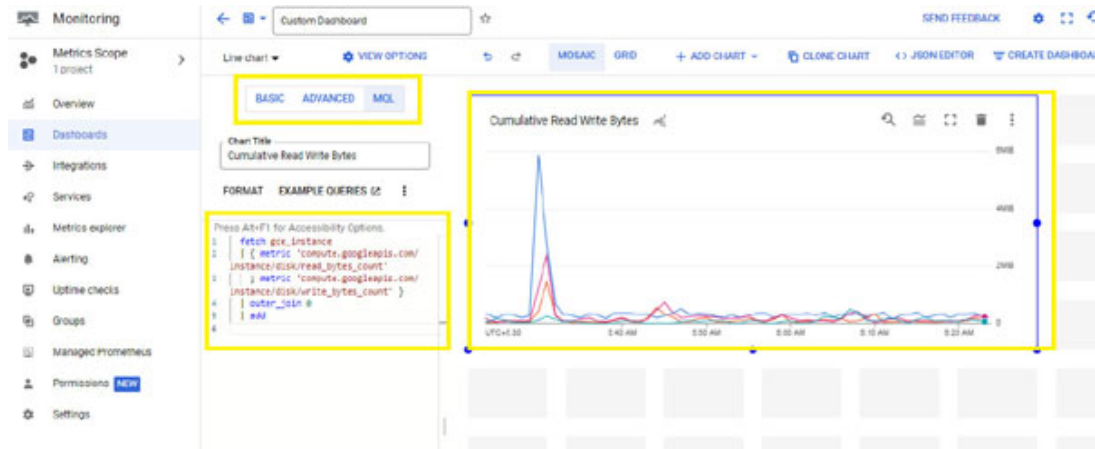
```
fetch gce_instance
  | { metric 'compute.googleapis.com/instance/disk/read_bytes_count'
    ; metric 'compute.googleapis.com/instance/disk/write_bytes_count'
    }
  | outer_join 0
  | add
```

This **Monitoring Query Language (MQL)** adds the bytes count for read and write.

Let us see how you can configure an available dashboard using this MQL. Follow the given steps:

1. Go to **Monitoring | Dashboards | Create Dashboards**.

2. On the **Create Dashboard** screen, apply the configuration as shown in [Figure 15.25](#):

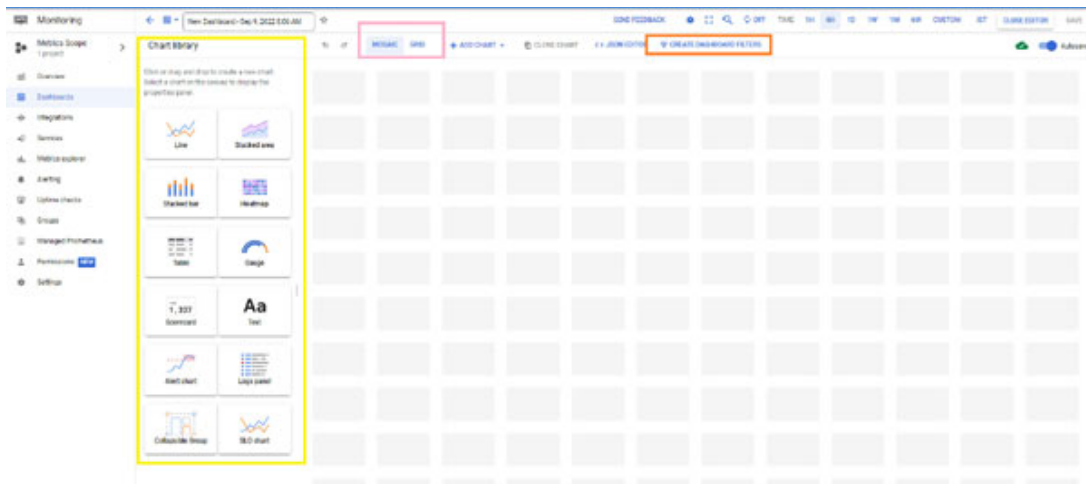


**Figure 15.25:** Using MQL

3. Select the option of MQL. In the MQL text box, mention the preceding MQL code and see the chart on the right side. For more details on MQL, please see more details - <https://cloud.google.com/monitoring/mql>

Along with support of giving custom queries (MQL), GCP monitoring also provides many chart options to create dashboards. Different chart options are used to represent data in an intuitive way.

Consider the following [Figure 15.26](#). The yellow rectangle covers the different chart options, and the pink rectangle configures the layout. You can apply filter on the metrics coming in the dashboard (orange). You can combine multiple charts into one dashboard, add documentations and share them easily with other teams. Please refer to the following figure:



**Figure 15.26:** Creating dashboards

For example, in the following [Figure 15.27](#), you can see multiple charts of different sizes and documentation created:



**Figure 15.27:** Dashboard of multiple charts

The dashboard design is created for not only the technical teams but also for the management teams, that want to have a look at metrics. The dashboard supported could be very easily shared. Moreover, each of the charts could be converted to an alert chart, and it could be ensured that alerts will be generated when the metrics go above threshold values. This is primarily used in cases when you want to get notified based on the value of some metrics. For example, if the CPU usage is 90% for the last 5 minutes, generate an alert. This alert will not make sense for SLI.



## Responsibilities of SRE

In this last section of the chapter, the reader will acquire better understanding of responsibilities performed by SRE teams. Although we did talk about them in the previous section, we will dive deep into this topic now.

### Incident management

Although everybody wants their services to run without a hiccup, practically speaking, they may fail or go down. When such an unwanted incident happens over a continued overtime window, it is known as an **Incident**. Though the primary aim of SRE is to make sure that such situations do not occur, if they do happen, how well they restore the system to normal, depends on the capability of the SRE.

Resolving an incident means restoring the service to normal or mitigating the impact of the issue. Managing incident means coordinating the efforts of multiple teams efficiently and ensuring effective communication between the engineering parties/teams. The basic aim is to respond to an incident in a very structured way. Incidents could be confusing and a well-thought-through strategy of action when such incidents occur increases the time to recover.

To ensure the readiness of the plan and effectively apply the plan, SRE team members lead, perform, and ensure the following actions/best practices:

1. **Prioritization:** The SRE team identifies the service-creating issue and may decide to shut down or restore the service, making sure that they preserve sufficient information behind the cause of the incident.
2. **Preparations:** The SRE team prepares for these unwanted situations or incidents by properly documenting the incident management process in advance, in collaboration with all the participants in the incidents.
3. **Managing incident resolution:** When the incident occurs, SRE team members manage the whole resolution activity. Incidents are tough to resolve and might take many days as well, and it might result in a few members' emotional states

becoming cranky. They manage the environment so that your team is moving continuously in the right direction of resolution.

4. **Periodic improvements:** The SRE team periodically re-evaluates the plan to improve it further and make it more effective. The plan gets updated with learning from each incident.
5. **Drills:** The SRE team performs drills where they synthetically create an incident and situation of an incident, to measure the effectiveness/correctness of the plan.

## Playbook maintenance

The playbook contains high-level instructions on how to react to alerts. It is important to keep in mind that this critical alert is just a temporary thing and might fade away in some time. Playbooks contain explanation about the why, how, and what of an alert, and attach the severity of it. It also has steps written to resolve the alert. Whenever an alert is generated, the playbook must be updated with every incident, to make sure we document the reason for the latest alerts.

