

**grokking**

# continuous delivery

Christie Wilson





**MEAP Edition**  
**Manning Early Access Program**  
**Grokking Continuous Delivery**  
**Version 6**

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](https://manning.com)

# welcome

---

Thank you for purchasing *Grokking Continuous Delivery*!

This book is for everyone who does the nitty-gritty day-to-day job of building software. Whether you build frontends, backends, tools, or infrastructure, this book is for you!

To get the most benefit from this book, you'll want to have some familiarity with the basics of Linux, programming language concepts, and testing. You'll also want to have some experience with version control, HTTP servers and containers. You don't need deep knowledge on any of these topics; and if needed you could definitely research them as you go.

I have been super passionate about Continuous Delivery (CD) for most of my career. I've often started a new position with the intention of switching my focus to something different, but it's such an intriguing space that I always find myself pulled back in. CD is at the heart of modern software development, and as software development becomes more and more ambitious, CD is the mechanism that enables it. At the same time, it's a field where it's hard to get your hands on concrete best practices and actions that you can take as an engineer; so many resources are aimed at selling the concepts to managers and directors or are tied to some specific vendor's product.

In this book, I hope you'll find practical takeaways for effectively practicing CD on your team, regardless of what space you're in or what language you're using. I'll be talking about the basic building blocks you'll need to have in place, but I won't recommend any specific CD tools: you'll be able to use the recommendations in the book to evaluate the tools available and make the best choices for your particular needs.

Consider this the missing manual for how to get started with CD and apply it effectively! As you read, if you notice any missing topics or details, please let me know in the [liveBook discussion forum](#). CD is a huge topic that spans the entire development process and multiple roles. With your feedback we can get the right balance of information to set folks up for success in this exciting and essential space!

You might notice that chapter 3 is missing, but don't worry; it is a work in progress and will be added soon. You may also notice that some chapters have exercises while some do not, and that is something you should also see fixed in subsequent updates.

Until then, happy reading!

—Christie Wilson

# *brief contents*

---

## **PART 1: INTRO**

- 1 Welcome*
- 2 A basic pipeline*

## **PART 2: KEEPING SOFTWARE IN A DELIVERABLE STATE AT ALL TIMES**

- 3 Version control is the only way to roll*
- 4 Use linting effectively*
- 5 Dealing with noisy tests*
- 6 Speeding up slow test suites*
- 7 Give the right signals at the right times*

## **PART 3: MAKING DELIVERY EASY**

- 8 Easy delivery starts with version control*
- 9 Building*
- 10 Deploying*

## **PART 4: PIPELINE DESIGN**

- 11 Starter pack: go from 0 to CD*
- 12 Scripts*
- 13 Graph design*

## **APPENDICES:**

- A CD systems*
- B Version control systems*



# Welcome

# 1



---

## In this chapter:

- why should you care about Continuous Delivery?
- understand the history of Continuous Delivery, Continuous Integration, Continuous Deployment and CI/CD
- define the different kinds of software that you might be delivering and explain how Continuous Delivery applies to them
- define the elements of Continuous Delivery: Keeping software in a deliverable state at all times; Making delivery easy

---

Hi there! Welcome to my book! I'm so excited that you've decided to not only learn about Continuous Delivery, but really understand it. That's what this book is all about: learning what you need to do to have Continuous Delivery really work for you on a day to day basis.

## 1.1 Do you need Continuous Delivery?

The first thing you might be wondering is if it's worth your time to learn about Continuous Delivery, and even if it is, is it worth the hassle of applying it to your projects?

The quick answer is YES if the following is true for you:

1. You are making software professionally
2. More than one person is involved in the project

If both of those are true for you, Continuous Delivery is worth investing in. *Even if just one is true*, (you're working on a project for fun with a group of people, or you're making professional software solo), you won't regret investing in Continuous Delivery.

*"But wait - you didn't ask what I'm making. What if I'm working on kernel drivers, or firmware, or microservices? Are you sure I need Continuous Delivery?" - You*

It doesn't matter! Whatever kind of software you're making, you'll benefit from applying the principles in this book. The elements of Continuous Delivery that we'll be explaining in this book are built on the principles that we've been gathering ever since we started making software; they're not a trend that will fade in and out of popularity, they are the foundations that will remain whether we're making microservices, monoliths, distributed container based services, or whatever comes next.

In this book we'll be covering the fundamentals of Continuous Delivery and will give you some examples of how you can apply them to your project; the exact details of how you do Continuous Delivery will probably be unique and you might not see them exactly reflected in this book, but what you WILL see is the components you need to put it together, and the principles to follow to be the most successful.

***But I don't need to deploy anything!***

Good point!

Deployment and the related automation are an exception and do NOT apply to all kinds of software - but Continuous Delivery is about far more than just deployment. We'll get into this in the rest of this chapter.

## 1.2 Why Continuous Delivery?

What's this thing we're here to learn about anyway? There are a lot of definitions out there, but before we get into those, I want to tell you what Continuous Delivery (CD) means to me, and why I think it's so important:

**Continuous Delivery is the process of modern professional software engineering.**

**Modern:** Professional software engineering has been around way longer than CD - though those folks working with punch cards would have been ecstatic for CD! One of the reasons why we can have CD today, and we couldn't then, is that CD costs a lot of CPU cycles. To have CD, you run a lot of code!

I can't even imagine how many punch cards you'd need to define a typical CD workflow!

**Professional:** If you're writing software for fun, it's kind of up in the air whether you're going to want to bother with CD. For the most part, CD is the processes you put in place when it's really important that the software works. The more important it is, the more elaborate the CD. And when we're talking about professional software engineering, we're probably not talking about one person writing code on their own. Most engineers will find themselves working with at least a few other people, if not hundreds, possibly working on exactly the same codebase.

**Software engineering:** Other engineering disciplines come with bodies of standards and certifications that are largely lacking when it comes to software engineering. So let's simplify it: software engineering is writing software. When we add the modifier "professional", we're talking about writing software professionally.

**Process:** Writing software professionally requires a certain approaches to ensure that the code we write actually does what we mean it to. These processes are less about how one software engineer is writing code (though that's important too!), and more about how that engineer is able to work with other engineers to deliver professional quality software.

Continuous Delivery is the collection of processes that we need to have in place to ensure that multiple software engineers, writing professional quality software, can create software that does what they want.

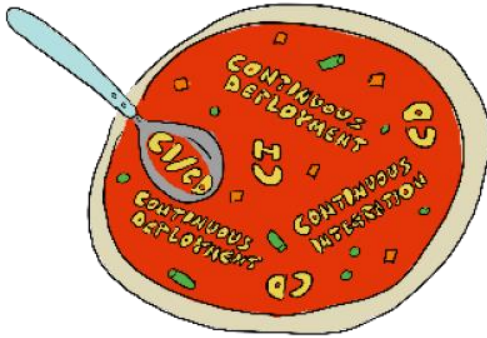


### QUESTION

**Q** Wait are you saying CD stands for Continuous Delivery? I thought it meant Continuous Deployment!

**A** Some people do use it that way, and the fact that both terms came into existence around the same time made this very confusing. Most of the literature I've encountered (not to mention the CD Foundation!) favors using CD for Continuous Delivery, so that's what this book will use.

## 1.3 Continuous Word Soup



Let's take a quick look at the evolution of these terms to understand more.

Continuous Integration, Continuous Delivery and Continuous Deployment are all phrases that were created intentionally (or in the case of Continuous Integration, evolved), and the creators had specific definitions in mind.

CI/CD is the odd one out: no one seems to have created this phrase. It seems to have popped into existence because lots of people were trying to talk about all the different continuous activities at the same time and needed a short form. (CI/CD/CD didn't take for some reason!)

The phrase CI/CD as it's used today refers to the tools and automation required for any and all of Continuous Integration, Delivery and Deployment.



You might be thinking: okay Christie, that's all well and good, but what does deliver actually mean? And what about Continuous Deployment? What about CI/CD?

It's true, we've got a lot of phrases to work with! And to make matters worse, people don't use them consistently. In their defense, that's probably because some of them don't even have definitions!

- 2016: "CI/CD" entry added to Wikipedia
- 2014: Earliest article mentioning "CI/CD"
- 2010: "Continuous Delivery" practice defined in book of the same name
- 2009: "Continuous Deployment" coined in blog post
- 1999: "Continuous Integration" practice defined in *Extreme Programming Explained*
- 1994: "Continuous Integration" coined in *Object Oriented Analysis and Design with Applications*

## 1.4 Continuous Delivery (CD)

Continuous Delivery is the collection of processes that we need to have in place to ensure that multiple software engineers, writing professional quality software, can create software that does what they want.

My definition captures what I think is really cool about CD, but it's far from the usual definition you'll encounter. Let's take a look at the definition of **Continuous Delivery** used by the Continuous Delivery Foundation (CDF):

A software development practice where working software is released to users as quickly as it makes sense for the project and built in such a way that it has been proven that this can safely be done at any time.

If you start to break this down, you'll notice there are two big pieces to CD. You're doing **Continuous Delivery** when:

1. You can safely deliver changes to your software at any time
2. Delivering that software is as simple as pushing a button

This book will be going into detail about the activities and automation that will help you achieve these two goals. Specifically:

1. To be able to safely deliver your changes at any time, you must always be in a deliverable state. The way to achieve this is with Continuous Integration (CI).
2. Once these changes have been verified with CI, the processes to deliver the changes should be automated and repeatable.

The big shift that CD represents over just CI is redefining what it means for a feature to be **done**. With CD, done means delivered. And the process for getting from implementation to delivered change is automated, easy, and fast.

Before we start digging into how you can achieve these goals in the next chapters, let's break these terms down a bit further.

A really interesting detail is that that **Continuous Delivery** is a set of goals that we aim for; the way we get there might vary from project to project. That being said, there are activities that have emerged as the best ways we've found for achieving these goals, and that's what this book is about!

## 1.5 Integration

**Continuous Integration (CI)**, is the oldest of the terms we're dealing with - but still a key piece of the Continuous Delivery pie. Let's start even simpler with looking at just **integration**.

What does it mean to **integrate software**? Actually part of that phrase is missing - to integrate you need to integrate something into something else. And in software, that something is code changes. When we're talking about integrating software, what we're really talking about is:

### **Integrating code changes into existing software.**

This is the primary activity that software engineers are doing on a daily basis: changing the code of some existing piece of software.

This is especially interesting when you look at what a team of software engineers does: they are constantly making code changes, often to the same piece of software. Combining those changes together is **integrating** them.

**Software integration is the act of combining together code changes made by multiple people.**

As you have probably personally experienced, this can really go wrong sometimes. For example, when I make a change to the same line of code as you do, and we try to combine those together, we have a conflict and have to manually decide how to integrate those changes.

On some rare occasions we may be creating software for the very first time, but from every point after the first successful compile, we are once again integrating changes into existing software!

There's one more piece missing from this definition; when we integrate code changes we do more than just putting the code changes together, *we also verify that the code works*. You might say that "V" for Verification is the missing letter in CI! Verification has been packed into the Integration piece, so when we talk about software integration, what we really mean is:

**Software integration is the act of combining together multiple code changes made by multiple people and verifying that the code does what it was intended to do.**



### **QUESTION**

**Q**  
**A**

Who cares about all these definitions? Show me the code already!!

It's hard to be intentional and methodical about what we're doing if we can't even define it. Taking the time to arrive at a shared understanding (via a definition) and getting back to core principles is the most effective way to level up!



## 1.6 Continuous Integration



Let's put the **continuous** into **continuous integration** with an example outside of software engineering.

Holly is a chef and she's cooking pasta sauce. She starts with a set of raw ingredients: onions, garlic, tomatoes, spices. In order to cook, she needs to **integrate** these ingredients together, in the right order and the right quantities, to get the sauce that she wants.

To accomplish this, every time she adds a new ingredient, *she takes a quick taste*. Based on the flavor, she might decide to add a little extra, or realize she wants to add an ingredient she missed.

By tasting along the way, she's evolving the recipe through a series of integrations. Integration here is expressing two things:

- Combining the ingredients
- Checking to verify the result

And that's what the **integration** in **continuous integration** means: combining code changes together, and also verifying that they work. **Combine and verify.**

Holly repeats this process as she cooks. If she waited until the end to taste the sauce, she'd have a lot less control and it might be too late to make the needed changes. That's where the **continuous** piece of **continuous integration** comes in. We want to be integrating (combining and verifying) our changes as frequently as we possibly can - as soon as you can.

And when we're talking about software, what's the soonest we can combine and verify? As soon as we make a change.

**Continuous Integration is the process of combining code changes frequently, where each change is verified on check in.**

Combining code changes together means that engineers using continuous integration are committing and pushing to shared version control every time we make a change, and they are verifying those changes work together by applying automated verification, including tests and static analysis.

Automated verification? Static analysis? Don't worry if you don't know what those are all about, that's what this book is here for! In the rest of the book, we'll be looking at how to create the automated verification that makes continuous integration work.

## 1.7 What do we deliver?

Now as we transition from looking at Continuous Integration to Continuous integration, we need to take a small step back. Almost every definition we explore is going to make some reference to delivering some kind of software (for example, we're about to start talking about *integrating and delivering change to software*). Probably good to make sure we're all talking about the same thing when say **software** - and depending on the project you're working on, it can mean some very different things.

When you are delivering software, there are several different forms of software you could be making (and integrating and delivering each of these will look slightly different):

**Library:** If your software doesn't do anything on its own, but is intended to be used as part of other software, it's probably a library

**Binary:** If your software is intended to be run, it's probably a binary executable of some kind. This could be a service or application, it could be a tool which is run and completes, or it could be an application which is installed onto a device like a tablet or phone.

**Configuration:** This refers to information that you can provide to a binary to change its behavior without having to recompile it. Typically this corresponds to the levers that a system administrator had available to make changes to running software.

**Image:** Container images are a specific kind of binary that are currently an extremely popular format for sharing and distributing services with their configuration, so they can be run in an operating system agnostic way.

**Service:** In general services are binaries that are intended to be up and running at all times, waiting for requests that they can respond to by doing something or returning information. Sometimes there are also referred to as applications.



The term software exists in contrast to hardware. Hardware is the actual physical pieces of our computers, i.e. the machines we do things with. And we do those things by providing the physical machines with instructions. Instructions can be built directly into hardware, or they can be provided to hardware when it runs via software.

### VOCAB TIME

At different points in your career you may find yourself dealing with some or all of the above kinds of software. But regardless of the particular form you are dealing with, in order to create it, you need to **integrate** and **deliver** changes to it.

## 1.8 Delivery

What it means to **deliver** changes software depends on what you are making, who is using it and how. Usually delivering changes refers to one or all of: building, releasing and deploying:

**Building:** Building software is the act of taking code (including changes) and turning it into the form required for it to be actually used. This usually means compiling the code written in a programming language into a machine language. Sometimes it also means wrapping the code into a package, such as an image, or something that can be understood by a package manager (e.g. pypi for Python packages).

Building is also done as part of integration; part of ensuring that changes work together will be building the software.

**Publishing:** You publish software by copying it to a software repository (a storage location for software). For example by uploading your image or library to a package registry.

**Deploying:** This is the act of copying the software where it needs to be to run and putting it into a running state.

You can **deploy** without **releasing**, e.g. deploying a new version of your software but not directly any traffic to it.

**Releasing:** You release software by making it available to your users. This could be by uploading your image or library to a repository, or by setting a configuration value to direct a percentage of traffic to a deployed instance.



### VOCAB TIME

We've been building software for as long as we've had programming languages. This is such a common activity that the earliest systems that did what we now call Continuous Delivery were called build systems. This terminology is so prevalent that even today you will often people refer to the build and what they usually mean is one or more phases in a CD pipeline (more on these in chapter 2!).

## 1.9 Continuous Delivery/Deployment

Now we know what it means to deliver software changes, but what does it mean when we say that delivery is **continuous**?

When we looked at Continuous Integration (CI), we learned that in that context **continuous** means "as soon as possible". Is that the case for **Continuous Delivery (CD)**? Yes and no.

The way that Continuous Delivery uses continuous actually would be better represented as a continuum:



Your software should be proven to be in a state where it could be built, released and/or deployed at any time - but how frequently you choose to deliver that software is up to you.

Around this time you might be wondering, "What about **Continuous Deployment**? How does that fit in?"

2010: "Continuous Delivery"  
practice defined in book of the  
same name

2009: "Continuous Deployment"  
coined in blog post

That's a great question. Looking at the history again, you'll notice that the two terms, Continuous Delivery and Continuous Deployment, came into existence pretty much back to back. What was going on when these terms were coined?

This was an inflection point for software: the old ways of creating software, which relied on humans doing things manually, a strong dev and ops divide (interestingly the term "devops" appeared at around the same time)

and sharply delineated processes (e.g. "testing phase") were starting to shift (left).



**Shifting left** is a process where efforts are made to find defects as early as possible in the software development process.

### VOCAB TIME

Both Continuous Deployment and Continuous Delivery were naming the set of practices that emerged at this time.

Let's look at the definition of Continuous Deployment:

**Working software is released to users automatically on every commit.**

Continuous Deployment is an optional step beyond Continuous Delivery. Whether you go this far is up to you and what your project needs.

The key is that Continuous Delivery enables Continuous Deployment; always being in a releasable state and automating delivery frees you up to decide what is best for your project.

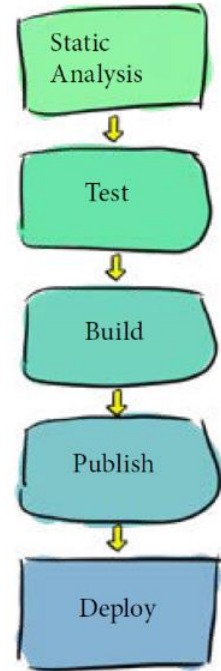
## 1.10 Elements of Continuous Delivery

The rest of this book will show you the fundamental building blocks of Continuous Delivery:

A software development practice where working software is released to users as quickly as it makes sense for the project and built in such a way that it has been proven that this can safely be done at any time.

You will learn how to use Continuous Integration (CI) to always be in a releasable state, and you will learn how to make delivery automated and repeatable, allowing you to choose whether you want to go to the extreme of delivering on every change (Continuous Deployment), or you'd rather deliver on some other cadence, but confident in the knowledge that you have the automation in place to deliver as frequently as you need.

And at the core of all of this automation will be your Continuous Delivery pipeline. In this book we'll dig into each of these tasks and what they look like. You'll find that no matter what kind of software you're making, many of these tasks will be useful to you.



### QUESTION

- Q** Pipeline? Task? What are those?  
**A** Read the next chapter to find out!

Let's look back at the different forms of software we explored and what it means to deliver each of them:

	Delivery includes building?	Delivery includes publishing?	Delivery includes deploying?	Delivery includes releasing?
Library	Depends	Yes	No	Yes
Binary	Yes	Usually	Depends	Yes
Configuration	No	No	Usually	Yes
Image	Yes	Yes	Depends	Yes
Service	Yes	Usually	Yes	Yes

### 1.11 Conclusion

There are a lot of terms in the Continuous Delivery space, and a lot of contradictory definitions. In this book, we use CD to refer to Continuous Delivery, and we'll be focusing on how to setup the automation you need in order to use CD for whatever kind of software you're delivering.

### 1.12 Summary

- Continuous Delivery is useful for all software, it doesn't matter what kind of software you're making.
- To enable teams of software developers to make professional quality software, you need Continuous Delivery.
- To be doing Continuous Delivery, you use Continuous Integration to make sure your software is always in a deliverable state.
- Continuous Integration is process of combining code changes frequently, where each change is verified on check in.
- The other piece of the Continuous Delivery puzzle is the automation required to make delivery as easy as pushing a button.
- Continuous Deployment is an optional step you can take if it makes sense for your project, where software is automatically delivered on every commit.

### 1.13 Up next . . .

We're going to learn all about the basics and terminology of Continuous Delivery automation, setting up the foundation for the rest of the book!



## A basic pipeline

# 2



---

### In this chapter:

- terminology that will be used in this book for basic building blocks: pipelines and tasks
- elements of a basic CD pipeline: static analysis, testing, building, publishing, deploying
- the role of automation in the execution of pipelines: webhooks, events and triggering.
- how the varied terminology in the CD space relates: Tasks, Stages, Pipelines, Workflows, Steps, Jobs, Nodes, Runners, Executors, Events, Triggers, Builds, Webhooks, Agents

---

Before we get into the nitty gritty of how to create great Continuous Delivery (CD) pipelines, let's zoom out and take a look at pipelines as a whole. In this chapter we'll look at some pipelines at a high level and identify the basic elements you should expect to see in most CD pipelines.

## 2.1 Cat Picture Website

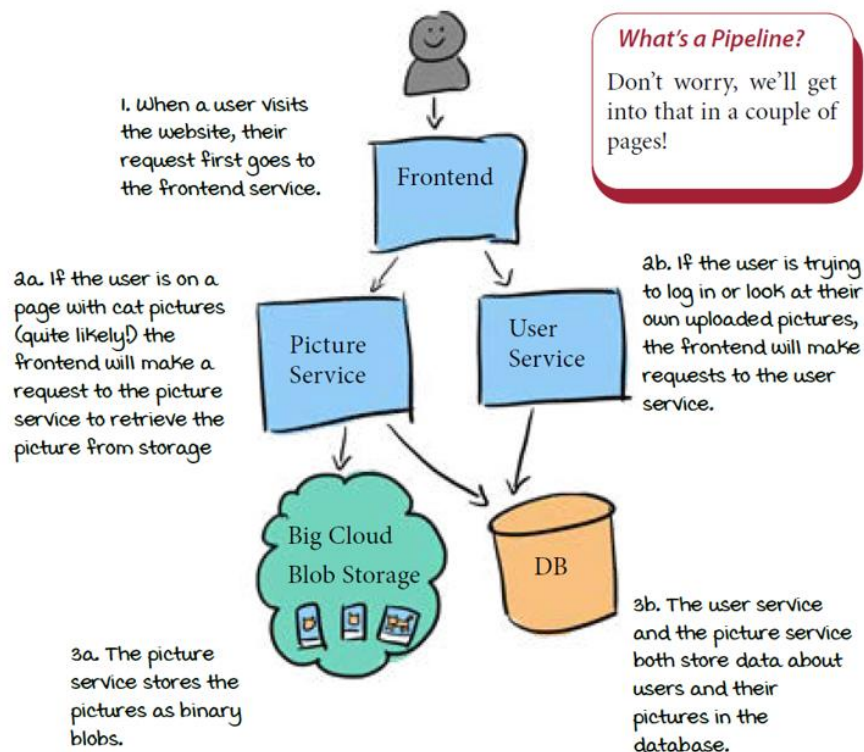
To understand what goes into basic CD Pipelines, we'll take a look at the Pipelines used for The Cat Picture Website.

The Cat Picture Website is the best website around for finding and sharing cat pictures! The way it's built is relatively simple, but since it's a very popular website, the company that works on it (Cat Picture Inc.) has architected it into several services.

### What's CD again?

We use CD in this book to refer to Continuous Delivery. See chapter 1 for more!

They run Cat Picture Website in the cloud (their cloud provider is called Big Cloud Inc) and they use some of Big Cloud's services, such as Big Cloud Blob Storage Services (BCBSS).



### What's a Pipeline?

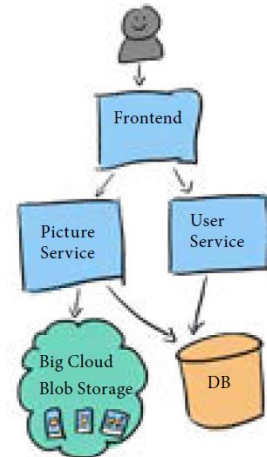
Don't worry, we'll get into that in a couple of pages!

## 2.2 Cat Picture Website Source Code

The architecture diagram tells us how the cat picture website is architected, but in order to understand the CD pipeline there's another important thing to consider: where does the code live?

In Chapter 1 we looked at the elements of Continuous Delivery, half of which is about using Continuous Integration (CI) to ensure we are always in a releasable state. Let's look at the definition again:

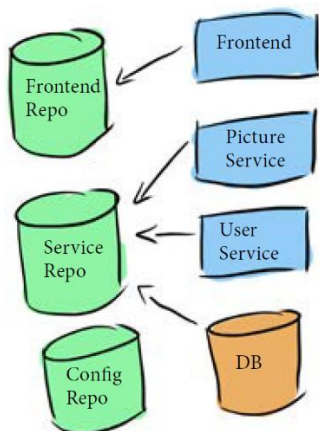
**Continuous Integration (CI)** is process of combining code changes frequently, where each change is verified on check in.



When we look at what we're actually doing when we do CD, we can see that the core is code changes. This means that the input to our CD pipelines is the source code. In fact this is what sets CD pipelines apart from other kinds of workflow automation: CD pipelines almost always take source code as an input.

### Version Control

Using a version control system such as git is a prerequisite for CD; without having your code stored with history and conflict detection, it is practically impossible to have CD.



Before we look at the Cat Picture Website CD pipelines, we need to understand how their source code is organized and stored.

The folks working on Cat Picture Website store their code in several code repositories (repos):

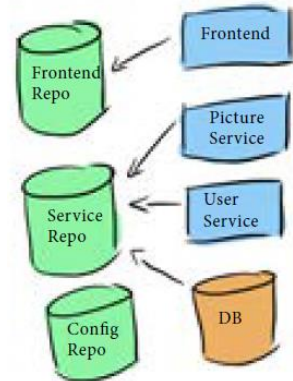
- The Frontend Repo holds the code for the Frontend.
- The Picture Service, User Service and the database schemas are all stored in the Service repo.
- Lastly, Cat Picture Website uses a gitops approach to configuration management (more on this in Chapter 11), so their configuration is stored in the Config Repo.

There are lots of other ways they could have organized their code, all with their own pros and cons.

## 2.3 Cat Picture Website Pipelines

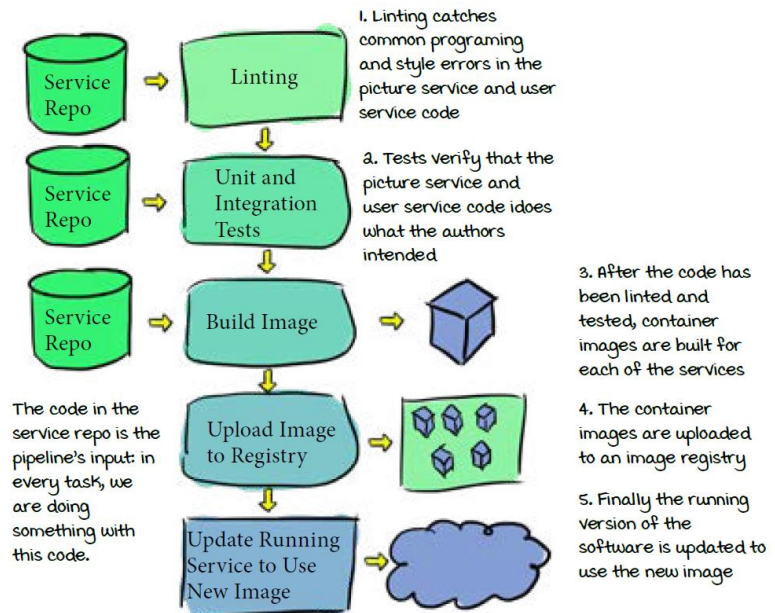
Since Cat Picture Website is made up of several services and all the code and configuration needed for it is spread across several repos, it is managed by several CD pipelines. We'll go over all of these pipelines in detail in future chapters where we examine more advanced pipelines, but for now we're going to stick to the basic pipeline that is used for the User Service and the Picture Service.

Since these two services are so similar, the same pipeline is used for both, and that pipeline will show as all of the basic elements we'd expect to see in a pipeline.



### VOCAB TIME

**Container images** are executable software packages that contain everything needed to run that software.



When does this thing actually get run? We'll get to that in a few pages, and go in depth in chapter 6.

Not only is this the pipeline used for the cat picture website, this pipeline has the basic elements that you'll see in my pipelines!

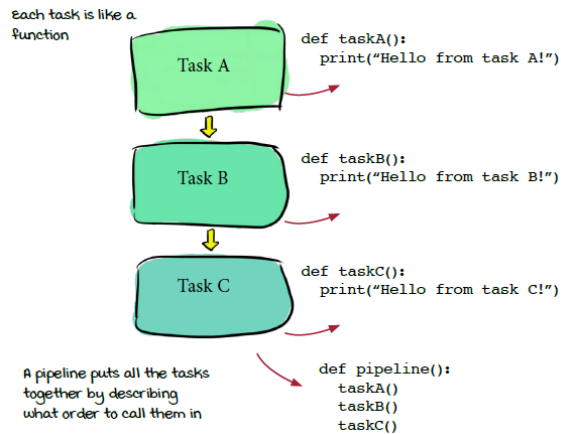
## 2.4 What's a pipeline? What's a task?

We just spent a few pages looking at the Cat Picture Website pipeline, but what is a “pipeline” anyway?

There's a lot of different terminology in the CD space. Where we're using the term “pipeline”, some CD systems use other terms like “workflow”. We'll have an overview of this terminology at the end of the chapter, but for now let's take a look at the terminology we'll be using in this book: pipelines and tasks.

Tasks are individual things you can do: you can think of them a lot like functions. And pipelines are like the entrypoint to code, which calls all the functions at the right time, in the right order.

Below is a **pipeline**, with 3 **tasks**: Task A runs first, then Task B, then Task C.



CD Pipelines will get run again and again; we'll talk more about when in a few pages. If we were to run the pipeline() function (representing the Pipeline on the left), we'd get this output:

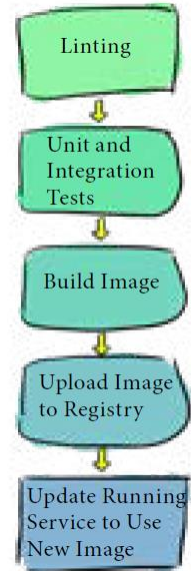
```
Hello from task A!
Hello from task B!
Hello from task C!
```

## 2.5 The basic tasks in a CD pipeline

The Cat Picture Website pipeline shows us all of the basic tasks that you will see in most pipelines. We'll be looking at these basic tasks in detail in the next chapters.

Let's review what each task in this pipeline is for:

- **Linting** catches common programming and style errors in the picture service and user service code
- **Unit and integration tests** verify that the picture service and user service code does what the authors intended
- After the code has been linted and tested, the **build image** task builds container images for each of the services
- Next we **upload the container images** to an image registry
- Finally the running version of the software is **updated to use the new images**



Each of the tasks in the cat picture website pipeline is representative of a basic pipeline element:

- **Linting** is the most common form of static analysis in CD pipelines
- Unit and integration tests are forms of **tests**
- These services are built into images; to use most software you need to **build** it into some other form before it can be used
- Container images are stored and retrieved from registries; as we saw in chapter 1, some kinds of software will need to be **published** in order to be used
- Cat Picture Website needs to be up and running so users can interact with it. Updating the running service to use the new image is how Cat Picture Website is **deployed**.

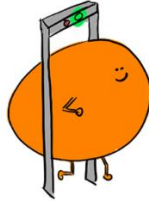
These are the basic types of tasks you'll see in a CI/CD pipeline:



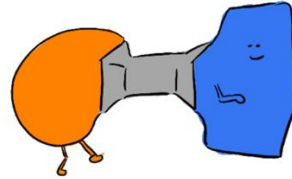


## 2.6 Gates and Transformations

Some tasks are about verifying your code. They are quality **gates** that your code has to pass through.



Other tasks are about changing your code from one form to another. They are **transformations** on your code: your code goes in as input and comes out in another form.



Looking at the tasks in a CD pipeline as gates and transformations goes hand in hand with the elements of Continuous Delivery. In chapter 1 we learned that you're doing Continuous Delivery when:

1. You can safely deliver changes to your software at any time
2. Delivering that software is as simple as pushing a button

If you squint at those, they map 1:1 to gates and transformations:

- **Gates** verify the quality of your code changes, ensuring it is safe to deliver them.
- **Transformations** build, publish, and, depending on the kind of software, deploy your changes.

And in fact, the **gates** usually comprise the **Continuous Integration (CI)** part of your pipeline!

CI is all about verifying your code! You'll often hear people talk about "running CI" or "CI failing" and usually they're referring to **gates**.

## 2.7 CD: Gates and Transformations

Let's look at our basic CD tasks again and see how they map to **gates** and **transformations**:

- Code goes into **gating tasks** and they either pass or fail. If they fail, the code should not continue through the pipeline.
- Code goes into **transformation tasks** and it changes into something completely different or changes are made to some part of the world using it.

**Linting** is all about looking at the code and flagging common mistakes and bugs, but without actually running the code. Sounds like a **gate** to me!



**Testing** activities verify that the code does what we intended it to do. Since this is another example of code verification, this sounds like a **gate** too.



**Building** code is about taking code from one form and transforming it into another form so that it can be used. Sometimes this activity will catch issues with the code, so it has aspects of CI, however in order to test our code, we probably need to build it, so the main purpose here is to **transform** (build) the code.



**Publishing** code is about putting the built software somewhere so that it can be used. Putting the software somewhere where it can be used is part of releasing that software. (For some code, such as libraries, this is all you need to do in order to release it!) This sounds like a kind of **transformation** too.



Lastly, **deploying** the code (for kinds of software that need to be up and running) is a kind of **transformation** of the state of the built software.



### QUESTION

**Q**

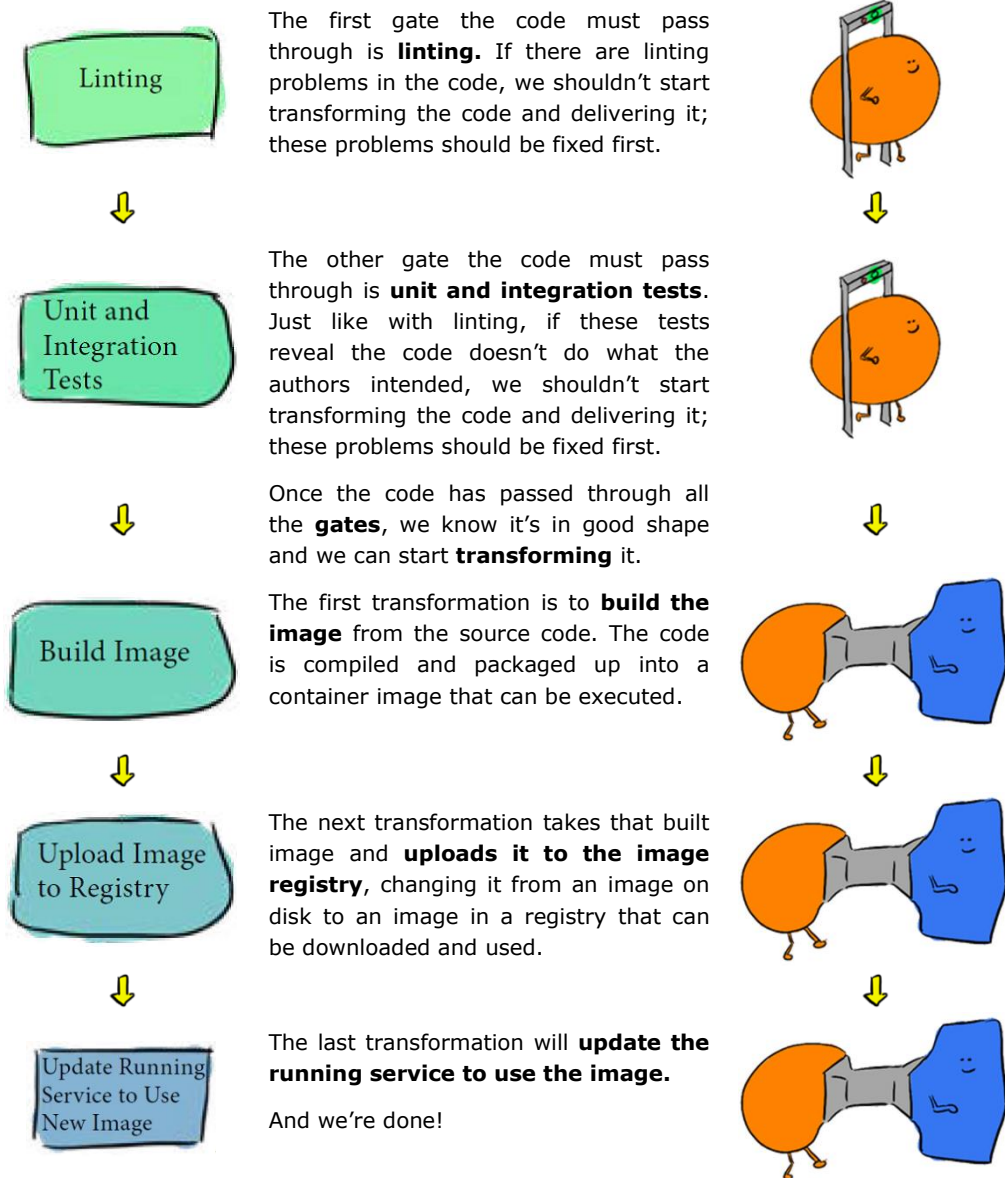
Okay you said the gates are the CI tasks - are you saying CI is just about tests and linting? I remember before Continuous Delivery, CI including building too.

**A**

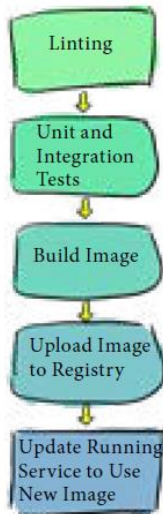
I hear you! CI does often include building, and sometimes folks throw publishing in there too. What really matters is having a conceptual framework for these activities, so in this book we choose to treat CI as being about verification, and not building/publishing/deploying/releasing.

## 2.8 Cat Picture Website Service Pipeline

What does the Cat Picture Website service pipeline look like if we view it as a pipeline of gates and transformations?



## 2.9 Running the pipeline



You might be starting to wonder how and when this pipeline actually gets run. That's a great question! The process evolved over time for the folks at Cat Picture Website Inc.

When Cat Picture Website Inc. started, there were only a few engineers: Topher, Angela and Sato. Angela wrote the cat picture website service pipeline in python and it looked like this:

```
def pipeline(source_repo, config_repo):
    linting(source_repo)
    unit_and_integration_tests(source_repo)
    image = build_image(source_repo)
    image_url = upload_image_to_registry(image)
    update_running_service(image_url, config_repo)
```

This is a simplification of the code Angela wrote, but it's enough info for us to use for now!

The pipeline function in the code above executes each of the tasks in the cat picture website as a function.

Both linting and testing happen on the source code, building an image builds from the source code, and then the outputs of each transformation (building, uploading, updating) are passed to each other as they are created.

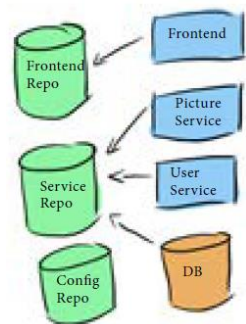
This is great, but how do you actually run it? Someone (or as we'll see later, some THING) needs to execute the pipeline function.

Topher volunteered to be in charge of running the pipeline, so he wrote an executable python file that looks like this:

```
if __name__ == "__main__":
    pipeline("https://10.10.10.10/catpicturewebsite/service.git",
            "https://10.10.10.10/catpicturewebsite/config.git")
```

This executable file calls the pipeline function, passing in the addresses of the service repo and config repo git repositories as arguments.

All Topher has to do is run the executable, and he'll run the pipeline and all of its tasks.



**QUESTION****Q**

Should I be writing my pipelines and tasks in Python like Angela and Topher ?

**A**

Probably not! Instead of reinventing a CD system yourself, there are lots of existing tools you can use. The appendices at the end of this book will give you a brief overview of some of the current options.

We'll be using Python to demonstrate the ideas behind these CD systems without suggesting any particular system to you - they all have their pros and cons and you should use the ones that work best for your needs.

## 2.10 Running once a day

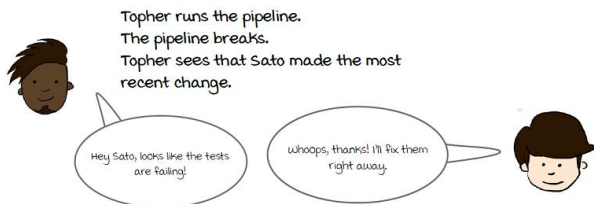
Topher is in charge of running the pipeline, by running the executable python file.

```
def pipeline(source_repo, config_repo):
    linting(source_repo)
    unit_and_integration_tests(source_repo)
    image = build_image(source_repo)
    image_url = upload_image_to_registry(image)
    update_running_service(image_url, config_repo)

if __name__ == "__main__":
    pipeline("https://10.10.10.10/catpicturewebsite/service.git",
            "https://10.10.10.10/catpicturewebsite/config.git")
```

When does he actually run it? He decides that he's going to run it every morning before he starts his day. Let's see what that looks like:

Tuesday 10am

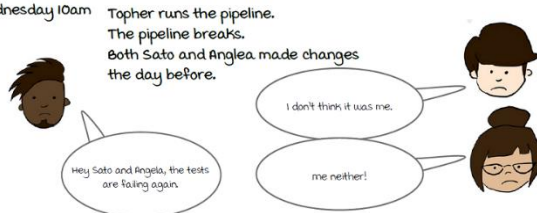


Saying a pipeline *breaks* means that some task in the pipeline encountered an error and pipeline execution stopped.

### VOCAB TIME

That worked okay, but look what happened the next day:

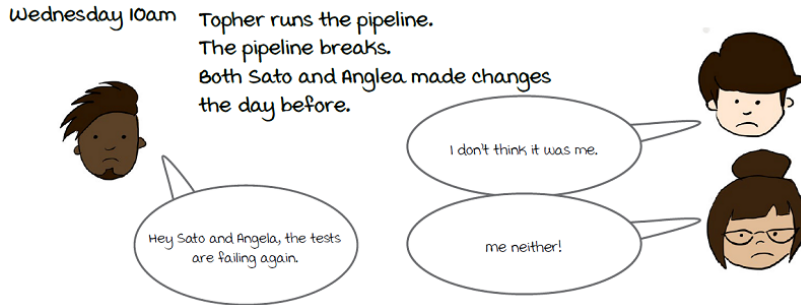
Wednesday 10am



This isn't working out like Topher has hoped: because he's running the pipeline once a day, he's picking up all of the changes that were made the day before. When something goes wrong, he can't tell which change caused the problem.