The contents of playbooks can very easily go out of date, and hence they need updates after every incident and reviews after every major release. If you had created a playbook that contains a lot of details, the frequency of change will also be high than if you create a generic playbook. It depends on team to team; some like to maintain lots of information and steps, meaning that they will rely more on the playbook to handle the situation, rather than put their mind onto it. On the other hand, it could contain just the basic details, and the SRE who is working, puts his mind behind analysing everything about an issue from scratch.

## Drills

A drill is a periodic activity, where the SRE team synthetically tries to replicate a failure scenario and take the planned action against it. If after doing that activity, the team is able to restore back the service completely, it is assumed to be a successful drill, or else the drill is marked as failed. In case of a failed drill, introspection of the process is needed and optionally review of the documents (playbooks) available for correctness and effectiveness, is also done.

Such drills help enterprises come up with **Mean Time to Recovery (MTTR)** numbers for the end users. The time taken by most of the drills is assumed to be the time that could be quoted for MTTR.

## Automating SRE actions

Till now in this chapter, we have investigated all the concepts and their implementation via UI console. It was good for understanding purpose, but when it comes to creating such dashboards, SLI, SLO and so on, in a real software project, we need it in an automated fashion. There are multiple ways in which this could be automated, and a few common ones are as follows:

- **Cloud Monitoring REST API:** The payload is a JSON payload which can be easily downloaded and modified and committed to a repo. The **Continuous Integration and Continuous Deployment (CI/CD)** pipelines can change the environment attributes and trigger the REST APIs.
- **Google Remote Procedure Call (gRPC):** There are monitoring libraires available in java and python. In case the library is not available in the language of your choice, these gRPC method could be used to develop one.
- **Terraform:** Monitoring dashboards can be created via using the Terraform. Terraform has the module for Google cloud monitoring. For more details: [https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/monitoring\\_dashboard](https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/monitoring_dashboard)

## Conclusion

Google team is probably one of the first teams that started thinking about processes like SRE. SRE is important to increase the reliability of new features in your projects. GCP offers built-in customizable logging, monitoring, and dashboarding tools such as PaaS. GCP also offers a way to define SLO and error budgets as well. You can easily configure multiple alerts as per the error budget burn down. These offerings from Google are easily customizable where you can override all the behaviours as per your use cases.

SRE plays a role in not only day-to-day observing the deployments, but they also play a key role in strategizing and planning the situation of incidents. For this, they maintain a playbook that contains details about an issue that can occur, and they practice the decisions to measure the effectiveness of the plan by doing periodic drills.

## **Points to remember**

1. SRE ensures the reliability of the system in the midst of fast-paced feature development.
2. GCP offers managed tool stack (PaaS) for all the possible SRE needs.
3. GCP offers facilities like logging, monitoring, dashboarding, and alerting features in the GCP operation suite. You do not need to worry about the scalability of the stack as it is completely auto-scalable and managed by GCP.
4. GCP monitoring provides a rich set of already created metrics. However, there is no restriction on incorporating new use cases.
5. The SRE team, apart from observing the system and resolving critical alerts, plays a key role at the time of occurrence of incidents.
6. The SRE team updates the playbook and performs drills to define the Meantime To Recover for an application.

## **Questions**

1. SLAs are decided after engineering teams and business owners agree to an availability number. True or False?
2. How does GCP allow defining custom mathematical functions for SLI calculation?
3. Working on a resolution of an incident is the responsibility of SRE. True or False?

## **Answers**

1. False. Ideally, a debate happens on SLO. If the service runs over a period meeting SLO numbers, then we define the SLA accordingly.
2. You can define custom functions using the Monitoring Query language. Once the metrics are aggregated, create an SLI then SLO, and configure alerts.

3. False. It is the responsibility of the complete team including DevOps.

# **CHAPTER 16**

## **SRE Use Cases**

### **Introduction**

Applications hosted on **Google Cloud Platform (GCP)** not only take advantage of a highly robust and reliable infrastructure for running the workloads, but they also have the advantage of integration with the components of cloud monitoring and cloud logging, which lets you easily manage and develop the SRE framework for your application. GCP provides services to host request-response and data processing applications and describes how to leverage monitoring metrics exposed by the application as **Service Level Indicators (SLI)**.

### **Structure**

In this chapter, we will discuss the following topics:

- GCP service grouping
- SRE practices in the microservices world
- SRE practices big data world

### **Objectives**

After reading this chapter, you will be able to understand and use the SLI metrics provided out of the box by GCP, for request-response and data-intensive applications. This chapter will bridge the gap between theory (discussed in [\*Chapter 15, Site Reliability Engineering\*](#)) and real-world scenarios (described in the current chapter). You have

already seen how to create custom SLI (not available out of the box) using log-based metrics in the previous chapter.

## **GCP service grouping**

Google cloud services can be grouped into the following types:

### **Request response services**

In this group of services, a user requests to do some tasks and wait for the request to complete. The following GCP offerings support this category:

- App engine
- Google Kubernetes engine
- Cloud Run and Cloud Functions
- Cloud end points

### **Data storage and retrieval services**

This group contains offerings from GCP which store and retrieve data based on the user request and supports appropriate user responses. The following GCP offerings support this category:

- Cloud big table
- Cloud spanner
- Datastore
- Cloud storage

### **Data processing services**

This group has offerings from GCP, which perform processing on massive datasets. The following GCP offerings support this category:

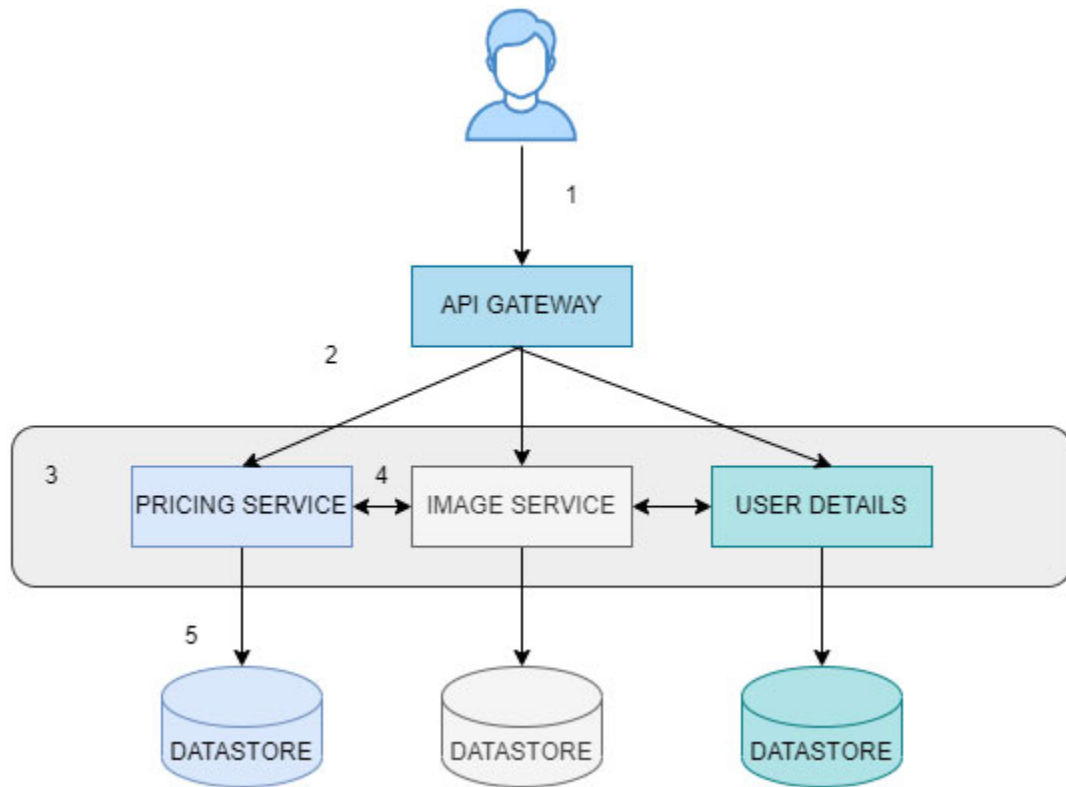


- Google Dataflow
- Google Dataproc

In the subsequent sections of this chapter, you will see all the preceding groups of services in action, explained via the two most widely used architecture – big data and microservice. You will see a very high-level architecture of both the use cases, sufficient enough to understand the SRE stack available on Google Cloud Platform.

## **SRE practices in the microservices world**

Microservices is an architectural approach to developing software, where the software is composed of small and independent services that communicate over well-defined APIs. These services are developed and managed by separate teams, and each microservice caters to a very specific role in the larger software package. Consider the following [Figure 16.1](#) which depicts an over simplified example of microservice:



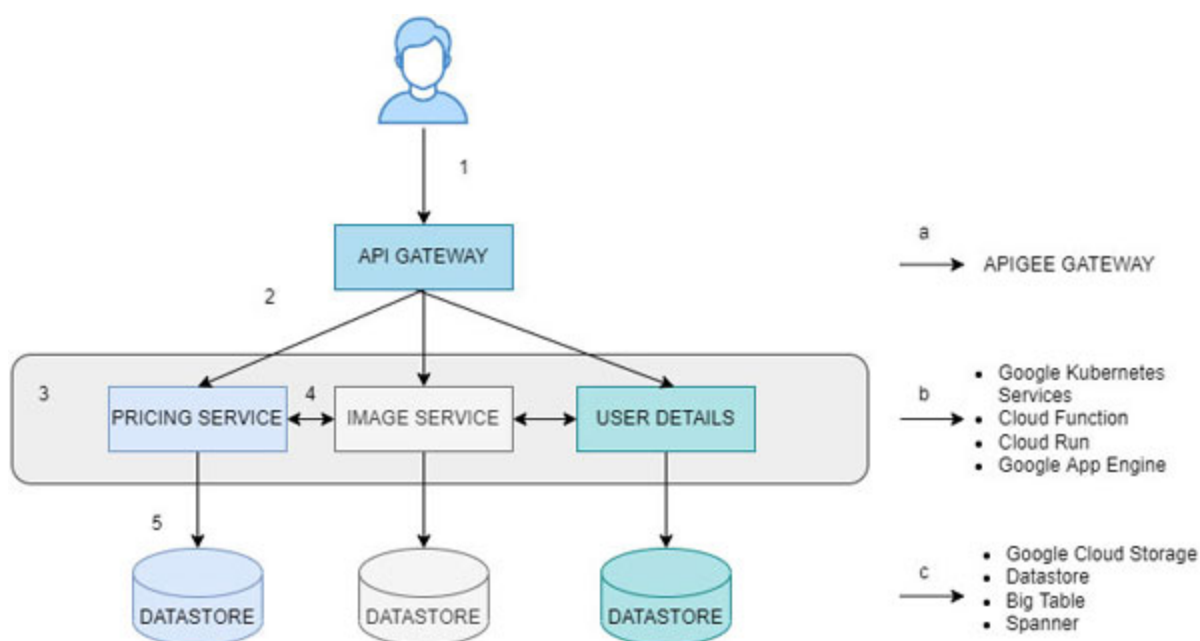
**Figure 16.1:** Simple microservice

Follow the numerical labelling in the figure with the explanations given as follows:

1. An external user/client requested some work to be done from the software system.
2. The request is received via an API gateway, which decides which service to trigger. This selection of a certain service could be based on the URL or some property in the header of the requests.
3. This section represents a set of microservices – pricing, image, and user details. These microservices have a very specific purpose; for example, the pricing service only deals with operations related to the pricing of a product.
4. Microservices can interact among themselves to orchestrate a complete user journey.

5. Each microservice has their own separate datastore. This datastore could be of totally different technology choices.

Let us repaint the generic picture featured in [Figure 16.1](#), with components specific to GCP in the following [Figure 16.2](#):



**Figure 16.2:** Simple microservice using GCP

From the preceding figure, follow the labels a, b and c with the following explanation:

- a. Apigee is a Google-provided API gateway.
- b. GCP provides multiple options to host microservices. These are GKE, Cloud Function, Cloud Run, and App Engine.
- c. GCP provided options for storing data depending on the nature of data – columnar vs. document vs. relational.

In the preceding microservices use case, the following SLIs in [Table 16.1](#) are the most common set of SLIs one can think of:

--	--

SLI	Description
Availability	Whether the service was able to accept the user request.
Latency	Time taken by service to accept a request and revert with a response.
Throughput	Number of requests handled concurrently by the system.
Success rate	Number of requests handled by API successfully.
Error rate	Number of requests that errored out by the API.

**Table 16.1:** *Microservice SLIs*

Now let us look at the implementation part of the preceding SLIs and the strategy for defining **Service Level Objective (SLO)** for each of the preceding SLIs.

## Availability

Take pricing microservices as an example for the section. Assume the pricing service is hosted on Google App Engine and Cloud Spanner as the backend. The overall availability of the service is a combination of availability of the Cloud Spanner and availability of service hosted on App Engine. It will be a fair assumption to make, that each request made on App Engine will ideally interact with Cloud Spanner. Hence, the availability of Cloud Spanner will impact the availability of services on the App Engine. For example, if the availability of Cloud Spanner is 99.9%, by no means does the pricing service on App Engine can have availability better than 99.9%.

The 99.9% is just a number and it depends on the class of infrastructure being used for deployment. If we denote the availability of storage as  $a\%$  and the availability of computing as  $b\%$ , then  $b$  can never be greater than  $a$ .

One can debate that in case of non-availability of the Cloud Spanner in the preceding case, one might apply retries or even park the data temporarily in memory. However, that

will affect the latency API or freshness SLI. So, in that case, your other SLIs will take a hit. The choice therefore depends on use case to use case. We need to define availability SLO and the SLA for the service, keeping into consideration that you will need time to debug the issues related to Spanner as well as issues related to App Engine.

App Engine standard environment pushes metric data to cloud monitoring using the `gae_app` monitored resource type, and `HTTP/server/response_count` metric type.