## 2.11 Trying Continuous Integration



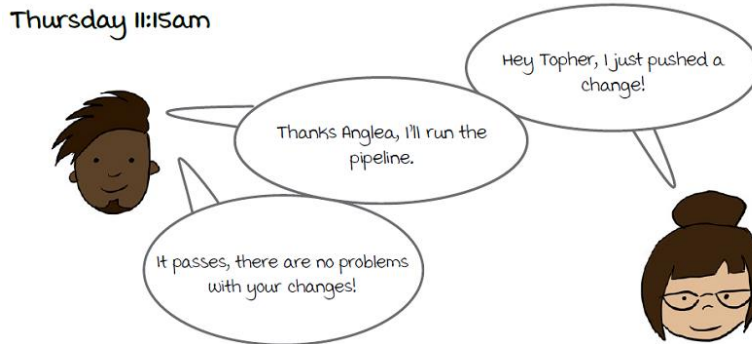
Because Topher is running the pipeline once a day, he's picking up all of the changes from the day before.

If we look back at the definition of Continuous Integration we can see what's going wrong:

- **Continuous integration** is process of combining code changes frequently, *where each change is verified on check in.*

Topher needs to run the pipeline on every change. This way every time the code is changed, the team will get a signal about whether that change introduced problems or not.

Topher asks his team to tell him each time they push a change, so that he can run the pipeline right away. Now the pipeline is being run on every change and the team is getting feedback immediately after they make their changes.





Saying a pipeline passes means everything succeeded, i.e. nothing broke.

### **VOCAB TIME**

#### ***Continuous Deployment***

By running the entire pipeline, including the transformation tasks, Topher is actually doing Continuous Deployment as well!

Many people will run their CI tasks and their transformation tasks as different pipelines. We'll explore the tradeoffs in chapter 11.

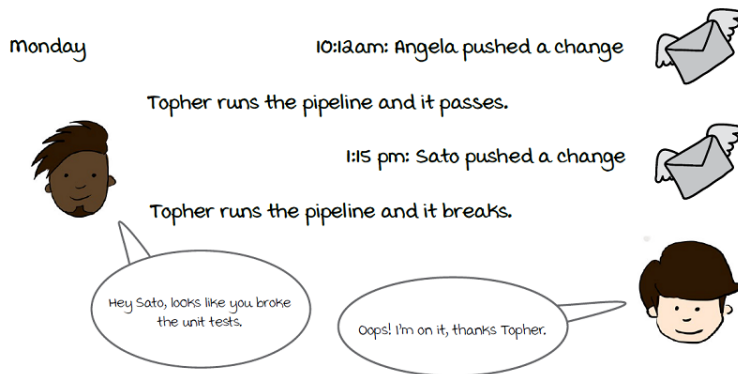
## 2.12 Using notifications

A few weeks have passed, and the team has been telling Topher every time they make a change. Let's see how it's going!



Once again, it didn't work quite as well as Topher hoped. Angela made a change and forgot to tell him, and now the team has to backtrack. How can Topher make sure he doesn't miss any changes?

Topher looks into the problem and realizes that he can get notifications from his **source code management** every time someone makes a change. Instead of having the team tell him when they make changes, he uses these email notifications.

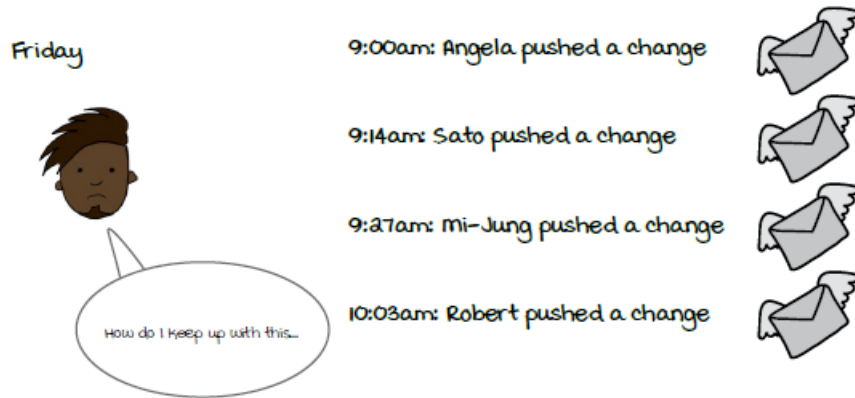


Source Code Management (SCM) is the term for systems like GitHub which combine version control with extra features such as code review tools. Other examples are GitLab and BitBucket.

### VOCAB TIME

## 2.13 Scaling manual effort

Things have been going so well for the team that two more team members have joined. What does this look like for Topher now?



Topher is now spending his entire day running the pipeline and has no time to do any other work. He has lots of ideas for things he wants to improve in the pipeline, and some features he wants to implement, but he can't find any time!

He decides to step back and think about what's happening so he can find a way to save his own time.

1. An email arrives in Topher's inbox
2. Topher's email application notifies Topher he has a new email
3. Topher sees the notification
4. Topher runs the pipeline script
5. Topher tells people when the pipeline fails

Topher looks at his own role in this process. Which parts require Topher's human intervention?

1. Topher has to see the email notification
2. Topher has to type the command to run the script
3. Topher tells people what happened

Is there some way Topher could take himself out of the process? He'd need something that could:

1. See the notification
2. Run the pipeline script
3. Tell people what happened

Topher needs to find something that can receive a notification and run his script for him.

## 2.14 Automation with webhooks

Time is precious! Topher has realized his whole day is being taken up running the pipeline, but he can take himself out of the process if he can find tools to:

1. See the notification
2. Run the pipeline script
3. Tell people what happened



Topher looks into the problem and realizes that his SCM (Source Code Management) system supports **webhooks**.

By writing a simple webserver, he can do everything he needs:

1. The SCM will make a request to his webserver every time someone pushes a change (Topher doesn't need to see the notification!)
2. When the webserver gets the request, it can run the pipeline script (Topher doesn't need to do it!)
3. The request the SCM system makes to the webserver contains the email of the person who made the change, so if the pipeline script fails, the webserver can send an email to the person who caused the problem.



Use **webhooks** to get a system outside of your control to run your code when events happen. Usually you do this by giving the system the URL of an HTTP endpoint that you control.

### VOCAB TIME

```
class Webhook(BaseHTTPRequestHandler):
    def do_POST(self):
        respond(self)
        email = get_email_from_request(self)
        success, logs = run_pipeline()
        if not success:
            send_email(email, logs)

if __name__ == '__main__':
    httpd = HTTPServer(('', 8080), Webhook)
    httpd.serve_forever()
```

Topher starts the webserver running on his workstation and voila: he has automated pipeline execution!

**QUESTION****Q**

How do I get notifications and events from my SCM?

**A**

You'll have to look at the documentation for your version control system to see how to set this up, but getting notifications for changes and webhook triggering is a core feature of most SCMs. If yours doesn't have that, consider changing to a different system that does!

## 2.15 Automation with webhooks

```
class Webhook(BaseHTTPRequestHandler):
    def do_POST(self):
        respond(self)
        email = get_email_from_request(self)
        success, logs = run_pipeline()
        if not success:
            send_email(email, logs)

if __name__ == '__main__':
    httpd = HTTPServer(('', 8080), Webhook)
    httpd.serve_forever()
```



Having your SCM call your webhook when an event happens is often referred to as triggering your pipeline.

### VOCAB TIME

Let's look at what happens now that Topher has automated execution with his webhook.

monday

9:14am: Angela pushes a change



The SCM triggers Topher's webhook



Topher's webhook runs the pipeline: it fails!



Topher's webhook sends an email to Angela telling her that her change broke the pipeline

The events from the SCM system and the webhooks are taking care of all that manual work Topher was doing before. Now he can move on to the work he actually wants to get done!



### QUESTION

**Q**  
**A**

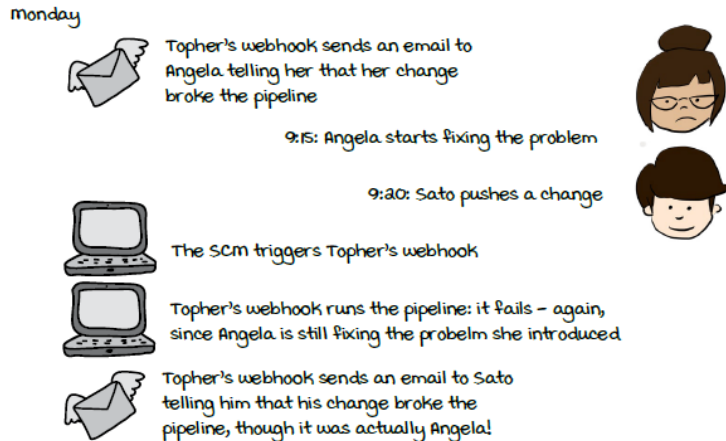
Should I write these webhooks myself like Topher did?

Again, probably not! We're using Python here to demonstrate how CD systems work in general, but instead of creating one yourself, look at the appendices at the end of this book to see existing CD systems you could use. Supporting webhooks is a key feature to look for!

## 2.16 Don't push changes when broken

There are a few more problems Topher will run into. Let's look at a couple of them here and we'll leave the rest for Chapter 7.

What if Angela introduced a change and wasn't able to fix it before another change was made?.



While Angela is fixing the problem she introduced, Sato pushes one of his changes. The system thinks that Sato caused the pipeline to break, but it was actually Angela, and poor Sato is confused.

Plus, every change that is added on top of an existing problem has the potential to make it harder and harder to fix the original problem.

The way to combat this is to enforce a simple rule:

**When the pipeline breaks, stop pushing changes.**

This can be enforced by the CD system itself, and also by notifying all the other engineers working on the project that the pipeline is broken, via notifications.

Stay tuned for chapter 7 to learn more!

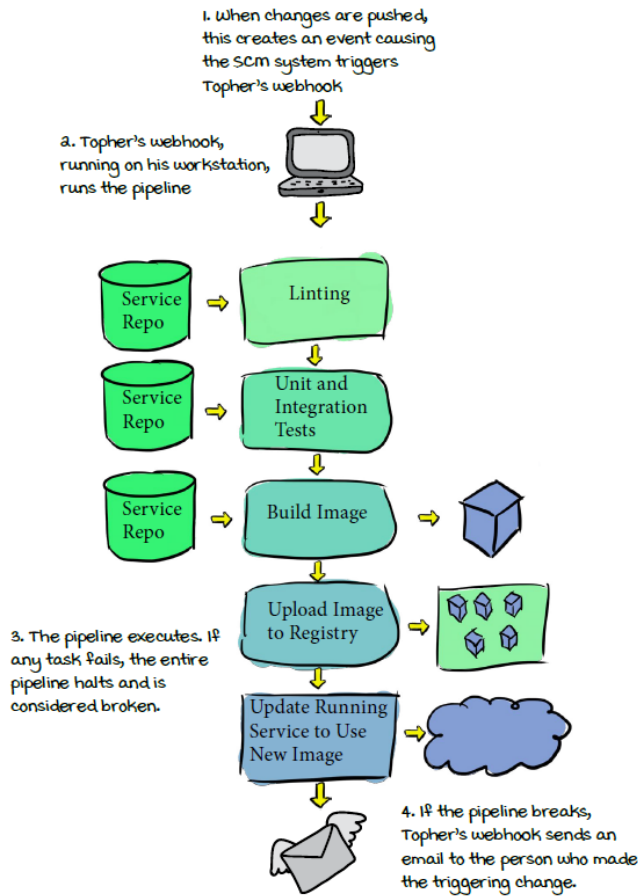
### *Why break the pipeline at all?*

Wouldn't it be better if Angela found out before she pushed that there was a problem? That way she could fix it before pushing and it wouldn't interfere with Sato's work. Yes, that's definitely better! We'll talk a bit more about this in the next chapter and get into more detail in chapter 7.



## 2.17 Cat Picture Website CD

Whew! Now we know all about Cat Picture Website's CD: the pipeline that they use for their services, and also how it is automated and triggered.



**Q**  
**A**

Should I run webhooks directly on my workstation too?

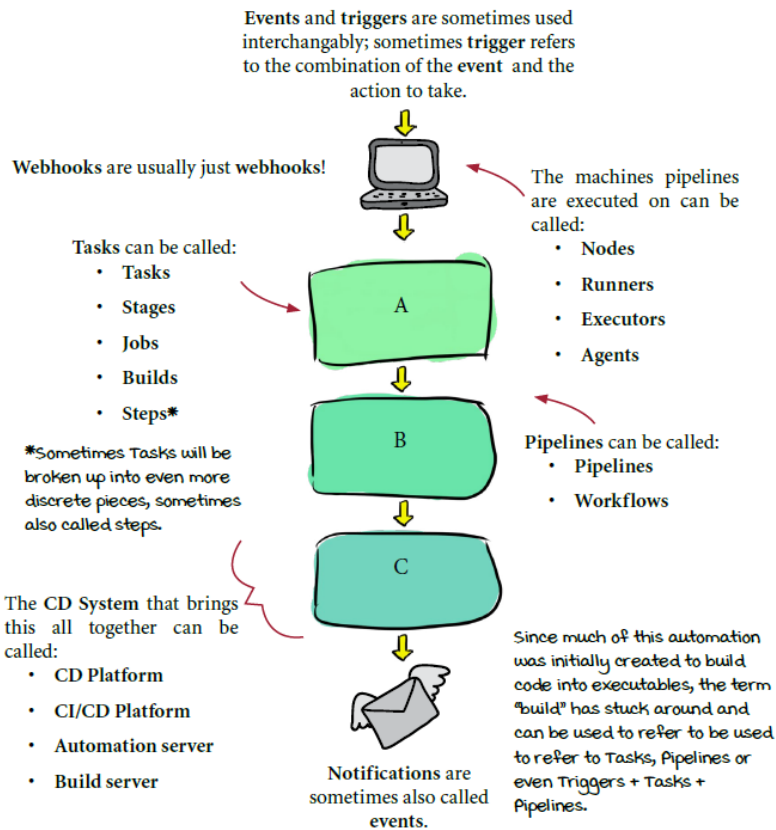
No! Running webhooks for you is another feature most CD systems will handle for you

## ANSWERS

## 2.18 What's in a name?

Once you start using a CD system, you might encounter terminology different from what we've been using in this chapter and will be using in the rest of this book. So here's an overview of the different terminology used across the space and how it relates to the terms we'll be using.

**Tasks** can be called:



## 2.19 Conclusion

The pipeline used by Cat Picture Website for their services shows us the same basic building blocks that you should expect to see in most CD pipelines. By looking at how the folks at Cat Picture Website run their pipeline, we've learned how important automation is in making CD scale, especially as a company grows.

In the rest of this book we'll be looking at the details of each element of the pipeline and how to stitch them together.

## 2.20 Summary

- This book will use the terms pipelines and tasks to refer to basic CD building blocks which can go by many other names
- Tasks are like functions. Tasks can also be called stages, jobs, builds and steps
- Pipelines are the orchestration that combines Tasks together. Pipelines can also be called workflows
- The basic components of a CD pipeline are static analysis, testing, building, delivering and deploying
- Static analysis and testing are gates (aka Continuous Integration (CI) tasks), while building, delivering and deploying are transformations
- Source Code Management (SCM) systems provide mechanisms such as events and webhooks to make it possible to automate pipeline execution
- When a pipeline breaks, stop pushing changes!

## 2.21 Up next . . .

In the next chapter we'll be looking at static analysis in detail: why we need it, what we can catch with it and what we can't, and when it makes sense to use it.

## Version control is the only way to roll

# 3



---

### In this chapter:

- Explain why version control is essential to Continuous Delivery
- Keep your software in a releasable state by keeping version control green and triggering pipelines based on changes in version control
- Define “config as code”
- Enable automation by storing all configuration in version control

---

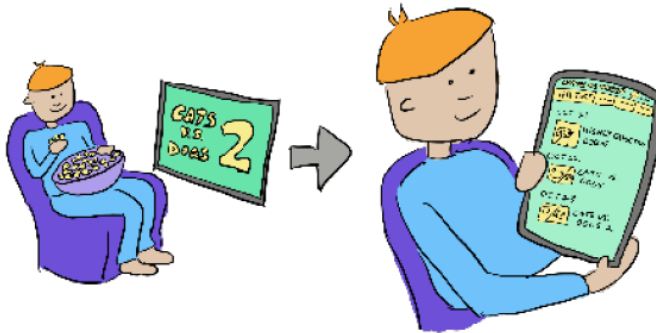
We’re going to start your Continuous Delivery journey at the very beginning with the tool that we need for the basis for absolutely everything we’re going to do next: version control.

In this chapter you’ll learn why version control is crucial to Continuous Delivery and how to use it to set you and your team up for success.

### 3.1 Sasha and Sarah’s start-up

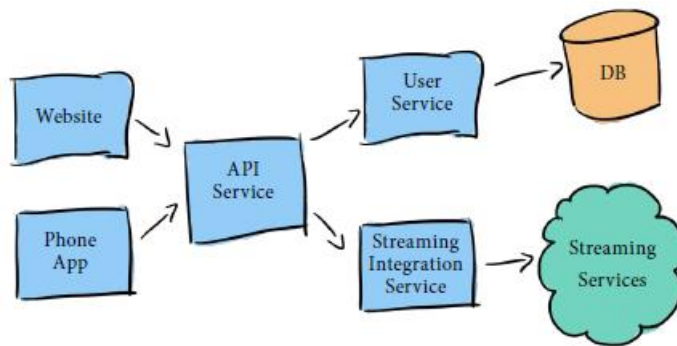
Recent university grads Sasha and Sarah have just gotten funding for an ambitious start-up idea: Watch Me Watch, a social networking site based around TV and movie viewing habits.

With Watch Me Watch, users can rate movies and TV shows as they watch them, see what their friends like, and get personalized recommendations for what to watch next.



Sasha and Sarah want the user experience to be seamless, so they are integrating with popular streaming providers. This means users don't have to tediously add movies and TV shows as they watch them, all of their viewing will automatically be uploaded to the app!

Before they get started, they've sketched out the architecture they want to build:



They're going to break up the backend logic into three services:

- The watch me watch API service, which handles all requests from the frontends
- The user service, which holds data about users
- The streaming integration service which integrates with popular streaming providers

They also plan to provide two different frontends for interacting with Watch Me Watch, a website and a phone app.

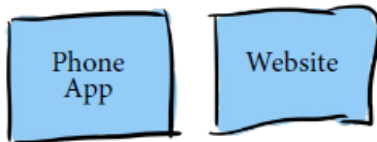
### 3.2 All kinds of data

As they stare proudly at this architecture diagram on their newly purchased white board, they realize that all the code they need to build is going to have to live somewhere. And they're going to both be making changes to it, so they'll need some kind of co-ordination.

They are going to create 3 services, which are designed and built in roughly the same way: they are written in Golang, and executed as running containers.



They'll also run a website and create and distribute a phone app, both of which will be ways for users to use Watch Me Watch.



The data to define the 3 services, the app and the website will include:

- Source code and tests written in Golang
- READMEs and other docs written in markdown
- Container image definitions (Dockerfiles) for the services
- Images for the website and phone app
- Task and Pipeline definitions for testing, building and deploying

The database (which will be running in the cloud) is going to need:

- Versioned schemas
- Task and Pipeline definitions for deploying

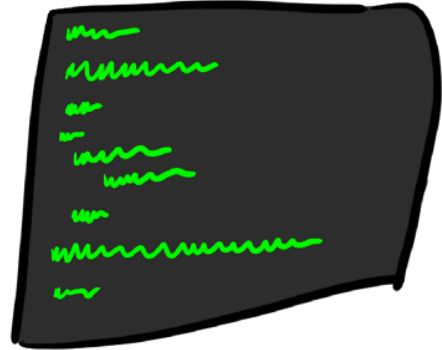


To connect to the streaming services they'll be integrating with, they're also going to need API keys and connection information.

### 3.3 Source and software

Even before they've written a single line of code, gazing at their architecture diagram and thinking about what each piece is going to need, Sasha and Sarah realize they are going to have a lot of data to store:

- Source code
- Tests
- Dockerfiles
- Markdown files
- Images
- Tasks and Pipelines
- Versioned schemas
- API keys
- Connection information



That's a lot! (And this is for a fairly straightforward system!) But what do all of these items have in common? They'll all **data**. And in fact, one step further than that, they are all **plain text**.

Even though each of the above is used differently, each of them is represented by **plain text data**. And when you're working on building and maintaining software, like Sasha and Sarah are about to be, you need to manage all that plain text data somehow.

And that's where **version control** comes in. Version control (also called **source control**) stores this data and tracks changes to it. It stores all of the data your software needs: the source code, the configuration you use to run it, supporting data like documentation and scripts: all the data you need to define, run and interact with your software.



#### VOCAB TIME

Plain text is data in the form of printable (or human readable) characters. In the context of software, plain text is often contrasted with binary data, which is data that is stored as sequences of bits which are not plain text. More simply: plain text is human readable data, the rest is binary data. Version control could be used for any data but it is usually optimized for plain text, so it doesn't handle binary data very well. This means you *can* use it to store binary data if you want, but some features won't work, or won't work well.

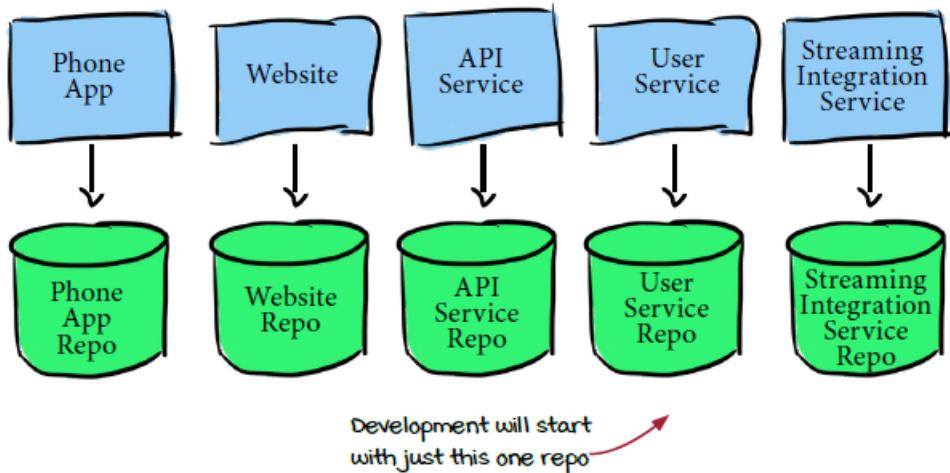
### 3.4 Repositories and versions

**Version control** is software for tracking changes to plain text, where each change is identified by a **version**, also called a **commit** or a **revision**. Version control gives you (at least) these two features for your software:

1. A central location to store everything, usually called **repository** (or **repo** for short!)
2. A history of all changes, where each change (or set of changes) results in a new, uniquely identifiable, **version**

The configuration and source code needed for projects can often be stored in multiple repos - sticking to just one repo for everything is exceptional enough that this has its own name: the **monorepo**.

Sasha and Sarah decide to have roughly one repo per service in their architecture, and they decide that the first repo they'll create will be for their user service.





### 3.5 Continuous Delivery and version control

**Version control** is the foundation for Continuous Delivery. I like the idea of treating Continuous Delivery as a “practice”, asserting that if you’re doing software development, you’re already doing Continuous Delivery (at least to some extent); however the one exception I’ll make to that statement is that if you’re not using version control, you’re not doing Continuous Delivery.

To be doing Continuous Delivery, you must use version control.

Why is it so important for Continuous Delivery? Remember that CD is all about getting to a state where:

1. You can safely deliver changes to your software at any time
2. Delivering that software is as simple as pushing a button

In Chapter 1 we looked at what was required to achieve (1) - specifically, **Continuous Integration (CI)**, which we defined as:

The process of combining code changes frequently, where each change is verified on check in.

We glossed over what “check in” means here - in fact we already assumed version control was involved! Let’s try to redefine CI without assuming version control is present:

The process of **combining code changes frequently**, where each change is verified on **when it is added to the already accumulated and verified changes**.

This definition suggests that in order to do CI we need:

1. Some way to combine changes
2. Somewhere to store (and add to) changes

And how do we store and combine changes to software? You guessed it: using version control. In every subsequent chapter after this, as we discuss elements you’ll want in your Continuous Delivery pipelines, we’ll be assuming that we’re starting from changes that are tracked in version control.



To be doing Continuous Delivery, you must use version control.

#### **TAKEAWAY**



Writing and maintaining software means creating and editing a lot of data, specifically plain text data. Use version control to store and track the history of your source code, configuration - all the data you need to define your software. Store the data in one or more repositories, with each change uniquely identified by a version.

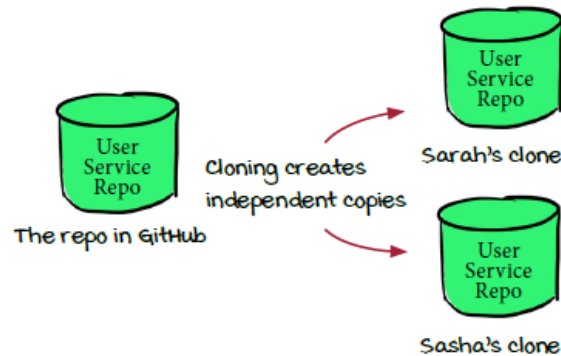
#### **TAKEAWAY**

### 3.6 Git and GitHub

Sarah and Sasha are going to be using **git** for version control. The next question is where their repository will be hosted and how they will interact with it. Sarah and Sasha are going to be using **GitHub** to host this repository and the other repositories they will create.

Git is a **distributed version control system**. What this means is that when you **clone** (that is, copy) a repository onto your own machine, you get a full copy of the entire repository which can be used independently of the remote copy - even the history is separate!

Sarah creates the project's first repository on GitHub and then clones the repo - this makes another copy of the repo on her machine, with all the same commits (none so far), but she can make changes to it independently. Sasha does the same thing, and they both have clones of the repo they can work on independently, and use to **push** changes back to the repo in GitHub.



#### SCM: Software Configuration Management and Source Code Management

Fun fact! Version control software is part of *Software Configuration Management (SCM)*, the process of tracking changes to the *configuration* that is used to build and run your software. In this context *configuration* actually refers to details about all of the data in the repo, including source code, and in fact the practice of *configuration management* for computers dates back to at least the 1970s. This terminology has fallen out of favor, leading both to a rebirth in *infrastructure as code* and later *configuration as code* (more on this in a few pages!) and a redefining of *SCM* as *Source Code Management*. *SCM* is now often used interchangeably with version control and sometimes used to refer to systems like GitHub which offer version control coupled with features like issue tracking and code review.

#### Which version control should I use?

At the time of writing, git is widely supported and popular and would be a great choice!

While we're using GitHub for our examples, there are other appealing options as well with different tradeoffs - see the appendices for an overview of other options such as GitLab and Bitbucket.

### 3.7 An initial commit - with a bug!

Sarah and Sasha both have clones of the user service repo and they're ready to work. In a burst of inspiration, Sarah starts working on the initial `User` class in the repo. She intends for it to be able to store all of the movies a user has watched, and the ratings that a given user has explicitly given to movies.

The `User` class she creates stores the name of the user, and she adds a method `rate_movie` which will be called when a user wants to rate a movie. The function takes the name of the movie to rate, and the score (as a floating point percentage) to give the movie. It tries to store these in the `User` object, but there's a bug in her code: the function tries to use `self.ratings`, but that object hasn't been initialized anywhere.

```
class User:
    def __init__(self, name):
        self.name = name

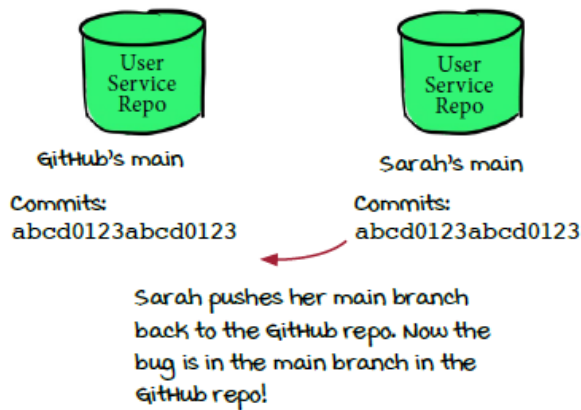
    def rate_movie(self, movie, score):
        self.ratings[movie] = score          #A
```

**#A** There's a bug here: `self.ratings` hasn't been initialized, so trying to store a key in it is going to raise an exception!

Sarah wrote a bug into this code, but she actually also wrote a unit test that will catch that error. She wrote a test (`test_rate_movie`) that tries to rate a movie and then verifies that the rating has been added:

```
def test_rate_movie(self):
    u = User("sarah")
    u.rate_movie("jurassic park", 0.9)
    self.assertEqual(u.ratings["jurassic park"], 0.9)
```

Unfortunately, Sarah forgets to actually run the test before she commits this new code! She adds these changes to her local repo, creating a new commit with ID `abcd0123abcd0123`. She commits this to the **main** branch on her repo, then pushes the commit back to the main branch in GitHub's repo.

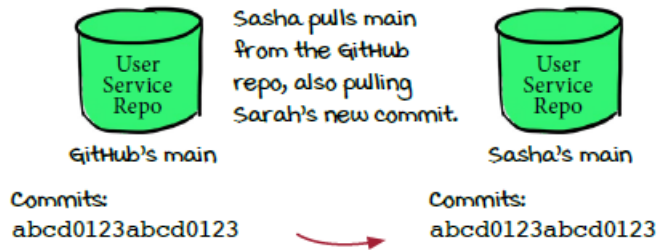


By default, the first branch created in git is called **main**. This default branch is used as the **source of truth** (the authoritative version of the code) and all changes are ultimately integrated here. See chapter 8 for a discussion of other branching strategies.

These example commit IDs are just for show; actual git commit IDs are the SHA-1 hash of the commit.

### 3.8 Breaking main

Shortly after Sarah pushes her new code (and her bug!), Sasha **pulls** the main branch from GitHub to her local repo, pulling in the new commit.



Sasha is excited to see the changes Sarah made:

```
class User:
    def __init__(self, name):
        self.name = name

    def rate_movie(self, movie, score):
        self.ratings[movie] = score
```

Sasha tries to use them right away, but as soon as she tries to use `rate_movie`, she runs smack into the bug, seeing the following error:

```
AttributeError: 'User' object has no attribute 'ratings'
```

"I thought I saw that Sarah included a unit test for this method," wonders Sasha. "How could it be broken?"

```
def test_rate_movie(self):
    u = User("sarah")
    u.rate_movie("jurassic park", 0.9)
    self.assertEqual(u.ratings["jurassic park"], 0.9)
```

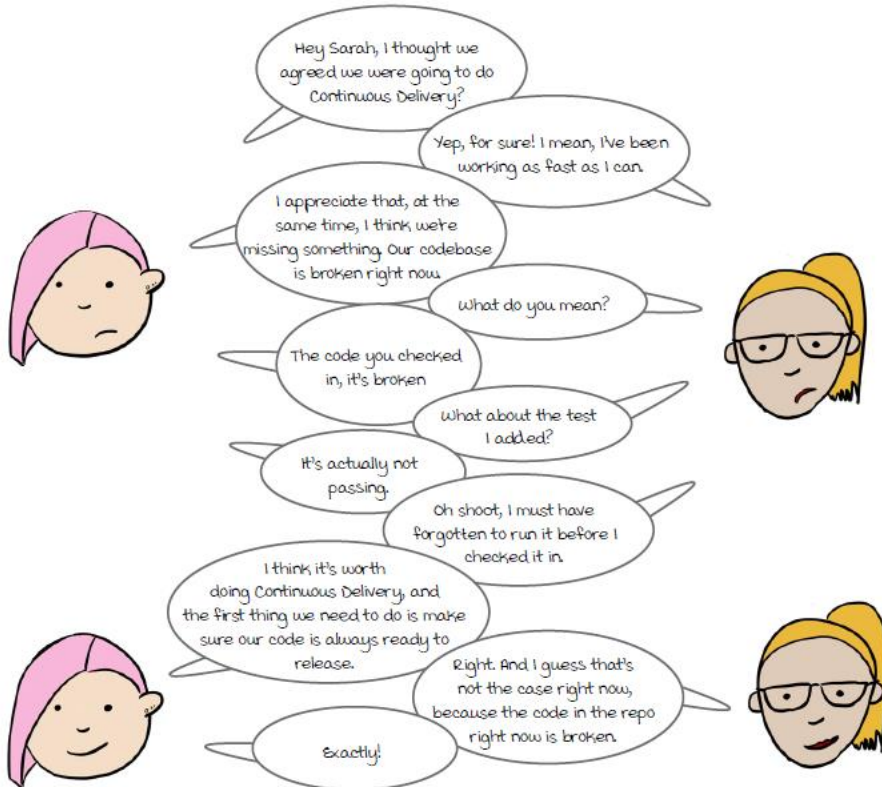
Sasha runs the unit test and, low and behold, the unit test fails too:

```
Traceback (most recent call last):
  File "test_user.py", line 21, in test_rate_movie
    u.rate_movie("jurassic park", 0.9)
  File "test_user.py", line 12, in rate_movie
    self.ratings[movie] = score
AttributeError: 'User' object has no attribute 'ratings'
```

Sasha realizes that the code in the GitHub repo is broken.

### 3.9 Are we doing Continuous Delivery?

Sasha is a bit frustrated after learning that the User Service code in the GitHub repo is broken and brings up the issue with Sarah.



### 3.10 Keep version control releasable

Sarah and Sasha have realized that by allowing broken code to be committed to the user service repo in GitHub, they're violating one of two pillars of Continuous Delivery.

Remember, to be doing CD you want to be trying to get to a state where:

1. You can safely deliver changes to your software at any time
2. Delivering that software is as simple as pushing a button

The user service cannot be safely delivered until the bug Sarah introduced is fixed. This means the user service is not in a state where it is safe to deliver.

Sarah is able to fix it and quickly push a commit with the fix, but how can Sarah and Sasha make sure this doesn't happen again? After all, Sarah had written a test that caught the problem she introduced, and that wasn't enough to stop the bug from getting in.

No matter how hard Sarah tries, she might forget to run the tests before committing at some point in the future - and Sasha might too - they're only human after all!

What Sasha and Sarah need to do is to *guarantee* that the tests will be run before changes are committed. When you need to guarantee that something happens (and if it's possible to automate that thing) your best bet is to automate it.

If you rely on humans to do something that always without fail needs to be done, sometimes they'll make mistakes - which is totally okay because that's how humans work! Let humans be good at what humans do, and when you need to guarantee that the same thing is done in the exact same way every time, and happens without fail, use automation.



#### TAKEAWAY

When you need to guarantee that something happens, use automation. Human beings are not machines, and they're going to make mistakes and forget to do things. Instead of blaming the person for forgetting to do something, try to find a way to make it so they don't have to remember.

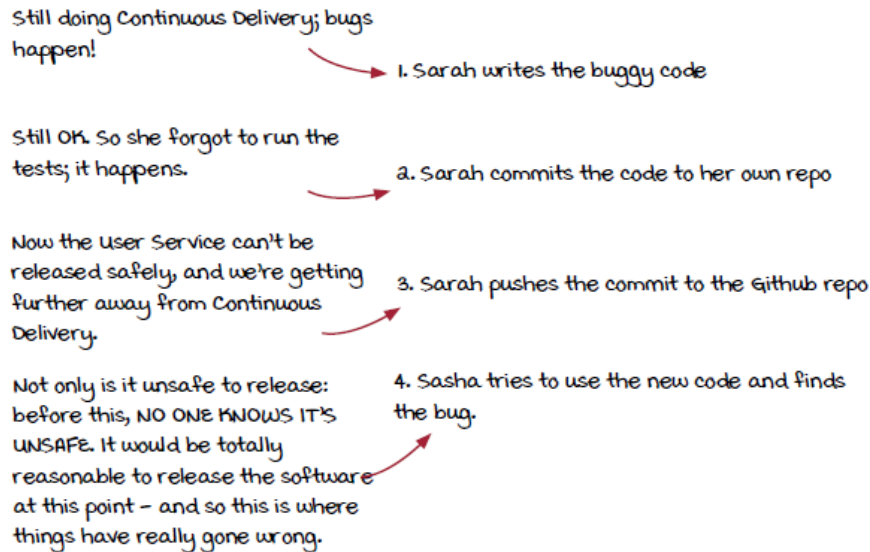
*Won't there always be broken code?*

You'll never catch every single bug, so in some sense, there will always be broken code committed to version control. The key is to always keep version control in a state where you feel confident releasing; introducing the occasional bug is par for the course but the goal is to be in a state where the need to roll back is minimal and the risk of a release or deployment is low. See chapters 8 and 10 for more on releasing.



### 3.11 Trigger on changes to version control

Looking at what led to user service repo being in an unsafe state, we realize that the point where Sarah went wrong wasn't when she introduced the bug, or even when she committed it. The problems started when she pushed the broken code to the remote repo:



So what's the missing piece between (3) and (4) above that would let Sarah and Sasha do Continuous Delivery?

In chapter 2 we learned an important principle for what to do when breaking change are introduced:

**When the pipeline breaks, stop pushing changes.**

But what - what pipeline? Sasha and Sarah don't have any kind of pipeline or automation set up at all. They have to rely on manually running tests to figure out when anything is wrong. And that's the missing piece that Sasha and Sarah need: not just having a pipeline to automate that manual effort and make it reliable, but *setting it up to be triggered on changes to the remote repo*.

**Trigger pipelines on changes to version control.**

If Sasha and Sarah had a pipeline that ran the unit tests whenever a change was pushed to the GitHub repo, Sarah would have immediately been notified of the problem she introduced.

*Can something be done even earlier to stop (3) from happening at all?*

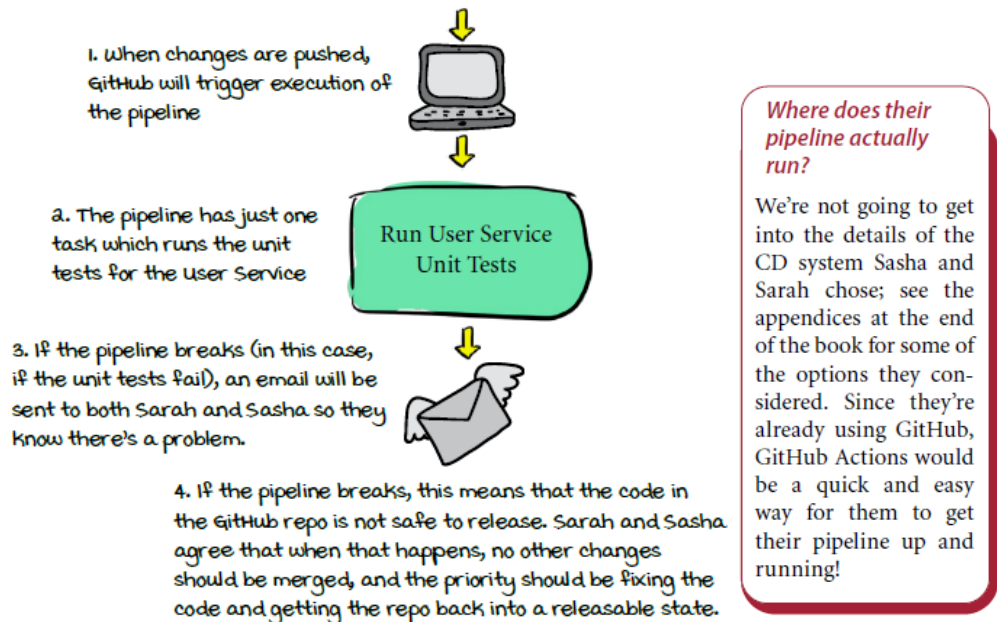
Absolutely! Becoming aware of when problems are introduced is a good first step, but even better is to stop the problems from being introduced at all. This can be done by running pipelines before commits are introduced to the remote main branch. See chapter 7 for more on this.

### 3.12 Triggering the User Service Pipeline

Sasha and Sarah create a pipeline. For now it has just one task to run their unit tests. They setup webhook triggering so that the pipeline will be automatically run every time commits are pushed to the repo in GitHub, and if the pipeline is unsuccessful, an email notification will be sent to both of them.

Now if any breaking changes are introduced, they'll find out right away. They agree to adopt a policy of dropping everything to fix any breakages that are introduced; i.e.:

When the pipeline breaks, stop pushing changes.



#### TAKEAWAY

Trigger pipelines on changes to version control. Just writing tests isn't enough; they need to be running regularly. Relying on people to remember to run them manually is error prone. Version control is not just the source of truth for the state of your software, it's also the jumping off point for all the CD automation we'll look at in this book.

### 3.13 Building the User Service

Sarah and Sasha now have a (small) pipeline in place that will make sure they know immediately if something breaks.

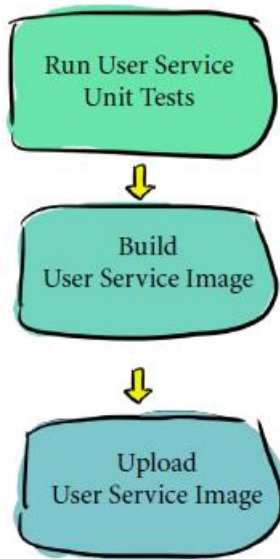
This code isn't doing them any good unless they're doing with it! So far this pipeline has been helping them with the first part of Continuous Delivery:

#### 1. You can safely deliver changes to your software at any time

Having a pipeline and automation to trigger it will also help them with the second part of Continuous Delivery:

#### 2. Delivering that software is as simple as pushing a button

They need to add tasks to their pipeline to build and publish the User Service. They decide to package the User Service as a container image and push it to an image registry.