You can apply a filter to data by using the `response_code` metric label to calculate “*success*” and “*total*” responses. You can create an SLI definition by creating a `TimeSeriesRatio` of “*success*” to the “*total*” count.

```
“serviceLevelIndicator”: {  
  “requestBased”: {  
    “goodTotalRatio”: {  
      “totalServiceFilter”:  
        “metric.type=\"appengine.googleapis.com/http/server/resp  
onse_count\"  
        resource.type=\"gae_app\"  
        metric.label.\"response_code\">\"499\"  
        metric.label.\"response_code\"<\"399\"\"\",  
      “successServiceFilter”:  
        “metric.type=\"appengine.googleapis.com/http/server/resp  
onse_count\"  
        resource.type=\"gae_app\"  
        metric.label.\"response_code\"<\"299\"\"\",  
    }  
  }  
}
```

Similar to App Engine’s availability, Cloud Spanner availability SLI could be defined as well. The concept remains the same, that is, the Spanner writes data to cloud monitoring using the `spanner_instance` resource type and `query_count` as metric type. As was the case with the App

Engine, now you can define a filter of “*success*” to the “*total*” number of database queries and create an SLI definition by creating a **TimeSeriesRatio** of “*success*” to the “*total*” count.

```
“serviceLevelIndicator”: {
  “requestBased”: {
    “goodTotalRatio”: {
      “totalServiceFilter”:
        “metric.type=\\”spanner.googleapis.com/query_count\\”
        resource.type=\\”spanner_instance\\”
        metric.label.\\”database\\”=\\”my_database\\””,
      “successServiceFilter”:
        “metric.type=\\”spanner.googleapis.com/query_count\\”
        resource.type=\\”spanner_instance\\”
        metric.label.\\”database\\”=\\”my_database\\”
        metric.label.\\”status\\”=\\”ok\\””,
    }
  }
}
```

## Latency

While defining the latency SLO and SLA, keep in mind the cumulative behaviour of latency of computing logic hosted on App Engine and data residing in cloud spanner. Individual latency of App Engine and cloud spanner could be calculated using the standard metrics on Monitoring APIs.

For measuring the latency of the App Engine standard environment, use the monitored resource type as **gae\_app** and **HTTP/server/response\_latencies** as metric type. You can represent the latencies in a distribution cut structure. The following SLO expects 95% of all requests to fall between 0 to 50 ms in latency over a rolling period of 1 min.

```
{
  “serviceLevelIndicator”: {
    “requestBased”: {
```

```

    "distributionCut": {
      "distributionFilter":
        "metric.type=\"appengine.googleapis.com/http/server/res
        ponse_latencies\"
        resource.type=\"gae_app\"",
      "range": {
        "min": 0,
        "max": 50
      }
    }
  },
  "goal": 0.95,
  "rollingPeriod": "60s",
  "displayName": "95% requests under 50 ms"
}

```

For measuring latencies in Spanner, you can use the **spanner\_instance** as monitored resource type and **API/request\_latencies** as metric type. You can filter the data using the method metric label to measure latencies. You can express the request-based SLI using the distribution as demonstrated in the following code:

```

{
  "serviceLevelIndicator": {
    "requestBased": {
      "distributionCut": {
        "distributionFilter":
          "metric.type=\"spanner.googleapis.com/api/request_laten
          cies\"
          resource.type=\"spanner_instance\"
          metric.label.\"database\"=\"scaling_gcp_database\"",
        "range": {
          "min": 0,
          "max": 50
        }
      }
    }
  }
}

```

```

    }
  },
  "goal": 0.95,
  "rollingPeriod": "60s",
  "displayName": "95% requests under 50 ms "
}

```

Other SLIs, throughput, success rate and error rate can be easily created using the preceding showcased metrics in different combinations. For example, for throughput, create an SLI on just the number of requests (successful or errored out). Similarly for success rate, count the total number of successful requests in a window. For error rate, the number of errored out request in a time window.

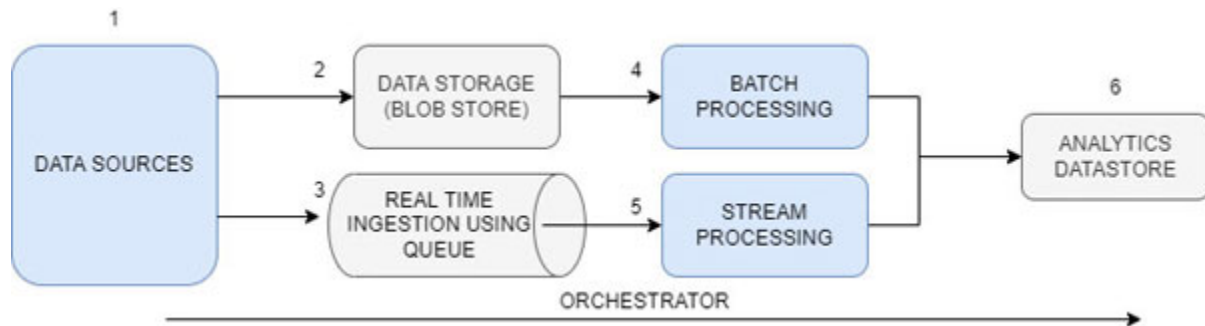
## **SRE practices big data world**

For use cases that include massive data processing (big data), SLIs like latency and availability do not make much sense. Availability and latency are terms more relevant in cases where a user is waiting for a response in real-time. This data processing happens either in batch mode or streaming mode, and in both cases, availability and latency are not the right measure of customer satisfaction, as no customer/client is waiting for a response. Rather, when we talk about SRE practices in Bigdata, we talk about two more terms:

- **Freshness:** Number of times the user received the latest data for read operations on the API, despite the underlying data store being updated with a certain write latency.
- **Correctness:** Correctness is the measure of how many processing errors the pipeline encounters.



You will see the implementation of the preceding two SLIs, but before that, let us go ahead and describe big data use cases. Consider the following generic high-level big data architecture diagram in [Figure 16.3](#):