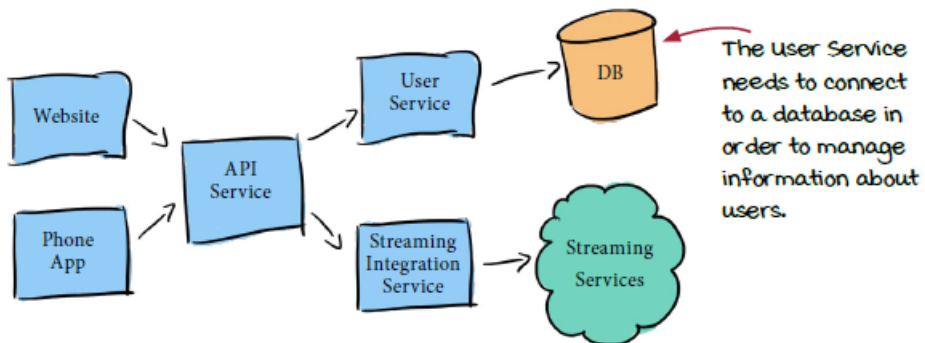
By adding this to their pipeline, they make this "as simple as pushing a button" (or in this case, even simpler, since it will be triggered by changes to version control!)

Now on every commit, the unit tests will be run, and if they are successful, the User Service will be packaged up and pushed as an image.

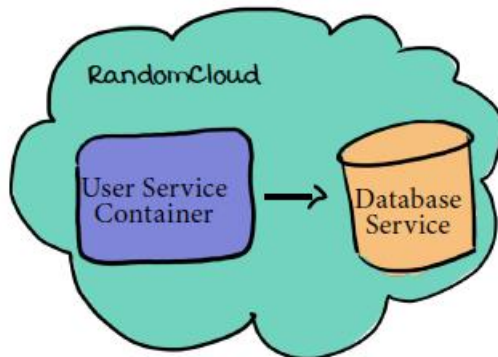
### 3.14 The User Service in the cloud

The last question Sarah and Sasha need to answer for the User Service is the image they are now automatically building will run. They decide they'll run it using the popular cloud provider RandomCloud.

RandomCloud provides a service for running containers, so running the User Service will be easy - except that in order to be able to run, the User Service also needs access to a database, where it stores information about users and movies:



Fortunately, like most cloud offerings, Random Cloud provides a database service which Sarah and Sasha can use with the User Service:



With the User Service pipeline automatically building and publishing the User Service image, all they need to do now is configure the User Service container to use RandomCloud's database service.

### 3.15 Connecting to the RandomCloud database

To get the User Service up and running in RandomCloud, Sasha and Sarah need to configure the User Service container to connect to RandomCloud's database service. To pull this off, two pieces need to be in place:

1. It needs to be possible to configure the User Service with the information the service needs to connect to a database.
2. When running the User Service, it needs to be possible to provide the specific configuration that allows it to Random Cloud's database service.

For (1), Sasha adds command line options that the User Service uses to determine what database to connect to:

```
./user_service.py \
--db-host=10.10.10.10 \
--db-username=some-user \
--db-password=some-password \
--db-name=watch-me-watch-users      #A
```

**#A** The database connection information is provided as command line arguments

For (2), the specifics of RandomCloud's database service can be provided via the configuration that RandomCloud uses to run the User Service container.

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest      #A
  args:
    - --db-host=10.10.10.10
    - --db-username=some-user
    - --db-password=some-password
    - --db-name=watch-me-watch-users          #B
```

**#A** This image is built and pushed as part of the User Service pipeline. It contains and runs `user_service.py`

**#B** These are the same arguments as above, now provided as part of the RandomCloud configuration

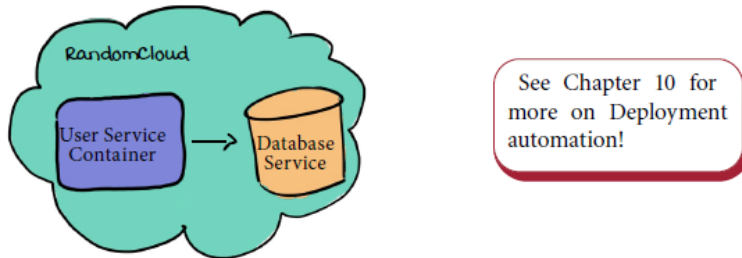
Using `:latest` to identify the image to deploy has some serious downsides - see Chapter 9 for what those are and what you can do instead!

#### *Should they be passing around passwords in plain text?*

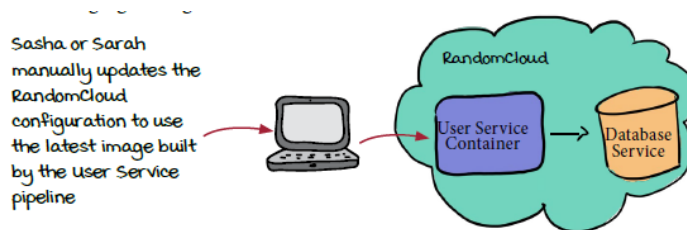
The short answer is no. Sasha and Sarah are about to learn that they want to store this configuration in version control, and they definitely don't want to commit the password there. More on this in a bit.

### 3.16 Managing the User Service

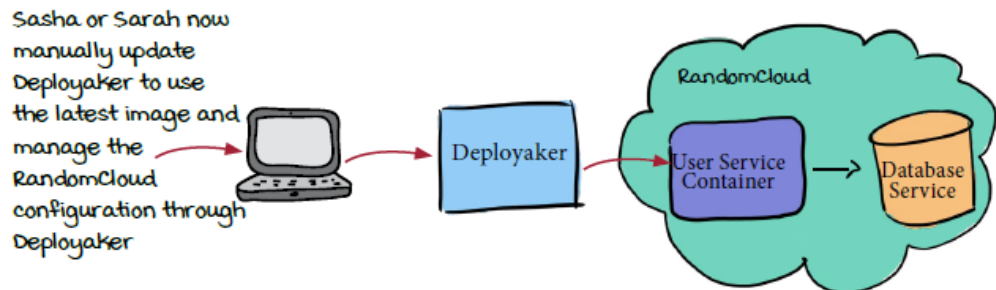
Sarah and Sasha are all set to run the User Service as a container using popular cloud provider RandomCloud.



For the first couple of weeks, every time they want to do a launch, they use the RandomCloud UI to update the container configuration with the latest version, sometimes changing the arguments as well.



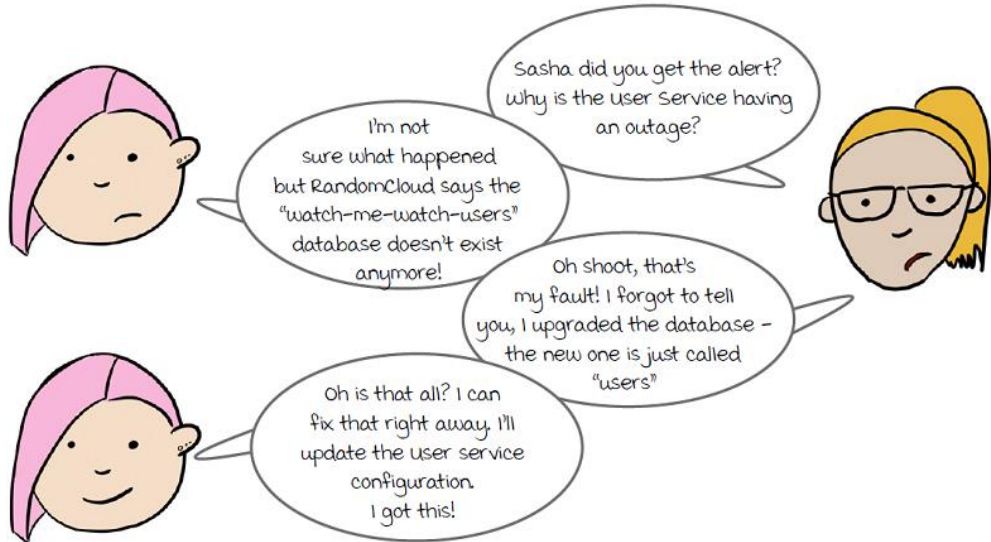
Soon Sarah and Sasha decide to invest in their deployment tooling a bit more, and so they pay for a license with Deployaker, a service which allows them to easily manage deployments of User Service (and later the other services that make up Watch Me Watch as well).



The User Service is now running in a container on RandomCloud, and that service is managed by Deployaker. Deployaker continually monitors the state of the User Service and makes sure that it is always configured as expected.

### 3.17 The User Service outage

One Thursday afternoon, Sasha gets an alert on her phone from RandomCloud, telling her the User Service is down. Sasha looks at the logs from the User Service and realizes that it can no longer connect to the database service. The database called `watch-me-watch-users` no longer exists!



Sasha races to fix the configuration - but she makes a crucial mistake. She completely forgets that Deployaker is managing the User Service now. Instead of using Deployaker to make the update, she makes the fix directly in the Random Cloud UI.

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
    - --db-host=10.10.10.10
    - --db-username=some-user
    - --db-password=some-password
    - --db-name=users      #A
```

**#A** Sasha updates the configuration to use the correct database, but she makes the change directly to RandomCloud and forgets about Deployaker completely

The User Service is fixed and the alerts from RandomCloud stop.

### 3.18 Outsmarted by automation

Sasha has rushed in a fix to the RandomCloud configuration to get the User Service back up and running, but she completely forgot that Deployaker is running behind the scenes.

That night, Sarah has been sleeping soundly when she is suddenly woken up by another alert from RandomCloud. The User Service is down again!



Sarah opens up the Deployaker UI and looks at the configuration it is using for the User Service:

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  - --db-host=10.10.10.10
  - --db-username=some-user
  - --db-password=some-password
  - --db-name=watch-me-watch-users #A
```

**#A** This configuration is still using the database that Sarah deleted!

In spite of being so tired that she can't think properly, Sarah realizes what happened. Sasha fixed the configuration in RandomCloud but didn't update it in Deployaker. Deployaker periodically checks the deployed User Service to make sure it is deployed and configured as expected. Unfortunately, when Deployaker checked that night, it saw the change Sarah had made - which didn't match what it expected to see. So Deployaker resolved the problem by overwriting the fixed configuration with the configuration it had stored - triggering the same outage again! Sarah sighs and makes the fix in Deployaker:



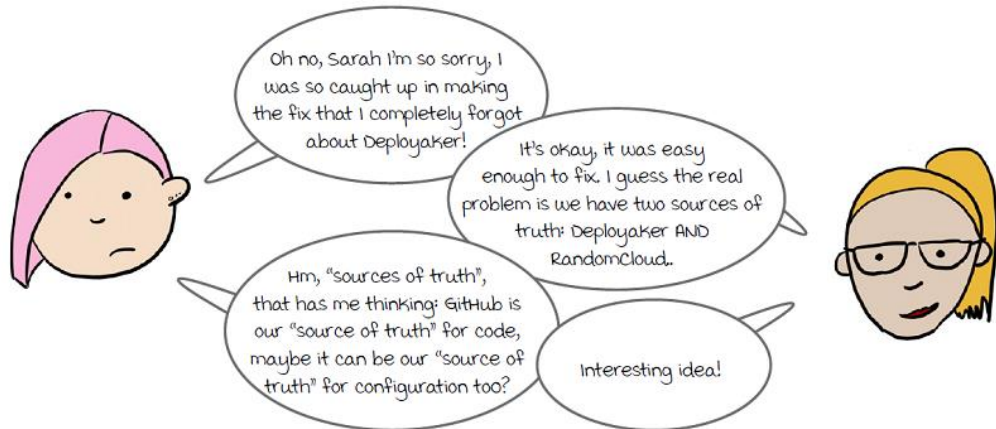
```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
    - --db-host=10.10.10.10
    - --db-username=some-user
    - --db-password=some-password
    - --db-name=users      #A
```

**#A** Now the correct configuration is stored in Deployaker and Deployaker will ensure that the service running in RandomCloud uses this configuration.

The alerts stop and she can finally go back to sleep.

### 3.19 What's the source of truth?

The next morning, bleary eyed over coffee, Sarah tells Sasha what happened.



The configuration that they are talking about is the RandomCloud configuration for the User Service container that needed to be changed to fix the outages the previous day:

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
    - --db-host=10.10.10.10
    - --db-username=some-user
    - --db-password=some-password
    - --db-name=users # OR --db-name=watch-me-watch-users
```

There were two sources of truth for this Configuration:

1. The configuration that RandomCloud was actually using
2. The configuration stored in Deployaker, which it would use to overwrite whatever RandomCloud was using if it didn't match

Sasha has suggested that maybe they can store this configuration in the GitHub repo alongside the User Service source code. But would this just end up being a third source of truth?

The final missing piece is to configure Deployaker to use the configuration in the GitHub repo as its source of truth as well.

### 3.20 Version Control and sensitive data



As a rule of thumb, all plain text data should go into version control. But what about sensitive data, like secrets and passwords? Usually you wouldn't want everyone with access to the repo to have access to this kind of information (and they usually don't need it). Plus, adding this information to version control will store it indefinitely in the history of the repo!

For Sasha and Sarah, the configuration for the User Service contains sensitive data: the username and password for connecting to the database service:

```
user-service.yaml
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
    - --db-host=10.10.10.10
    - --db-username=some-user
    - --db-password=some-password
    - --db-name=users
```

Sasha and Sarah want this config file in version control, but they don't want to commit these sensitive values.

But they want to commit this config file to version control - how do they do that without committing the username and password? The answer is to store that information somewhere else and have it managed and populated for you. Most clouds provide mechanisms for storing secure information, and many CD systems will allow you to populate these secrets safely - which will mean trusting the CD system enough to give it access.

Sasha and Sarah decide to store the username and password in a storage bucket in RandomCloud, and they configure Deployaker so that it can access the values in this bucket and populate them at deploy time.

```
user-service.yaml
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
    - --db-host=10.10.10.10
    - --db-username=randomCloud:watchMeWatch:userServiceDBUser
    - --db-password=randomCloud:watchMeWatch:userServiceDBPass
    - --db-name=users
```

These keywords indicate to Deployaker that it needs to fetch the real values from RandomCloud

### 3.21 User Service config as code

Now that Sasha and Sarah have setup Deployaker such that it can fetch sensitive data (the User Service database username and password) from RandomCloud they want to commit the config file for the User Service the repo:

```
user-service.yaml
---
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
  - --db-host=10.10.10.10
  - --db-username=randomCloud:watchMeWatch:userServiceDBUser
  - --db-password=randomCloud:watchMeWatch:userServiceDBPass
  - --db-name=users
```

They make a new directory in the User Service repo called config where they store this config file, and they'll put any other configuration they discover that they need along the way. Now the User Service repo structure looks like this:

```
docs/
config/          #A
  user-service.yaml
service/         #B
test/
setup.py
LICENSE
README.md
requirements.txt
```

#A This new directory will hold the User Service configuration used by Deployaker as well as any other configuration they need to add in the future

#B All of the source code is in the service directory

#### *Can I store the configuration in a separate repo instead?*

Sometimes this makes sense, especially if you are dealing with multiple services and you want to manage the configuration for all of them in the same place. However keeping the configuration near the code it configures makes it easier to change both in tandem. Start with using the same repo and move the configuration to a separate one only if you later find you need to.

### 3.22 Hard-coded data



user-service.yaml

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice...
  args:
    - --db-host=10.10.10.10
    - --db-username=randomCloud:watchMeWatch:userServiceDBUser
    - --db-password=randomCloud:watchMeWatch:userServiceDBPass
    - --db-name=users
```

Even if Deployaker popluates some of these values, the database connection information is essentially hard-coded and this config can't be used in other environments

With the database connection information hardcoded, it can't be used in any other environments - for example when spinning up a test environment or developing locally. This defeats one of the advantages to config as code, which is that by tracking the configuration you are using when you run your software in version control, you can use this exact configuration when you develop and test. But what can you do about those hardcoded values?

The answer is usually to make it possible to provide different values at runtime (i.e. when the software is actually being deployed), usually by either:

- **Using templating.** For example instead of hard-coding `--db-host=10.10.10.10`, you'd use a templating syntax such as `--db-host={{ $db-host }}` and use a tool to populate the value of `$db-host` as part of deployment
- **Using layering.** Some tools for configuration allow you to define layers which override each other, for example committing the hard-coded `--db-host=10.10.10.10` to the repo for when the User Service is deployed, and using tools to override certain values when running somewhere else (e.g. something like `--db-host=localhost:3306` when running locally).

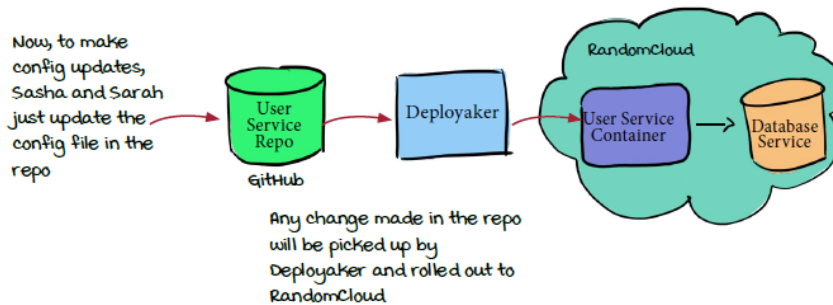
Both of the above approaches have the downside of the configuration in version control not representing entirely the actual configuration being run. For this reason, sometimes people will choose instead to add steps to their pipelines to explicitly **hydrate** (i.e. fully populate the configuration with the actual values for a particular environment) the configuration and commit this "hydrated" configuration back to version control.

Even if Deployaker popluates some of these values, the database connection information is essentially hard-coded and this config can't be used in other environments

### 3.23 Configuring Deployaker

Now that the User Service configuration is committed to GitHub, Sasha and Sarah no longer need to supply this configuration to Deployaker. Instead they configure Deployaker to connect to the User Service GitHub repo and give it the path to the config file for the User Service: `user-service.yaml`.

This way, Sarah and Sasha never need to make any changes directly in RandomCloud or Deployaker. They commit the changes to the GitHub repo and Deployaker picks up the changes from there and rolls them out to RandomCloud.



#### *Is it reasonable to expect CD tools to use configuration in version control?*

Absolutely! Many tools will let you point them directly at files in version control, or at the very least will be programmatically configurable so you can use other tools to update them from config in version control. In fact, it's a good idea to look for these features when evaluating CD tooling and steer clear of tools that only let you configure them through their UIs.

#### *Wait, what about the config that tells Deployaker's how to find the service config in the repo? Should that be in version control too?*

Good question - to a certain extent you need to draw a line somewhere and not strictly *everything* will be in version control (e.g. sensitive data). That being said, Sasha and Sarah would benefit from at least writing some docs on how Deployaker is configured and committing *those* to version control, to record how everything works for themselves and new team members, or if they ever need to setup Deployaker again. Plus, there's a big difference between configuring Deployaker to connect to a few git repos and pasting and maintaining all of the Watch Me Watch service configuration in it.

### 3.24 Config as code

How does configuration fit into Continuous Delivery? Remember that the first half of Continuous Delivery is about getting to a state where:

**You can safely deliver changes to your software at any time**

When many people think about their delivering their software, they only think about the source code. But as we saw at the beginning of this chapter, there are actually all kinds of by **plain text data** that make up your software - and that includes the configuration you use to run it.

We also took a look at Continuous Integration to see why version control was key. Continuous integration is:

**The process of combining code changes frequently, where each change is verified on when it is added to the already accumulated and verified changes.**

In order to really be sure you can safely deliver changes to your software, you need to be accumulating and verifying changes to *all the plain text data that makes up your software* - including the configuration.

This practice of treating software configuration the same way you treat source code (i.e. storing it in version control and verifying it with CI) is often called **config as code**. Doing config as code is key to practicing Continuous Delivery, and doing config as code is as simple as versioning your configuration in version control, and as much as you can, applying verification to it such as static analysis and using it when spinning up test environments.

Config as code is not a new idea! You may remember earlier in the chapter we mentioned that the practice *configuration management* for computers dates back to at least the 1970s - sometimes we forget the ideas we've already discovered them and have to rediscover them with new names.

#### *What's the difference between infrastructure as code and config as code?*

The idea of *infrastructure as code* came along first, but *config as code* was hot on its heels. The basic idea with *infrastructure as code* is to use code/configuration (stored in version control) to define the infrastructure your software runs on, e.g. machine specs and firewall configuration. Where *config as code* is all about configuring the running software, *infrastructure as code* is more about defining the environment the software runs in (and automating its creation). The line between the two is especially blurry today when so much of the infrastructure we use is cloud based - when you deploy your software as a container, are you defining the infrastructure, or configuring the software? But the core principles of both are the same: treat everything required to run your software "like code": store it in version control and verify it.

### 3.25 Rolling out software and config changes

Sarah and Sasha have begun doing config as code by storing the User Service configuration in Deployaker.

They almost immediately see the payoff a few weeks later when they decide that they want to separate the data they are storing in the database into two separate databases. Instead of one giant `User` database, they want a `User` database and a `Movie` database. To do this they need to make two changes:

1. The User Service previously only took one argument for the database name: `--db-name`, now it needs to take two arguments

```
./user_service.py \
--db-host=10.10.10.10 \
--db-username=some-user \
--db-password=some-password \
--db-users-name=users \      #A
--db-movies-name=movies
```

#A The User Service has to be updated to recognize these two new arguments

2. The configuration for the User Service needs to be updated to use the two arguments instead of just the `--db-name` argument it is currently using

```
apiVersion: randomcloud.dev/v1
kind: Container
spec:
  image: watchmewatch/userservice:latest
  args:
    - --db-host=10.10.10.10
    - --db-username=some-user
    - --db-password=some-password
    - --db-users-name=users      #A
    - --db-movies-name=movies
```

#A And the configuration has to be updated to actually use the new arguments as well

Back when they were making configuration changes directly in Deployaker, they would have had to roll these changes out in two phases:

1. After making the source code changes to the User Service, they'd need to build a new image
2. At this point, the new image would be incompatible with the config in Deployaker; they wouldn't be able to do any deployments until Deployaker was updated

But now that they source code and the configuration live in version control together, they can make all the changes and once, and they'll all be smoothly rolled out together by Deployaker!





Only use tools that let you store their configuration in version control. Some tools assume you'll configure the view their UIs (e.g. websites and CLIs); this can be fine for getting something up and running quickly, but in the long run to practice continuous delivery you'll want to be able to store this configuration in version control. Avoid tools that don't let you.

#### **TAKEAWAY**



Treat ALL the plain text data that defines your software like code and store it in version control. You'll run into some challenges in this approach around sensitive data and environment specific values, but the extra tooling you'll need to fill these gaps is well worth the effort. By storing everything in version control you can be confident that you are always in a safe state to release - accounting for ALL the data involved, not just the source code.

#### **TAKEAWAY**

### 3.26 Conclusion

Even though it's early days for Watch Me Watch, Sarah and Sasha quickly learned how critical version control is to Continuous Delivery. They learned that far from being just passive storage, it's the place where the first piece of Continuous Delivery happens: it's where code changes are combined, and those changes are the triggering point for verification - all to make sure that the software remains in a releasable state.

Though at first they were only storing source code in version control, they realized that they could get a lot of value from storing configuration there as well - and treating it like code!

As the company grows, they'll continue to use version control as the single source of truth for their software. Changes made in version control will be the jumping off point for any and all of the automation they add from this point forward, from automatically running unit tests to doing canary deployments.

### 3.27 Summary

- You must use version control in order to be doing Continuous Delivery
- Trigger CD pipelines on changes to version control
- Version control is the source of truth for the state of your software, and it's also the jumping off point for all the CD automation in this book
- Practice config as code and store all plain text data that defines your software (not just source code but configuration too) in version control. Avoid tools that don't let you do this.

### 3.28 Up next . . .

In the next chapter we'll look at how to use linting in CD pipelines to avoid common bugs and enforce quality standards across codebases, even with many contributors.

## Use linting effectively

# 4



---

### In this chapter:

- identify the types of problems linting can find in your code: bugs, errors, and style problems
- aim for the ideal of 0 problems identified but temper this against the reality of legacy codebases
- lint large existing codebase by approaching the problem iteratively
- weigh the risks of introducing new bugs against the benefits of addressing problems

---

Let's get started actually building your pipelines! Linting is a key component to the continuous integration (CI) portion of your pipeline: it allows you to identify and flag known issues and coding standard violations, reducing bugs in your code and making it easier to maintain.

### 4.1 Becky and Super Game Console

Becky just joined the team at Super Game Console and she's really excited!

Super Game Console is a video game console that runs simple python games, and it's very popular. The best feature is its huge library of python games, which anyone can contribute.



The folks at Super Game Console have a submission process that allows everyone from the hobbyist to the professional sign up as a developer and submit their own games.



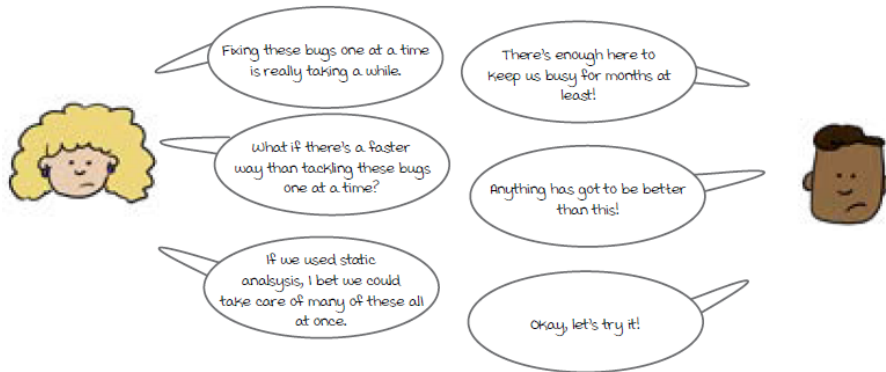
But there are a *lot* of bugs in the games and it has been starting to become a problem.

Becky and Ramon, who has been on the team for a while now, have been working their way through the massive backlog of game bugs. Becky has noticed a few things:

- Some of the games wouldn't even compile! And a lot of the other bugs are caused by simple mistakes like trying to use variables that aren't initialized
- There are lots of mistakes which do not actually cause bugs but get in the way of Becky's work, for example unused variables.
- The code in every single game looks different from the one before it! The inconsistent style makes it hard for her to debug.

## 4.2 Linting to the rescue!

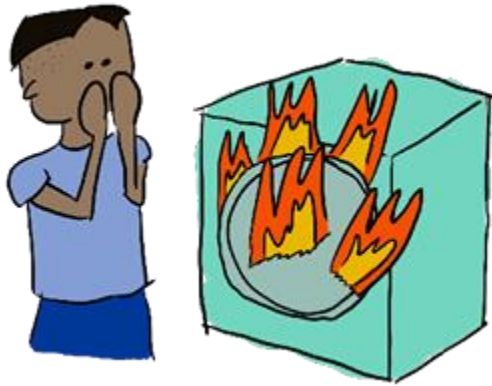
Looking at the types of problems causing the bugs she and Ramon has been fixing, they remind Becky a lot of the kinds of problems that linters catch.



What is linting anyway? Well it's the action of finding lint, using a linter! And what's lint? You might think of the lint that accumulates in your clothes dryer.



By themselves, the individual fibers don't cause any problems, but when they build up over time, they can interfere with the effective functioning of your dryer. Eventually, if they are neglected for too long, the lint builds up and the hot air in the dryer eventually sets it on fire!

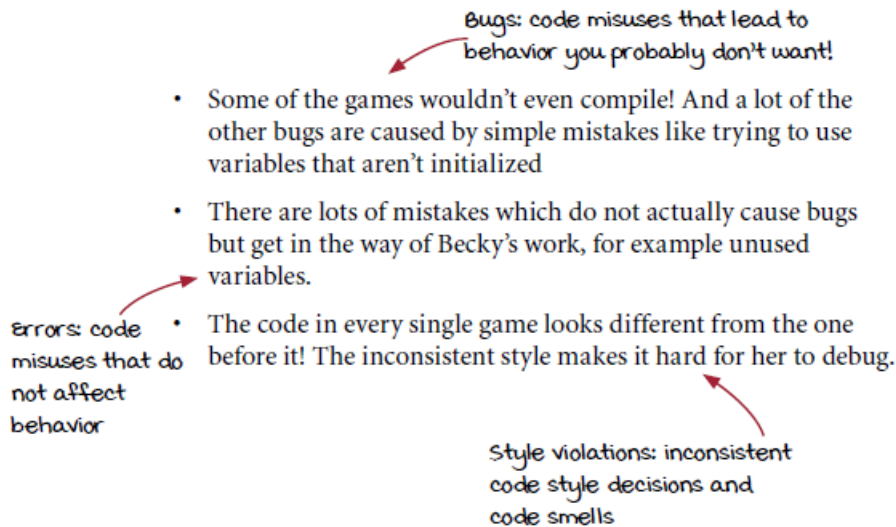


And it's the same for programming errors and inconsistencies that may seem minor: they build up over time! Just like in Becky and Ramon's case: the code they are looking at is inconsistent and full of simple mistakes. Not only are these problems causing bugs, they're also getting in the way of maintaining the code effectively.

### 4.3 The lowdown on linting

There are linters of all different shapes and sizes. Since they analyze and interact with code, they are usually specific to a particular language, e.g. `pylint` for Python. Some linters apply generically to anything you might be doing in the language, and some are specific to particular domains and tools, for example linters for working effectively with http libraries.

We'll be focusing on linters that apply generically to the language you are using. Different linters will categorize the problems they raise differently, but they can all be viewed as falling into one of three buckets. Let's take a look at the problems Becky noticed and how they demonstrate the three kinds of problems:



#### *What's the difference between static analysis and linting?*

Using **static analysis** lets you analyze your code without actually executing it. In the next chapter we'll talk about tests, which are a kind of **dynamic analysis** because you have to actually execute your code.

**Linting** is a kind of static analysis; the term static analysis can encompass many different ways of analyzing code. In the context of CI/CD, most of the static analysis we'll be discussing is done with tools called linters, otherwise the distinction isn't that important.

The name linter comes from the 1978 tool of the same name created by Bell Labs. Most of the time, especially in the context of CI/CD, the terms static analysis and linting can be used interchangeably, but technically there are forms of static analysis that go beyond what linters can do.

## 4.4 The tale of pylint and many many issues

Since the games for Super Game Console are all in Python, Becky and Ramon decide that using the tool `pylint` is a good place to start.

This is the layout of the Super Game console codebase:

```
console/
docs/
games/      #A
test/
setup.py
LICENSE
README.md
requirements.txt
```

**#A The games directory is where they store all the developer submitted games!**

The folder `games` has thousands of games in it! Becky is excited to see what `pylint` can tell them about all these games. She and Ramon watch eagerly as Becky types in the command and presses enter...

```
pylint games
```

And they are rewarded with screen after screen filled with warnings and errors! This is a small sample of what they see:

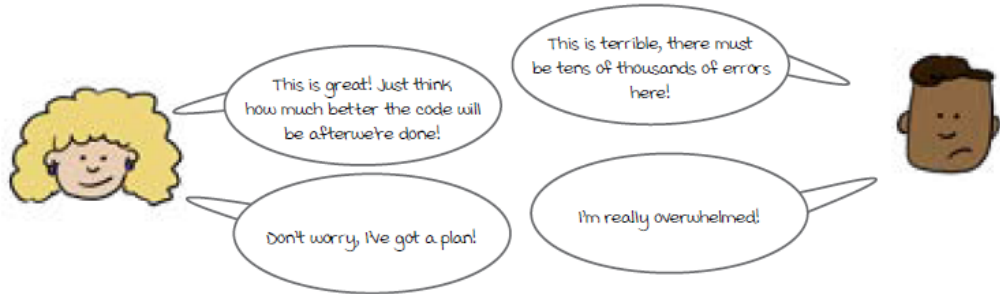
```
games/bridge.py:40:0: W0311: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
games/bridge.py:41:0: W0311: Bad indentation. Found 4 spaces, expected 8 (bad-indentation)
games/bridge.py:46:0: W0311: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
games/bridge.py:1:0: C0114: Missing module docstring (missing-module-docstring)
games/bridge.py:3:0: C0116: Missing function or method docstring (missing-function-
docstring)
games/bridge.py:13:15: E0601: Using variable 'board' before assignment (used-before-
assignment)
games/bridge.py:8:2: W0612: Unused variable 'cards' (unused-variable)
games/bridge.py:23:0: C0103: Argument name "x" doesn't conform to snake_case naming style
(invalid-name)
games/bridge.py:23:0: C0116: Missing function or method docstring (missing-function-
docstring)
games/bridge.py:26:0: C0115: Missing class docstring (missing-class-docstring)
games/bridge.py:30:2: C0116: Missing function or method docstring (missing-function-
docstring)
games/bridge.py:30:2: R0201: Method could be a function (no-self-use)
games/bridge.py:26:0: R0903: Too few public methods (1/2) (too-few-public-methods)
games/snakes.py:30:4: C0103: Method name "do_POST" doesn't conform to snake_case naming
style (invalid-name)
games/snakes.py:30:4: C0116: Missing function or method docstring (missing-function-
docstring)
games/snakes.py:39:4: C0103: Constant name "httpd" doesn't conform to UPPER_CASE naming
style (invalid-name)
games/snakes.py:2:0: W0611: Unused import logging (unused-import)
games/snakes.py:3:0: W0611: Unused argv imported from sys (unused-import)
```



***What about other languages and linters?***

Becky and Ramon are using python and pylint, but the same principles apply regardless of the language or linter you are using. All good linters should give you the same flexibility of configuration we'll demonstrate with pylint and catch the same variety of issues.

## 4.5 Legacy code: using a systematic approach



The first time you run a linting tool against an existing codebase, the number of issues it finds can be overwhelming! (In a few pages we'll talk about what to do if you don't have to deal with a huge existing codebase.)

Fortunately Becky has dealt with applying linting to legacy codebases before and has a systematic approach that she and Ramon can use to both speed things up and use their time effectively.

1. Before doing anything else, they need to configure the linting tools. The options that pylint is applying out of the box might not make sense for Super Game Console.
2. Next, measure a baseline and keep measuring. Becky and Ramon don't necessarily need to fix every single issue; if all they do is make sure the number of issues goes down over time, that's time well spent!
3. Once they've got the measurements, every time a developer submits a new game, they can measure again, and stop the game from being submitted if it introduces more problems. This way the number won't ever go up!
4. At this point, Becky and Ramon have ensured things won't get any worse; with that in place they can start tackling the existing problems. Becky knows that not all linting problems are created equal, so she and Ramon will be dividing and conquering so that they can make the most effective use of their valuable time.

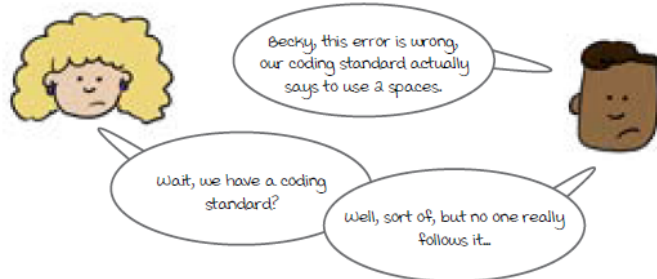


The key to Becky's plan is that she knows that they don't have to fix everything: just by preventing new problems from getting in, they've already improved things. And the truth is, not everything has to be fixed - or even should be.

## 4.6 Step 1: Configure against coding standards

Ramon has been looking through some of the errors pylint has been spitting out and notices that it's complaining they should be indenting with 4 spaces instead of 2:

```
bridge.py:2:0: W0311: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
```



**Already familiar with configuring linting?**

Then you can probably skip this! Read this page if you've never configured a linting tool before.

This is often the case when coding standards aren't backed up by automation, so Becky isn't surprised. But the great news is that the (currently ignored) coding standards have most of the information that Becky and Ramon need, information like:

- Indent with tabs or spaces? If spaces, how many spaces?
- Are variables named with snake\_case or camelCase?
- Is there a maximum line length? What is it?

The answers to these questions can be fed into pylint as configuration options, into a file usually called `.pylintrc`.

Becky didn't find everything she needed in the existing coding

style, so she and Ramon had to make some decisions themselves. They invited the rest of the team at Super Game Console to give input as well, but there were some items that no one could agree on; in the end, Becky and Ramon just had to make a decision. When in doubt, they leaned on Python language idioms, which mostly meant sticking with pylint's defaults.

### Features to look for

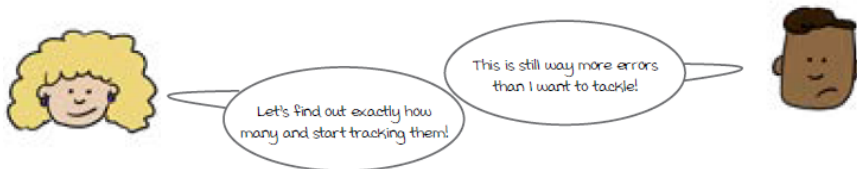
When evaluating linting tools, expect them to be configurable. Not all codebases and teams are the same, so it's important to be able to tune your linter. Linters need to work for you, not the other way around!

### Automation is essential to maintain coding standards

Without automation, it's up to individual engineers to remember to apply coding standards, and it's up to reviewers to remember to review for them. People are people: we're going to miss stuff! But machines aren't: linters let us automate coding standards so no one has to worry about them.

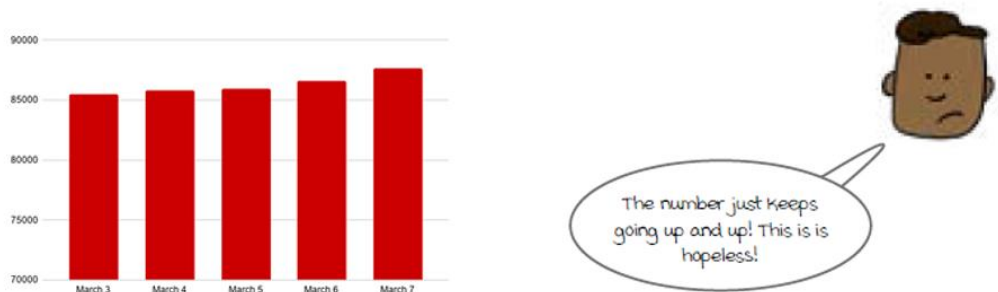
## 4.7 Step 2: Establish a baseline

Now that Becky and Ramon had tweaked pylint according to the existing and newly established coding standard, they had slightly less errors, but still in the tens of thousands.



Becky knows that even if she and Ramon left the codebase exactly the way it is, by just reporting on the number of issues and observing it over time, this can help motivate the team to decrease the number of errors. And in the next step they'll use this data to stop the number of errors from going up.

Becky writes a script that runs pylint and counts the number of issues it reports. She creates a pipeline that runs every night and publishes this data to a blob store. After a week she collects the data and creates this graph showing the number of issues:



The number keeps going up because even as Becky and Ramon work on this, developers are eagerly submitting more games and updates to Super Game Console. Each new game and new update has the potential to include a new issue.

### *Do I need to build this myself?*

Yes, in this case, you probably will. Becky had to write the tool herself to measure the baseline number of issues and track them over time. If you want to do this, there's a good chance you'll need to build the tool yourself. This will depend on the language you are using and the tools available; there are also services you can sign up for that will track this information for you over time. Most CI/CD systems do not offer this functionality because it is so language and domain specific.

## 4.8 Step 3: Enforce at submission time

Ramon noticed that as submissions came in, the number of issues pylint was finding was going up, but Becky has a solution for that: block submissions that increase the number of issues. This means enforcing a new rule on each pull request:

Every pull request must either reduce the number of linting issues or leave it the same.

Becky creates this script to add to the pipeline that Super Game Console runs against all pull requests:

```
# when the pipeline runs, it will pass to this script
# paths the files that changed in the pull request
paths_to_changes = get_arguments()

# run the linting against the files that changed to see
# how many problems are found
problems = run_lint(paths_to_changes)

# becky created a pipelines that runs every night and
# writes the number of observed issues to a blob store;
# here the lint script will download that data
known_problems = get_known_problems(paths_to_changes)

# compare the number of problems seen in the changed code
# to the number of problems seen last night
if len(problems) > len(known_problems):
    # the pull request should not be merged if it increase the
    # number of linting issues
    fail('number of lint issues increased from {} to {}'.format(
        len(known_problems), len(problems)))
```

*Shouldn't you look at more than just the number of issues?*

It's true that just comparing the number of issues glosses some things over; for example, the changes could fix one issue but introduce another. But the really important thing for this situation is the overall trend over time, and not so much the individual issues.

The next step is for Becky to add this to the existing pipeline that runs against every pull request.

Every pull request must either reduce the number of linting issues or leave it the same.



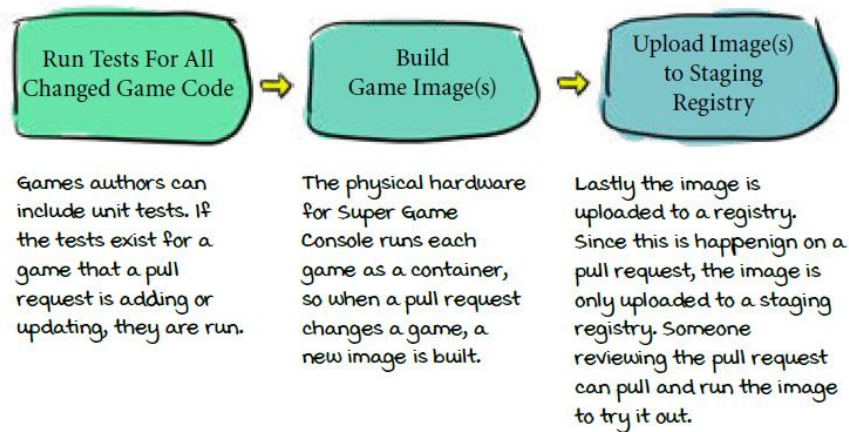
### *Green field or small code base*

We'll talk about this a bit more in a few pages, but if you're working with a small or brand new codebase, you can skip measuring the baseline and just clean up everything at once. Then, instead of adding a check to your pipeline that ensures the number doesn't go up, add a check that fails if linting finds any problems at all.

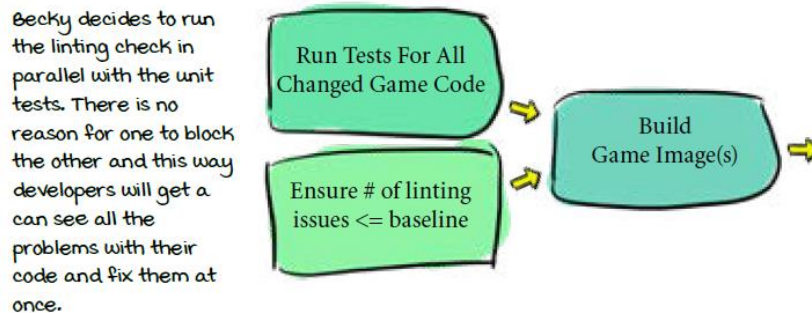
## 4.9 Step 3: Enforce at submission time

Becky wants her new check to be run every time a developer submits a new game or an update to an existing game.

Super Game Console accepts new games as pull requests to their GitHub repository. They already make it possible for developers to include tests with their games and they run those tests on each pull request. This is what the pipeline looks like before Becky's change:

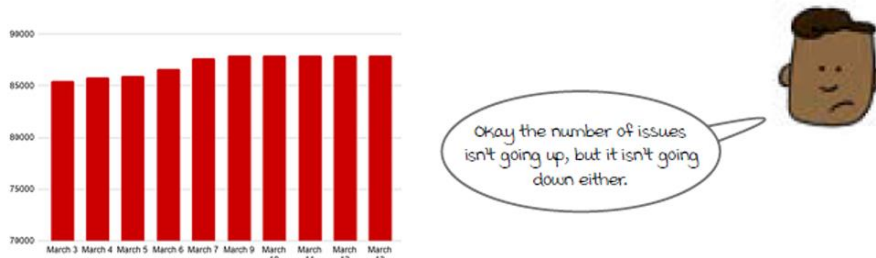


Becky wants to add her new check to the pipeline that Super Game Console runs against every pull request.



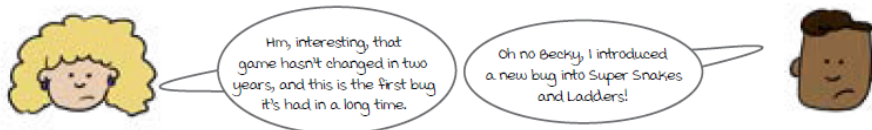
Now, whenever a developer opens a pull request to add or change a Super Game Console game, Becky's script will run. If this pull request increase the number of linting issues in the project, the pipeline will stop. The developer must fix this before the pipeline will continue to building images.

## 4.10 Step 4: Divide and conquer



Becky and Ramon have stopped the problem from getting worse. Now the pressure is off and they are free to start tackling the existing issues, confident that more won't be added.

It's time to start fixing issues! But Ramon quickly runs into a problem...



Making ANY changes, including changes that fix linting issues, has the risk of introducing more problems.

So why do we do it? Because the reward outweighs the risk! And it ONLY makes sense to do it when that's the case. Let's take a look at the rewards and the risks when we fix linting problems:

We can determine some interesting things from this list. The first reward is about catching bugs, which we need to weigh against the first risk of introducing new bugs.

**Reward 1:** Linting can catch bugs

**Reward 2:** Linting helps remove distracting errors

**Reward 3:** Consistent code is easier to maintain

**Risk 1:** Making changes can introduce new bugs

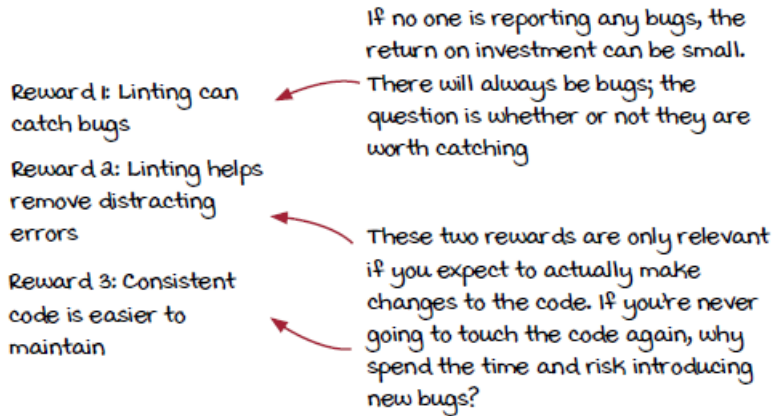
**Risk 2:** Fixing linting issues takes time

Ramon introduced a new bug into a game that didn't have any open reported bugs. Was it worth the risk of adding a bug to a game that, as far as everyone could tell, was working just fine? Maybe not!

The other two rewards (2 and 3) are only relevant when the code is being changed. If you don't ever need to change the code, it doesn't matter how many distracting errors it has, or how inconsistent it is.

Ramon was updating a game that hasn't had a change in two years. Was it worth taking the time and risking introducing new bugs into a game that wasn't being updated? Probably not! He should find a way to isolate these games so he can avoid wasting time on them.

## 4.11 Isolation: Not everything should be fixed



Becky and Ramon look at all the games they have in their library, and they identify the ones that change the least. These are all more than a year old and the developers have stopped updating them. They also look at the number of user reported bugs with these games. They select the games which haven't changed in more than a year and don't have any open bugs, and move them into their own folder.

Their codebase now looks like this:

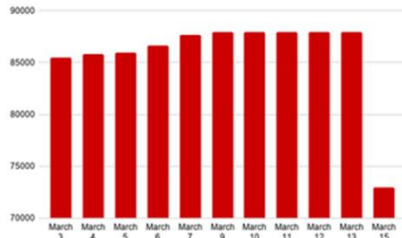
```
.pylintrc          #A
console/
docs/
games/
  frozen/          #B
  ...
test/
setup.py
LICENSE
README.md
requirements.txt
[MASTER]
ignore=games/frozen
```

#A The configuration file for pylint that Becky and Ramon made in Step 1

#B These games haven't been updated in more than a year and have no open bugs. They don't expect changes, so it's okay to exclude them from the linting check.



## 4.12 Enforcing isolation



And to be extra safe, Becky created a new script that made sure that no one was making changes to the games in the frozen directory:

```
# when the pipeline runs, it will pass to this script
# paths the files that changed in the pull request
paths_to_changes = get_arguments()

# instead of hardcoding this script to look for changes
# to games/frozen, load the ignored directories from
# .pylintrc to make this check more general purpose
ignored_dirs = get_ignored_dirs_from_pylintrc()

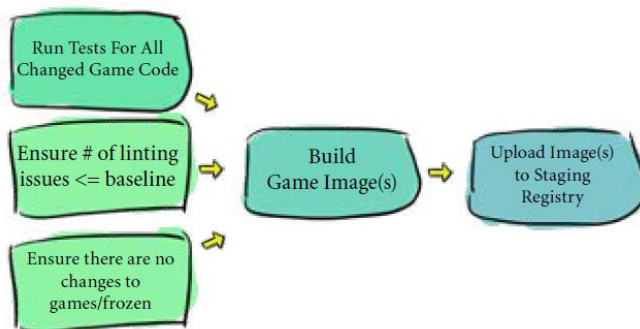
# check for any paths that are being changed which are in
# the directories being ignored
ignored_paths_with_changes = get_common_paths(
    paths_to_changes, ignored_dirs)

if len(ignored_paths_with_changes) > 0:
    # the pull request should not be merged if it
    # includes changes to ignored directories
    fail('linting checks are not run against {}, '
        'therefore changes are not allowed'.format(
            ignored_paths_with_changes))
```

*But what if you  
NEED to change a  
frozen game?*

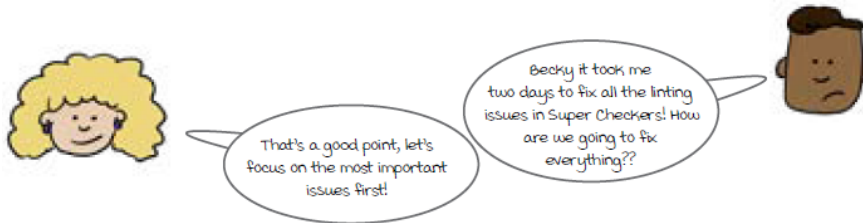
This error message should include some guidance for what to do if that's the case - and the answer is that the submitter will need to then move the game out of the frozen folder - and then deal with all the linting issues which would undoubtedly be surfaced as a result!

Next she added it to the Pipeline that runs against pull requests:



### 4.13 Not all problems are created equal

Okay NOW it was finally time to start fixing problems, right? Ramon dove right in, but two days in he was frustrated:



Becky and Ramon want to focus on fixing the most impactful issues first. Let's look again at the rewards and risks of fixing linting issues for some guidance:

Reward 1: Linting can catch bugs

Reward 2: Linting helps remove distracting errors

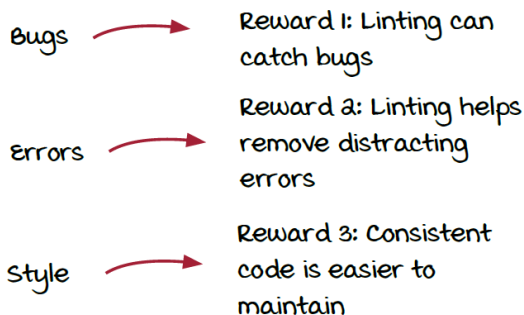
Reward 3: Consistent code is easier to maintain

Risk 1: Making changes can introduce new bugs

Risk 2: Fixing linting issues takes time

Ramon is running smack into Risk 2: it's taking a lot of time for him to fix all the issues. So Becky has a counterproposal: fix the most impactful issues first. That way they can get the most value for the time they do spend, without having to fix absolutely everything.

So which issues should they tackle first? The linting rewards happen to correspond to different types of linting issues:



## 4.14 Types of linting issues

The types of issues that linters are able to find can fall into three buckets: bugs, errors and style.

**Bugs** found by linting are common misuses of code that lead to undesirable behavior, for example:

- Uninitialized variables
- Formatting variable mismatches

Bugs

**Errors** found by linting common misuses of code that do not affect behavior but either cause performance problems or interfere with maintainability. For example:

- Unused variables
- Aliasing variables

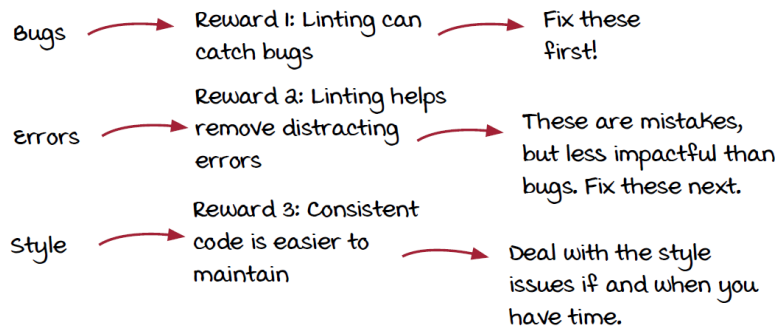
Errors

And lastly the **style** problems found by linters are inconsistent application of code style decisions and code smells, for example:

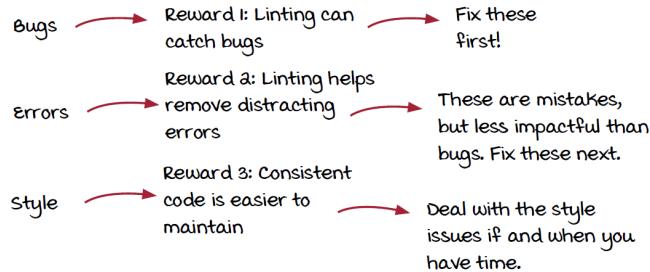
- Long function signatures
- Inconsistent ordering in imports

style

While it would be great to fix all of these, if you only had time to fix one set of linting issues, which would you choose? Probably bugs, right? Makes sense, since these affect the actual behavior of your programs! And that's what the hierarchy looks like:



## 4.15 Bugs first, style later



Becky recommends to Ramon that they tackle the linting issues systematically. That way if they need to switch to another project, they'll know they time they spent fixing issues as well used. They might even decide to time box their efforts: see how many issues they can fix in two weeks, then move on.

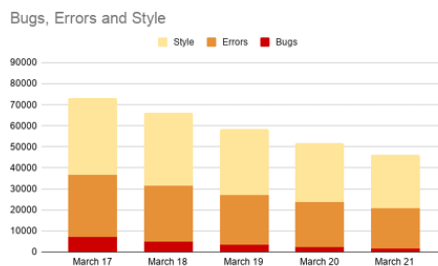
How can they tell which issues are which? Many linting tools categorize the issues they find. Let's look again at some of the issues pylint found:

```

games/bridge.py:46:0: W0311: Bad indentation. Found 2 spaces, expected 4 (bad-indentation)
games/bridge.py:1:0: C0114: Missing module docstring (missing-module-docstring)
games/bridge.py:13:15: E0601: Using variable 'board' before assignment (used-before-assignment)
games/bridge.py:8:2: W0612: Unused variable 'cards' (unused-variable)
games/bridge.py:30:2: R0201: Method could be a function (no-self-use)
games/bridge.py:26:0: R0903: Too few public methods (1/2) (too-few-public-methods)
games/snakes.py:30:4: C0103: Method name "do_POST" doesn't conform to snake_case naming style (invalid-name)
  
```

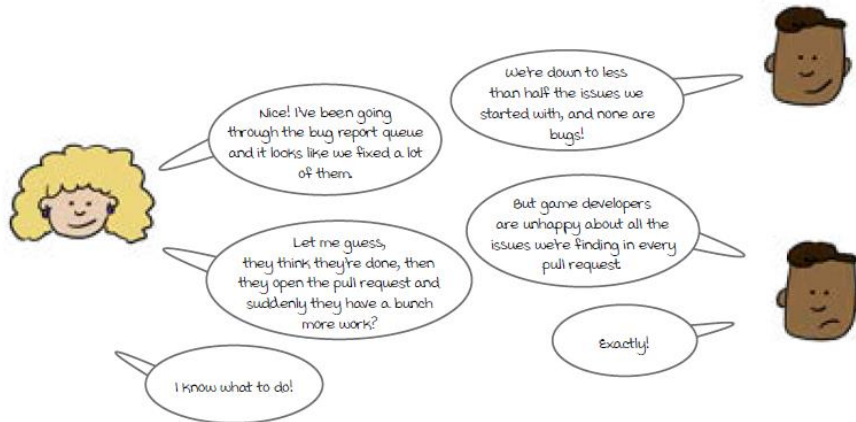
Each issue has a letter and a number. pylint recognizes 4 categories of issues: E is for error, which is the type we are calling bugs. W for warning is what we are calling errors, and the last two, C for convention and R for refactor are what we are calling style.

Ramon creates a script and tracks the number of errors of each type as they work for the next week:



The overall number of issues stays fairly high, but the number of bugs - the most important type of linting issue - is steadily decreasing!

## 4.16 Jumping through the hoops



It can be frustrating to think you're done, just to encounter a whole new set of hoops to jump through.

But the answer here is pretty simple: incorporate linters into your development process. How do you do this, and how do you make it easy for the developers you are working with?

1. Commit the configuration files for your linting alongside your code. Becky and Ramon have checked in the `.pylintrc` code they're using right into the Super Game Console repo. This way developers can use the exact same configuration that will be used by the CI/CD pipeline and there will be no surprises.
2. Run the linter as you work. You could run it manually, but the easiest way to do this is to use your IDE (Integrated Development Environment). Most IDEs, and even editors like vim, will let you integrate linters and run them as you work. This way when you make mistakes, you'll find out immediately.

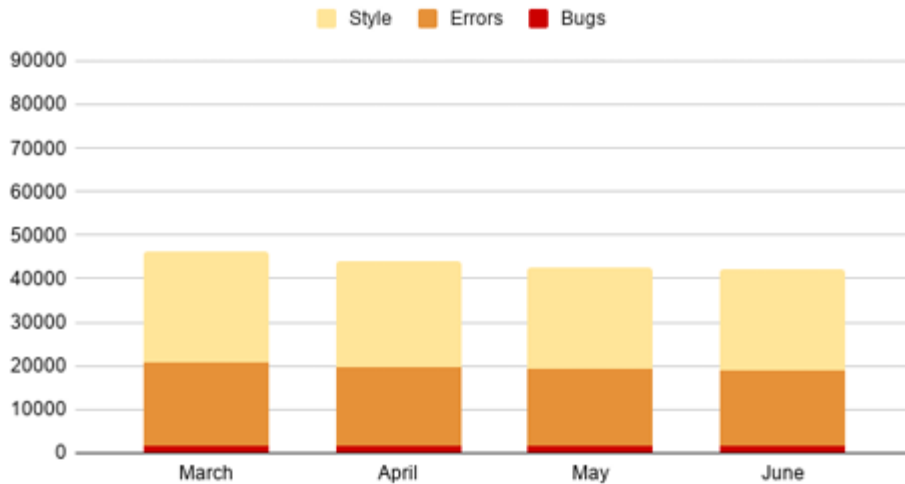
Becky and Ramon send out a PSA to all the developers they work with recommending they turn on linting in their IDEs. They also add a message when the linting task fails on a pull request reminding the game developers that they can turn this on.

### *What about formatters?*

Some languages make it easy to go a step beyond linting and eliminate many coding style questions by providing tools called formatters which automatically format your code as you work. They can take care of issues such as making sure imports are in the correct order and making sure spacing is consistent. If you work in a language with a formatter, this can save you a lot of headaches! Make sure to integrate the formatter with your IDE, and in your pipeline, run the formatter and compare the output to the submitted code.

## 4.17 Legacy code vs the ideal

Becky and Ramon didn't get a chance to fix every single issue because there was a lot of code that already existed before they started linting. This means they have to keep tracking the baseline and making sure that the number of issues doesn't increase, or they have to keep tweaking the pylint configuration to ignore the issues they've decided to just live with.



But what does the ideal look like, i.e. what if Becky and Ramon could spend as much time as they wanted on linting, what state would they want to end up in?

If you are lucky enough to be working on a brand new or relatively small codebase, you can shoot directly for this ideal.

**The ideal: The linter produces 0 problems when run against your codebase.**

Is this a reasonable goal to aim for? Yes! And even if you never get there, shooting for the stars and landing on the moon isn't too bad.

If you're dealing with a new or small codebase, you don't have to do everything that Becky and Ramon did.

In steps 2 and 3 you'll notice that Becky and Ramon spend a lot of time focusing on measuring and tracking the baseline. Instead of doing that, take the time to work through all of the problems. You can still apply the order as described in Step 4, that way if you get interrupted for some reason, you've still dealt with the most important issues first, but the goal is to get to the point where there are 0 problems.

Then, apply a similar check to the one that Becky and Ramon added in step 3, but instead of comparing the number of linting problems to the baseline, require it to always be 0!

## 4.18 Conclusion

Super Game Console had a huge backlog of bugs and issues, and the lack of consistent style across all of their games made them hard to maintain.

Even though their existing codebase was so huge, Becky was able to add linting to their processes in a way that brought immediate value. She did this by approaching the problem iteratively. After re-establishing the project's coding standards, she worked with Ramon to measure the number of linting issues they currently had, and add checks to their pull request pipeline to make sure that the number didn't increase.

As Becky and Ramon started working through the issues, they realized they were not all equally important, so they focused on code that was likely to change, and tackled the issues in priority order.

## 4.19 Summary

- Linting identifies bugs and helps keep your codebase consistent and maintainable.
- The ideal situation is that running linting tools will raise 0 errors, but with huge legacy codebases, we can settle for at least not introducing more errors.
- Changing code always carries the risk of introducing more bugs, so it's important to be intentional and consider if the change is worth it. If the code is changing a lot and/or has a lot of known bugs, it probably is, but otherwise, you can isolate it and leave it alone.
- Linting typically identifies three different kinds of issues, and they are not equally important. Bugs are almost always worth fixing. Errors can lead to bugs and make code harder to maintain, but aren't as important as bugs. Lastly, fixing style issues makes your code easier to work with, but these issues aren't nearly as important as bugs and errors.

## 4.20 Up next . . .

In the next chapter we'll look at how to effectively include unit tests in your pipelines.

## Dealing with noisy tests

# 5



---

### In this chapter

- Explain why tests are crucially important to continuous delivery
  - Create and execute a plan to go from noisy test failures to a useful signal
  - Understand what makes tests noisy
  - Treat test failures as bugs
  - Define flakey tests and understand why they are harmful
  - Retry tests appropriately
- 

It'd be nearly impossible to have Continuous Delivery without tests! For a lot of folks, tests are synonymous with at least the Continuous Integration (CI) side of CD, however, over time some test suites seem to degrade in value. In this chapter we'll take a look at how to take care of noisy test suites.



## 5.1 Continuous Delivery and tests

How do tests fit into Continuous Delivery? Let's look again at what we discussed in chapter 1. Continuous Delivery is all about getting to a state where:

1. You can safely deliver changes to your software at any time
2. Delivering that software is as simple as pushing a button

How do you know you can safely deliver changes? You need to be confident that your code will do what you intended it to do. In software, we gain confidence about our code by testing it. Tests confirm to us that our code does what we meant for it to do.

This book isn't going to teach you to write tests - there are many great books written on the subject you can refer to! We're going to assume that not only do you know how to write tests, but also that most modern software projects have at least *some* tests defined for them. It has become common knowledge that production software needs tests.

In chapter 3 we talked about the importance continuously verifying every change. **It is crucially important that tests are run not only frequently, but on every single change!** This is all well and good when a project is new and only has a few tests, but as the project grows, so do the suites of tests and they can become slower and less reliable over time. In this chapter we're going to look at how to maintain these tests over time so you can keep getting a useful signal, and be confident you're always in a releasable state!



### Vocab time

*Test suite* is a term for a grouping of tests. It often means “the set of tests which test this particular piece of software.”



### Question

Q: Where does QA fit into Continuous Delivery?

A: With all this focus on test automation, you might wonder if Continuous Delivery means getting rid of the QA (Quality Assurance) role. It doesn't! The important thing is to let humans do what humans do best: explore and think outside the box. Automate when you can, but automated tests will always do exactly what you tell them. If you want to discover new problems you've never even thought of, you'll need humans performing a QA role!

## 5.2 Ice Cream for All outage

One company that's really struggling with their test maintenance is the wildly successful ice cream delivery company, Ice Cream for All. Their unique business proposition is that they connect you directly to ice cream vendors in your area so that you can order your favorite ice cream and have it delivered directly to your house within minutes!

Ice Cream for All connects users to thousands of ice cream vendors. To do this, the Ice Cream service needs to be able to connect to each vendor's unique API.

July 4 is a peak day for Ice Cream for All. Every year on July 4, Ice Cream for All receives the most ice cream orders they receive all year. But this year, they had a terrible outage, during the busiest part of the day! The Ice Cream Service was down for more than an hour.

The team working on the Ice Cream Service wrote up a retrospective to try to capture what went wrong and fix it in the future, and had an interesting discussion in the comments:



### Vocab time

A *retrospective*, sometimes called a post mortem, is an opportunity to reflect on processes, often when something goes wrong, and decided how to improve in the future.

#### Retrospective: Ice Cream Service Outage July 4

##### Impact:

80% of Ice Cream Service requests errored with 500 from July 4 19:00 UTC to 20:13 UTC

##### Root cause:

Pull Request #20034 introduced a regression (previously fixed in issue #9877) into the Ice Cream API Adapter class

Duration: 73 minutes

Resolution: Piyush reverted the changes from #20034 and manually built and pushed a new image for the Ice Cream Service

% of service impacted: 93% of requests to the Ice Cream service failed

Detection: The on call engineer (Piyush) as paged when the SLO violation was detected



Nishi

I'm a bit confused, if we fixed this already, why did it happen again? Didn't we have tests?



Piyush

We do have tests for it, and actually it looks like those tests failed on #20034



Nishi

What?? Why did we merge #20034 anyway?



Pete

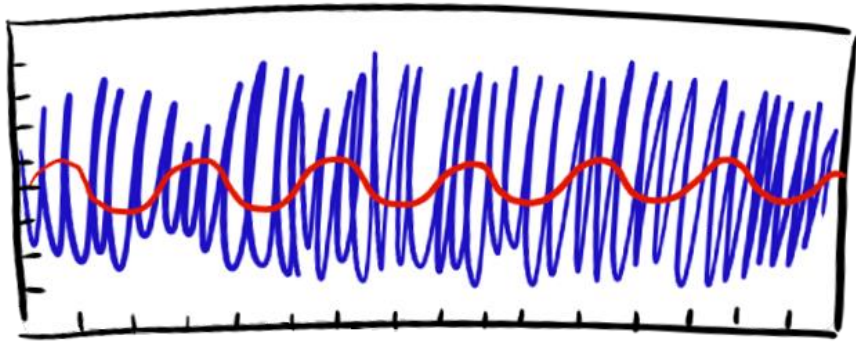
That test fails all the time, so unfortunately we didn't realize it had caught a real problem this time :(

### 5.3 Signal vs. noise

Ice Cream for All has a problem with noisy tests. Their tests fail so frequently that their engineers often ignore the failures. And this caused them real world problems: ignoring a noisy test cost them business on their busiest day of the year!

What should the team do about their noisy tests? Before they do anything, they need to understand the problem. *What does it mean for tests to be noisy?*

The term “noisy” comes from “the signal to noise ratio” which compares some desired information (the **signal**) to interfering information that obscures it (the **noise**).



When we’re talking about tests, what is the signal? What is the information that we’re looking for? This is an interesting question, because your gut reaction might be to say the signal is passing tests. Or maybe the opposite, that failures are the signal.

The answer is: both! The signal is the information, the noise is anything that distracts us from the information.

When tests pass, this gives us information: we know the system is behaving as we expect it to (as defined by our tests). When tests fail, that gives us information too. And it’s even more complicated than that. In the chart below you can see that both failures and successes can be signals and they can be noise.

Tests	Succeed	Fail
Signal	Passes and should pass (i.e. catches the errors it was meant to catch)	Failures provide new information
Noise	Passes but shouldn't (i.e. the error condition is happening)	Failures do not provide any new information

## 5.4 Noisy successes

This can be a bit of a paradigm shift, especially if you are used to thinking of passing tests as providing a good signal, and failing tests as causing noise. This can be true, but as we've just seen it's a bit more complicated:

*The signal is the information, the noise is anything that distracts us from the information.*

- Successes are signals unless they are covering up information
- Failures are signals when they provide new information and noise when they don't

When can a successful test cover up information? One example is a test that passes but really shouldn't, aka a **noisy success**. For example, in the `Orders` class, a method recently added to the Ice Cream For All codebase was supposed to return the most recent order, and this test was added for it:

```
def test_get_most_recent(self):
    orders = Orders()
    orders.add(datetime.date(2020, 9, 4), "swirl cone")
    orders.add(datetime.date(2020, 9, 7), "cherry glazed")
    orders.add(datetime.date(2020, 9, 10), "rainbow sprinkle")

    most_recent = orders.get_most_recent()
    self.assertEqual(most_recent, "rainbow sprinkle")
```

The test currently passes - but it turns out that the method `get_most_recent` is actually just returning the last order in the underlying dictionary:

```
class Orders:
    def __init__(self):
        self.orders = collections.defaultdict(list)

    def add(self, date, order):
        self.orders[date].append(order)

    def get_most_recent(self):
        most_recent_key = list(self.orders)[-1]
        return self.orders[most_recent_key][0]          #A
```

**#A** There are a number of things wrong with this method, including not handling the case where no orders have been added, but more importantly, what if the orders are added out of order?

The method `get_most_recent` is not actually paying attention to when the orders are made at all, it is just assuming that the last key in the dictionary corresponds to the most recent order. And since the test just so happens to be adding the most recent order last (and since Python 3 dictionary ordering is now guaranteed to be insertion order), the test is passing.

But since the underlying functionality is actually broken, the test really shouldn't be passing at all - and this is what we call a **noisy success**: by passing, this test is covering up the information that the underlying functionality actually does not work as intended.

## 5.5 How failures become noise

We've just seen how a test success can be noise - but what about failures? Are failures always noise? Always signal? Neither! The answer is that Failures are signals when they provide new information and noise when they don't. Remember:

*The signal is the information, the noise is anything that distracts us from the information.*

- Successes are signals unless they are covering up information
- Failures are signals when they provide new information and noise when they don't

When a test fails initially, it gives us new information: it tells us there is some kind of mismatch between the behavior the test expects and the actual behavior. This is a signal.

That same signal can become noise if we ignore the failure. The next time the same failure occurs, it's giving us information that we already know: we already knew that the test had failed previously, this new failure is not new information. *By ignoring test failure, we have made that failure into noise.*

This is especially common if it's hard to diagnose the cause of the failure, and if the failure doesn't always happen (say the test passes when run as part of the CI automation, but fails locally), it's much more likely to get ignored, therefore creating noise.



## It's your turn: evaluating signal vs. noise

Let's take a look at some of the test situations that Ice Cream for All is dealing with and categorize them as noise or signal.

1. While working on a new feature around favoriting ice cream flavors, Pete creates a pull request for his changes. One of the UI tests fails because Pete's changes accidentally moved the "order" button from the expected location to a different one.
2. While running the integration tests on his machine, Piyush sees a test fail: `TestOrderCancelledWhenPaymentRejected`. He looks at the output from the test, looks at the test and the code being tested, and doesn't understand why it fails. When he re-runs the test, it passes.
3. Although `TestOrderCancelledWhenPaymentRejected` failed once for Piyush, he can't reproduce the issue and so he merges his changes. Later on, he submits some other change and sees the same test fail against his pull request. He reruns it, and it passes, so he ignores the failure again and merges the changes.
4. Nishi has been refactoring some of the code around displaying order history. While doing this, she notices that the logic in one of the tests is incorrect: `TestPaginationLastPage` is expecting the generated page to include 3 elements but it should only include 2, and there is actually a bug in the pagination logic.



## Answers

1. Signal. The failure of the UI test has given Piyush new information: that he moved the "order" button.
2. Signal. Piyush didn't understand what caused the failure of this test, but something caused the test to fail, and this revealed new information.
3. Noise. Piyush suspected from his previous experience with this test that something might be wrong with it. Seeing the test fail again tells him that the information he got before was legitimate, but by allowing this failure and merging, he has created noise.
4. Noise. The test Nishi discovered should have been failing; by passing it was covering up information.

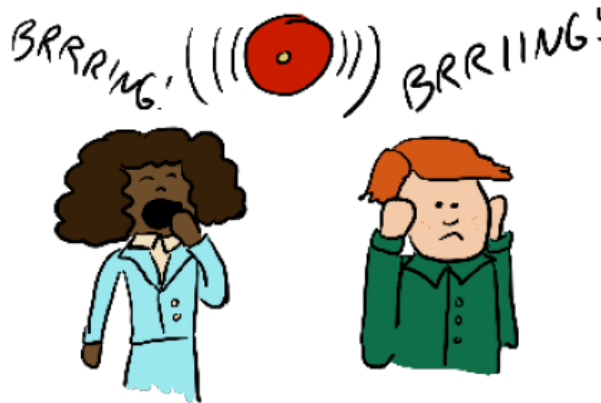
## 5.6 Going from noise to signal

Alerting systems are only useful if people pay attention to the alerts. When they are too noisy, people stop paying attention, so they may miss the signal.

Car alarms are an example of this: if you live in a neighborhood where a lot of cars are parked, and you hear an alarm go off, are you rushing to your window with your phone out, ready to phone in an emergency? It probably depends on frequently it's happened; if you've never heard a alarm like that, you might. But if you hear them every few days, more likely you're thinking, "Oh someone bumped into that car. I hope the alarm gets turned off soon."



What if you live or work in an apartment building and the fire alarm goes off? You probably take it seriously: begrudgingly exit the building. What if it happens again the next day? You'll probably leave the building anyway because those alarms are LOUD but you'd probably start to doubt that it's an actual emergency, and the next day you'd definitely think it's a false alarm.



The longer we tolerate a noisy signal, the easier it is to ignore it and the less effective it is.

## 5.7 Getting to green

The longer we tolerate noisy tests, the easier it is to ignore them - even when there is real information there - and the less effective they are. Leaving them in this state seriously undermines their value. People get desensitized to the failures and feel comfortable ignoring them.

This is the same position that Ice Cream for All is in: their engineers have gotten so used to ignoring their tests that they've let some major problems slip through, which were actually caught by the tests, and they've actually lost money as a result.

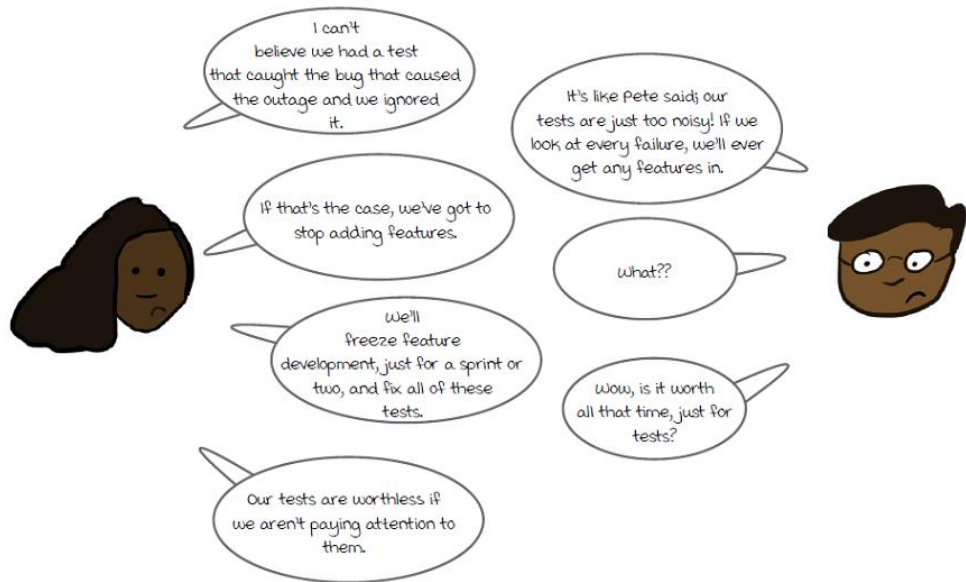


### Vocab time

Test successes are often visualized as *green* while failures are often red; getting to green means that all your tests are passing!

How do they fix this? The answer is to get to green as fast as possible, i.e. get to a state where the tests are in a consistent state (passing) and any change in this state (failing) is a real signal that needs to be investigated.





Nishi is totally right: creating and maintaining tests isn't something we do for the sake of the tests themselves; we do this because we believe they add value and most of that value is the signals they give us.

So she has made a hard decision: stop adding features until all the tests are fixed.

Tests provide value in other ways too, for example creating unit tests can help improve the quality of your code; but that's outside the scope of this book. Look for a book about unit testing to learn more!

## 5.8 Another outage!

The team did what Nishi requested: they froze feature development for two weeks and during that time did nothing but fix tests. After the end of week 3, the test task in their pipeline was consistently passing. They had gotten to green!



The team felt confident about adding new features again and in the third week, went back to their regular work. At the end of that week, they had another release - and a small party to celebrate. But at 3am, Nishi was roused from her sleep by an alert telling her there had been another outage.



### Noodle on it

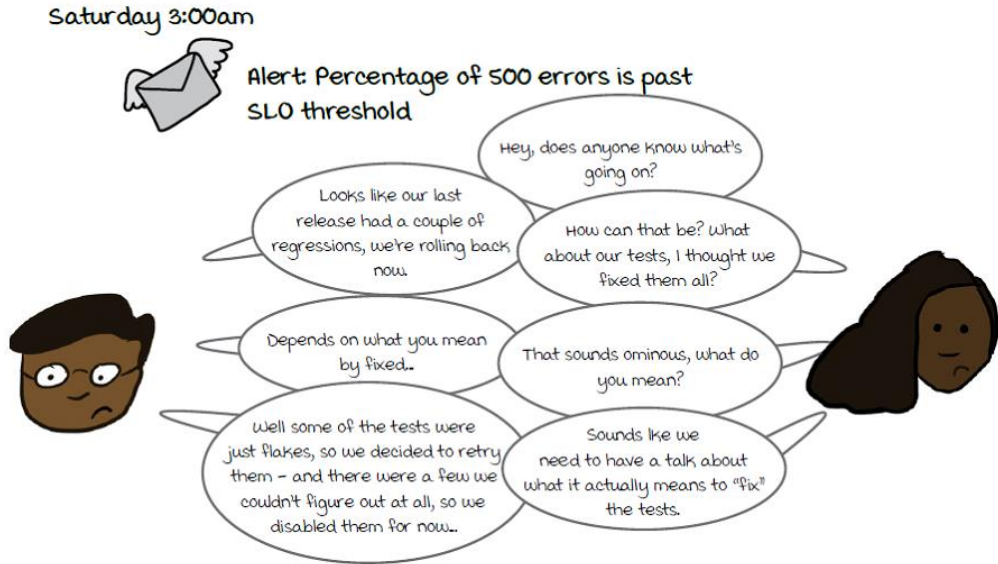
Did Nishi make a good call? She decided to freeze feature development for two weeks, to focus on fixing tests, and the end result was another outage. Looking back at that decision, an easy conclusion to draw is that the (undoubtedly expensive) feature freeze was not worth it and did more harm than good.

Nishi was faced with a decision that many of us will often face: maintain the status quo (where, for Ice Cream for All, this meant noisy tests that unexpectedly cause outages) or take some kind of action. Taking action means trying something out and making a change; anytime you change the way you do things, you're taking a risk: the change could be good or it could be bad. Often, it's both! And, often, when changes are made, there can be an adjustment period during which things are definitely worse, though ultimately they end up better.

What would you do if you were in Nishi's position? Do you think she made the right call? What would you do differently in her shoes, if anything?

## 5.9 Passing tests can be noisy

Nishi jumped into the team's group chat to investigate the outage she had just been alerted to:



The team had felt good about their test suite because all the tests were passing, but unfortunately they hadn't actually removed the noise - they'd just changed it. Now, the successful tests were the noise.

Getting the test suite from noise to signal was the right call to make, and getting the suite from often failing to green was a good first step, because it combats desensitization.

But just getting to green isn't enough: test suites that pass can still be noisy, and can hide serious problems.



### Vocab time

Test successes are often visualized as **green** while failures are often red; getting to green means that all your tests are passing!

*The team at Ice Cream for All had fixed their desensitization problem, but they hadn't actually fixed their tests.*



## Takeaway

Nishi made a good call, but getting to green on its own isn't enough. The goal is to get to a state where the tests are in a consistent state (passing) and any change in this state (failing) is a real signal that needs to be investigated. When dealing with noisy test suites:

1. Get to green as fast as possible.
2. Actually FIX every failing test; just silencing them adds more noise.

## 5.10 Fixing test failures

You might be surprised to learn that it's not totally straightforward to know if you have fixed a test! It comes back to the question of what constitutes a signal and what is noise when it comes to tests.

People often think that fixing a test means going from a failing test to a passing test. But there is more to it than that!

Technically **fixing the test means that you have gone from the state of the test being noise to it being a signal**. This means that there are tests that are currently passing which may need "fixing." More about that in a bit, for now let's talk about fixing tests that are currently failing.

Every time a test fails, this means one (or both) of two things have happened:

1. The test was written incorrectly (i.e. the system was not intended to behave in the way the test was written to expect).
2. There is a bug in the system (i.e. the test is correct and it's the system that isn't behaving correctly).

What's interesting is that we write tests with situation (2) in mind, but when tests fail (especially if we can't immediately understand why), we tend to assume that the situation is (1), i.e. that the tests themselves are the problem.

This is what is usually happening when people say their tests are noisy: their tests are failing and they can't immediately understand why, so they jump to the conclusion that something is wrong with the tests.

But both (1) and (2) have something in common:

**When a test fails, there is a mismatch between how the test expects the system to behave and how the system is actually behaving.**

Regardless of whether the fix is to update the test or to update the system, there is a mismatch that needs to be investigated.

This is the point in the test's lifecycle where there is the greatest chance that noise will be introduced. The test's failure has given you information, specifically that there is a mismatch between the tests and the system. If you ignore that information, then every new failure isn't telling you anything new, it's repeating what you already know: there is a mismatch. *This is how test failures become noise.*

The other way you can introduce noise is by misdiagnosing case (2) as case (1). It is often easier to change the test than it is to figure out why the system is behaving the way it is; if you do this without really understanding the system's behavior, you've created a noisy successful test. Every time that test passes, it's covering up information: the fact that there was a mismatch between the test and the system that was never fully investigated.

**Treat every test failure as a bug and investigate it fully.**

## 5.11 Ways of failing: flakes

Complicating the story around signal and noise in tests, we have the most notorious kind of test failure: **the test flake**.

Tests can fail in two ways:

1. Consistently: every time the test is run, it will fail
2. Inconsistently: sometimes the test succeeds, sometimes it fails, and the conditions that make it fail are not clear

Tests that fail inconsistently are often called **flakes** or **flakey**, and when these tests fail, this is often called **flaking**, because in the same way that you cannot rely on a flakey friend to follow through on plans you make with them, you cannot depend on these tests to consistently pass or fail.



Consistent tests are much easier to deal with than flakes - and much more likely to be acted on (hopefully in a way that reduces noise). Flakes are the most common reason that a test suite ends up in a noisy state.

And maybe because of that, or maybe just because it's easier, people do not treat flakes as seriously as consistent failures.

- Flakes make test suites noisy
- Flakes are likely to be ignored and treated as not serious

This is kind of ironic, because we've seen that the noisier a test suite is, the less valuable it is. And what kind of test is likely to make a test suite noisy? The flake, which we are likely to ignore. What is the solution?

**Treat flakes like any other kind of test failure: like a bug.**

Just like any other case of test failure, flakes represent a mismatch between the system's behavior and the behavior that the system expects, the only difference is that there is something about that mismatch that is non-deterministic.

## 5.12 Reacting to failures

What went wrong with Ice Cream For All's approach? They had the right initial idea.

**When tests fail, stop the line: don't move forward until they are fixed.**

This means: if you have failing tests in your codebase, it's important to get to green as fast as possible, i.e. stop all merging into your main branch until those failures are fixed. And if it's happening in a branch, don't merge that branch until the failures are fixed.

But the question is: *how do you fix those failures?* You have a few options

1. **Actually fix it.** Ultimately the goal is to understand why the test is failing and either fix the bug that is being revealed or update an incorrect test.
2. **Delete the test.** This is rare, but your investigation may reveal that this test was not adding any value and its failure is not actionable. In that case, there's no reason to keep it around and maintain it.
3. **Disable the test.** This is an extreme measure and if it is done, it should only be done temporarily. Disabling the test means that you are hiding the signal. Any disabled tests should be investigated as fast as possible and either actually fixed (see above) or deleted.
4. **Retry the test.** This is another extreme measure, and also hides the signal. This is a common way of dealing with flakey tests. The reasoning behind this is rooted in the idea that ultimately what we want the tests to do is pass, but this is incorrect: what we want the tests to do is provide us information. If a test is sometimes failing, and we cover that up by retrying it, we're actually hiding the information and creating more noise. Retrying is sometimes appropriate, but rarely at the level of the test itself.

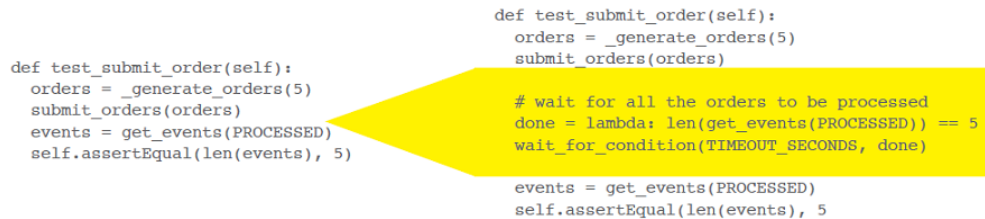
Looking at these options, really the only good options are (1) and in some rare cases, (2). Both (3) and (4) are stop gap measures that should only be taken temporarily, if at all, because they add noise to your signal by hiding failures.

## 5.13 Fixing the test: change the code or the test?

Ice Cream for All had rolled back their latest release and once again frozen feature development as they looked into the tests they had tried to “fix” previously.

Looking back through some of the fixes that had been merged, Nishi noticed a disturbing pattern: many of the “fixes” were changing only the tests, very few of them were changing the actual code being tested. Nishi knew this was an antipattern.

For example, this test had been flaking, so it was updated to wait longer for the success condition:



```
def test_submit_order(self):
    orders = _generate_orders(5)
    submit_orders(orders)
    events = get_events(PROCESSED)
    self.assertEqual(len(events), 5)
```

```
def test_submit_order(self):
    orders = _generate_orders(5)
    submit_orders(orders)

    # wait for all the orders to be processed
    done = lambda: len(get_events(PROCESSED)) == 5
    wait_for_condition(TIMEOUT_SECONDS, done)

    events = get_events(PROCESSED)
    self.assertEqual(len(events), 5)
```

The test was initially written with an assumption in mind: that the order would be considered acknowledged immediately after it had been submitted. And in fact that code that called `submit_orders` was built with this assumption as well. But this test was flaking because there was a race condition in `submit_orders`!

Instead of fixing this problem in the `submit_orders` function, someone had updated the test instead, which covered up the bug, and added a noisy success to the test suite.

### They were in fact hiding the bug!

Whenever you deal with a test that is failing, before you make any changes, you have to understand:

Is the test failing because of a problem with the actual code that is being tested? That is, if the code acts like this when it is actually being used, is that what it should be doing? If it is, then it's appropriate to fix the test. But if not, the fix shouldn't go in the test: it should be in the code.

This means making a mental shift from “let's fix the test”, i.e. “making the test pass” to “let's understand the mismatch between the actual behavior of the code - and make the fix in the appropriate place.”

### Treat every test failure as a bug and investigate it fully.

Nishi asked the engineer who updated the test to investigate further; after finding the source of the race condition, they were able to fix the underlying bug and the test didn't need to be changed at all.



## 5.14 The dangers of retries

Retrying an entire test is usually not a good idea, because ANYTHING that causes the failure will be hidden.

Take a look at this test in the Ice Cream Service integration test suite, one of the tests for their integration with Mr. Freezie:

```
# We don't want this test to fail just because
# the MrFreezie network connection is unreliable
@retry(retries=3)
def test_process_order(self):
    order = _generate_mr_freezie_order()
    mrf = MrFreezie()
    mrf.connect()
    mrf.process_order(order)
    _assert_order_updated(order)
```

During the development freeze, Pete had made the decision that this test should be retried. His reasons were sound: the network connection to Mr. Freezie's servers were known to be unreliable, so this test would sometimes flake because it couldn't establish a connection successfully, and would immediately pass on a retry.

But the problem is that Pete is retrying the entire test: this means that if the test fails for some other reason, the test will still be retried. And that's exactly what happened - it turned out there was a bug in how they were passing orders to Mr. Freezie which made it so that the total charge was sometimes incorrect - and when this happened in the live system, users were being charged the wrong amount, leading to 500 errors and an outage.

What should Pete do instead? Remember that test failures represent a mismatch:

**When a test fails, there is a mismatch between how the test expects the system to behave and how the system is actually behaving.**

Pete needs to ask himself the question we need to ask every time we investigate a test failure:

**Which represents the behavior we actually want: the test or the system?**

A reasonable improvement on Pete's strategy would be to change the retry logic to just be around the network connection:

```
def test_process_order_better(self):
    order = _generate_mr_freezie_order()
    mrf = MrFreezie()

    # We don't want this test to fail just because
    # the MrFreezie network connection is unreliable
    def connect():
        mrf.connect()
    retry_network_errors(connect, retries=3)

    mrf.process_order(order)
    _assert_order_updated(order)
```

## 5.15 Retrying revisited

Pete had improved his retry based solution by only retrying the part of the test that he felt was okay to have fail sometimes, but in code review, Piyush took it a step further:



**Piyush**

Thanks for the fix Pete!! This is way better :)

Just wondering, in the code that actually calls `MrFreezie.Connect()`, do we do the same retrying? I'm thinking that if the connection is so unreliable, users are going to run into the same problem.



**Pete**

Oh that's a good point - you're right, if the `Connect()` call fails for any of our integrations, we just immediately give up. I'll update the Ice Cream Service code so that we are a bit more tolerant of network errors.

There were actually two bugs being covered up by the retry: in addition to missing the bug with how orders were being passed to Mr. Freezie, there was a larger bug in that none of the Ice Cream Service code was tolerant of network failures either (you don't want your ice cream order to fail just because of a temporary network problem, do you?).

Ice Cream for All was actually lucky that they caught the issues that the retry was introducing so quickly. If there hadn't been an outage, they may never have noticed, and they probably would have used this retry strategy to deal with more flakey tests. You can imagine how this can built up over time: imagine how many bugs they would be hiding after a few years of applying this strategy.

**Causing flakey tests to pass with retries introduces noise: the noise of tests that pass but shouldn't.**

The nature of software projects is that we are going to keep adding more and more complexity, which means the little shortcuts we take are going to get blown up in scope as the projet progresses.

Slowing down a tiny bit and rethinking stop gap measures like retries will pay off in the long run!



## It's your turn

Piyush is trying to deal with another flakey test in the Ice Cream service. This test fails, but it happens less than once a week, even though the tests run at least a hundred times per day, and it's very hard to reproduce locally:

```
def test_add_to_cart(self):
    cart = _generate_cart()
    items = _generate_items(5)
    for item in items:
        cart.add_item(item)
    self.assertEqual(len(cart.get_items()), 5)
```

This assertion  
sometimes fails

The cart is backed by a database. Every time an item is added to the cart, the underlying database is updated, and when items are read from the cart, they are read from the database.

1. Assume that what when the tests fail, the number of items in the cart is 4 instead of 5. What do you think might be going wrong?
2. What if the problem is that the number of items is 6 instead of 5, what might be going wrong then?
3. If Piyush deals with this by retrying the test, what bugs might he risk hiding?
4. Let's say Piyush noticed this problem while he was trying to fix a critical production issue. What can he do to make sure he doesn't add more noise, while not blocking his critical fix?



## Answers

1. If the number of items read is less than the number written, suggesting there is a race somewhere; some kind of synchronization needs to be introduced to ensure that reads actually reflect the writes.
2. If the number is greater, there might be a fundamental flaw in how items are being written to the database.
3. Either of the above scenarios suggest flaws in the cart logic that could lead to customer orders being lost and customers being over or under charged.
4. In this scenario, adding a temporary retry to unblock this work might be reasonable, as long as the issue with `test_add_to_cart` is subsequently treated as a bug and the retry logic is quickly removed.

## 5.16 Why do we retry?

Given what we just looked at, you might be surprised that anyone retries failing tests at all. If it's so bad, why do so many people do it, and why do so many test frameworks support it?

There are a few reasons:

1. There are often good reasons to have some kind of retrying logic; for example in Pete's case he was right to want to retry network connections when they fail. But instead of taking the extra step of making sure the retry logic is in the appropriate place, it's easier to retry the whole test.
2. Another very compelling reason is that if you've setup your pipelines appropriately (more on this in the next chapter!), then a failing test blocks development and slows people down. It's reasonable that people often want to do the quickest easiest thing they can do to unblock development; and in situations like that, using retries as a temporary fix can be appropriate - as long as it's only temporary.
3. It feels good to fix something, and it feels even better to fix something with a clever piece of technology; retries let you get immediate satisfaction.
4. Most importantly, people often have the mentality that the goal is to get the tests to pass, but that's a misconception. We don't make tests pass just for the sake of making tests pass. We maintain tests because we want to get information from them (the signal). When we cover up failures without addressing them properly, we're actually reducing the value of our test suite by introducing noise.

*Temporary is forever*

Be careful whenever you make a temporary fix: these fixes reduce the urgency of addressing the underlying problem, and before you know it, 2 years have gone by and your temporary fix is now permanent.

So if you find yourself tempted to retry a test, try to slow down and see if you can understand what's actually causing the problem. Retrying can be appropriate if it is:

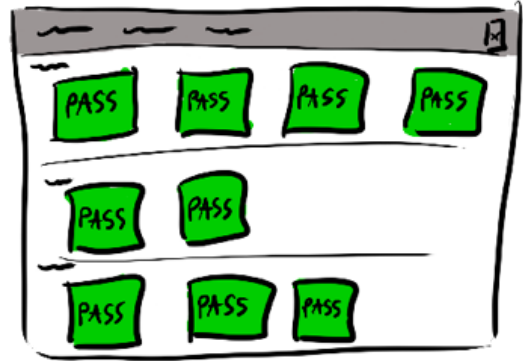
- Applied only to non-deterministic elements that are outside of your control (for example, integrations with other running systems)
- Isolated to precisely the operation you want to retry (for example in Pete's case, retrying the `Connect()` call only, vs. retrying the entire test)

## 5.17 Get to green and stay green

It seems like no matter what Ice Cream for All did, something went wrong. In spite of that, they had the right approach; they just ran into some valuable lessons that they needed to learn along the way - and hopefully we can learn from their mistakes!

Regardless of your project, your goal should be to get your test suite to green and keep it green.

If you currently have a lot of tests that fail (whether they fail consistently or are flakes), it makes sense to take some drastic measures in order to get back to a meaningful signal:



- Freezing development to fix the test suites will be worth the investment. If you can't get the buy-in for this (it's expensive!) all hope is not lost, it'll just be harder.
- Disabling and retrying problematic tests, while not approaches you want to take in the long run, can help you get to a green, i.e. get back to a signal people will listen to - as long as you prioritize properly investigating them afterward!

Remember, there's always a balance: no matter how hard you try and how well you maintain your tests, there are always going to be bugs. The question is, what is the cost of those bugs?

If you're working on critical healthcare technology, the cost of those bugs is enormous, and it's worth taking the time to carefully stamp out every bug you can. But if you're working on a website that let's people buy ice cream, you can definitely get away with a lot more. (Not to say ice cream isn't important - it's delicious!)

Get to green and stay green. Treat every failure as a bug, but also don't failures any more seriously than you need to.

### *Okay come on: lots of tests are flakey and they don't all cause outages right?*

It's probably extreme that poor Ice Cream for All had multiple outages that could have been caught by these neglected tests, but it's not out of the question. Usually the issues these cause are a bit more subtle, but the point is that you never know. And the bigger problem is that treating tests like this undermines their value over time. Imagine the difference between a fire alarm that sometimes means fire and sometimes doesn't (or even worse, sometimes fails to go off when there is a fire!) versus one that ALWAYS means fire. Which is more valuable?



## Build Engineer vs. Developer

Depending on your role on the team, you may be reading this chapter in horror, thinking: But I can't change the tests!

It's pretty common to divide up roles on the team such that someone ends up responsible for the state of the test suite, but they are not actually the same people developing the features, or writing the tests. This can happen to folks in a role called "build engineer", or "engineering productivity" or similar: these are roles which are adjacent to, and supporting the feature development of a team.

If you find yourself in this role, it can be very tempting to lean on solutions that don't require input or work from the developers on the team working on features. This is another big reason why we end up seeing folks trying to rely on automation (e.g. retrying) instead of trying to tackle the problems in the tests directly.

But if the evolution of software development so far has taught us anything, it's that drawing lines of responsibility too rigidly between roles is an antipattern: just take a look at the whole DevOps movement: an attempt to break down the barriers between the developers and the operations team. Similarly, if we draw a hard line between build engineering and feature development, we'll find ourselves walking down a similar path of frustration and wasted effort.

When we're talking about CD in general, and about testing in particular, the truth is that we can't do this effectively without effort from the feature developers themselves. Trying to do this will lead to a degradation in the quality and effectiveness of the test suite over time.

So if you're a build engineer, what do you do? You have three options:

1. You can apply fancy automation like retries (and the strategies we'll see in the next chapter) and accept the reality that this will cause the test suite to degrade over time.
2. You can learn to put on the feature developer hat and make these required fixes (to the tests AND the code you're testing).
3. You can get buy in from the feature developers and work closely with them to address any test failures, e.g. opening bugs to track failing tests and trusting them to treat the bugs with appropriate urgency.

## 5.18 Conclusion

Testing is the beating heart of Continuous Delivery! Without testing we don't know if the changes that we are trying to Continuously Integrate are safe to deliver. But the sad truth is that the way we maintain our tests suites over time often causes them to degrade in value. In particular this often comes from a misunderstanding about what it means for tests to be noisy - but it's something we can proactively address!

## 5.19 Summary

- Tests are crucial to Continuous Delivery
- Both failing AND passing tests can be causing noise; noisy tests are any tests that are obscuring the information that your test suite is intended to provide
- The best way to restore the value of a noisy test suite is to get to green (a passing suite of tests) as quickly as possible
- Treat test failures as bugs and understand that often the appropriate fix for the test is in the code and not the test itself; either way the failure represents a mismatch between the system's behavior and the behavior the test expected and it deserves a thorough investigation
- Retrying entire tests is rarely a good idea and should be done with caution

## 5.20 Up next . . .

In the next chapter, we'll continue to look at the kinds of issues that plague test suites as they grow over time, particularly their tendency to become slower, often to the point of slowing down actual feature development.

## Speeding up slow test suites

# 6



---

### In this chapter:

- Speed up slow test suites by running faster tests first
- Use the test pyramid to identify the most effective ratio of unit to integration to system tests
- Use test coverage measurement to get to and maintain the appropriate ratio
- Get a faster signal from slow tests using parallel and sharded execution
- Understand when parallel and sharded execution are viable and how to use them

---

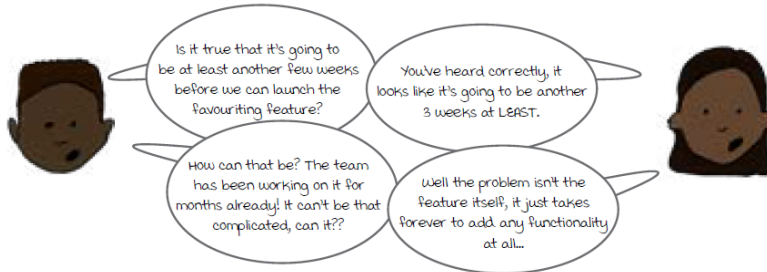
In the last chapter we learned how to deal with test suites that weren't giving us a good signal - but what about tests that are just plain old slow? No matter how good the signal is, if it takes too long to get it, it'll slow down your whole development process! Let's see what we can do with even the most hopelessly slow suites.



## 6.1 Dog Picture Website

Remember Cat Picture website from Chapter 2? Their biggest competitor, Dog Picture Website, has been struggling with their velocity.

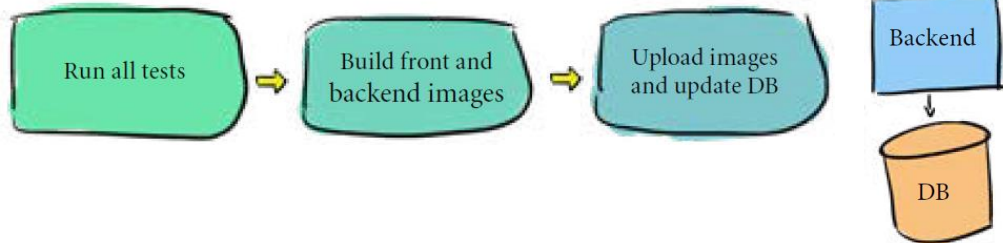
Jada, the product manager is upset because it's taking months for even the simple features that users are demanding to make it to production.



To understand why development is so slow for Dog Picture Website, let's take a quick look at their architecture and their pipeline.

You might notice that the Dog Picture Website architecture is a bit less complex than some of the other architectures we've looked at: they have separated their frontend and backend services, but they haven't gone any further than that, and they haven't moved any of their storage to the cloud.

With such a simple architecture, why are they running into trouble?



### *Is moving to the cloud the answer?*

One big difference between Dog Picture Website and Cat Picture Website you might notice is that Cat Picture Website uses cloud storage. Is that the answer here? Not to solve this problem! If anything, that would complicate the testing story because less of the components would be in the engineer's control. (There are other benefits that outweigh these downsides, but that's a topic for a different book!)

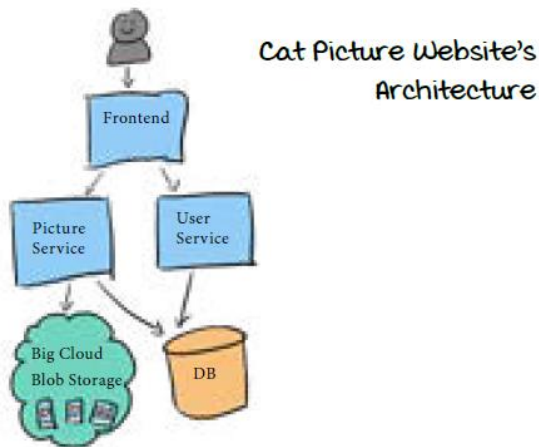
## 6.2 When simple is too simple

The pipeline that Dog Picture Website is using seems simple and reasonable - and at first glance, it might seem the same as the pipelines we've looked at so far. But there is an important difference.



This is the *only* pipeline that Dog Picture website uses. They use this to test, build and upload both their frontend and backend images. There is no other pipeline.

Back in chapter 2 we looked at the architecture and pipeline design used by Dog Picture Website's biggest competitor: Cat Picture Website.



Cat Picture Website uses a separate pipeline for each of their services:



Dog Picture website has decided instead to have one pipeline for their entire system; which is a reasonable starting point, but also one that they never evolved beyond. In particular, the task that runs their tests runs all of their tests at once.

In the sophistication of their pipeline design, Dog Picture Website is way behind their closest competitor!

## 6.3 New engineer tries to submit code

Let's take a look at what it's like to try to submit code to Dog Picture Website and how the pipeline design, particularly the test aspect, impacts velocity.

Sridhar, who is new to Dog Picture Website, has been working on the new favoriting feature that Jada was asking about. In fact, he's already written the code that he thinks the feature needs and he's written some tests as well.

What happens next?

Tuesday

2:00pm: Sridhar pushes his code changes

3:14pm: Another developer pushes changes

3:30pm: Another developer pushes yet more changes

11:00pm The CD system starts the nightly run of the pipeline



Wednesday

1:42am The tests fail



The CD system emails Sridhar and the other 2 developers who pushed changes, telling them the pipeline is broken.

4:02pm: After spending all day trying to debug the problem, Sridhar and the other 2 developers revert their changes to fix the pipeline. Sridhar will try to debug the failures and hopefully push again tomorrow.



Dog Picture Website's problems are different from the ones we looked at in the previous chapter: their test suite is always green, but the tests are only run once a day in the evening, and in the morning they have to sort out who broke what. And just like we saw in Chapter 2, this really slows things down!

## 6.4 Tests and Continuous Delivery

This might be a good time to ask an interesting question: with this process, is Dog Picture Website actually practicing Continuous Delivery? To some extent, the answer is always yes, in that they have some elements of the practice, including deployment automation and **Continuous Testing**, but let's look back again at what we learned in chapter 1. You're doing **Continuous Delivery** when:

1. You can safely deliver changes to your software at any time
2. Delivering that software is as simple as pushing a button

Thinking about the first element, can Dog Picture Website safely deliver changes at any time? Sridhar merged his changes hours before the nightly automation noticed that the tests were broken. What if Dog Picture Website had wanted to do a deployment that afternoon, would that have been safe?

No! Definitely not! Because their tests run only at night:

- They will always have to wait until at least the day after a change has been pushed to deploy it.
- The only time they know they are actually in a releasable state is immediately after the tests pass, before any other changes are added (say the tests pass at night and someone pushes a change at 8am: that immediately puts them back into the state where they don't know if they can release or not

In conclusion, Dog Picture Website is falling short of the first element of Continuous Delivery.

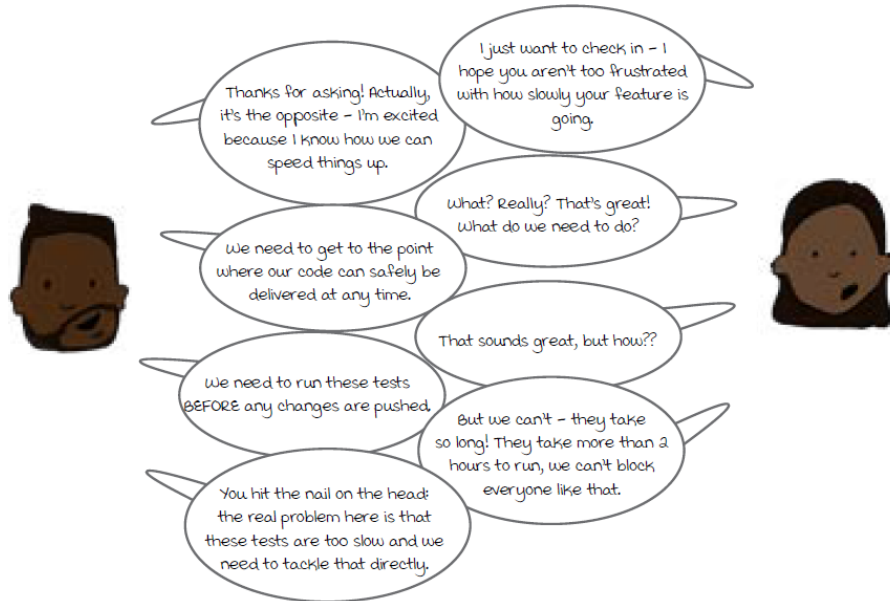


### VOCAB TIME

*Continuous Testing* is a phrase that refers to running tests as part of your Continuous Delivery pipelines. It's not so much a separate practice on its own, as it is an acknowledgement that tests need to be run *continuously*. Just having tests isn't enough: you have have tests, but never run them, or you may automate your tests, but only run them once in a while.

## 6.5 Diagnosis: too slow

Fortunately Sridhar is an experienced engineer and has seen this kind of problem before!



His manager is skeptical, but Sridhar is confident and Jada, their product manager, is overjoyed at the idea of doing something to fix their slow velocity.

Sridhar looks at the average runtimes of the test suite over the past few weeks: 2 hours and 35 minutes. He sets the following goals:

- Tests should run on every change, before the change gets pushed
- The entire test suite should run in an average of 30 minutes or less
- The integration and unit tests should run in less than five minutes
- The unit tests should run in less than one minute

The numbers you choose to aim for with your test suite will depend on your project, but in most cases should be in the same order of magnitude as the ones Sridhar chose.

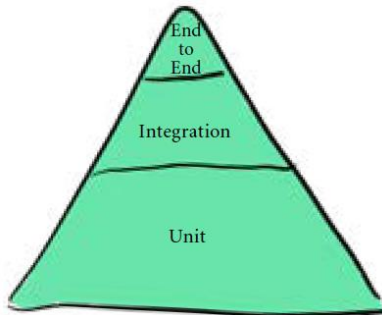
***If it hurts, bring the pain forward!***

When something is difficult or takes a long time, our instinct might be to delay it as long as possible, but the best approach is the opposite! If you isolate yourself from the problem, you'll be less motivated to ever fix it. So the best way to deal with bad processes is do them more often!

## 6.6 The test pyramid

You may have noticed that the goals Sridhar set are different depending on the type of test involved:

- The entire test suite should run in an average of 30 minutes or less
- The integration and unit tests should run in less than five minutes
- The unit tests should run in less than one minute

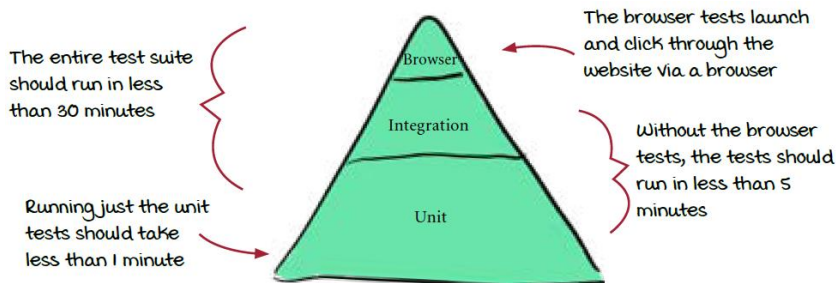


What are these kinds of tests that we're talking about? Sridhar is referring to the test pyramid, a common visualization for the kinds of tests that most software projects need and the approximate ratio of each kind of test that's appropriate.

The idea is that the vast majority of tests in the suite will be unit tests, and there will be a significantly smaller number of integration tests and finally a small number of end to end tests.

We're not going to go into detail about the specific differences between these kinds of tests in general - take a look about testing to learn more!

Sridhar has used this pyramid to set the goals for the Dog Picture website test suite:



### *Service tests vs UI tests vs end to end tests vs integration tests vs...*

If you have seen test pyramids before, you may have seen slightly different terminology used; the terminology is less important than the idea that there are different kinds of tests, where the tests at the bottom of the pyramid are the least coupled and the tests at the top are the most coupled (where “coupled” refers to the increasing interdependencies between components being tested, usually resulting in more complicated tests that take longer to run).

## 6.7 Fast tests first

One of the big reasons why Sridhar is taking an approach to the tests based on the pyramid is that he knows that one immediate way to get feedback faster is to start grouping and running the tests based on the kinds of tests they are.

### Run the fastest tests first.

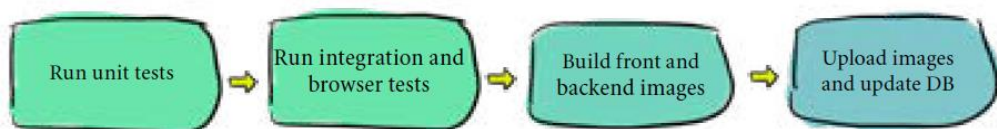
At the moment, Dog Picture Website is running all of their tests simultaneously, but when Sridhar identifies the unit tests in the code base and runs them on their own, he finds that they already run in less than a minute. He's already accomplished his first goal!

If he can make it easy for all the Dog Picture Website developers to run just the unit tests, they'll have a quick way to get some immediate feedback about their changes. They can run these tests locally, and they can immediately start running these tests on their changes before they get merged.

All he needs to do is find a way to make it easy to run these tests in isolation. He has a few choices of how to do this:

- Conventions around test location is the easiest way, for example, you could always store your unit tests beside the code that they test, and keep integration and system tests in different folders. To run just the unit tests, run the tests in the folders with the code (or in a folder called unit); to run the integration tests run the tests in the integration test folder, etc.
- Many languages allow you to specify the type of test somehow, for example by using a build flag in Golang (you can isolate integration tests by requiring them to be run with a build flag `integration`) or in Python if you use the `pytest` package you can use a decorator to mark tests of different types.

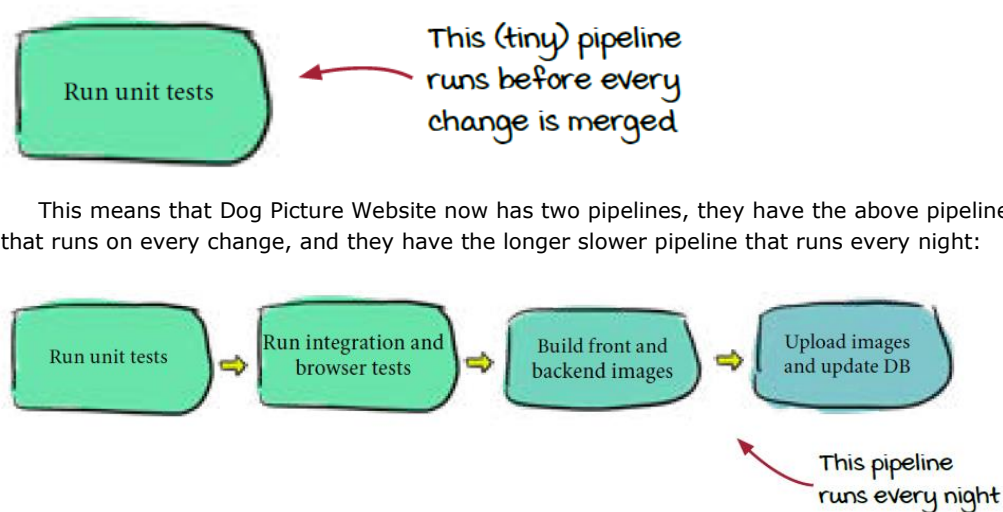
Fortunately Dog Picture Website has already been more or less following a convention based on test location: browser tests are in a folder called `tests/browser` and the unit tests live next to the code. The integration tests were mixed in with the unit tests, so Sridhar moved them into a folder called `tests/integration` and then updated their pipeline to look like this:



## 6.8 Two pipelines

Up until now, engineers had to wait until the nightly pipeline run to get feedback on their changes, because the pipeline takes so long to run. However the new “Run unit tests” task that Sridhar has made runs in less than a minute, so it’s safe to run that on every change, even before the change is merged.

Sridhar updates the Dog Picture Website automation so that the following pipeline, containing only one task, runs on every change before merging:



This means that Dog Picture Website now has two pipelines, they have the above pipeline that runs on every change, and they have the longer slower pipeline that runs every night:

Is it bad that they have two pipelines? The goal is always get “shift left” and get as much information as early as possible (more about this in the next chapter), so this situation is not ideal, but by creating the separate, faster pipeline that can run on every change, Sridhar was able to improve the situation: previously, engineers got no feedback at all on their changes before they were merged, now they will at least get some feedback. Depending on your project’s needs, you may have one pipeline, or you may have many. See the chapter on graph design for more on this.



When dealing with a slow suite of tests, get an immediate gain by making it possible to run the fastest tests on their own, and by running those tests first, before any others. Even though the entire suite of tests will still be just as slow as ever, this will let you get some amount of the signal out faster.

### TAKEAWAY

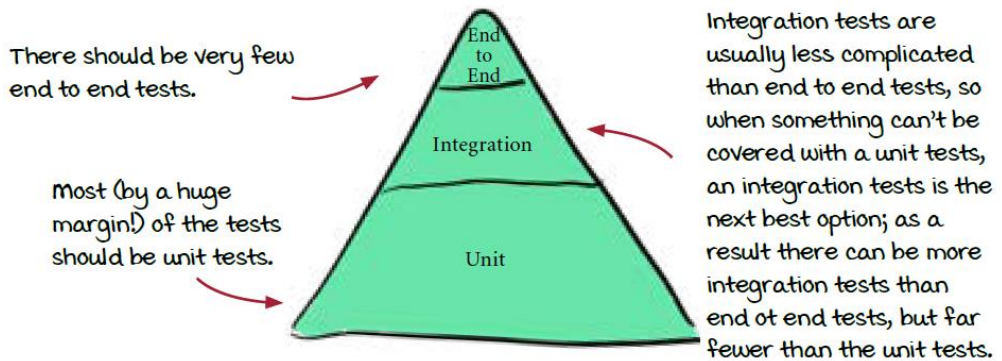


## 6.9 Getting the right balance

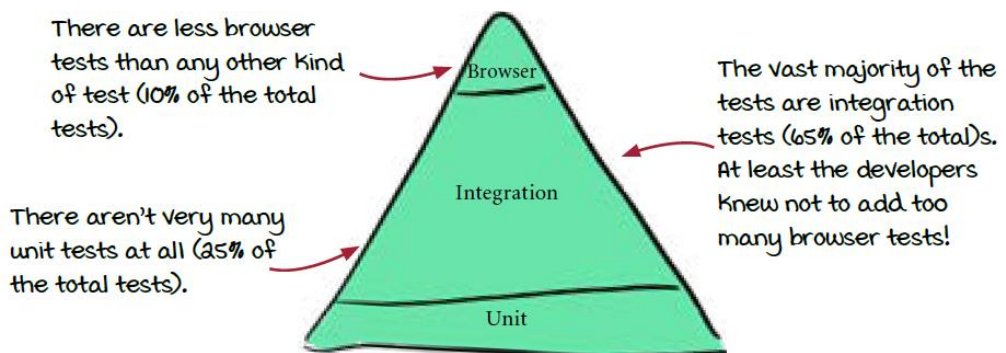
Sridhar has improved the situation, but his change has had virtually no effect on the integration and browser tests - they are just as slow as ever and developers still have to wait until the next morning after pushing their changes to find out the results.

For his next improvement, Sridhar is once again going back to the testing pyramid. When he last looked at it, he was thinking about the relative speed of each set of tests. But now he's going to look at the relative distribution of tests.

The pyramid also gives us guidelines as to how many (literally the quantity) tests of each type we want to aim for. Why is that? Because as you go up the pyramid, the tests are slower. (And also harder to maintain but that's a story for another book!)



Sridhar counts up the tests in the Dog Picture Website suite so he can compare their pyramid to the ideal. The Dog Picture Website looks more like this:



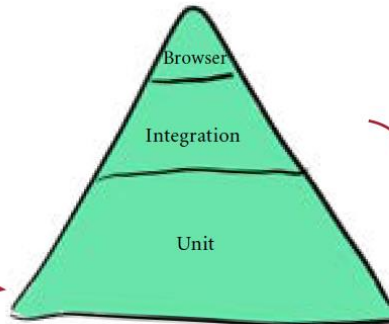
## 6.10 Changing the pyramid

Why is Sridhar looking at ratios in the pyramid? Because he knows that the ratios in this pyramid are not set in stone. Not only is it possible to change these ratios, but changing the ratios can lead to faster test suites.

Let's look again at the goals he set around execution time:

The entire test suite should run in less than 30 minutes

Running just the unit tests should take less than 1 minute



Without the browser tests, the tests should run in less than 5 minutes

Sridhar wants the integration and unit tests to run in less than 5 minutes. Currently the integration tests are 65% of the total number of tests. The rest are 10% browser tests and 25% unit tests. Given that integration tests are slower than unit tests, imagine what a difference it could make if the ratio was changed (assuming the same total number of tests): if the integration tests were only 20% of the total number of tests, and the unit tests were instead 70%. This would mean removing about 2/3 of the existing (slow) integration tests, and replacing them with (faster) unit tests - which would immediately impact the overall execution time.

With the ultimate goal of adjusting the ratios in order to speed up the test suite overall, Sridhar sets some new goals:

- Increase the % of unit tests from 25% to 70%
- Decrease the % of integration tests from 65% to 20%
- Keep the % of browser tests at 10%

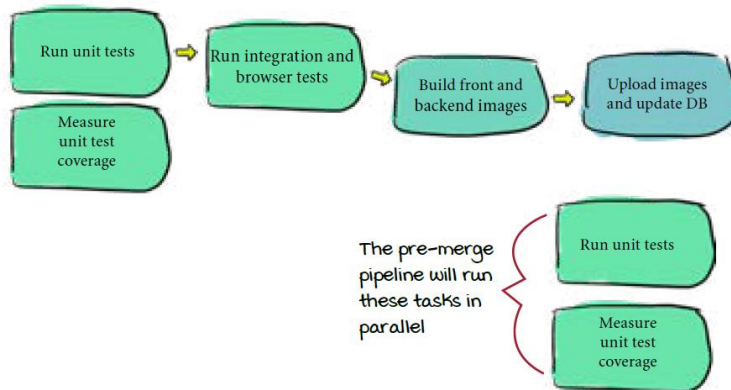
## 6.11 Safely adjusting tests

Sridhar wants to make some changes in the ratio of unit tests to integration tests. He wants to:

- Increase the % of unit tests from 25% to 70%
- Decrease the % of integration tests from 65% to 20%

This means he needs to increase the number of unit tests, while decreasing the number of integration tests. How will he do this safely, and where can he even start?

Sridhar noticed that Dog Picture Website's pipeline doesn't include any concept of test coverage measurement. The pipeline runs tests, then builds and deploys, but at no point does it measure the code coverage provided by any of the tests. The very first change he's going to make is to add test coverage measurement into this pipeline, in parallel to running the tests:



Measuring coverage will also run the unit tests, so sometimes you'll see these combined as one task. (If the unit tests task fails, the coverage measurement task will probably fail too!)

Since the coverage task is just as fast as the unit test test, he's able to add it to the pipeline that runs before changes are merged also.



### QUESTION

**Q** Wait! Where's the linting? I read chapter 4 and I know linting is important too, shouldn't Sridhar be adding linting too?

**A** Totally agree - and that's probably going to be Sridhar's next step once he deals with these tests, but he can only tackle one problem at a time! In chapter 2 you can see an overview of all the elements a CD pipeline should have, including linting.

## 6.12 Test Coverage

Sridhar decided the first step toward safely adjusting the ratio of unit test to integration tests was to start measuring test coverage. What is test coverage measurement and why is it important?

Test coverage is a way of evaluating how effectively your tests exercise the code they are testing. Specifically, test coverage reports will tell you, line by line, which code under test is being used by tests, and which isn't.

For example, Dog Picture Website has this unit test for their search by tag logic:

```
def test_search_by_tag(self):
    search = _new_search()
    results = search.by_tags(['fluffy'])
    self.assertDogResultsEqual(results, 'fluffy', [Dog('sheldon')])
```

This test is testing the method `by_tags` on the `Search` object, which looks like this:

```
def by_tags(self, tags):
    try:
        query = build_query_from_tags(tags)
    except EmptyQuery:
        raise InvalidSearch()
    result = self._db.query(query)
    return result
```

Test coverage measurement will run the test `test_search_by_tag` and observe which lines of code in `by_tags` are executing, producing a report about the percentage of lines covered. The coverage for `by_tags` by `test_search_by_tag` looks this, where yellow indicates lines that are executed by the test and red indicates lines that aren't:

```
def by_tags(self, tags):
    try:
        query = build_query_from_tags(tags)
    except EmptyQuery:
        raise InvalidSearch()
    result = self._db.query(query)
    return result
```

It's reasonable that the test above doesn't exercise any error conditions, good unit testing practice would leave that for another test - but in this case `test_search_by_tag` is the only unit test for `by_tags`. So those lines are not covered by any test at all. For this method, the test coverage is 3 out of 5 lines, or 60%.

### *Coverage critiera*

The above example uses a coverage criteria called *statement coverage*, which evaluates each statement to see if it has been executed or not. There are also other, more fine grained criteria that can be used such as *condition coverage*: if an if statement has multiple conditions, *statement coverage* would consider it covered if it's hit at all, but *condition coverage* coverage would require that every condition be explored fully. In this chapter we'll stick to *statement coverage* which is a great go-to.

## 6.13 Enforcing test coverage

It's important to remember that while Sridhar is making these changes, people are still working and submitting features! People are submitting more features (and bug fixes), and sometimes (hopefully most of the time!) tests as well. This means that even as Sridhar looks at the test coverage, it could be going down!

But fortunately Sridhar knows a way that not only stop this from happening, he can actually use this to help his quest to increase the number of unit tests.

Before going any further, Sridhar is going to update the coverage measurement task to fail the pipeline if the coverage goes down. From the moment that he introduces this change onward, he can be confident that the test coverage in the code base will at the very least not go down, but ideally go up as well.

(Besides helping the overall problem, this is a great way to share the load such that Sridhar isn't the only one doing all the work!)

He updates the task that runs the test coverage to run this script:

```
# when the pipeline runs, it will pass to this script
# paths the files that changed in the pull request
paths_to_changes = get_arguments()

# measure the code coverage for the files that were changed
coverage = measure_coverage(paths_to_changes)

# measure the coverage of the files before the changes;
# this
# could be by retrieving the values from storage somewhere,
# or it could be as simple as running the coverage again
# against the same files in trunk (i.e. before the changes)
prev_coverage = get_previous_coverage(paths_to_changes)

# compare the coverage with the changes to the previous
# coverage
if coverage < prev_coverage:
    # the changes should not be merged if they decrease
    # coverage
    fail('coverage reduced from {} to
        {}'.format(prev_coverage, coverage))
```

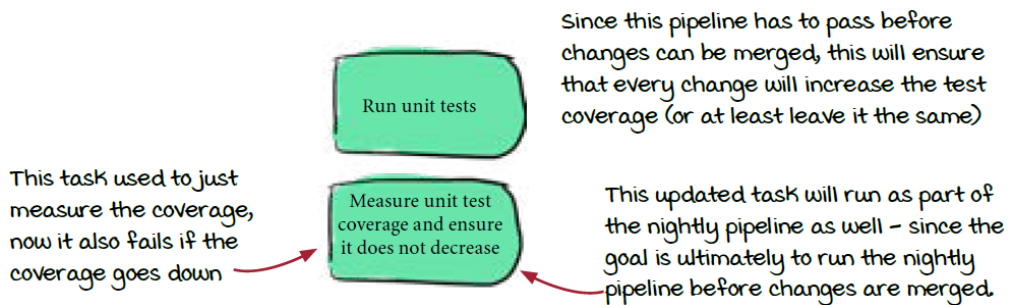
Does this sound familiar? You might recognize this as a very similar approach to the one Becky took in chapter 4 with linting! Measuring linting and measuring coverage have a lot in common!

You might recognize this as a variation on the linting script that Becky created in chapter 4!

## 6.14 Test coverage in the pipeline

By introducing this script into the pre-merge pipeline, Sridhar has triaged the existing coverage problem: the problem was that folks weren't being fastidious about how they introduced unit test. By adding automation to measure coverage and block them, they can make more informed decisions about what they cover and what they don't.

With Sridhar updating the unit test coverage task to actually enforce requirements on test coverage, the pre-merge pipeline looks like this:



It's a very subtle change from the previous iteration, but now Sridhar can continue on with his work and be sure that the features and bug fixes being merged as he works are going to either increase the coverage, or in the worst case, leave it the same.

### *Do I need to build this myself?*

It depends! For most languages, there are a lot of existing tools you can make use of to measure your coverage, and even store and report on it over time. Many folks choose to write their own regardless, because it's not very hard to implement and you have slightly more control over the behavior. You'll need to investigate the tools available and decide for yourself.

## 6.15 Moving tests in the pyramid with coverage

At this point, the number of unit tests is likely to start to steadily increase, even without any further intervention, because Sridhar has made it a requirement to include unit tests alongside the changes that the engineers are making.

Will this be enough for him to achieve his goals? Remember his goals are:

- Increase the % of unit tests from 25% to 70%
- Decrease the % of integration tests from 65% to 20%

Over time the ratios will likely trend in these directions, but not fast enough to make the dramatic kinds of changes Sridhar is looking for. Sridhar is going to need to write additional unit tests and probably also remove existing integration tests. How will he know which to add and which to remove?

Sridhar looks at the code coverage reports, finds the code with the lowest coverage percentages and looks at which lines are not covered. For example he looks at the coverage of the `by_tags` function we saw a few pages ago.