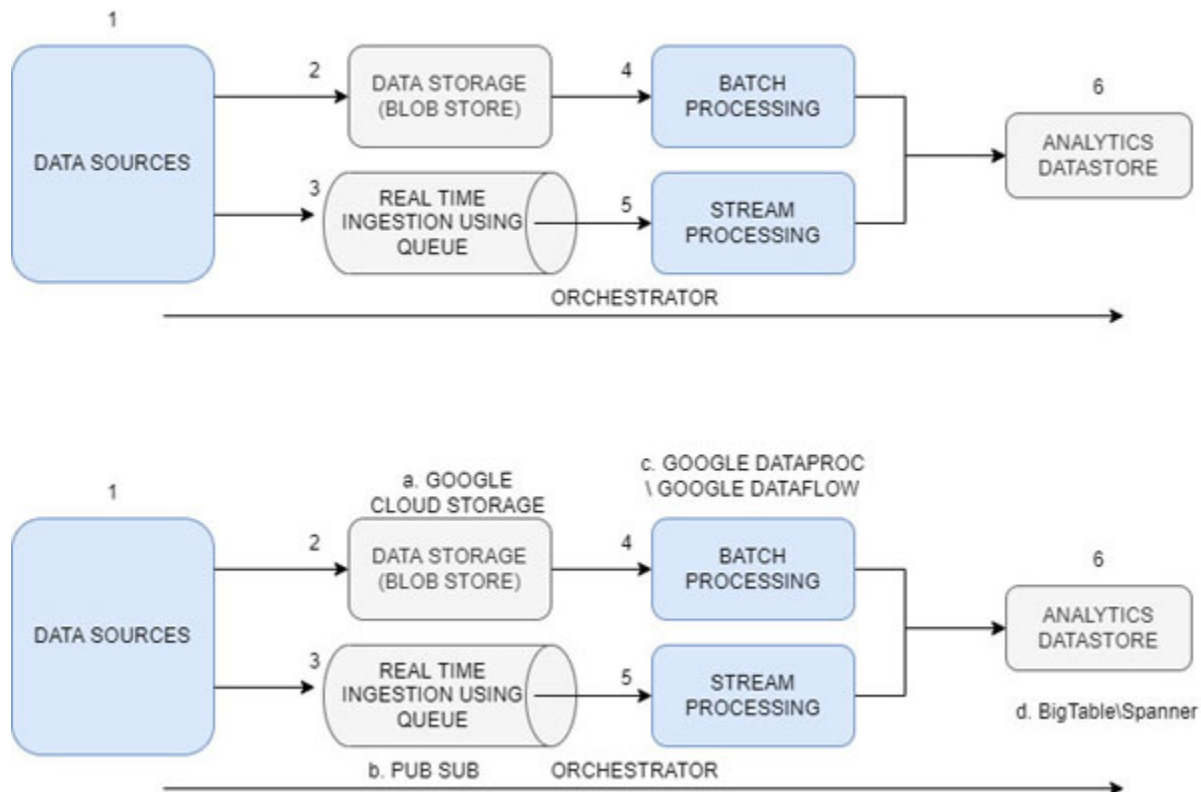
**Figure 16.3:** Simple big data processing

Follow the numerical labelling in the figure with the explanations given as follows:

1. Data sources are devices that generate a huge volume of data. For example, in IoT devices, this data is sensor data. Similarly, in the case of a mobile app, it is the data generated by the user using your app.
2. Data is either stored in Blob Store or any other data store.
3. Real-time data is published onto a queue for real-time streaming scenarios.
4. Batch processing is a form of data processing that has a well-defined start time and end time, and is usually associated with collecting a huge volume of data and then triggering processing.
5. Stream processing is a form of processing, where the data is processed in real-time. Generally, systems do not wait for data to accumulate, and processing happens either in real-time or in near real-time.
6. The streaming and batch processing applications often save their data in a well-structured format in a data store. This datastore contains data that is as per the

queries defined by consumption systems. For example, let us consider a downstream system that needs to get an average income per city, and this average income per city is calculated by the processing pipelines and the result is saved in a data store.

Let us now see what the preceding picture looks like when we superimpose GCP technology with the architecture described above. Consider [Figure 16.4](#):



**Figure 16.4:** Simple big data processing on GCP

In the preceding figure, follow the labels a, b, c, and d with the following explanation.

- Blob store in GCP is known as Google Cloud Storage.
- The queue service available in GCP is Google Pub/Sub.
- For processing, GCP offers Dataproc and Dataflow for both batch and streaming pipelines.
- BigTable\Spanner

- d. The most common data store for analytics purpose on GCP is Cloud Spanner and Bigtable.

You can track the availability and latency metrics for cloud storage and analytics datastore, but as said, the nature of the application does not involve an entity waiting for a response right away. Hence, freshness and correctness are better SLIs for such use case. Google Cloud provides a mechanism to define correctness and freshness for both Dataproc, as well as Dataflow pipelines.

## Correctness SLI

Correctness SLI is defined as the chunk of data that produced the correct result. The outcome of processing is generally validated against heuristics; for example, an image processing pipeline cannot produce an image of zero bytes. Similarly, in a financial transaction, the difference in balance before the transaction, to the balance after the transaction, is equal to the transaction or not.

Another way to configure this, is by using a known synthetic dataset and triggering it periodically to check the results of processing with a pre-known result.

For this example, let us assume we have the pipelines written in Google Dataflow. Dataflow writes metric to cloud monitoring using the monitored resource type `dataflow_job` and metric type `job/element_count`, which counts the number of elements added to Pcollection so far. Adding all of them grouped with a job name, gives the total records processed by the job.

From your data flow job, emit a log-based metric in case of error with a severity metric label. You can use the `logging.googleapis.com/log_entry_count` metric type in the `dataflow_job` resource and count the number of errors logged with configured severity count. You can use the ratio to

express the correctness of SLI by using a **TimeSeriesRatio** structure.

```
"serviceLevelIndicator": {
  "requestBased": {
    "goodTotalRatio": {
      "totalServiceFilter":
        "metric.type=\"dataflow.googleapis.com/job/element_count\"
        resource.type=\"dataflow_job\"
        resource.label.\"job_name\"=\"scaling_gcp_job\"",
      "badServiceFilter":
        "metric.type=\"logging.googleapis.com/log_entry_count\"
        resource.type=\"dataflow_job\"
        resource.label.\"job_name\"=\" scaling_gcp_job \"
        metric.label.\"severity\"=\"error\"",
    }
  }
}
```

The preceding configuration will count the total number of elements in the job named **scaling\_gcp\_job** as well as the total number of logs whose severity is marked as error.

## **Freshness SLI**

In a batch pipeline, Freshness can be defined as the time elapsed since a processing run was completed successfully for a given output. In other words, the processing was completed, and the data got uploaded to the datastore. When we query the data, the new data should be the output. The time between pushing the data and successfully pulling the updated data is Freshness. Similarly, in the case of streaming applications, freshness SLI can be defined as the time difference between the most recent processed record present and current time.

Dataflow pushes metrics to cloud monitoring using the `dataflow_job` monitored resources type and `job/per_stage_system_lag` metric type, which tracks the maximum duration that a data item has been processing or awaiting processing. Freshness SLI is generally represented by using the `DistributionCut` structure. In the following example, the SLO expects that the oldest data element is processed in under 50 seconds, 90% of the time, over a rolling one-minute window:

```
{
  "serviceLevelIndicator": {
    "requestBased": {
      "distributionCut": {
        "distributionFilter":
          "metric.type=\"dataflow.googleapis.com/job/per_stage_sy
            stem_lag\"
            resource.type=\"dataflow_job\"
            resource.label.\"job_name\"=\"scaling_gcp_job\"",
        "range": {
          "min": 0,
          "max": 50
        }
      }
    }
  },
  "goal": 0.90,
  "rollingPeriod": "60s",
  "displayName": "90% data elements processed under 50 s"
}
```

Apart from the preceding two SLIs, there are other SLIs available as well. Two of them, which make sense in the case of data-intensive applications, are as follows:

## [Coverage as an SLI](#)

This is the portion of valid data processed successfully. To determine this, you must first identify a valid record and skip the bad records. Next, you must do a simple count operation of all the good records.

## **Throughput as an SLI**

The portion of time when the data processing rate was higher than the threshold. The most common way to measure the amount of work done regardless of the size of data is bytes processed per second.

## **Conclusion**

Services hosted on GCP push the metrics to Google cloud monitoring and once the metric is pushed, the same could be used to define SLI and SLO for the applications. One key aspect to reaching an SLO number for a service, is to understand the SLO numbers of any service which is been triggered from inside of the original service. For metrics, which are not available, out of the box, one very common strategy is to use the log-based metric to generate logs with appropriate labels or some other information, and then utilize them while defining SLIs.

The recommendation is to use the out-of-the-box metrics since that requires less management by engineering teams.

## **Points to remember**

- Request-response type systems have availability and latency as the two most critical SLIs.
- Data-intensive applications have a freshness rate and correctness SLIs.
- Google service push metric to cloud monitoring, which could be used for SRE purposes.

## **Questions**

1. How will you configure a metric for SLI which is not available out of the box in cloud monitoring?
2. Name SLIs apart from freshness and correctness for data-intensive applications.
3. Name SLIs apart from availability and latency for a request-response use case.

## **Answers**

1. Custom metrics are supported via the log-based metrics strategy.
2. Coverage, and Throughput SLI.
3. Success and Error rate.

# Index

## **A**

- ACID properties [197](#)
- advanced metrics [29](#), [30](#)
  - latency [32](#)
  - response time [31](#)
- Airflow 1.x [8](#)
- Airflow 2.x [8](#)
- Airflow environment
  - active workers, monitoring [223](#), [224](#)
  - commands [225](#), [226](#)
  - DAGs, running [222](#)
  - database CPU and memory usage, monitoring [224](#)
  - environment pre-set [222](#)
  - observing [222](#)
  - optimizing [221](#), [222](#)
  - running and queued tasks, monitoring [224](#)
  - scheduler CPU and memory, monitoring [223](#)
  - task scheduling latency, monitoring [225](#)
  - total parse time of DAGs, monitoring [223](#)
  - web server CPU and memory, monitoring [225](#)
  - worker Pod evictions, monitoring [223](#)
  - workers CPU and memory usage, monitoring [224](#)
- alerts
  - creating [297](#)
- Amazon Elastic Kubernetes Service (EKS) [111](#)
- Amazon Kubernetes Service (AKS) [41](#)
- Amazon Web Services (AWS) [1](#)
- Apache Airflow [209](#)
- Apache Beam pipelines [253](#), [254](#)
- App Engine [151-154](#)
  - standard App Engine, versus flex App Engine [157](#), [158](#)
  - web application, deploying [154-157](#)
- automatic scaling [12](#), [13](#)
- auto scalability [6](#)
- Autoscaler architecture [201](#)
  - Cloud scheduler [201](#)
  - end to end working [202](#), [203](#)
  - poller cloud function [202](#)
  - scalar cloud function [202](#)
- Autoscaler on node group
  - enabling [99](#), [100](#)



- autoscaler configurations [91](#)
  - autoscaling policy, creating based on multiple signals [97](#), [98](#)
  - cloud monitoring metrics based autoscaling [92](#), [93](#)
  - CPU-based autoscaling [91](#)
  - for per group metric [93](#), [94](#)
  - for per instance metric [93](#)
  - load balancing serving capacity based autoscaling [92](#)
  - schedule based scaling [95-97](#)
  - scheduling, based on prediction [97](#)
- Autoscaler deployment topology [203](#)
  - centralized deployment topology [204](#)
  - deployment of Autoscaler per project [204](#)
  - distributed deployment [204](#), [205](#)
- autoscaling [36](#)
  - implementing, in Managed Instances Group [87-90](#)
- autoscaling algorithms
  - BASIC [265](#)
  - NONE [264](#)
- autoscaling, Bigtable
  - benefits [188](#)
  - CPU utilization target [188](#)
  - limitations [191](#), [192](#)
  - manual node allocation [190](#), [191](#)
  - maximum number of nodes [189](#), [190](#)
  - minimum number of nodes [189](#)
  - programmatically autoscaling [191](#)
  - storage utilization target [189](#)
- autoscaling container instances
  - CPU allocation, configuring [173](#), [174](#)
  - maximum and minimum limit, configuring [175](#)
  - maximum concurrency, configuring [174](#), [175](#)
- autoscaling, in Dataproc
  - Autoscaling Policies API [240](#)
  - considerations [237](#), [238](#)
  - working [238](#), [239](#)
- Autoscaling Policies API [240](#)
  - applying, to Dataproc cluster [246](#), [247](#)
  - BasicAutoscalingAlgorithm resource [241](#), [242](#)
  - BasicYarnAutoscalingConfig resource [242](#), [243](#)
  - CRUD operations [245](#), [246](#)
  - InstanceGroupAutoscalingPolicyConfig resources [244](#), [245](#)
  - limitations [247](#), [248](#)
  - resource [240](#), [241](#)
- autoscaling policies, VMware Engine
  - configuring [147-149](#)
- availability metric [3](#), [27](#), [28](#)
- Average Response Time (ART) [31](#)

## B

- BASIC algorithm [265](#)
  - THROUGHPUT\_BASED [265](#)
- Bigtable [181](#)
  - advantages, over HBase [185](#)
  - architecture [185](#), [186](#)
  - atomic writes [183](#)
  - data handling [182](#)
  - data selection [183](#)
  - fast writes [183](#)
  - high throughput [183](#)
  - infrastructural footprint [184](#), [185](#)
  - scaling options [186](#)
  - strong consistency [183](#)
  - versioning changes [183](#)
  - voluminous datasets [182](#)
- Burn Rate threshold [297](#)

## C

- challenges, of scaling
  - cloud native and hybrid deployments [13](#)
  - housekeeping services [14](#)
  - load balancing [13](#), [14](#)
- cloud agnostic
  - advantages [68](#)
  - disadvantages [68](#)
- cloud elasticity [35](#)
  - and cost relationship [43-45](#)
  - benefits [41](#)
  - challenges [45](#), [46](#)
  - defining [36-38](#)
  - examples [38-41](#)
  - versus, scalability [46](#)
- cloud elasticity, benefits
  - considerable capacity [42](#)
  - high availability [42](#)
  - justified costs [42](#)
  - painless and optimal scaling [42](#)
  - redundancy and flexibility [42](#)
  - simple management [43](#)
- cloud elasticity use cases
  - eCommerce application [47-49](#)
  - song streaming application [49](#), [50](#)
- Cloud Functions [176](#), [177](#)
  - event driven functions [176](#)

- HTTP functions [176](#)
  - maximum and minimum instances, configuring [178](#), [179](#)
  - memory, configuring [177](#), [178](#)
  - traffic spikes above max limits, addressing [179](#)
- cloud monitoring dashboard
  - logs, aggregating for [300](#)
- Cloud Run [170](#), [171](#)
  - autoscaling container instances [173](#)
  - features [171](#), [172](#)
  - infrastructural footprint [172](#), [173](#)
- cloud scalability [2](#)
  - auto scalability [6](#)
  - diagonal scaling [6](#), [7](#)
  - horizontal scalability [3](#), [4](#)
  - indirect KPI impact [28](#)
  - versus, elasticity [46](#)
  - vertical scalability [4](#), [5](#)
- cloud scalability metrics
  - advanced metrics [29](#), [30](#)
  - availability [27](#), [28](#)
  - costs [24](#)
  - performance [20](#), [21](#)
  - reliability [23](#)
- cloud scaling
  - automatic scaling [12](#)
  - key challenges [13](#)
  - manual scaling [11](#), [12](#)
  - risks, of improper scaling [16](#)
  - scenarios [10](#), [11](#)
  - scheduled scaling [12](#)
- cloud scaling, benefits [7](#)
  - cost saving [9](#)
  - Disaster Recovery [9](#)
  - ease of use [8](#)
  - flexibility [8](#)
  - global presence [9](#)
  - maintenance [8](#)
  - speed [8](#)
- Cloud Spanner [195](#), [196](#)
  - autoscaling, with Autoscaler [201](#)
  - infrastructural footprint [198](#), [199](#)
  - manual scaling [199-201](#)
  - multi-regional spanner [197](#)
- cluster autoscaler [137](#)
  - scaling limits [138](#)
- clusters
  - decommissioning [248](#)
- Composer Autoscalars [220](#)