```
def by_tags(self, tags):
```

```
    try:
```

```
        query = build_query_from_tags(tags)
```

```
    except EmptyQuery:
```

```
        raise InvalidSearch()
```

```
    result = self._db.query(query)
```

```
    return result
```

The error case of having an empty query is not covered by unit tests. So Sridhar knows that this is a place where he can add a unit test. Additionally, if he can find an integration test that covers the same logic, he can potentially delete it. So he goes looking through the integration tests and find a test called `test_invalid_queries`. This test creates an instance of the running backend service (this is what all the integration tests do), then makes invalid queries, and ensures that they fail. Looking at this test, Sridhar realizes he can cover all of the invalid query test cases with unit tests. He writes the unit tests, which execute in less than a second, and is able to delete `test_invalid_queries`, which took around 20 seconds or more, and still feel confident that the test suite would catch the same errors that it did before the change.



### QUESTION

**Q**  
**A**

Should I measure coverage for my integration and end to end tests?

To get a complete idea of your test suite coverage, you may be tempted to measure coverage for your integration and end to end tests. This is sometimes possible, usually requiring the systems under test to be built with extra debug information that can be used to measure code coverage while these higher level tests execute. You may find this useful, however it's usually something you have to build yourself, and might give you a false sense of confidence; i.e. your best bet will always be high unit test coverage, so that metric is important in isolation, and you might miss it if you only look at the total test suite coverage as a whole.



## 6.16 What to move down the pyramid?

In order to continue to increase the percentage of unit tests, Sridhar applies this pattern to the test suite:

1. He looks for gaps in unit test coverage; i.e. literally lines of code that are not covered. He looks at the packages and files with the lowest percentages first in order to maximize his impact.
2. For the code he finds that isn't covered, he adds unit tests that cover those lines
3. He looks through the slower tests, specifically in this case the integration tests, to find any tests that cover the logic now covered by the unit tests, and updates or deletes them.

By doing this he is able to both dramatically increase the amount of unit tests and reduce the amount of integration tests, that is to increase the number of fast tests and decrease the number of slow tests.

Lastly he audits the integration tests to look for duplicate coverage: for every integration tests he asks these questions:

1. Is this case covered in the unit tests?
2. What would cause this test case to fail when the unit tests pass?

If the case is covered in the unit tests already (1), and if there isn't anything (that isn't covered somewhere else) that would cause the integration test to fail when the unit tests pass (2), it is safe to delete the integration test.



### QUESTION

**Q**

Hold on, surely I'm going to lose some information if I do this! Aren't my integration tests better than my unit tests? I've seen the memes, unit tests aren't enough.

**A**

You're right! The question is: how many integration tests do you need? The purpose of the integration tests is to make sure that all the individual units are wired together correctly. If you test the individual units, and then you test that the units are connected together correctly, you've covered nearly everything. At this point it becomes a cost benefit tradeoff: is it worth the cost of running and maintaining integration tests that cover the same ground as unit tests, on the off chance that they might catch a corner case you missed? The answer depends on what you're working on. If people's lives are at stake, the answer may be yes; it's important to make the right tradeoff for your software.



### It's your turn

Sridhar has found that the Search class has very low coverage in general, and he's working his way through the reports to increase it. Working his way through the reports, he looks at the coverage for the function `from_favorited_search` and sees:

```
def from_favorited_search(self, favorite):
    try:
        cached_result = self.cache.get_result(favorite.query())
    except CacheError:
        cached_result = None
    if cached_result is None:
        result = self.db.query(favorite.query())
    else:
        result = cached_result.result()
    return result
```

He looks for the integration tests that cover the favorited search behavior and finds these tests:

```
test_favorited_search_many_results
test_favorited_search_no_results
test_favorited_search_cache_connection_error
test_favorited_search_many_results_cached
test_favorited_search_no_results_cached
```

Which integration tests should Sridhar consider removing? What unit tests might he add?



### ANSWERS

This looks like a classic scenario where the integration tests are doing all the heavy lifting. The unit tests are covering only one path: the path where there is no cached result and there are no errors, and the integration tests are trying to cover everything. Sridhar's plan is basically to invert this: instead of covering one happy path with unit tests, and handling all the other cases with integration tests, he'll replace all of the above integration tests with `test_favorited_search`, and he'll add unit tests to cover all of the integration test cases above.

## 6.17 Legacy tests and FUD

It can feel scary to make changes to, or even remove, tests that have been around for a long time! This is a place where we can often encounter FUD: Fear, Uncertainty and Doubt.

If we listen to the FUD, we might decide it's too dangerous to make changes to the existing test suites: there are too many tests, it's too hard to tell what they're testing, and we become afraid of being the person who removed the test that it turned out was holding the whole thing up.

If you find yourself thinking this way, it's worth taking a moment to think ultimately about what FUD really is, and where it comes from. It's ultimately all about the F: fear. It's fear that we might do something wrong, or make things worse, and it holds us back from making changes.

Then, think about why we have all the tests we do: the tests are meant to empower us, to make us feel confident that we can make changes that do what we want them to, without fear.

FUD is the very opposite of what our tests are meant to do for us. Our tests are meant to give us confidence, and FUD takes that confidence away.

Don't let FUD hold you back! When you hear FUD whispering to you that it's too dangerous to make any changes, you can counter it with cold hard facts. Remember what tests are: they are nothing more or less than a codification of how the test author thought the system was supposed to behave. Nothing more or less than that. They aren't even the system itself! Instead of giving in to the fear, take a deep breath and ask yourself: do I understand what this test is trying to do? If not, then take the time to read it and understand it. If you understand it, then consider yourself empowered to make changes. If you don't make them, maybe no one will, and the sense of FUD that people feel about the test suite will only grow over time.

In general, working from a fear based mindset, and giving into FUD, will prevent you from trying anything new, and that will prevent you from improving, and if you don't improve your test suite over time I can guarantee you that it will only get worse.

Just say no to FUD!



### TAKEAWAY

When dealing with slow tests suites, looking at the test suite through the lens of the testing pyramid can help you focus on where things are going wrong. If your pyramid is too top heavy (a common problem!) you can use test coverage to immediately start to improve your ratios, and point you in the direction of what tests can be replaced with faster and easier to maintain unit tests.

## 6.18 Running tests in parallel

After working hard on the integration and unit tests, Sridhar has made as much improvement as he thinks he can for now and he met his the goals he set for their relative quantities:

- He has increased the % of unit tests from 25% to 72% (his goal was 70%)
- He has decreased the % of integration tests from 65% to 21% (his goal was 20%)

The unit tests still run in less than a minute, but even meeting the goals above, the integration tests still take around 35 minutes to run. His overall goal was for the integration and unit tests together to run in less than five minutes. Even though he has improved the overall time (shaving more than 1 hour from the total), these tests are still slower than he wants them to be. He'd like to be able to include them in the pre-merge tests, and at 35 minutes, this might be almost reasonable, but he has trick up his sleeve that will let him improve this substantially before he adds them.

He's going to run the integration tests in parallel! Most test suites will by default run tests one at a time. For example, here are some of the integration tests which are left after Sridhar has reduced their number, and their average execution time:

1. `test_search_query` (20 seconds)
2. `test_view_latest_dog_pics` (10 seconds)
3. `test_log_in` (20 seconds)
4. `test_unauthorized_edit` (10 seconds)
5. `test_picture_upload` (30 seconds)

Running these tests one at a time takes  $20 + 10 + 20 + 10 + 30 = 90$  seconds on average. Instead, Sridhar updates the integration test task to run these tests in parallel, running as many of them as possible at once individually. In most cases, this means running one test at a time per CPU core. On an 8 core machine, the above five tests can easily run in parallel, meaning that executing them all will only take as long as the longest test: 30 seconds, instead of the entire 90 seconds.

After his cleanup, Dog Picture Website has 116 integration tests. Running at an average of 18 seconds each, one at a time, they take about 35 minutes to run. Running them in parallel on an 8 core machine means that 8 tests can execute at once, and the entire suite can execute in approximately  $1/8$  of the time, or about 4 and a half minutes! By running the integration tests in parallel, Sridhar is able to finally meet his goal of being able to run the unit + integration tests in less than 5 minutes.

If one test runs waaaaay longer than the others, you'll still be held hostage to this test; for example if one test took 30 minutes on it's own, parallelization isn't going to help and the solution is going to be to fix the test itself.

## 6.19 When can tests run in parallel?

Can any tests be run in parallel? Not exactly. In order for tests to be able to run in parallel, they need to meet these criteria:

- The tests must not depend on each other
- The tests must be able to run in any order
- The tests must not interfere with each other (e.g. sharing common memory)

It is good practice to write tests that do not depend on or interfere with each other in any way, so if you are writing good tests, then you might not have any trouble at all making them run in parallel.

The trickiest requirement is probably making sure that tests do not interfere with each other. This can easily happen by accident, especially when testing code that makes use of any kind of global storage. With a little finesse, you'll be able to find ways to fix your tests so that they can be totally isolated, and then the result will likely be better code overall (i.e. code that is less coupled and more cohesive).

When Sridhar updated the Dog Picture Website test suite to run in parallel, he found a few tests that interfered with each other and had to be updated, but once he made those fixes, he was able to run both the unit and integration tests in less than five minutes.

### *Running unit tests in parallel is a smell*

Remember the goal of unit tests is to test functionality in isolation and to be FAST, as in on the order of seconds or faster. If your unit tests are taking minutes or longer, making you tempted to speed them up by running them in parallel, this is a sign that your unit tests are doing too much and are likely integration or system tests - there is a good chance you are missing unit tests entirely.



### QUESTION

**Q**  
**A**

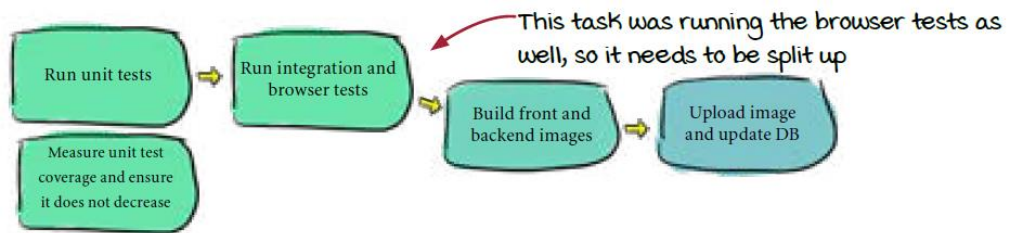
Do I need to build this “tests in parallel” functionality myself too?

Probably not! This is such a common way of optimizing test execution that most languages will provide you with a way to run your tests in parallel, either out of the box or with the help of common libraries. For example, you can run tests in parallel with Python by using a library such as `testtools` or an extension to the popular `pytest` library, and in Golang you get the functionality out of the box via the ability to mark a test as parallelizable when you write it with `t.Parallel()`. Find the relevant information for your language by looking up documentation on running tests in parallel or concurrently.

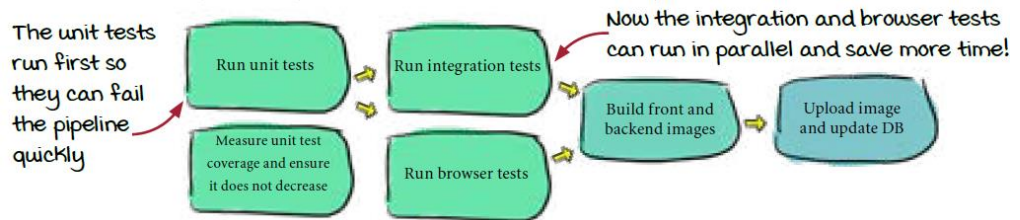
## 6.20 Updating the pipelines

Now that Sridhar had met his goal of running both the unit tests and the integration tests in less than 5 minutes, he could add the integration tests to the pre merge pipeline and engineers would get feedback on both the unit and integration tests before their changes merged.

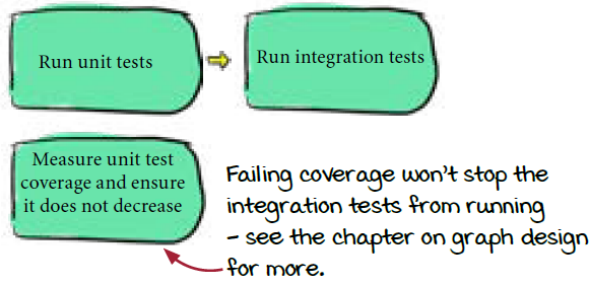
This meant he had to make some tweaks to the set of tasks in the Dog Picture Website Pipeline - there was still one task that ran the integration and browser tests together.



Fortunately the tests were already setup well for this change. You may recall that the browser tests are already in a separate folder called `tests/browser` and when Sridhar updated the pipeline to run the unit tests first, he separated the integration tests and put them into a folder called `tests/integration`. This makes it easy to take the final step of running the integration and browser tests separately:



And then Sridhar can add the integration test task to the pre merge pipeline. The pipeline will fail quickly if there is a problem with the unit tests; and the entire thing will run in less than five minutes.



***Why not run all the test tasks in parallel?***

Sridhar has assumed that if the unit tests, which take seconds to execute, fail, there's no point in running the integration tests because they'll probably fail, but both are good options. See the chapter on graph design for more.

## 6.21 Still too slow!

After working hard on the integration and unit tests, Sridhar has made as much improvement as he thinks he can for now and he met his the goals he set for their relative quantities:

- He has increased the % of unit tests from 25% to 72% (his goal was 70%)
- He has decreased the % of integration tests from 65% to 21% (his goal was 20%)

Is he done? He steps back and looks at his overall goals:

- **Tests should run on every change, before the change gets pushed** - he's almost there, now the unit and integration tests run, but not the browser tests
- **The entire test suite should run in an average of 30 minutes or less** - Sridhar has reduced the execution time of the integration tests - they used to take 35 minutes and now take around 5. The entire suite used to take 2 hours and 35 minutes and now is down to just over 2 hours. This is a big improvement, but Sridhar still hasn't met his goal.
- **The integration and unit tests should run in less than five minutes** - done!
- **The unit tests should run in less than one minute** - done!

The entire test suite is running in an average of 2 hours and 5 minutes:

- Unit tests: Less than 1 minute
- Integration tests: Around 5 minute
- Browser tests: The other 2 hours

The last remaining problem is the browser tests. All along, the browser tests have been the slowest part of the test suite. At an average runtime of 2 hours, no matter how much Sridhar optimizes the rest of the test suite, if he doesn't do something about the browser tests, it's always going to take more than 2 hours.

Can Sridhar take a similar approach and remove browser tests, replacing them with integration and unit tests? This is definitely an option, but when Sridhar looks at the suite of browser tests, he can't find any candidates to remove! The tests are already very focused and well factored, and at only 10% of the total test suite (with around 50 individual tests), the number of browser tests is quite reasonable.



## 6.22 Test sharding aka parallel++

Sridhar is stuck with the browser tests as they are, and they take about 2 hours to run. Does this mean he has to say goodbye to his goals of running the entire suite on every change in less than 30 minutes?

Fortunately not! Because Sridhar has one last trick up his sleeve: **sharding**. Sharding is a technique that is very similar to running tests in parallel, but increases the number of tests that can be executed at once *by parallelizing them across multiple machines*.

Right now, all of the 50 browser tests run on one machine, one at a time. Each test runs in an average of about 2 and a half minutes. Sridhar first tries running the tests in parallel, but they are so CPU and memory intensive that the gains are negligible (and in some cases the tests steal resources from each other, effectively slowing down). One executing machine can really only run one test at a time.

By sharding the test execution, Sridhar will divide up the set of browser tests so that he uses multiple machines which will each execute a subset of the tests, one at a time, allowing him to decrease the overall execution time.



### QUESTION

**Q** What if Sridhar beefed up the machines? Maybe then he could get away with running the tests in parallel on one machine?

**A** This might help, but as you probably know, machines are getting more and more powerful all the time - and we respond by creating more complex software and more complex tests! So while using more powerful machines might help Sridhar here, we're going to look at what you can do when this isn't an option; and we're not going to dive into the specific CPU and memory capacity of the machines he's using because what seems powerful today will seem trivial tomorrow!



### VOCAB TIME

We're referring to parallelizing tests across multiple machines as sharding, but you will find different terminology used by CD systems. Some systems will call this test splitting, and others will simply also refer to this as running tests in parallel, where "in parallel" means across multiple machines as opposed to how in this chapter we've used "in parallel" to refer to running multiple tests on one machine. Regardless you can think of sharding as the same basic idea as test parallelization, but across multiple machines.

## 6.23 How to shard

Sharding test execution allows you to take a suite of long running tests, and execute it faster by running it across more hardware, i.e. several executing machines instead of just one. But how does it actually work? You might be imagining a complex system requiring some kind of worker nodes co-ordinating with a central controller, but don't worry, it can be much much simpler than that!

The basic idea is that you have multiple shards, and each is instructed to run a subset of the tests. There are a few different ways you can decide which tests to run on which shard. In increasing order of complexity:

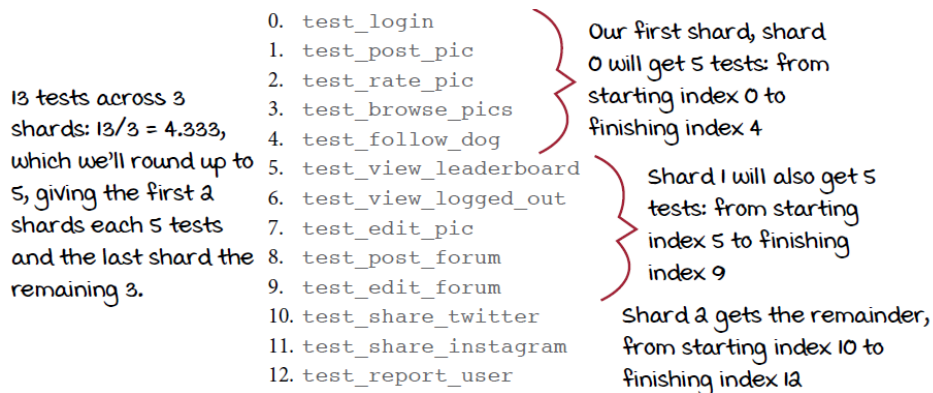
1. Run tests in a deterministic order and assign each shard a set of indexes to run
2. Assign each shard an explicit set of tests to run (for example, by name)
3. Keep track of attributes of tests from previous test runs (for example, how long it takes each to run) and use those attributes to distribute tests across shards (probably using their names like in option 2)



### VOCAB TIME

Each machine available to execute a subset of your tests is referred to as a **shard**.

Let's get a better handle on test sharding by looking at option 1 in a bit more detail. For example, imagine sharding the following 13 tests across 3 executing machines:



We can shard these tests using the first method by running a subset of the above tests on each of our 3 shards. If we're using python, one way to do this is with the python library `pytest-shard`:

```
pytest --shard-id=$SHARD_ID --num-shards=$NUM_SHARDS
```

For example, shard 1 would run:

```
pytest --shard-id=1 --num-shards=3
```

## 6.24 More complex sharding

Sharding by index is fairly straight forward, but what about outliers? Sridhar's browser tests run in an average of 2.5 minutes, but what if some of them take waaaaay longer?

This is where more complex sharding schemes come in handy, for example the third option we listed: keeping track of attributes of tests from previous test runs and use those attributes to distribute tests across shards using their names.

In order to do this, you need to store timing data for tests as you execute them. For example, take the 13 tests we ran in the last example and imagine we'd been storing how many minutes each had taken to run across the last 3 runs:

0) test_login (1.5, 1.7, 1.6)	Average = 1.6 minutes
1) test_post_pic (3, 3.1, 3.2)	Average = 3.1 minutes
2) test_rate_pic (0.8, 0.9, 0.7)	Average = 0.8 minutes
3) test_browse_pics (2, 2, 2)	Average = 2.0 minutes
4) test_follow_dog (0.8, 0.8, 0.8)	Average = 0.8 minutes
5) test_view_leaderboard (1.8, 2.0, 1.9)	Average = 1.9 minutes
6) test_view_logged_out (1.7, 2.1, 1.9)	Average = 1.9 minutes
7) test_edit_pic (2.1, 2.6, 2.2)	Average = 2.3 minutes
8) test_post_forum (1.8, 1.9, 1.7)	Average = 1.8 minutes
9) test_edit_forum (1.6, 1.5, 1.7)	Average = 1.6 minutes
10) test_share_twitter (2.1, 1.9, 2.0)	Average = 2.0 minutes
11) test_share_instagram (2.0, 1.9, 2.1)	Average = 2.0 minutes
12) test_report_user (1.3, 1.2, 1.1)	Average = 1.2 minutes

To determine the sharding for the next run, you'd look at the average timing data and create groupings such that each of the 3 shards would execute the test in roughly the same amount of time.

We're going to skip going into the details of this algorithm (though it does make for a fun and surprisnly practical interview question!). If you want this kind of sharding, it's possible that you might need to build it yourself, but you also might find that the CD system you're using (or tools in your language) will do it for you. For example, the CD system CircleCI lets you do this by feedng the names of your tests into a language agnostic splitting command:

```
circleci tests split --split-by=timings
```

**Shard 0: 7.4 minutes**  
test\_edit\_pic (2.3)  
test\_share\_instagram (2.0)  
test\_post\_forum (1.8)  
test\_report\_user (1.2)  
test\_follow\_dog (0.8)

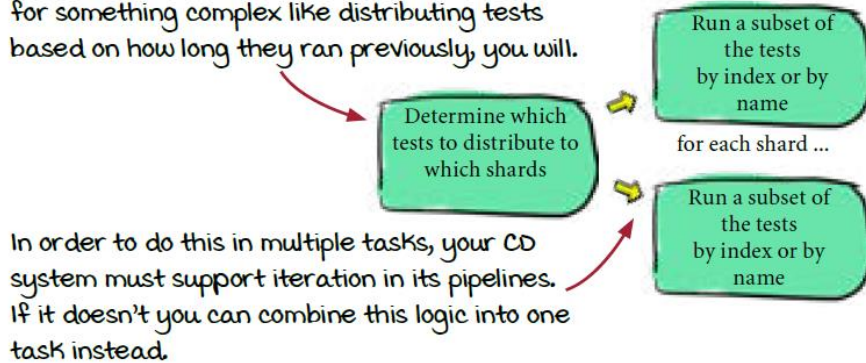
**Shard 1: 8.1 minutes**  
test\_post\_pic (3.1)  
test\_view\_leaderboard (1.9)  
test\_login (1.6)  
test\_rate\_pic (0.8)

**Shard 2: 7.5 minutes**  
test\_browse\_pics (2.0)  
test\_share\_twitter (2.0)  
test\_view\_logged\_out: (1.9)  
test\_edit\_forum: (1.6)

## 6.25 Sharded pipeline

You may decide to do all of the steps for sharding within one task of your pipeline, or if your CD system supports it, you might break this out into multiple tasks.

If you're doing something simple like sharding by index, you probably don't need this first step, but for something complex like distributing tests based on how long they ran previously, you will.



In order to support being run with sharding, a set of tests must meet the following requirements:

- The tests must not depend on each other.
- The tests must not interfere with each other; if the tests share resources, for example all connecting to the same instance of a dependency, they may conflict with each other (or maybe not - the easiest way to find out is to try).
- If you want to distribute your tests by index, it must be possible to run the tests in a deterministic order so that the test represented by an index must be consistent across all shards.

### *If running unit tests in parallel is a smell, then sharding them is a stink!*

As mentioned earlier, if your unit tests are slow enough that you need to run them in parallel for them to run in a reasonable length of time, then that's a smell that something is not quite right with your unit tests (they are probably doing too much). If they are SO SLOW that you want to shard them, then I can say with 99.99999% certainty that what you have are not unit tests, and your code base would be well served by replacing some of these integration/system tests in disguise with unit tests. *p.s. I hearby propose calling really bad code smells "code stinks"*

## 6.26 Sharding the browser tests

Sridhar is going to solve the problem of the slow browser tests by applying sharding! The overall goal Sridhar is aiming for is:

- **The entire test suite should run in an average of 30 minutes or less**

The unit and the integration tests take an average of 5 minutes in total, so Sridhar needs to get the browser tests to run in about 25 minutes.

The browser tests take an average of 2 and a half minutes, and there are 50 of them. The time each test tasks to execute is fairly uniform, so Sridhar decides to use the simpler route and shard by index. How many shards does he need to meet his goal?

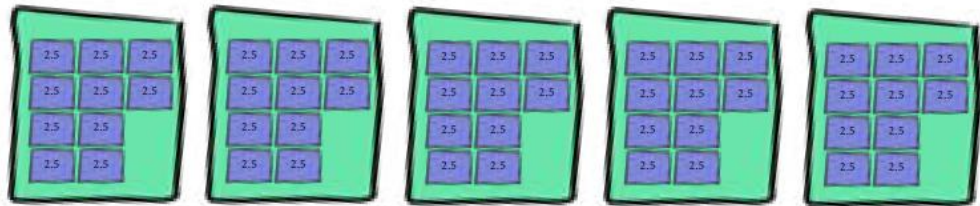


Since the goal is to complete all the tests in 25 minutes, this means each shard can run for up to 25 minutes. How many browser tests can run in 25 minutes?

If they each take an average of 2.5 minutes,  $25 \text{ minutes} / 2.5 \text{ minutes} = 10$ . In 25 minutes, one shard can run 10 tests.



With 50 tests in total, and each shard able to run 10 tests in 25 minutes, Sridhar needs  $50/10 = 5$  shards.



Using 5 shards will meet his goal, but he knows they have enough hardware available that he can be even more generous, and he decides to allocate 7 shards for the browser tests.



With 7 shards, each shard will need to run  $50/7$  tests; the shards with the most will run the ceiling of  $50/7 = 8$  tests. 8 tests at an average of 2.5 minutes will complete in 20 minutes. This lets Sridhar slightly beat his goal of 25 minutes, and gives everyone a bit more room to add more tests, before more shards will need to be added.

## 6.27 Sharding in the pipeline

Simple index based sharding will work for the Browser tests, so all the Shridar has to to is add tasks that run in parallel, one for each shard, and have each use pytest-shard to run their subset of the tests.

His sharded browser test tasks will run this python script, using python to call pytest:

```
# when the pipeline runs, it will pass to this script
# the index of the the shard and the total number of shards
# as arguments
shard_index, num_shards, path_to_tests = get_arguments()
# we'll invoke pytest as command to run the correct set of tests
# for this shard
run_command(
    "pytest --shard-id={} --num-shards={} {}".format(
        shard_index, num_shards, path_to_tests
    )
)
```

To add this script to his pipeline, all he has to do is add a set of tasks that run in parallel, in his case 7, one for each of the 7 shards.

Run pytest-shard  
for shard index 0

Run pytest-shard  
for shard index 1

Run pytest-shard  
for shard index 2

Run pytest-shard  
for shard index 3

Run pytest-shard  
for shard index 4

Run pytest-shard  
for shard index 5

Run pytest-shard  
for shard index 6

Does he need to hard code 7 individual tasks into his pipeline to make this happen? It depends on the features of the CD system he's using. Most will provide a way to parallelize tasks, allowing you to specify how many instances of the task you'd like to run, and then providing as arguments (often environment variables) information to the running tasks on how many instances are running in total and which instance they are.

For example, using GitHub actions you can use a matrix strategy to run the same job multiple times:

```
jobs:  #A
  tests:
    strategy:
      fail-fast: false
      matrix:
        total_shards: [7]
        shard_indexes: [0, 1, 2, 3, 4, 5, 6]
```

**#A** GitHub actions uses “jobs” to refer to what this book calls “tasks”

With the above configuration, the tests job would be run 7 times, and steps in each job can be provided with the following context variables so they'll know how many shards there are in total and which shard they are running as:

```
{{ matrix.total_shards }}
{{ matrix.shard_indexes }}  #A
```

**#A** These matrix option names are arbitrary; see the GitHub Actions `jobs.<job_id>.strategy.matrix` documentation for more

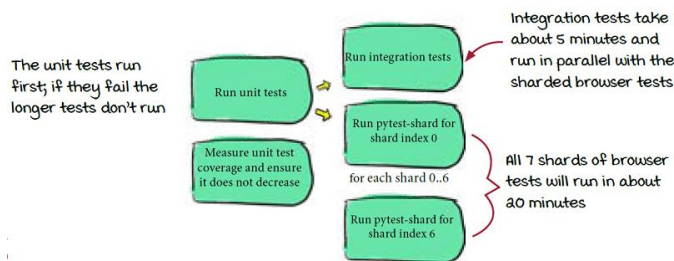


## 6.28 Dog Picture Website's pipelines

Now that Sridhar has met his goal of running the browser tests in 25 minutes - in fact, in 20 minutes! - he can combine all the tests together and the entire suite can run in an average of 30 minutes or less. This means he can go back to his last goal:

- **Tests should run on every change, before the change gets pushed**

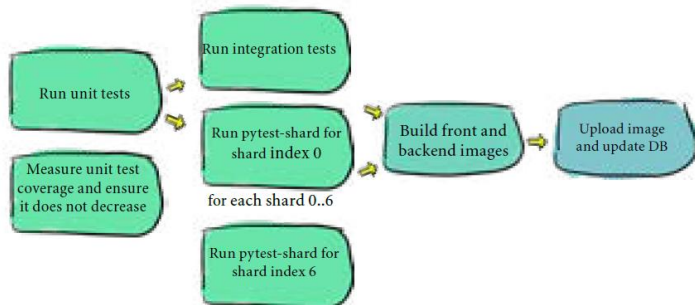
Sridhar adds the browser tests to the pre merge pipeline, running them in parallel with the integration tests. The pre merge pipeline can now run all of the tests and it will take only the length of the sharded browser tests (20 minutes) + the unit tests (less than 1 minute).



*Why is the pre merge pipeline different from the nightly pipeline?*

That's a good question! It doesn't have to be - see the chapter on graph design for more about the tradeoffs.

Sridhar makes the same updates to the nightly release pipeline as well so that it gets the same speed boost.



### Takeaway

Running tests in parallel will increase your hardware footprint, but it will save you one another invaluable asset: time! When tests are slow, first optimize the test distribution by leveraging unit tests, then leverage parallelization and sharding if needed.

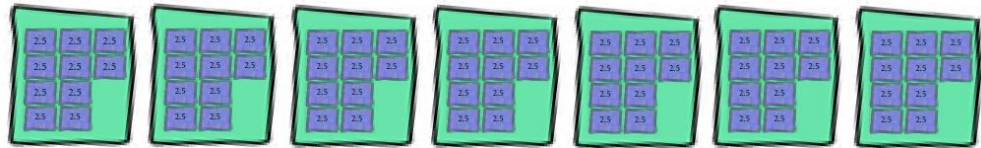




### Noodle on it

Sridhar needed 5 shards to run the 50 tests in 25 minutes or less, and he added an extra 2 shards for a total of 7, speeding up the test execution time and adding some buffer for future tests. But what if the number of tests keeps growing, does that mean adding more and more shards? Will that work?

Once the number of browser tests increases from 50 to 70, each shard of the 7 shards will be running 10 tests, and the overall execution time will be 25 minutes.



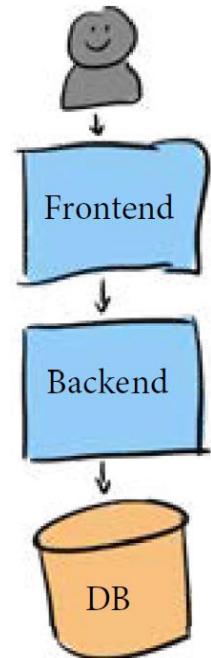
This means if any more tests are added, the browser tests will take more than 25 minutes to run and more shards will need to be added. Does this mean they'll have to keep adding shards indefinitely? Won't that eventually be too much?

That could happen; you may remember that the architecture of Dog Picture Website is quite monolithic:

If Dog Picture Website continues to grow its feature base, they will likely want to start dividing up responsibilities of the "backend service" into separate services - which can each have their own tests suites.

This will mean that when something is changed, only the tests that are related to that change can be run, instead of needing to run absolutely everything. This kind of division of responsibilities will probably be required in order to match the growth of the company as well, i.e. as more people are added, they will need to be divided into effective teams which each have independent areas of ownership.

Food for thought: fast forward to the future, where Dog Picture Website is made up of multiple services, each with their own set of end to end tests. Is running each set separately enough to be confident that the entire system works? Should all of the tests be run together before a release in order to be certain? The answer is: it depends, but remember, you can never be 100% certain. The key is to make the tradeoffs that work for your project.



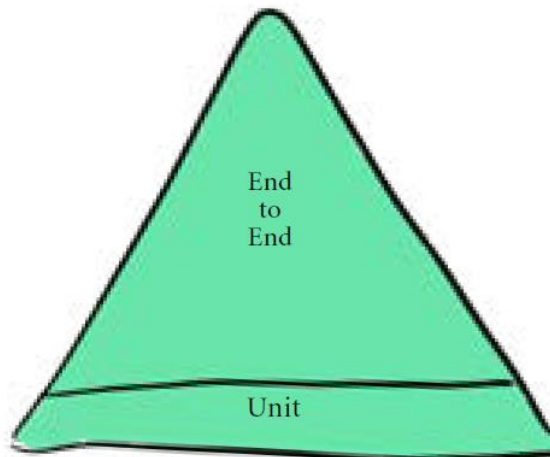


### It's your turn

Dog Picture Website and Cat Picture Website actually share a common competitor: the up and coming Bird Picture Website. Bird Picture Website is actually dealing with a similar problem around slow tests, but their situation is a bit different. Their entire test suite runs in about 3 hours, but unlike Dog Picture Website, they run this entire suite for every pull request. This means that when their engineers are ready to submit changes, they open up a pull request, and then leave it, often until the next day, to wait for the tests to run. One advantage to this approach is that they catch a lot of problems before they get merged, but it means that engineers will often spend days trying to get their changes merged (sometimes called "wrestling with the tests").

The test suite Bird Picture website uses has the following distribution:

- 10% unit tests
- No integration tests
- 90% end to end tests



The unit tests cover 34% of the codebase, and they take 20 minutes to run. Given the above, what are some good next steps for Bird Picture Website to go about speeding up their test suite?



## ANSWERS

A few things stand out immediately about Bird Picture Website's test suites:

- The tests they are calling "unit tests" are quite slow for unit tests; ideally they would run in a couple of minutes max, if not in seconds. There is a good chance there are actually more like integration tests.
- The unit test (or maybe "integration test") coverage is quite low
- There are a LOT of end to end tests in comparison to the amount of unit tests; it could be that there just aren't very many tests in general, but there's also a good chance that Bird Picture Website is relying too much on these end to end tests.

Based on this information, there are a few things that the folks at Bird Picture website could do:

- Sort through the slow unit tests; if any of these are actually unit tests (i.e. running in seconds or less), run those separately from the other slower tests (which are actually integration tests). These unit test can be run quickly first and give an immediate signal.
- Measure the coverage of these fast unit tests - it will be even lower than the already low 34% coverage. Compare the areas without coverage to the huge set of end to end tests, and identify end to end tests that can be replaced with unit tests.
- Introduce a task to measure and report on unit test coverage on every pull request, and don't merge any pull requests that decrease the unit test coverage.
- From there, take a fresh look at the distribution of tests and decide what to do next. There's a good chance that many of the end to end tests could be downgraded to integration tests; i.e. instead of needing the entire system to be up and running, maybe the same cases could be covered with just a couple of components, which will probably be faster.

## 6.29 Conclusion

Over time, Dog Picture Website's test suite had taken longer and longer to run. Instead of facing this problem directly and finding ways to speed up the tests, they had removed the tests from their daily routine, basically postponing dealing with the pain as long as possible. Though this may have helped them speed up initially, it was now slowing them down. Sridhar knew that the answer was to look critically at the test suite and optimize it as much as possible. And when it couldn't be optimized any further, he was able to use parallelization and sharding to make the tests fast enough that the tests could once again become part of the pre merge routine and engineers could get feedback faster.

## 6.30 Summary

- Get an immediate gain from a slow test suite by making it possible to run the fastest tests independently and running them first
- Before solving slow test suite problems with technology, first take a critical look at the tests themselves. Using the test pyramid will help you focus your efforts, and enforcing test coverage will help you maintain a strong unit test base
- That being said, perhaps your test suite is super solid, but they just take a long time to run. When you've reached this point, you can use parallelization, and sharding (parallelization) to speed up your tests by trading time for hardware

## 6.31 Up next . . .

In the next chapter we'll expand on the theme of getting signals at the right time in the development lifecycle - in Dog Picture Website's case, by shifting the tests to earlier in their process, often called shifting left. We'll look at the various signals that are a part of the software lifecycle, as well as when and how to make the signals available.

## Give the right signals at the right times

# 7



---

### This chapter covers:

- identify the points in a change's lifecycle when bugs can be introduced
- Describe how to guarantee that bugs will not be introduced by conflicting changes; weigh the pros and cons of each approach
- Catch bugs at all points in a change's lifecycle by running CI before merging, after merging and periodically

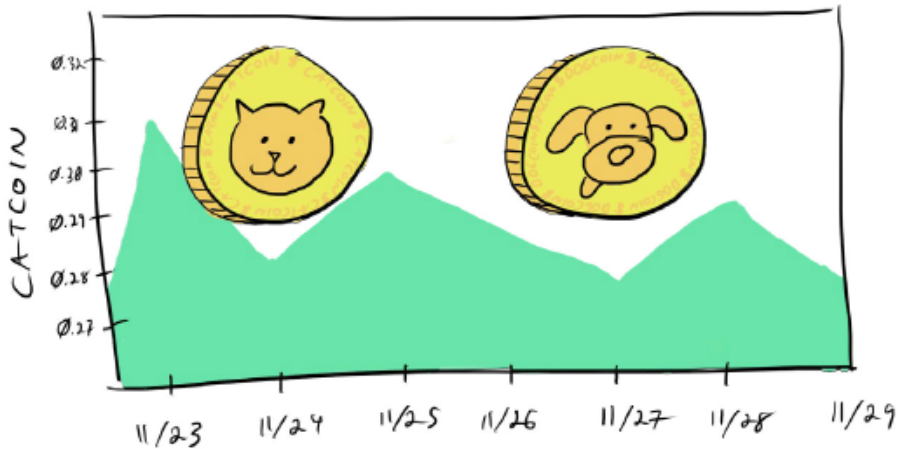
---

In the previous chapters we've seen Continuous Integration pipelines running at different stages in a change's lifecycle. We've seen them run after a change is committed, leading to an important rule: *when the pipeline breaks, stop merging*. We've also seen cases where linting and tests are made to run before changes are merged, ideally to prevent getting to a state where the code base is broken.

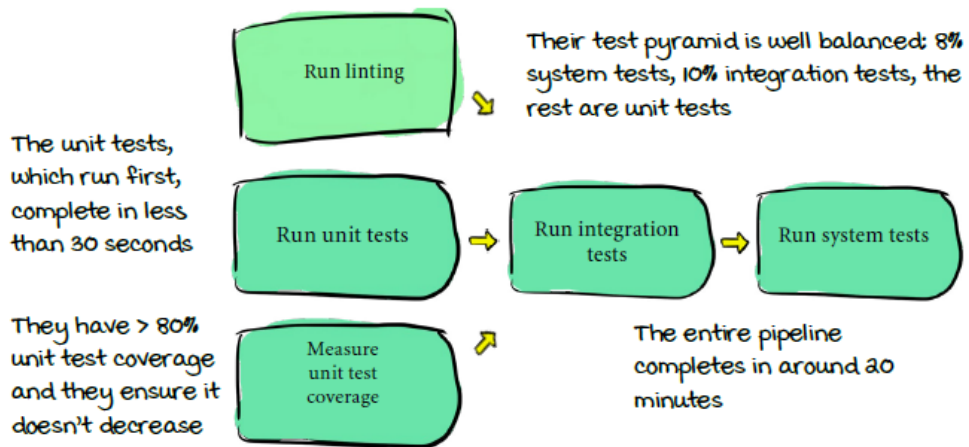
In this chapter we'll look at the lifecycle of a change, all the different places where bugs can be introduced, and how to run pipelines at the right times to catch and fix these bugs as quickly as possible.

## 7.1 CoinExCompare

CoinExCompare is a website that publishes exchange rates between digital currencies. Users can log onto their website and compare exchange rates, for example between currencies such as CatCoin and DogCoin.



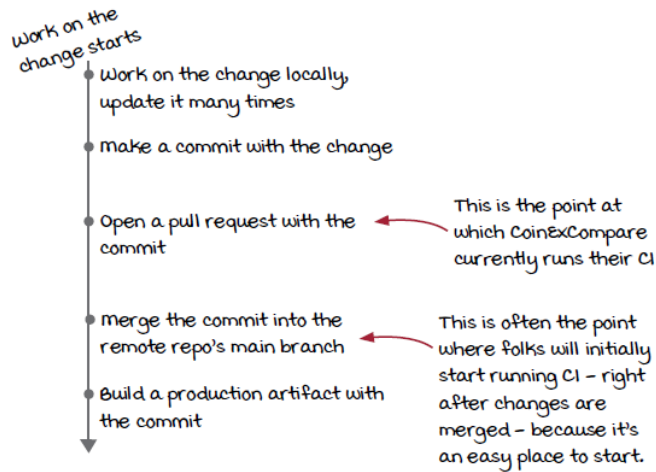
The company has been growing rapidly, but lately they've been facing bugs and outages. They're especially confused because they've been looking carefully at their pipelines, and they think they've done a pretty good job of covering all the bases:



With a great CI pipeline like that, what could they be doing wrong?

## 7.2 Lifecycle of a change

To figure out what might be going wrong for CoinExCompare, they map out the timeline of a change, so they can think about what might go wrong along the way:



### Vocab time

The term *production* is used to refer to the environment where you make your software available to your customers. If you run a service, the endpoint(s) available to your customers can be referred to as *production*. Artifacts such as images and binaries that run in this environment (or are distributed directly to your customers) can be called *production* artifacts (e.g. “*production* images”). This term is used to contrast with any intermediate environments (e.g. a “staging” environment) or artifacts which may be used for verification or testing along the way, but aren’t ever made directly available to your customers.

### 7.3 CI before and after merge

If you're starting from no automation at all, the easiest place to start running CI is often right after a change is merged.

We saw this in chapter 2, when Topher setup webhook automation for Cat Picture website that would run their tests whenever a change was pushed. This quickly led to them adopting an important rule:

This depends on what tools you're already using. Some tools, like GitHub, make it very easy to setup pull request based CI.

#### When the pipeline breaks, stop pushing changes.

This is still a great place to start and the easiest way to hook in automation, especially if you're using version control software that doesn't come with additional automation features out of the box and you need to build it yourself (like Topher did in chapter 2).

However it has some definite downsides:

- You will only find out about problems AFTER they are already added to the codebase. This means that your codebase can get into a state where it isn't safe to release - and part of Continuous Delivery is **getting to a state where you can safely deliver changes to your software at any time**. Allowing your codebase to become broken on a regular basis directly interferes with that goal.
- Requiring that everyone stop pushing changes when the CI breaks stops everyone from being able to make progress which is at best frustrating, and at worst, expensive.

This is where CoinExCompare was about 6 months ago, but they decided to invest in automation that would allow them to run their CI before merging instead - so they could prevent their codebase from getting into a broken state. This mitigates the two downsides of running CI after the changes are already merged:

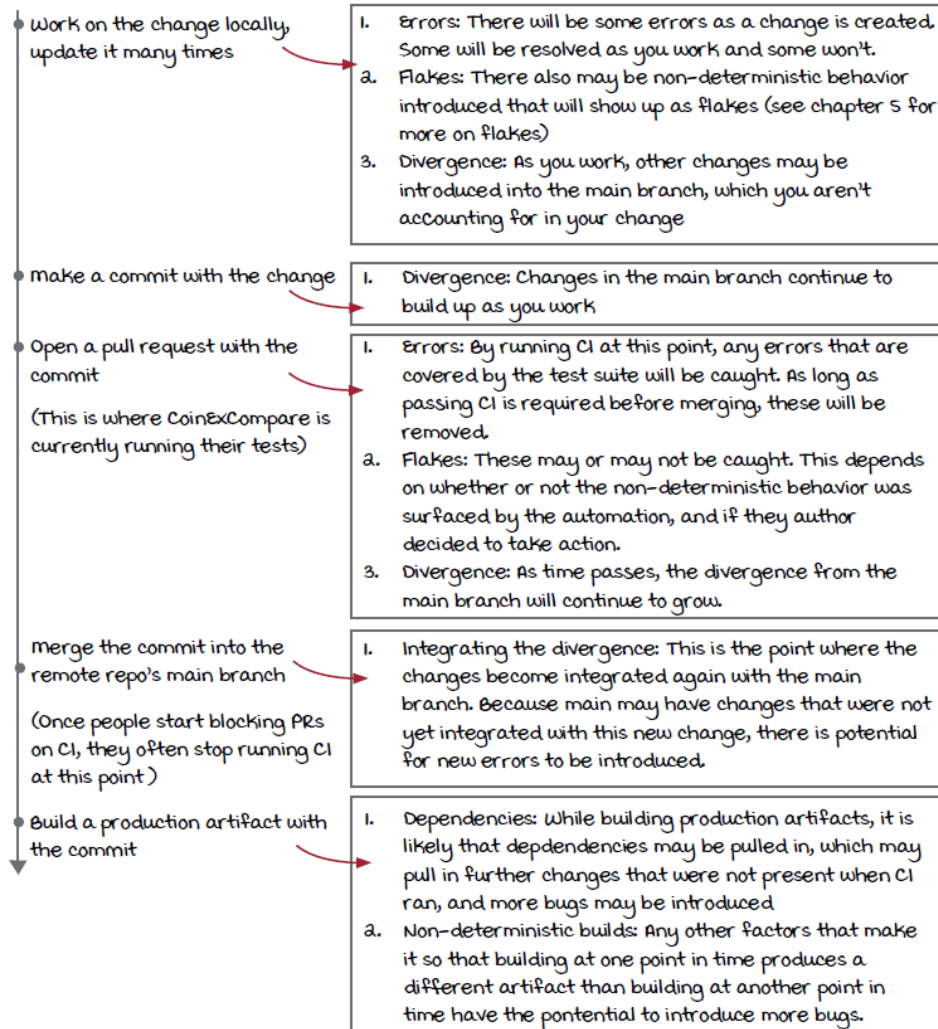
- Instead of finding out about problems after they've already been added, stop them from being added to the main codebase at all.
- Avoid blocking everyone when a change is bad; instead let the author of the change deal with the problem. Once it's fixed, the author will be able to merge the change.

This is where CoinExCompare is today: they run CI before changes are merged, and they don't merge changes until the CI passes.



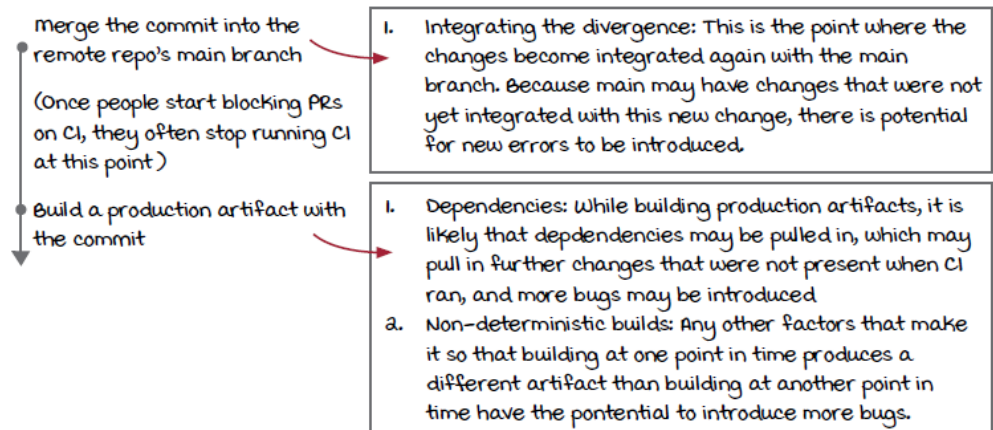
## 7.4 Timeline of a change's bugs

CoinExCompare requires CI to pass before a change is merged, but they're still running into bugs in production. How can that be? To understand, let's take a look at all the places bugs can be introduced for a change:



## 7.5 CI only before merging misses bugs

CoinExCompare is currently blocking PR merges on CI passing, but that is the **ONLY** time they're running their CI. And as it turns out, there are a few more places that bugs can creep in after that point:



This comes down to three sources of bugs:

1. **Divergence from the main branch:** If CI runs only before a change is integrated back into the main branch, this means that there might be changes in main that the new change didn't take into account, and CI was never run for.
2. **Changes to dependencies:** Most artifacts will require packages and libraries outside of its own codebase in order to operate. When building production artifacts, some version of these dependencies will be pulled in. If these are not the same version that you ran CI with, new bugs can be introduced.
3. **Non-determinism:** this pops up both in the form of flakes that aren't caught and also subtle difference from one artifact build to the next which have the potential to introduce bugs.

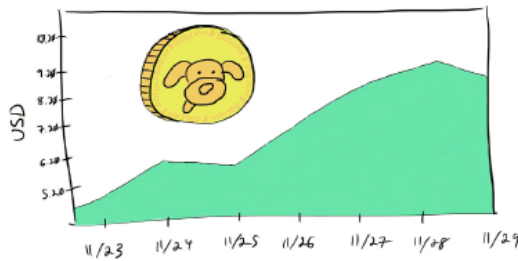
Let's take a look at how CoinExCompare can tackle each of these sources of bugs.

## 7.6 A tale of two graphs: default to seven days

CoinExCompare recently ran into a production bug that was caused by the first source of post-merge bugs:

### Divergence from the main branch

Nia has been working on a feature to graph the last 7 days worth of coin activity for a particular coin. For example, if a user went to the landing page for DogCoin, they would see a graph like this, showing the closing price of the coin in USD on each of the last 7 days:



While she's working on this functionality, she finds an existing function that looks like it'll make her job a lot easier. The function `get_daily_rates` will return the peak daily rates for a particular coin (relative to USD) for some period of time. By default the function will return the rates for all time, indicated by a value of 0 (aka MAX).