- Cluster Autoscaler [221](#)
- Horizontal Pods scalar [221](#)
- node auto provisioning [221](#)
- Composer autoscaling
  - Airflow worker set controller, using [219](#), [220](#)
  - benefits [217-219](#)
  - factors affecting [220](#)
- Compound Metrics [19](#)
- cost KPIs [25](#)
- cost metric [24](#), [25](#)
  - forecasted cost [26](#), [27](#)
  - total cloud cost [26](#)
- CRUD operations, on autoscalers
  - autoscaler, deleting [99](#)
  - autoscaler, describing [98](#)
  - autoscaler, turning off [99](#)
  - autoscaler, updating [99](#)
- Customer Satisfaction Score (CSAT) [29](#)
- custom scale metric
  - AND condition [41](#)
  - OR condition [41](#)

## D

- Dataflow autotuning [260](#)
  - dynamic work rebalancing [264](#)
  - horizontal autoscaling [260](#)
  - scaling, for batch jobs [261](#)
  - scaling, for streaming jobs [261](#), [262](#)
  - streaming pipeline, horizontal scaling [262](#)
  - vertical autoscaling [263](#), [264](#)
- Dataflow infrastructure
  - disk size [257](#), [258](#)
  - machine type [258](#)
  - public IPs, disabling [258](#)
  - right regions, selecting [258](#)
- Dataflow job
  - max nodes, limiting [266](#)
  - persistent disk, scaling [267](#)
- Dataflow job lifecycle [259](#)
  - distribution [259](#)
  - execution graph [259](#)
  - fusion optimization [259](#)
  - optimizations, combining [259](#)
  - parallelization [259](#)
- Dataflow Prime [257](#)
- Dataflow Prime right fitting

- scaling [265](#), [266](#)
- Dataflow shuffle [256](#)
  - for optimizing data shuffle [267](#)
- Dataflow streaming engine [256](#)
- diagonal scaling [6](#), [7](#)
- Directed Acyclic Graph (DAG) [210-212](#)
- Disaster Recovery [2](#)
- Distributed Denial-of-Service (DDoS) attacks [56](#)
- distributed deployments [55](#)
  - analytics hybrid/multi-cloud [56](#)
  - partitioned multi-cloud [55](#)
  - tiered hybrid [55](#)

## E

- elasticity. See cloud elasticity
- Elastic Load Balancing (ELB) [13](#)
- error budget [276](#)
  - tracking [296](#)
- exponential scaling
- fault-tolerant workloads [134](#)

## F

- fault-tolerant workloads
  - exponential scaling [134](#)
- Flex App Engine [163](#)
  - autoscaling, configuring [164](#), [165](#)
  - manual scaling, configuring [164](#)
- forecasted cost [26](#), [27](#)
- Function as A Service (FaaS) [176](#)

## G

- GCP service grouping [308](#)
  - data processing services [308](#), [309](#)
  - data storage and retrieval services [308](#)
  - request response services [308](#)
- Google Cloud Function (GCF) [173](#)
- Google Cloud Platform (GCP) [1](#), [71](#), [73](#)
  - console\UI portal, using [73](#)
  - GCloud commands, using [73](#)
  - REST APIs [73](#)
- Google Cloud Storage (GCS) bucket [176](#)
- Google Composer [209](#), [210](#)
  - components [213](#), [214](#)
  - horizontal scaling options [214](#)

- vertical scaling options [216](#)
- Google Dataflow [251-253](#)
- Google Dataproc [229-234](#)
  - autoscaling [237](#)
  - manual scaling [234-236](#)
- Google Kubernetes Engine (GKE) [41](#), [111](#)
- governance [66](#)
  - communication [66](#)
  - effective planning [67](#)
  - guidelines, creating [66](#)
  - proper auditing [67](#)

## H

- Hadoop Distributed File System (HDFS) [4](#), [181](#)
- Horizontal Pod Autoscaler (HPA) [121](#)
- horizontal Pod scaler [122](#)
- horizontal Pod scaling
  - configuring [121](#), [122](#)
  - metric threshold definition [122](#), [123](#)
  - multiple metrics, configuring [123](#)
- horizontal scalability [3](#), [4](#)
- horizontal scaling [36](#)
- horizontal scaling options, Composer [214](#)
  - maximum and minimum number of worker nodes, adjusting [215](#)
  - number of schedulers, adjusting [215](#)
- hybrid-cloud deployments
  - business drivers [58-60](#)
  - challenges [60-62](#)
  - defining [54](#)
  - IaaS, versus PaaS [62](#)
  - model [57](#), [58](#)

## I

- IaaS versus PaaS [65](#)
  - distributed deployment [64](#)
  - Docker and Kubernetes [65](#)
  - Hadoop cluster [65](#)
  - infrastructure as code [66](#)
  - OpenShift Container Platform [66](#)
  - redundant deployment [62-64](#)
- image registry [115](#)
- indirect KPI impact, cloud scalability [28](#), [29](#)
  - customer satisfaction [29](#)
  - software development and operational KPIs impacts [29](#)
- Infinite scalability [43](#)

- Infrastructure as a Service (IaaS) [3](#)
- Infrastructure as Code tools
  - Ansible [8](#)
  - Terraform [8](#)
- Input/Output Operations Per Second (IOPS) [19](#)
- instance groups [73](#), [75](#)
  - autoscaling [82](#), [83](#)
  - Managed Instance Groups [76](#)
  - unmanaged instance group [80](#)

## K

- KPIs [17](#)
  - defining [18](#), [19](#)
- Kubelet [115](#)
- Kube-proxy [115](#)
- Kubernetes [111](#)
  - application, building [113](#)
  - application, packaging [113](#)
  - application, scaling [118](#)
  - architecture [114](#)
  - image registry [115](#)
  - master node [114](#)
  - worker node [114](#)
- Kubernetes considerations
  - load balancing [140](#)
  - network policies, for scaling [139](#)
  - node pool configurations [139](#)
  - storage [140](#)

## L

- latency metrics [32](#)
  - average end-to-end latency [32](#)
  - number of slow end-to-end transactions [32](#)
  - number of very slow end to end latency times [32](#)
  - throughput [32](#)
- load balancing [107](#)
  - backend service, aligning with MIG [107](#), [108](#)
  - cross regions load balancing [109](#)
  - instance group, adding to load balancer [107](#)
  - MIG, adding to target pool [108](#)
  - multi regional external load balancer, configuring [109](#)
- log based metrics [286-289](#)
- logs
  - aggregating, for setting up cloud monitoring dashboard [300-302](#)
- lookback duration [297](#)

## M

### Managed Instance Group (MIG)

- autoscaling [83-85](#)

- creating [77](#)

- features [76](#), [77](#)

- predictive scaling [85](#), [86](#)

- scaling [83](#)

- Stateful Managed Instance Group [79](#), [80](#)

- Stateless Managed Instance Group [77-79](#)

manual scaling [11](#), [12](#), [37](#)

master node, Kubernetes [114](#)

- API Server [114](#)

- controller [114](#)

- etcd [114](#)

- scheduler [114](#)

Mean Time to Recover (MTTR) [9](#), [304](#)

metrics, for SLIs

- key SLIs [289](#), [290](#)

- log based metrics [286-289](#)

- out of box SLI metrics, using [285](#), [286](#)

- selecting [284](#)

Microsoft Azure [1](#)

multiple metrics, horizontal pod scaling

- autoscaling, on custom metric [129](#), [130](#)

- autoscaling, on external metric [125-129](#)

- autoscaling, on resource utilization [124](#), [125](#)

- configuring [123](#)

- scaling requests, issuing [123](#), [124](#)

- trashing [123](#)

multi-cloud deployments

- business drivers [58-60](#)

- challenges [60-62](#)

- defining [54](#)

- IaaS, versus PaaS [62](#)

- model [56](#), [57](#)

## N

Network Load Balancing (NLB) [108](#)

node groups

- autoscaling [99](#), [100](#)

NONE algorithm [264](#)

## O

out of box SLI metrics



using [285](#), [286](#)

## P

performance metric [20](#), [21](#)  
    big data use case [21](#), [22](#)  
    microservices use case [22](#)  
    REST API use case [22](#), [23](#)  
pipeline execution graph [259](#)  
Platform as a Service (PaaS) [3](#)  
pods [115](#)  
Point-In-Time Recovery (PITR) [197](#)  
pre-defined configurations  
    accelerator optimized machines [74](#)  
    compute optimized [74](#)  
    general purpose [74](#)  
    memory optimized [74](#)  
predictive scaling [37](#)  
preemptible VMs  
    using [136](#), [137](#), [249](#)  
private cloud  
    creating [145-147](#)  
Probability of Failure on Demand (POFOD) [24](#)

## R

Rate of Occurrence of Failure (ROCOF) [24](#)  
redundant deployments [55](#)  
    business continuity multi-cloud or hybrid cloud storage [55](#)  
    cloud bursting [55](#)  
    hybrid environment [55](#)  
reliability metric [3](#), [23](#)  
    mean time to failure [23](#), [24](#)  
    mean time to repair [24](#)  
    Probability of Failure on Demand (POFOD) [24](#)  
    Rate of Occurrence of Failure (ROCOF) [24](#)  
request based SLI [285](#)  
reservations, consuming  
    instances, consuming from matching reservation [104](#), [105](#)  
    instances, creating without [107](#)  
    specific shared reservation, consuming [106](#)  
reserved resources  
    consuming [104](#)  
    consumption type [101](#), [102](#)  
    for effective auto scaling [101](#)  
    shared project zonal reservations [103](#), [104](#)  
    shared type [101](#)

- single project zonal reservations [102](#), [103](#)
- response time metric
  - average response time [31](#)
  - data in and out [31](#)
  - infrastructure utilization [31](#)
  - peak response time [31](#)
  - request per second [31](#)
- runbooks [274](#)

## S

- Saturation Point [30](#)
- scale
  - versus, cost relationship [14-16](#)
- scale-in controls [86](#)
  - maximum allowed reduction [86](#)
  - trailing time window [86](#)
- scaling [83](#)
- scaling in Kubernetes [118](#)
  - horizontal pod scaling, configuring [121](#), [122](#)
  - multi-dimensional pod scaling, configuring [132](#), [133](#)
  - scale-down mechanism [120](#)
  - scale-up mechanism [119](#), [120](#)
  - vertical pod scaling, configuring [130-132](#)
- scaling options, Bigtable
  - autoscaling [187](#)
  - triggers, autoscaling [186](#), [187](#)
- scaling strategies, as per load [206](#)
  - linear scale up [207](#)
  - stepwise scale up [206](#)
- scheduled scaling [12](#), [36](#), [37](#)
- security [3](#)
- Service Level Agreement (SLA) [3](#), [10](#), [276](#)
- Service Level Indicators (SLI) [275](#), [276](#)
  - creating [291](#), [292](#)
- Service Level Objectives (SLO) [275](#)
  - alerts [297](#)
  - creating [293-295](#)
  - setting [290](#), [291](#)
- service monitoring
  - with Google Cloud monitoring [276-284](#)
- Simple Metrics [19](#)
- Single Point of Failure (SPOF) [28](#), [56](#)
- Site Reliability Engineering (SRE) [14](#), [271](#)
  - actions, automating [304](#)
  - drills [304](#)
  - incident management [302](#), [303](#)

- playbook maintenance [303](#)
- principles [273](#)
- typical SRE process [274](#)
- SLO alerts
  - probes and uptime checks [298](#), [299](#)
- S.M.A.R.T. [20](#)
- Software as a Service (SaaS) [3](#)
- Spot Pods
  - using [134-136](#)
- Spot VMs
  - using [136](#)
- SRE practices big data world [314](#), [315](#)
  - correctness SLI [316](#)
  - freshness SLI [317](#)
  - SLI coverage [318](#)
  - SLI throughput [318](#)
- SRE practices, in microservices world [309](#), [310](#)
  - availability [311](#), [312](#)
  - latency [312](#), [313](#)
- Standard App Engine
  - automatic scaling [159](#)
  - autoscaling scaling, configuring [161-163](#)
  - basic scaling [158](#)
  - basic scaling, configuring [160](#)
  - manual scaling [158](#)
  - manual scaling, configuring [161](#)
  - scaling, configuring [160](#)
- Stateful Managed Instance Group [79](#), [80](#)
- Stateless Managed Instance Group [77-79](#)

## T

- target pool [108](#)
- total cloud cost
  - KPI, defining for cost in non-production environments [26](#)
  - KPI, defining for cost in production environments [26](#)
- Total Cost to Ownership (TCO) [14](#)
- typical SRE process
  - defining [274](#)

## U

- unmanaged instance group [80](#), [81](#)

## V

- Vertical Pod Autoscaling (VPA) [131](#)

- vertical scalability [4](#), [5](#)
- vertical scaling [36](#)
- vertical scaling options, Composer
  - environment size, adjusting [217](#)
  - parameters, adjusting [216](#)
- VMware Engine [143-145](#)

## W

- web application, Kubernetes
  - building [115](#), [116](#)
  - deploying [117](#), [118](#)
- window-based SLI [285](#)
- Wordcount Dataflow job [254](#), [255](#)
  - Dataflow Prime [257](#)
  - Dataflow Shuffle service [256](#)
  - Dataflow streaming engine [256](#), [257](#)
  - fusion optimization [255](#)
  - infrastructure, configuring [257](#)
  - optimizations, combining [256](#)
- worker node, Kubernetes [114](#)
  - container runtime [115](#)
  - Kubelet [115](#)
  - Kube-proxy [115](#)
  - pods [115](#)

## Y

- Yet Another Resource Negotiator (YARN) cluster [229](#)