```
MAX=0

def get_daily_rates(coin, num_days=MAX):
    rate_hub = get_rate_hub(coin)
    rates = rate_hub.get_rates(num_days)
    return rates
```

Looking around the codebase, Nia is surprised to see that none of the callers are making use of the logic that defaults `num_days` to MAX. Since she has to call this function a few times, she decides that defaulting to 7 days is reasonable, and it gives her the functionality she needs, so she changes the function to default to 7 days instead of MAX and adds a unit test to cover it.

```
def get_daily_rates(coin, num_days=7):
    rate_hub = get_rate_hub(coin)
    rates = rate_hub.get_rates(num_days)
    return rates

...
def test_get_daily_rates_default(self):
    rates = get_daily_rates("catcoin")
    self.assertEqual(rates, [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0])
```

All the tests, including her new one, pass, so she feels good about opening up a pull request for her change.

## 7.7 A tale of two graphs: default to thirty days

But Nia doesn't realize that someone else is making changes to the same code!

Fellow CoinExCompare employee Zihao is working on a graph feature for another page. This feature shows the last 30 days worth of data for a particular coin.

Unfortunately neither Nia nor Zihao have realized that there is more than one person working on this very similar logic!

And great minds think alike: Zihao also noticed the same function that Nia did and thought it would give him exactly what he needed:

```
MAX=0

def get_daily_rates(coin, num_days=MAX):
    rate_hub = get_rate_hub(coin)
    rates = rate_hub.get_rates(num_days)
    return rates
```

We just saw that Nia changed this function, but her changes haven't been merged yet, so Zihao isn't at all aware of them.

Zihao did the same investigation that Nia did, and noticed that no one was using the default behavior of this function. Since he has to call it a few times, he felt it would be reasonable to change the default behavior of the function so that it would return rates for the last 30 days instead of for all time.

He makes the change a bit differently than Nia:

```
MAX=0

def get_daily_rates(coin, num_days=MAX):
    rate_hub = get_rate_hub(coin)
    rates = rate_hub.get_rates(30 if num_days==MAX else num_days)
    return rates
```

Zihao also adds a unit test to cover his changes:

```
def test_get_daily_rates_default_thirty_day(self):
    rates = get_daily_rates("catcoin")
    self.assertEqual(rates, [2.0]*30)
```

Both Nia and Zihao have changed the same function to behave differently, and are relying on the changes they've made. Nia is relying on the function returning 7 days worth of rates by default, and Zihao is relying on it returning 30 days worth of data.

### *Who changed it better?*

Nia changed the argument default, while Zihao left the argument default alone and changed the place where the argument was used. Nia's change was the better approach: in Zihao's version the default is being set twice to two different values - not to mention that the `MAX` argument will no longer work because even if someone provides it explicitly, the logic will return 30 days instead. This is the sort of thing that hopefully would be pointed out in code review. In reality this example is a bit contrived so that we can demonstrate what happens when conflicting changes are made but not caught by version control.

## 7.8 Conflicts aren't always caught

Nia and Zihao have both changed the defaulting logic in the same function, but at least when it comes time to merge, these conflicting changes will be caught, right?

Unfortunately no! For most version control systems, the logic to find conflicts is simple and has no awareness of the actual semantics of the changes involved. When merging changes together, if exactly the same lines are changed, the version control system will realize that something is wrong, but it can't go much further than that.

Nia and Zihao changed different lines in the `get_daily_rates` function, so the changes can actually be merged together without conflict!

Zihao merges his changes first, changing the state of `get_daily_rates` in the main branch to have his new defaulting logic:

```
MAX=0

def get_daily_rates(coin, num_days=MAX):
    rate_hub = get_rate_hub(coin)
    rates = rate_hub.get_rates(30 if num_days==MAX else num_days)
    return rates
```

Meanwhile, Nia merges her changes in as well. Zihao's changes are already present in main, so her changes to the line two lines above Zihao's changes are merged in, resulting in this function:

```
MAX=0

def get_daily_rates(coin, num_days=7):
    rate_hub = get_rate_hub(coin)
    rates = rate_hub.get_rates(30 if num_days==MAX else num_days)
    return rates
```

Nia's change sets the default value for the argument

meanwhile Zihao was relying on the argument defaulting to max

The result is that Zihao's graph feature is merged first, and it works just fine, until Nia's changes are merged, resulting in the function above. Nia's changes break Zihao's: now that the default value is 7 instead of `MAX`, Zihao's ternary condition will be false (unless some unlucky caller tries to explicitly pass in `MAX`), and so the function will now return 7 days worth of data by default. This means Nia's functionality will work as expected, but Zihao's is now broken.

### *Does this really happen?*

It sure does! This example is a little contrived since the more obvious solution for Zihao would be to also change the default argument value, which would have immediately been caught as a conflict. A more realistic scenario that comes up more frequently in day to day development might involve changes that span multiple files, for example making changes that depend on a specific function, while someone else makes changes to that function.

## 7.9 What about the unit tests?

Nia and Zihao both added unit tests as well. Surely this means that the conflicting changes will be caught?

If they had added the tests at the same point in the file, the version control system would catch this as a conflict, since they would both be changing the same lines. Unfortunately in our example, the unit tests were introduced at different points in the file so no conflict was caught! The end result of the merges would be both unit tests being present:

```
def test_get_daily_rates_default(self):
    rates = get_daily_rates("catcoin")
    self.assertEqual(rates, [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0])
...
def test_get_daily_rates_default_thirty_day(self):
    rates = get_daily_rates("catcoin")
    self.assertEqual(rates, [2.0]*30)
```

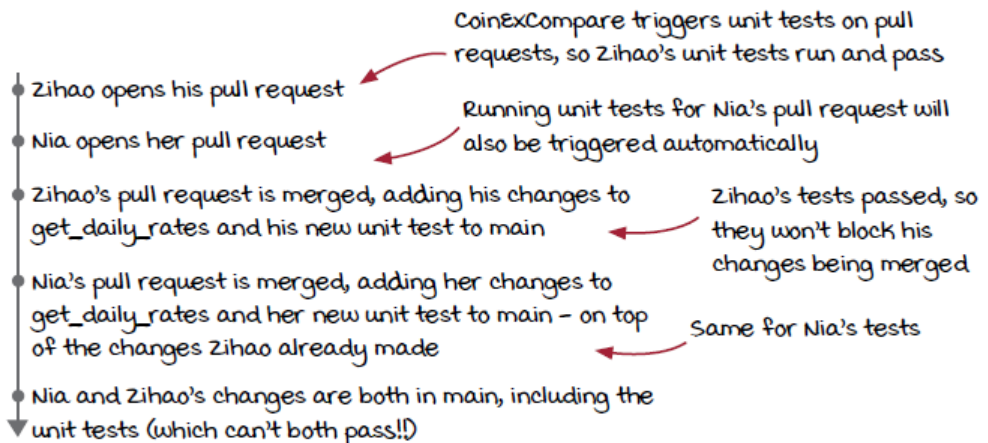
Nia's unit test expects the function to return 7 days worth of data by default

Zihao's unit test expects the exact same function, called with exactly the same arguments, to return 30 days worth of data

The version control system couldn't catch the conflict, but at least it should be impossible for both tests to pass, right? So surely the problem will be caught when the tests are run?

Yes and no! If both of these tests are run at the same time, one of them will fail (it is impossible for both to pass unless something undeterministic is happening).

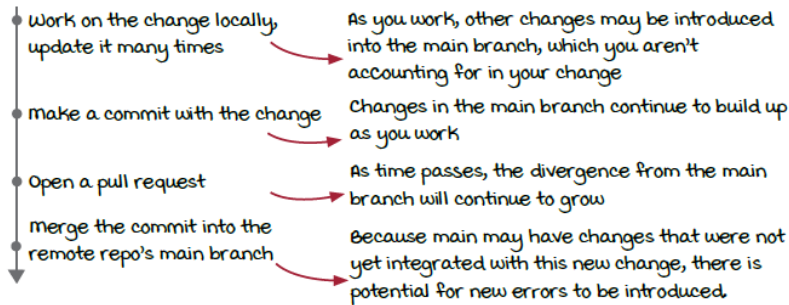
But will both tests be run together? Let's look at a timeline of what happens to Nia and Zihao's changes and when the tests will be run:



Tests are run automatically for each pull request *only*. CoinExCompare is relying solely on running their CI (including tests) on each pull request, but there is no automation to run CI on the combined changes after they have been merged together.

## 7.10 Pull request triggering still lets bugs sneak in

Running CI triggered by pull requests is a great way to catch bugs before they are introduced into the main branch. But as we saw with Nia and Zihao,



The longer your changes are in your own branch and aren't integrated back into the main branch, the more chance there will be that a conflicting change will be introduced that will cause unforeseen bugs.

Another way to mitigate this risk is to merge back into main as quickly as possible - more on this in the next chapter!



### Question

Q: I regularly pull in changes from main as I work, doesn't that fix the problem?

A: That certainly reduces the chances of missing conflicting changes introduced into main, but unless you can guarantee the latest changes are pulled in, CI is run immediately before merge, and no further changes sneak in during that time, there is still a chance you'll miss something with only pull request triggered CI.



### Takeaway

Running CI on pull requests before merging won't catch all conflicting changes. If the conflicting changes are changing exactly the same lines, version control can catch the conflict and force updating (and re-running CI) before merging, but if the changes are on different lines - or in different files - you can end up in a situation where CI has passed before merge, but after merging, the main branch is in a broken state.

## 7.11 CI before AND after merge

What can CoinExCompare do to avoid getting into a state where main is broken because conflicts haven't been caught?

Both Nia and Zihao added tests to cover their functionality - if those tests had been run once the changes had been combined (merged) the issue would have been caught right away.

CoinExCompare sets a new goal:

**Require changes are combined with the latest main and CI passes before merging**

What can CoinExCompare do to meet this goal? They have a few options:

1. Run CI periodically on main
2. Require branches to be up to date before they can be merged into main
3. Use automation to merge changes with main and re-run CI before merging (aka using a **merge queue**)

We'll look at each option in more detail, but at a glance each comes with its own set of tradeoffs:

Option (1) will catch these errors but only after they've actually been introduced into main; this means main can still get into a broken state.

Option (2) will prevent the kind of errors that we've been looking at from getting in, and it's supported out of the box by some version control systems (for example GitHub). But in practical application it can be a huge nuisance.

Option (3) if implemented correctly can also prevent these errors from getting in. As an out of the box feature it's works very well, but it can be complicated if you need to implement and maintain it yourself.



## 7.12 Option 1: Run CI periodically

Let's look at each of the options in more detail.

With Nia and Zihao's situation, one of the most frustrating aspects was that the issue wasn't caught until it was seen in production - even though there were unit tests that could have caught it!

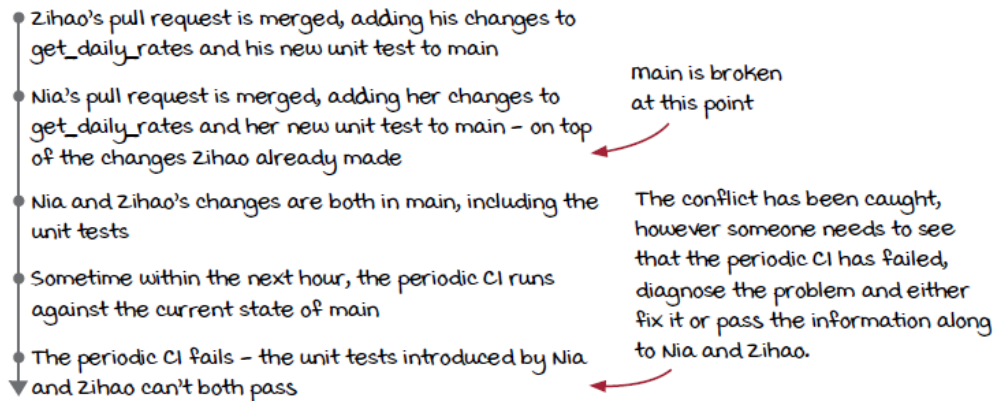
With this option, we focus less on stopping this edge case from happening, and more on easily detecting it if it does. The truth is that bugs like these, which are caused by the interaction of multiple changes, are unlikely to happen very often.

An easy way to detect these problems is to run your CI **periodically** against main, in addition to running it against pull requests. This could look like a nightly run of the CI, or even more often (e.g. hourly) if the tasks are fast enough.

Of course it has a couple of downsides:

1. This approach will let main get into a broken state
2. This requires someone to monitor these periodic tests, or at least be responsible for acting on them when they break

What would it look like for Nia and Zihao if CoinExCompare decided to use periodic CI as their solution to addressing these conflicting changes? Let's say CoinExCompare decides to run their periodic tests every hour:



At least now the problem will be caught, and might be stopped before it makes it to production, but does this meet CoinExCompare's goal?

**Require changes are combined with the latest main and CI passes before merging**

Since everything happens post merge, option 1 doesn't meet their bar.

## 7.13 Setting up periodic CI

CoinExCompare isn't going to move ahead with periodic CI (yet) but before we move on to the other options let's take a quick look at what it would take to set this up.

CoinExCompare is using GitHub actions, so making this change is easy. Say they wanted to run their pipeline every hour. In their GitHub actions workflow, they can use the schedule syntax to do by including a `schedule` directive in the `on` triggering section:

```
on:
  schedule:
    - cron: '0 * * * *'
```

The GitHub actions schedule directive uses cron tab syntax to express when to run

Stay tuned: there are some other upsides to running CI periodically which we'll look at a bit later in this chapter.

Though it's easy to setup the periodic (aka scheduled) triggering, the bigger challenging is actually doing something with the results.

When running CI against a pull request, it's much more clear who needs to take action when it fails: the author(s) of the pull request itself. And they will be motivated to do this because they need the CI to pass before they can merge.

With periodic CI, the responsibility is much more diffused. In order to make your CI useful, you need someone to be notified when failures occur, and you need a process for determining who actually needs to fix the failures.

Notification could be handled through a mailing list or by creating a dashboard; the harder part is deciding who needs to actually take action and fix the problems.

A common way to handle this is to setup a rotation (similar to being on call for production issues) and share the responsibility across the team. When failures occur, whoever is currently responsible needs to decide how to triage and deal with the issue.

If the periodic CI frequently has problems, dealing with the issues that pop up can have a significant negative impact on the productivity of whoever has to handle them and can be a drain on morale. This makes it (even more) important to make a concerted effort to make CI reliable so that the interruptions are infrequent.

See Chapter 5 for techniques for fixing noisy CI.



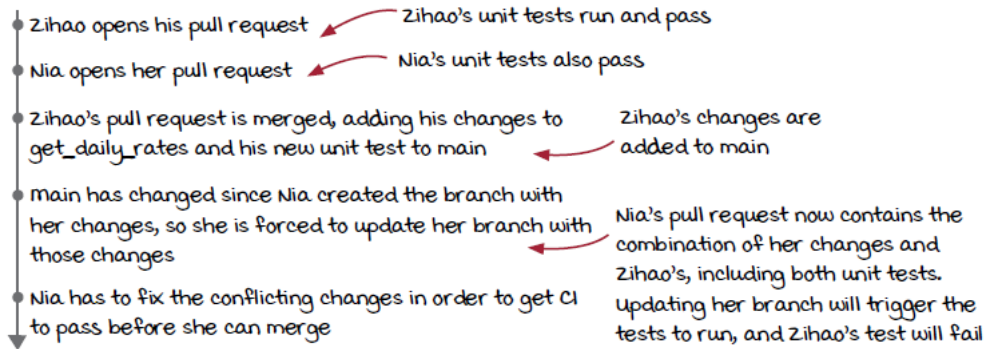
### Takeaway

**Takeaway:** Running CI periodically can catch (but not prevent) this class of bugs. While it is very easy to get up and running, for it to be effective you need someone to be monitoring these periodic tests and acting on failures.

## 7.14 Option 2: Require branch to be up to date

Option 1 will detect the problem, but won't stop it from happening. In option 2, you are guaranteed that problems won't sneak in. This works because if the base branch is updated, you'll be forced to update your branch before you can merge- and at the point that you update your branch, CI will be triggered.

Would this have fixed Nia and Zihao's problem? Let's take a look at what would have happened.



As soon as Zihao merged, Nia would be blocked from merging until she pulled in the latest main, including Zihao's changes. This would trigger CI to run again - which would run both Nia and Zihao's unit tests. Zihao's would fail, and the problem would be caught!

This strategy comes with an additional cost though: anytime main is updated, all pull requests for branches which don't contain these changes will need to be updated. In Nia and Zihao's case this was important because their changes conflicted, but this policy will be universally applied, whether it is important to pull in the changes or not.

### *Can you automate updating every single branch?*

It's possible (and this would be a nice feature for version control systems to support!) However, remember that for distributed version control systems like git, the branch that is backing the pull request is a copy of the branch the developer has been working on on their machine. This means the developer will need to pull any changes automatically added if they need to continue working. Not a deal breaker, but some extra complication. Also with this kind of automation, we're starting to get into the territory of option 3.

## 7.15 Option 2: At what cost?

Requiring that a branch be up to date with main before being merged would have caught Nia and Zihao's problem, but also, this approach would impact every pull request and every developer.

- Pull Request #45 is opened
- Pull Request #46 is opened
- Pull Request #47 is opened
- Pull Request #45 is merged, updating main
- Main was updated, so PRs #46 and #47 are now blocked from merging until they are updated
- Pull Request #48 is opened
- Pull Request #46 is updated and then merged
- Main was updated again, so now PR #48 is also blocked from merging until it is updated, and PR #47 continues to be blocked by the changes merged for PR #45 and now for PR #46 as well

Is it worth the cost? Let's see how this policy would impact several pull requests:

Each time a pull request is merged, it impacts (and blocks) all other open pull requests!

CoinExCompare has around 50 developers, and each of them try to merge their changes back into main every day or so. This means there are around 20-25 merges into main per day.

Why merge so frequently? See chapter 8 for more!

Imagine that 20 PRs are open at any given time, and the authors try to merge them within a day or so of opening. Each time a PR is merged, it will block the other 19 open PRs until they are updated with the latest changes.

The strategy in option 2 will guarantee that CI always runs with the latest changes, but at the cost of potentially a lot of tedious updates to all open PRs. In the worst case, developers will find themselves constantly racing to get their PRs in so they don't get blocked by someone else's changes.



### Takeaway

Requiring the branch to be up to date before changes can be merged will prevent conflicting changes from sneaking in, but it is most effective when only a few people are contributing to a codebase. Otherwise the headache may not be worth the gain.

## 7.16 Option 3: Automated merge CI

CoinExCompare decides that the additional overhead and frustration of always requiring branches to be up to date before merging isn't worth the benefit. What else can they do?

With CoinExCompare's current setup, tests ran against both Nia and Zihao's pull requests before merging. Those tests would be triggered to run again if anything in those PRs changed. This worked out just fine for Zihao's changes, but didn't catch the issues introduced when Nia's changes were added.

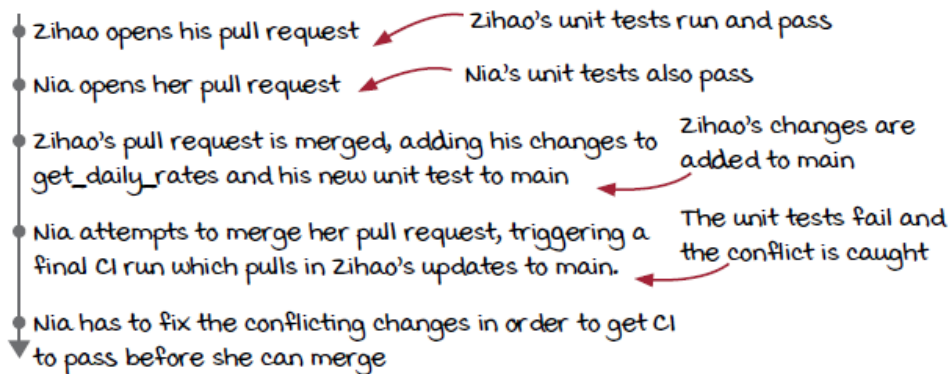
If only Nia's CI had been triggered to a) run one more time before merging, and b) included the latest changes from main when running those tests, the problem would have been caught.

So another solution to the problem is to introduce automation to run CI which runs final time before merging, against the changes merged with the latest code from main.

Accomplish this by doing the following:

1. Before merging, even if the CI has passed previously, run the CI again, including the latest state of main (even if the branch itself isn't up to date)
2. If the main branch changes during this final run, run it again. Repeat until it has been run successfully with exactly the state of main that you'll be merging into

What would have happened to Nia and Zihao's changes if they'd had this automation?



With the CI had pulling in the latest main (with Zihao's changes) and running a final time before allowing Nia to merge, the conflicting changes would be caught and won't make it into main.

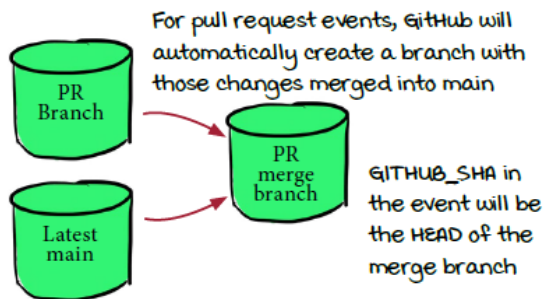
## 7.17 Running CI with the latest main

In theory it makes sense to run CI before merging, with the latest main, and make sure main can't change without re-running CI, but how do you actually pull this off? We can break the elements down a little further. We need:

1. A mechanism to combine the branch with the latest changes in main which CI can use
2. Something to run CI before merge and block the merge from occurring until it passes
3. A way to detect updates to main (and trigger the pre-merge CI process again) OR a way to prevent main from changing while the pre-merge CI is running

How do you combine your branch with the latest changes in main? One way is to do this yourself in your CI tasks by pulling the main branch and doing a merge.

But you often don't need to because some version control systems will actually take care of this for you. For example, when GitHub triggers webhook events (or when using GitHub actions), GitHub provides a **merged commit** to test again: it creates a commit that merges the PR changes with main.



Using this merged commit for all pull request triggered CI will increase your chances of catching these sneaky conflicts.

As long as your tasks fetch this merge commit (provided as the `GITHUB_SHA` in the triggering event), you've got (1) covered!

A different way of looking at this option (automated merge CI) is that it's an alternative approach to option 2. Option 2 requires branches to be up to date before merging, and option 3 makes sure branches are up to date when CI runs by introduction automation to update branch to that state before running CI, vs. blocking and waiting for the author to update their pull request with the latest changes.

## 7.18 Merge Events

Now that we've covered the first piece of the recipe, let's look at the rest. We need:

2. Something to run CI before merge and block the merge from occurring until it passes
3. A way to detect updates to main (and trigger the pre-merge CI process again) OR a way to prevent main from changing while the pre-merge CI is running

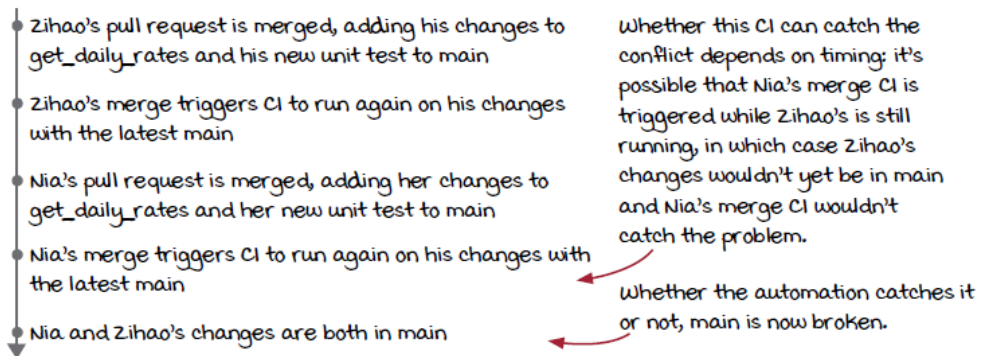
Most version control systems will give you some way to run CI in response to events, such as when a pull request is opened, when it is updated, or in this case, when it is merged, aka a **merge event**. If you run your CI in response to the merge event, you can be alerted when a merge occurs, and run your CI in response.

However this doesn't quite address requirements (2) and (3) above

GitHub makes triggering on a merge a bit complex: the equivalent of the merge event is a `pull_request` event with the activity type `close` when the `merged` field inside the payload has the value `true`. Not terribly straightforward!

2. The merge event will be triggered AFTER the merge occurs, i.e. after the PR is merged back into main, so if a problem is found it will have already made its way into the main branch. At least you'll know about it, but main will be broken.
3. There is no mechanism to ensure that any changes to main that occur while this automation is running will trigger the CI to run again, so some conflicts can still slip through the cracks.

What would this look like for Nia and Zihao's scenario?



So unfortunately triggering on merges won't give us exactly what we're looking for. It will increase the chances that we'll catch conflicts, but only after they've been introduced, and more conflicts can still sneak in while the automation is running.

## 7.19 Merge queues

If triggering on the merge event doesn't give us the whole recipe, what else can we do? The complete recipe we are looking for requires:

1. A mechanism to combine the branch with the latest changes in main which CI can use
2. Something to block the merge from occurring until CI passes
3. A way to detect updates to main (and trigger the pre-merge CI process again) OR a way to prevent main from changing while the pre-merge CI is running

We have an answer for (1) but the complete solution to (2) and (3) is lacking. The answer is to create automation which is entirely responsible for merging PRs. This automation is often referred to as a **merge queue** or **merge train**, i.e. merging is never done manually, it is always handled by automation which can enforce (2) and (3).

You can get this functionality by building the merge queue yourself, but fortunately you shouldn't need to! Many version control systems now provide a merge queue feature out of the box.

Merge queues, as their name implies, will manage queues of pull requests which are eligible to merge (e.g. they've passed all the required CI):

- Each eligible pull request is added to the merge queue
- For each pull request in order, the merge queue creates a temporary branch that merges the changes into main (the same logic as the merged commit GitHub provides in pull request events)
- The merge queue runs the required CI on the temporary branch
- If CI passes, the merge queue will go ahead and do the merge. If it fails, it won't. Nothing else can merge while this is happening because all merges need to happen through the merge queue.

For very busy repos, some merge queues optimize by **batching** together pull requests for merging and running CI. If the CI fails, an approach like binary search can be used to quickly isolate the offending PRs, for example, split the batch into two groups, re-run CI on each, and repeat until you discover which PR(s) broke the CI. Given how rare post merge conflicts are and if enough PRs are in flight that waiting for the merge queue becomes tedious, this optimization can save a lot of time.



## 7.20 Merge queue for CoinExCompare

Let's see how a merge queue would have addressed Nia and Zihao's conflict:

- Zihao's pull request is ready to merge
  - Zihao's pull request is added to the merge queue
  - The merge queue creates a branch from main, merges in Zihao's changes and starts CI
  - Nia's pull request is ready to merge
  - Nia's pull request is added to the merge queue
  - The merge queue is currently running CI for Zihao's PR, so Nia's PR has to wait
  - Zihao's PR passes and the merge queue merges his changes into main
  - It's Nia's turn: the merge queue creates a branch from main (including Zihao's recently merged changes), merges in Nia's changes and starts CI
  - CI fails and Nia must deal with the conflicting changes before she can merge her PR
- even if Zihao and Nia's attempts to merge overlap, the merge queue handles the race condition and prevents any conflicts from sneaking in
- The conflict was caught before and didn't make it into main!



### Building your own merge queue

This is doable but it's a lot of work. At a high level what you need to do is to a) create a system which is aware of the state of all pull requests in flight, b) block your pull requests on merging (e.g. via branch protection rules) until this system gives the green light, c) have this system select PRs which are ready to merge, merge them with main, and run CI, and finally d) have this system do the actual merging. This complexity has a lot of potential for error, but if you absolutely need to guarantee that conflicts do not sneak in, and you're not using a version control system with merge queue support, it might be worth the effort.



### Takeaway

Merge queues prevent conflicting changes from sneaking in by managing merging and ensuring CI passes for combination of the changes being merged and the latest state of main. Many version control systems provide this functionality, which is great because building your own may not be worth the effort.



## It's your turn: match the downsides

Let's look again at the three options for catching conflicts that are introduced between merges. Match these three options to the 2 downsides that fit best below:

1. Run CI periodically
2. Require branches to be up to date
3. Use a merge queue

For each of the three options above, select the 2 downsides that fit best:

1. Slows down the time to merge a PR
2. Requires someone to monitor and be responsible for the results
3. Results in many pull requests being blocked from merging until the authors unblock them
4. Allows main get into a broken state
5. Complex to implement if your version control system doesn't support it
6. Tedious when more than a few developers are involved



## Answers

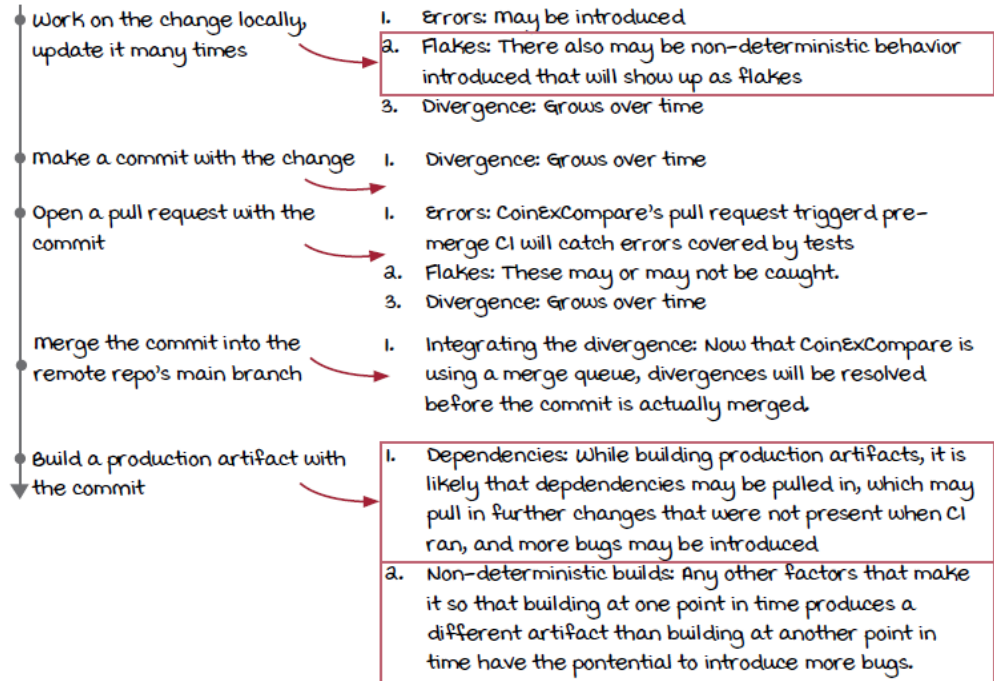
1. Run CI periodically: (4) conflicts introduced between pull requests will be caught when the periodic CI runs, after the merge into main has already occurred. (2) if no one pays attention to the periodic runs, they'll have no benefit.
2. Require branches to be up to date: (3) every time a merge happens, all other open pull requests will be blocked until they update. (6) when only a few developers are involved this is feasible but for a larger team this can be tedious.
3. Use a merge queue: (1) every PR will need to run CI an additional time before merging, and may need to wait for PRs ahead of it in the queue. (5) creating your own merge queue system can be complex.

## 7.21 Where can bugs still happen?

CoinExCompare decides to use a merge queue, and with GitHub they're able to opt into this functionality quite easily by adding the setting to their branch protection rules for main to Require Merge Queue.

Now that they are using a merge queue, have the folks at CoinExCompare successfully identified and mitigated all the places where bugs can be introduced?

Let's take a look again at the timeline of a change and when bugs can be introduced:



Even with the introduction of a merge queue, there are still several potential sources of bugs CoinExCompare hasn't tackled:

1. Divergence from and integration with the main branch (now handled!)
2. Changes to dependencies
3. Non determinism: in code and/or tests (i.e. flakes), and/or how artifacts are built

## 7.22 Flakes and pull request triggered CI

We learned in Chapter 5 that **flakes** occur when tests fail inconsistently: sometimes they pass, sometimes they fail. We also learned that this can be caused equally by a problem in the test or by a problem in the code under test, so the best strategy is to treat these like bugs and investigate them fully.

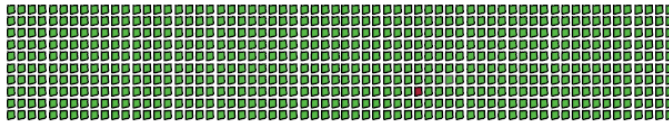
But since flakes don't happen all the time, they can be hard to catch!

CoinExCompare now runs CI on each pull request and before a pull request is merged. This is where flakes would show up, and the truth is that they would often get ignored. It's hard to resist the temptation to just run the tests again, merge, and call it a day - especially if your changes don't seem to be involved.

Is there a more effective way CoinExCompare can expose and deal with these flakes?

A few pages ago we looked at periodic CI, and decided it wasn't the best way to address sneaky conflicts, however it turns out the periodic CI can be a great way to expose flakes.

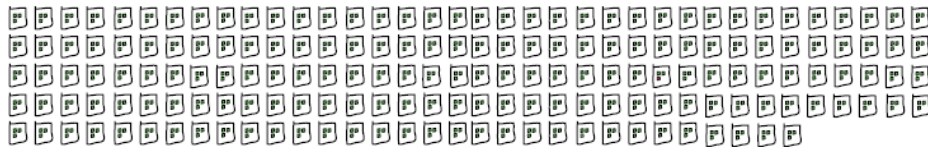
Imagine a test that flakes only once out of every 500 runs.



CoinExCompare developers have about 20-25 PRs open per day. Let's say the CI runs at least three times against each PR: once initially, once with changes, and finally again in the merge queue. This means every day there are about  $25 \text{ PRs} \times 3 \text{ runs} = 75$  chances to hit the failing test.



Over a period of about 7 days that's 525 changes to fail, so it's likely this test will fail one of those PRs. (And it's also likely the developer who created the pull request will just ignore it and run the CI again!)



## 7.23 Catching flakes with periodic tests

When relying only on pull request and merge queue based tests to uncover flakes, CoinExCompare will be able to reproduce a flake that occurs 1/500 times around once every 7 days. And when the flake is reproduced, there's a good chance that the author of the impacted pull request will simply decide to run the tests again and move on.



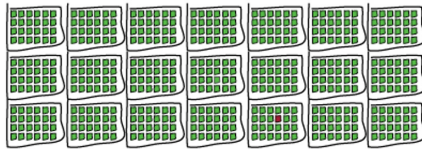
Is there anything that CoinExCompare can do to make it easier to reproduce flakes and not have to rely on the good behavior of the impacted engineer to fix it?

A few pages back we talked about periodic tests, and how they were not the best way to prevent conflicts from sneaking in, but it turns out that catching flakes with periodic tests works really well!

What if CoinExCompare sets up periodic CI to run once an hour? With the periodic CI running once an hour, it would run 24 times a day.



The flakey test fails 1/500 runs, so it would take 500/24 days, or approximately 21 days to reproduce the failure.



Reproducing the failure once every 21 days via periodic CI might not seem like a big improvement, but the main appeal is that if the periodic tests catch the flake, *they aren't blocking someone's unrelated work*. As long as the team has a process for handling failures discovered by the periodic CI, a flake discovered this way has a better chance of being handled and investigated thoroughly than when it pops up and blocks someone's unrelated work.



### Takeaway

Periodic tests help identify and fix non-deterministic behavior in code and tests, without blocking unrelated work

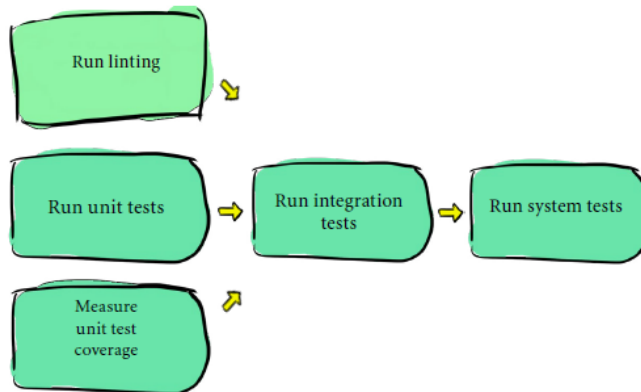
## 7.24 Bugs and building

By adding a merge queue and periodic tests, CoinExCompare has successfully eliminated most of their potential sources of bugs, but there are still ways bugs can sneak in:

1. Divergence from and integration with the main branch
2. Changes to dependencies
3. Non determinism: in code and/or tests (i.e. flakes) (caught via periodic tests), and/or how artifacts are built

Both of these sources of bugs revolve around the build process. In chapter 9 we're going to look at how to structure your build process to avoid these problems, but in the meantime, without overhauling how CoinExCompare builds their images, what can be done to catch and fix bugs introduced at build time?

Spoiler, the best answer to (2) is to always pin your dependencies



Let's take a look again at their pipeline:

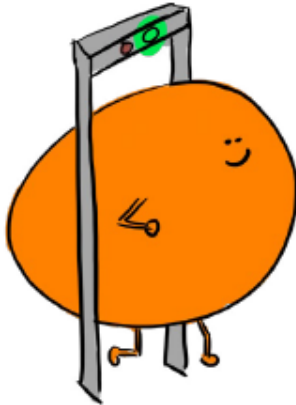
The last task in the pipeline runs the system tests. As with any system tests, these tests test the CoinExCompare system as a whole. System tests need something to run against, so part of this task must include setting up the **system under test (SUT)**. In order to create the SUT, the task needs to build the images used by CoinExCompare.

The **system under test (SUT)** refers to the system that system tests run against to verify the correct operation of.

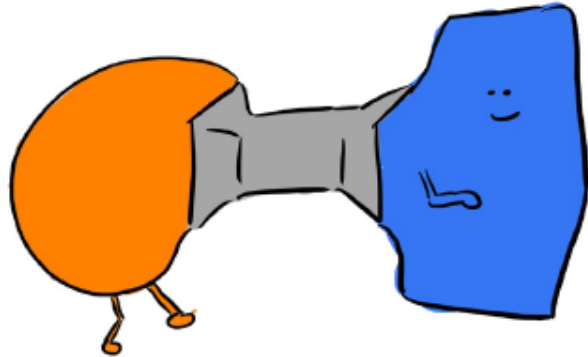
The types of bugs we're currently looking at sneak in while the images are being built - so can they be caught by the system tests? The answer is yes BUT the problem is that the images being built for the system tests are not the same as the ones being built and deployed to production. Those images will be built at some point later on, at which point the bugs can sneak back in.

## 7.25 CI vs. build and deploy

In chapter 2 we looked at two different kinds of tasks: gates and transformations.



Tasks that verify your code are quality gates that your code has to pass through.



Tasks that change your code from one form to another are transformations: your code goes in as an input and comes out in another form.

CoinExCompare separates their gate and transformation tasks into two different pipelines. The purpose of the pipeline we've been looking at so far, their CI pipeline, is to verify code changes (aka gating code changes).

CoinExCompare uses a different pipeline to build and deploy their production image (aka transforming the source code into a running container):

See chapter 13 for more on pipeline graph design.



The reality is that the line between these two kinds of tasks can blur. If you want to be confident in the decisions made by your gate tasks, i.e. your CI, you need to do a certain amount of transformation in your CI as well.

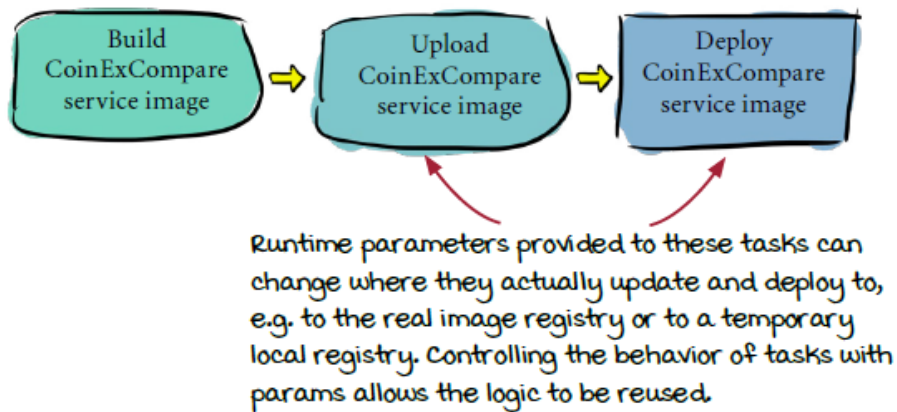
This often shows up in system tests, which are often secretly doing some amount of building and deploying.

## 7.26 Build and deploy with the same logic

The CoinExCompare system test task is actually doing a few things:

1. Setting up an environment to run the system under test
2. Building an image
3. Pushing the image to a local registry
4. Running the image
5. And only THEN running the system tests against the running container

But - and this is very common - it's not actually using the same logic that the deployment pipeline is using to build and deploy their images. If it was, it would be making use of the same tasks that are used in that pipeline:



This means there is a potential for bugs to sneak in when the actual images are built and deployed, specifically:

- Differences based on **when the build happens**, for example pulling in the latest version of a dependency during the system tests, but when the production image is built an even newer version is pulled in.
- Difference based on **the build environment**, for example running the build on a different version of the underlying operating system.

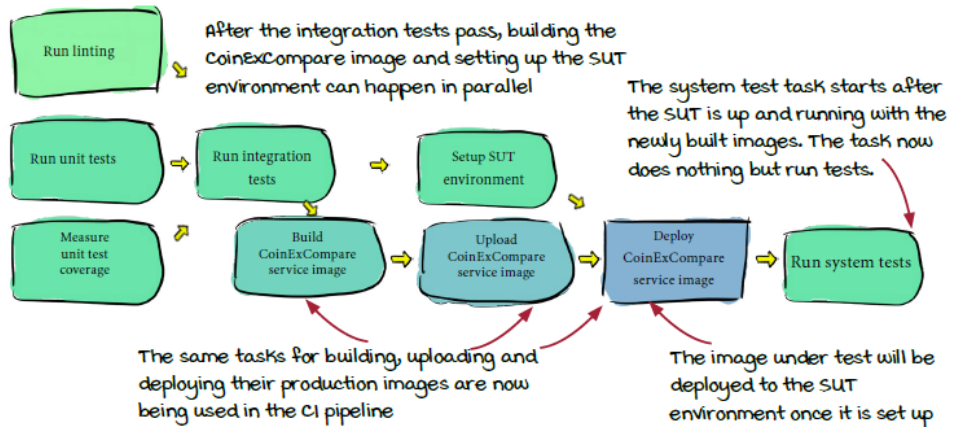
There are 2 changes that CoinExCompare can make to minimize these differences:

- Run the deployment tasks periodically as well
- Use the same tasks to build and deploy for the system tests as are used for the actual build and deploy



## 7.27 Improved CI pipeline with building

CoinExCompare updates their CI pipeline so that the system tests will use the same tasks they use for production building and deploying and it looks like this:



1. Divergence from and integration with the main branch
2. Changes to dependencies
3. Non determinism: in code and/or tests (i.e. flakes), and/or how artifacts are built

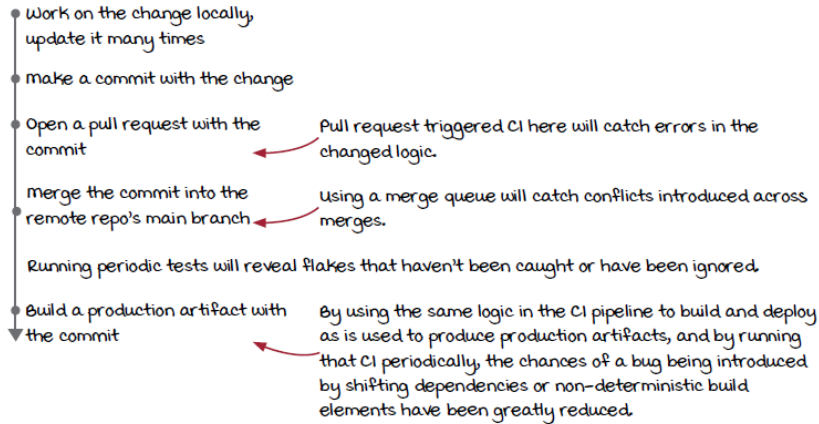
CoinExCompare isn't using Continuous Deployment; see chapter 10 for more on different deployment techniques.

- Changes in dependencies are mitigated because the images are now being built (and tested) every hour. If a change in a dependency introduces a bug, it now has a window of only about an hour to do it, and it will likely be caught the next time the periodic CI runs.
- Non-deterministic builds are mitigated because by using exactly the same tasks to build images for CI, we've reduced the number of variables that can differ.

(See Chapter 9 for more on how to completely defeat these risks.)

## 7.28 Timeline of a change revisited

Did they get them all? The folks at CoinExCompare sit down to look one final time at all the places a bug could be introduced:



CoinExCompare has successfully eliminated or at least mitigate all of the places that bugs can sneak in by:

- Continuing to use their existing pull request triggered CI
- Adding a merge queue
- Running CI periodically
- Updating their CI pipelines to use the same logic for building and deploying as their production release pipeline

With these additional elements in place, they are very happy to see a dramatic reduction in their production bugs and outages.

### *Treating periodic CI artifacts as release candidates*

The last two sources of bugs we observed have only been mitigated, not completely removed. Something quick and easy CoinExCompare would be to start treating the artifacts of their periodic CI as release candidates, and releasing those images as-is; i.e. no longer running a separate pipeline to build again before releasing. More on this in chapter 9!



## It's your turn: identify the gaps

For each of the following triggering setups, identify bugs that can sneak in and any glaring downsides to this approach (assume no other CI triggering):

1. Triggering CI to run periodically
2. Triggering CI to run after a merge to main
3. Pull request triggered CI
4. Pull Request triggered CI with merge queues
5. Triggering CI to run as part of a production build and deploy pipeline



## Answers

1. Periodic CI alone will catch errors, and also sometimes catch flakes, however this will be after they are already introduced to main. Making periodic CI alone work will require having people paying attention to periodic CI who will need to triage errors that occur back to their source.
2. Triggering after a merge to main will catch errors, but only after they are introduced to main. Since it runs immediately after merging, it will be easier to identify who is responsible for the changes, but also chances are high that any flakes revealed will be ignored. This will also require a “don’t merge to main when CI is broken” policy or errors can compound on each other and grow.
3. Pull request triggered CI is quite effective but it will miss conflicts introduced between pull requests and flakes that are revealed are likely to be ignored.
4. Adding a merge queue to pull request triggered CI will eliminate conflicts between pull requests, but it is likely flakes will still be ignored.
5. Running CI as part of a production release pipeline will ensure that errors introduced by updated dependencies (and some non-deterministic elements) are caught before a release, but following up on these errors will interrupt the release process. If they can’t be immediately fixed, and re-running makes the error appear to disappear, there is a good chance they will be ignored.

## 7.29 Conclusion

CoinExCompare thought that running CI triggered on each pull request was enough to catch all the errors that can be introduced by a change, however on closer examination they realized that this approach can't catch everything. By using merge queues, adding periodic tests, and updating their CI to use the same logic as their release pipelines, they've now got just about everything covered!

## 7.30 Summary

- Bugs can be introduced as part of the changes themselves, as conflicts between the changes and a diverging main branch, and as part of the build process
- Merge queues are a very effective way to prevent changes that conflict between PRs from sneaking in. If they aren't available in your version control system, requiring branches to be up to date can work well for small teams, or periodic tests are effective (though this means main may get into a broken state).
- Periodic tests are worth adding regardless as they can be a way to identify flakes without interrupting unrelated PRs, but using them effectively requires setting up some process around them.
- Building and deploying in your CI pipelines in the same way as your production releases are performed will mitigate the errors that can sneak between running the CI and release pipelines.

## 7.31 Up next . . .

In the next chapter, we'll start transitioning into looking at the details of Continuous Delivery pipelines which go beyond Continuous Integration: the transformation tasks that are used to build and deploy your code. The next chapter will dive into effective approaches to version control which can make the process run more smoothly, and how to measure that effectiveness.

## Easy delivery starts with version control

# 8



---

### In this chapter:

- Explain the DORA metrics which measure velocity: deployment frequency and lead time for changes
  - Increase speed and communication by avoiding long lived feature branches and code freezes
  - Decrease lead time for changes by using small, frequent commits
  - Increase deployment frequency safely by using small, frequent commits
- 

In the previous chapters we've been focusing on Continuous Integration, but from this chapter onward we'll start transitioning to the details of the rest of the activities in a Continuous Delivery Pipeline, specifically the transformation tasks that are used to build and deploy your code.

Good CI practices have a direct impact on the rest of your CD. In this chapter we'll dive into effective approaches to version control to make CD run more smoothly, and how to measure that effectiveness.

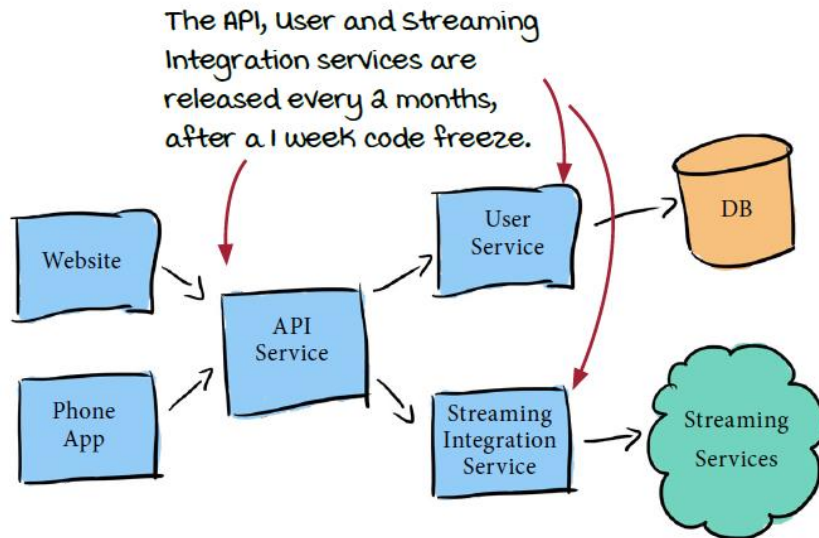
## 8.1 Meanwhile at Watch Me Watch

Remember the startup Watch Me Watch from chapter 3? Well they're still going strong - and in fact growing as a company! In the past two years they've grown from just Sasha and Sarah to a company of more than 50 employees.



From the very beginning they invested in automating their deployments, but as they've grown they've gotten nervous that these deployments are riskier and riskier, so they've been slowing them down.

Each of their services is now only released during specific windows, once every 2 months. For a week before a release, the code base is frozen and no new changes can go in.



In spite of these changes, somehow it feels like the problem is only getting worse: every deployment still feels extremely risky, and even worse, features are taking too long to get into production. Since Sasha and Sarah started on their initial vision, competitors have sprung up, and with the slow pace of features being released, it feels like the competitors are getting ahead!

It feels like now matter what they do, they're going slower and slower.

## 8.2 The DORA metrics

Sasha and Sarah are stumped, but new employee Sandy (they/them) has some ideas of what they can do differently. One day they approach Sasha in the hallway.



As they both stand in the hallway and Sandy starts to explain the DORA metrics, Sasha realizes that the whole team could really benefit from what Sandy knows, and asks Sandy if they'd mind giving a presentation to the company.

Sandy eagerly puts some slides together and gives everyone a quick introduction to the DORA metrics:

<p><b>Origin of the DORA metrics</b></p> <p>The DevOps Research and Assessment (DORA) team created the DORA metrics from nearly a decade of research</p>	<p><b>What are the DORA metrics ?</b></p> <p>The DORA metrics are 4 key metrics that measure the performance of a software team</p>
<p><b>The DORA metrics: Velocity</b></p> <p><b>Velocity</b> is measured by 2 metrics:</p> <ul style="list-style-type: none"> <li>• DeploymentFrequency</li> <li>• Lead Time for Changes</li> </ul>	<p><b>The DORA metrics: Stability</b></p> <p><b>Stability</b> is measured by 2 metrics:</p> <ul style="list-style-type: none"> <li>• Change Failure Rate</li> <li>• Time to Restore Service</li> </ul>



### 8.3 Velocity at Watch Me Watch

After her presentation on the DORA metrics, Sandy continues to discuss them with Sarah and Sasha, and how they can help with the problems Watch Me Watch is facing around how slowly they are moving.

Sandy suggests they focus on the two velocity related DORA metrics and measure these metrics for Watch Me Watch.

Wondering about the other 2 DORA metrics? We'll be looking at them in more detail in Chapter 10 when we talk about deploying.

#### The DORA metrics: Velocity

Velocity is measured by 2 metrics:

- Deployment Frequency
- Lead Time for Changes

In order to measure these, we need to look at them in a bit more detail.

- **Deployment Frequency** measures how often an organization successfully releases to production
- **Lead Time for Changes** measures the amount of time it takes a commit to get into production

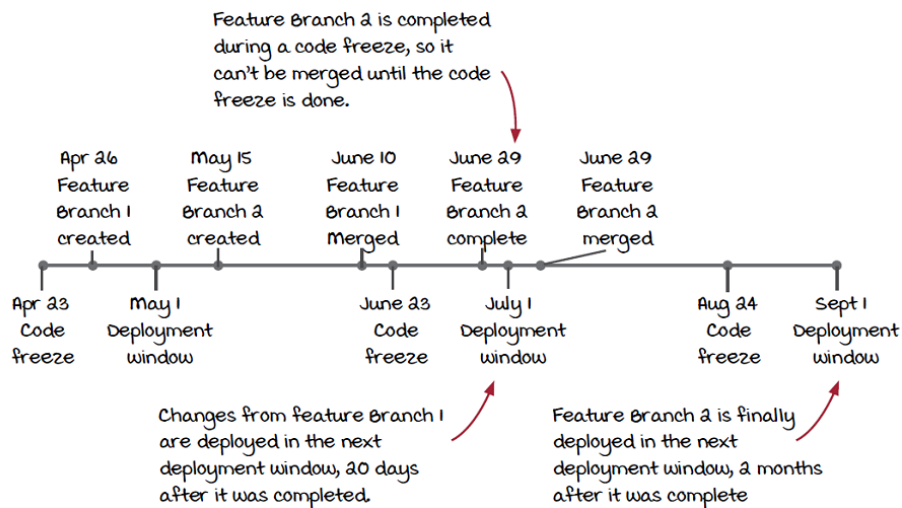
At Watch Me Watch, deployments can only occur as frequently as the deployment windows, which are every two months. So for Watch Me Watch, the **Deployment Frequency** is once every 2 months.

*production* refers to the environment where you make your software available to your customers, v.s intermediate environments you might use for other purposes, such as testing.

## 8.4 Lead time for changes

To measure Lead Time for Changes Sandy needs to understand a bit about the development process at Watch Me Watch. Most features are created in a feature branch, and that branch is merged back into main when development has finished on the feature. Some features can be completed in as little as a week, but most take at least a few weeks.

Here is what this process looks like for a two recent features, which were developed in Feature Branch 1 and Feature Branch 2:



The lead time for the changes in Feature Branch 1 was 20 days. Even though Feature Branch 2 was completed immediately before a deployment window, this was during the code freeze window so it couldn't be merged until after that, delaying the deployment until the next deployment window, two months later. This made the lead time for the changes in Feature Branch 2 two months, or around 60 days.

Looking across the last year worth of features and feature branches, Sandy finds that the average lead time for changes is around 45 days.



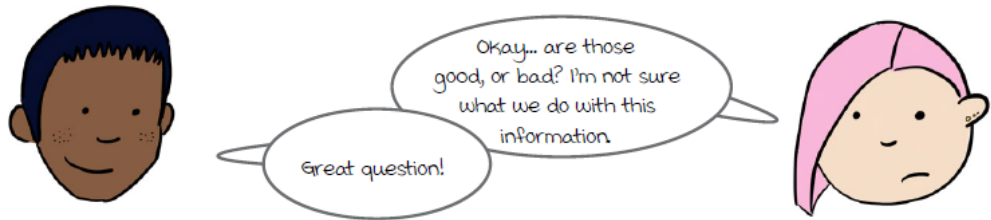
### Vocab time

*feature branching* is a branching policy where every time development starts on a new feature, a new branch (called a *feature branch*) is created. Development on this feature in this separate branch continues until the feature is completed, at which point it is considered ready to be merged back into the main codebase.

## 8.5 Watch Me Watch and elite performers

Sandy has measured the two velocity related DORA metrics for Watch Me Watch:

- **Deployment Frequency:** once every 2 months
- **Lead Time for Changes:** 45 days



Looking at these values in isolation, it's hard to draw any conclusions or take away anything actionable. As part of determining these metrics, the DORA team also ranked the teams they were measuring in terms of overall performance and put them into four buckets: low, medium, high and elite performing teams. For each metric, they reported what that metric looked like for teams in each bucket.

For the the velocity metrics, the breakdown (from the 2021 report) looked like this:

Metric	Elite	High	Medium	Low
Deployment Frequency	Multiple times a day	Once per week to once per month	Once per month to once every six months	Fewer than once every six months
Lead Time for Changes	Less than an hour	One day to one week	One month to six months	More than six months

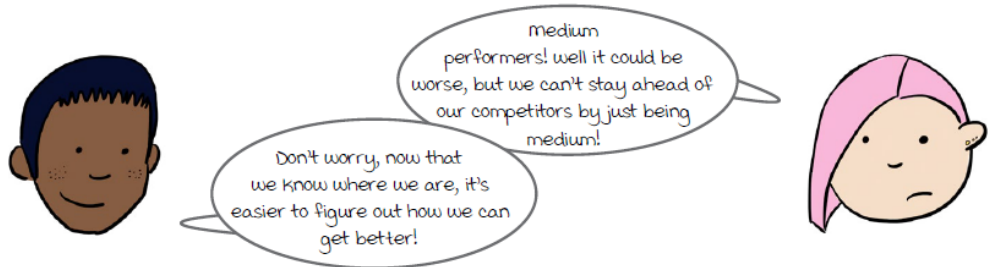
On the elite end of the spectrum, multiple deployments happen every day and the lead time for changes is less than an hour! On the other end, low performers deploy less frequently than once every six months, and changes take more than six months to get to production.

Comparing the metrics at Watch Me Watch with these values, they are solidly aligned with the medium performers.

### *What if we're between two buckets?*

The results reported by the DORA team are clustered such that there is a slight gap between buckets - this is based on the values they saw in the teams that they surveyed, and isn't meant to be an absolute guideline. If you find your values falling between buckets, it's up to you whether you want to consider yourself on the high end of the lower bucket or the low end of the higher bucket. It might be more interesting to step back and look at your values across all of the metrics to get an overall picture of your performance.

## 8.6 Increasing velocity at Watch Me Watch

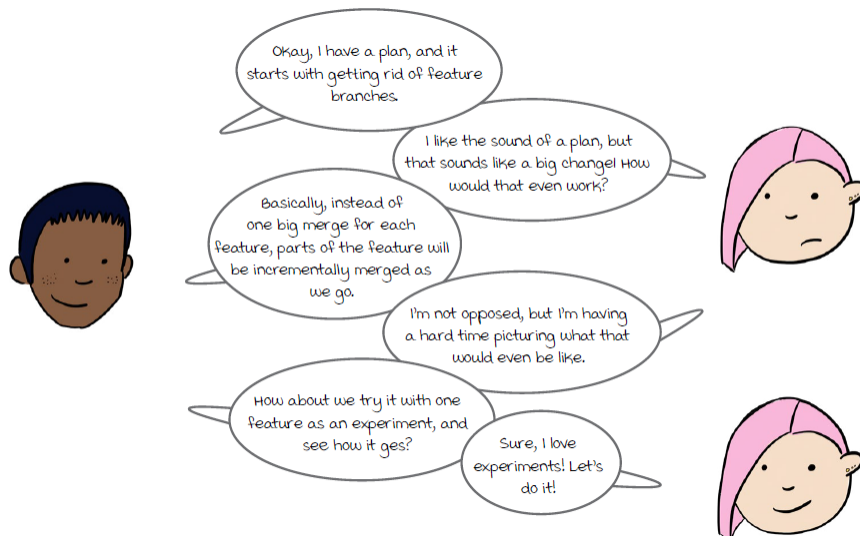


Sandy sets out to create a plan to improve the velocity at Watch Me Watch.

- **Deployment Frequency:** to move from being a medium performer to a high performer, they need to go from deploying once every 2 months to deploying at least once a month
- **Lead Time for Changes:** to move from being a medium to a high performer, they need to go from an average lead time of 45 days to one week or less

Their deployment frequency is currently determined by the fixed deployment windows they use, once every two month. And their lead time for changes is impacted by this as well: feature branches aren't merged until the entire feature is complete, and can only be merged between code freezes, and if the author misses a deployment window, their changes are delayed by two months until the next one.

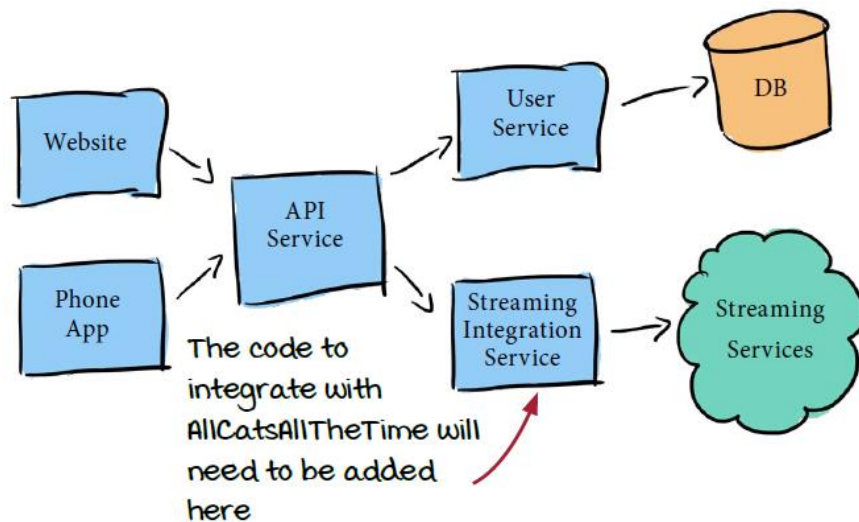
Sandy theorizes that both metrics are heavily influenced by the deployment windows (and the code freeze immediately before deployment), and made worse by the use of feature branches.



## 8.7 Integrating with AllCatsAllTheTime

To experiment with getting rid of feature branches, Sandy starts to work with Jan to try out this new approach for the next feature he's working on.

Jan has taken on integrating with the new streaming provider AllCatsAllTheTime (a streaming provider featuring curated cat related content). To understand the changes Jan will need to make, let's look again at the overall architecture of Watch Me Watch. Even though the company has grown since we last looked at their architecture, the original plans that Sasha and Sarah created have been working well for them, so the architecture hasn't changed:



Integrating AllCatsAllTheTime as a new streaming service provider means changing the Streaming Integration service. Inside the Streaming Integration service codebase, each integrated streaming service is implemented as a separate class, and is expected to inherit from the class `StreamingService`, implementing the following methods:

```
def GetCurrentlyWatching(self):  
    ...  
def GetWatchHistory(self, time_period):  
    ...  
def GetDetails(self, show_or_movie):  
    ...
```

// This interface enables most functionality that Watch Me Watch needs from streaming service providers: revealing what a user has been watching, and getting details for particular shows or movies the user has watched



## Vocab time

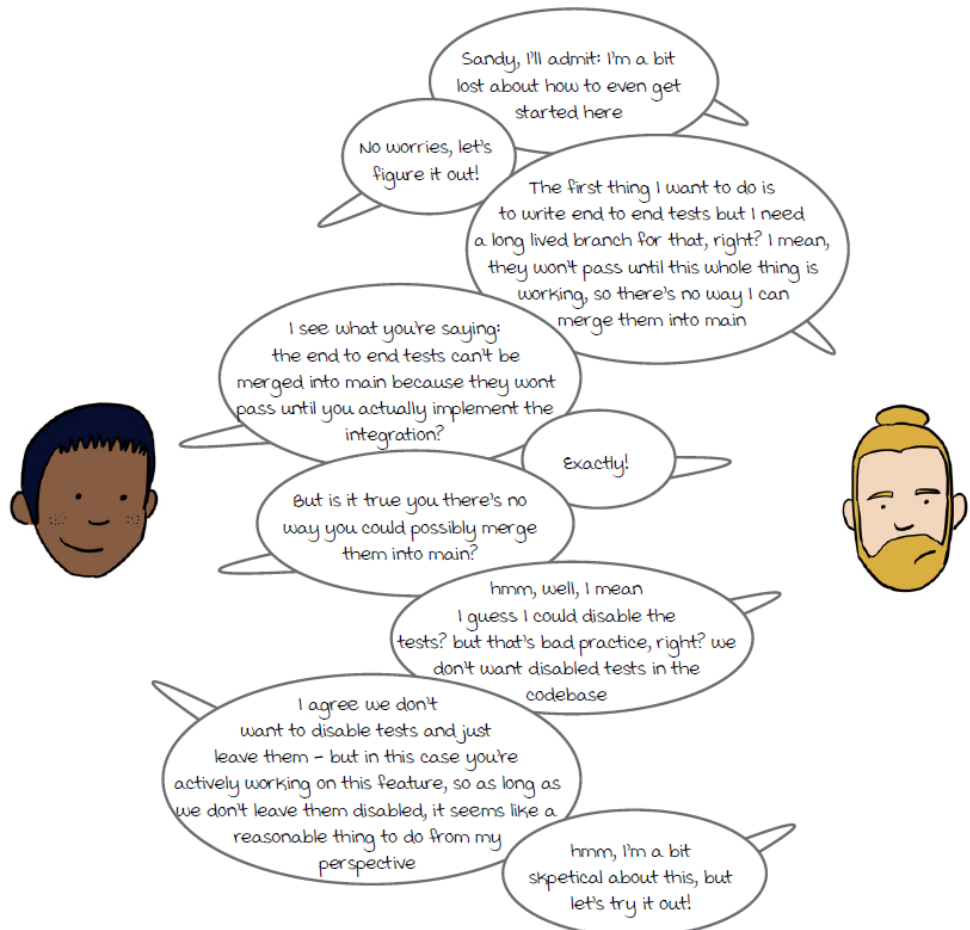
The approach Sandy is advocating for is often called *trunk based development*, where instead of relying on long lived branches, developers frequently merge back into the *trunk* of the repository, aka the main branch.

## 8.8 Incremental feature delivery

Sandy and Jan talk through how Jan would normally approach this feature:

1. Make a feature branch off of main
2. Start work on end to end tests
3. Fill in the skeleton of the new streaming service class, with tests
4. Start making each individual function work, with more tests and new classes
5. If he remembers, from time to time, he'll merge in changes from main
6. When it's all ready to go, merge the feature back into main

With the approach Sandy is suggesting, Jan will still create branches, but these branches will be merged back to main as quickly as possible, multiple times a day if he can. Since this is so different from how he usually works, they talk through how he's going to do this initially.



## 8.9 Committing skipped tests

Sandy has convince Jan that he can create his initial end to end tests, and even though they won't all pass until the feautre is done, he can commit them back to main as disabled tests. This will allow him to commit quickly back to main instead of keeping the tests in a long lived feature branch.

Jan creates his initial set of end to end tests for the new AllCatsAllTheTime integration. These tests will interact with the real AllCatsAllTheTime service, so he sets up a test account (WatchMeWatchTest01) and seeds the account with some viewing activity that his tests can interact with.

For example, this is one of the end to end tests that covers the `GetWatchHistory` method:

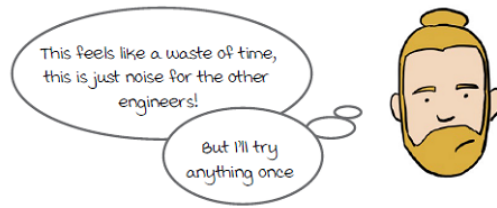
```
def test_get_watch_history(self):
    service = AllCatsAllTheTime(ACATT_TEST_USER)
    history = service.GetWatchHistory(ALL_TIME)

    self.assertEqual(len(history), 3)
    self.assertEqual(history[0].name, "Real Cats of NYC")
```

When he runs the tests, they of course fail, because he hasn't actually implemented any of the functions that the tests are calling. He feels very skeptical about it, but he does what Sandy suggested and disables the tests using `unittest.skip`, with a message explaining that the implementation is a work in progress. He includes a link to the issue for the AllCatsAllTheTime integration in their issue tracking system (#2387) so other engineers can find more information if they need to:

```
@unittest.skip("#2387) AllCatsAllTheTime integration WIP")
def test_get_watch_history(self):
```





***Jan is so skeptical! Is he a bad engineer?***

Absolutely not! It's natural to be skeptical when trying new things, especially if you've got a lot of experience doing things differently. The important thing is the fact that Jan is willing to try things out. In general, being willing to experiment and give new ideas a fair shot is the key element you need to make sure you and your team can keep growing and learning. And that doesn't mean everyone has to like every new idea right away.

## 8.10 Code review and “incomplete” code

How does taking an approach like this work with code review? Surely tiny incomplete commits like this are hard to review? Let’s see what happens!

Jan creates a Pull Request that contains his new skipped end to end tests and submits it for review. When another engineer from his team, Melissa, goes to review the PR, she’s understandably a bit confused, because she’s used to reviewing complete features. Her initial round of feedback reflects her confusion:



**Melissa**

Hey Jan, I’m not sure how to review this, it doesn’t seem like the pull request is complete? Are there maybe some files you forgot to add?

Up until this point, engineers working on Watch Me Watch have expected that a **complete** pull request includes a working feature, and all the supporting tests (all passing and none skipped) and documentation for that feature.

Getting used to a more incremental approach will mean redefining **complete**. Sandy lays some groundwork for how to move forward by redefining a **complete** pull request as a PR where:

- All code complies with linting checks
- Docstrings for incomplete functions explain why they are incomplete
- Each code change is supported by tests and documentation
- Disabled tests include an explanation and refer to a tracking issue

Sandy and Jan meet with Melissa and the rest of the team to explain what they are trying to do and share their new definition of complete. After the meeting Melissa goes back to the PR and leaves some new feedback.



**Melissa**

Okay I think I get it now! With this new incremental approach, I think the only thing missing is an update to our streaming service integration docs?

Jan realizes Melissa is right: he’s added tests but the documentation in the repo that explains their streaming service integrations hasn’t been updated, so he adds a change to the PR to add some very cursory initial docs:

```
* AllCatsAllTheTime - (#2387) a WIP integration with the provider of cat related content
```

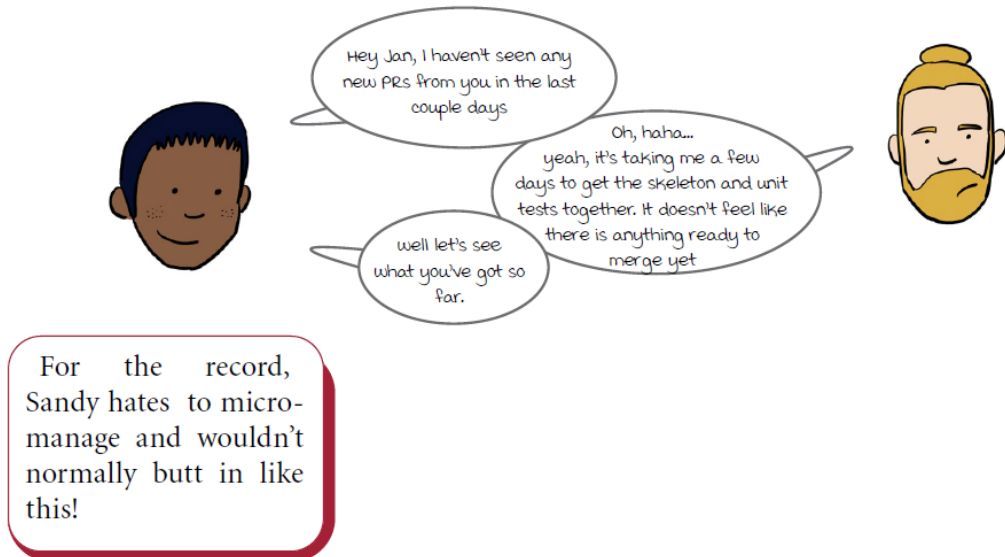
Melissa approves the changes and the disabled end to end tests are merged.

## 8.11 Keeping up the momentum

Jan's merged his initial (disabled) end to end tests. What's next? Jan's still taking the same approach he would to implementing a new feature, but without a dedicated feature branch:

1. ~~Make a feature branch off of main~~ (not using feature branches)
2. ~~Start work on end to end tests~~ (done, merged to main)
3. Fill in the skeleton of the new streaming service class, with tests (The next step)
4. Start making each individual function work, with more tests and new classes
5. If he remembers, from time to time, he'll merge in changes from main
6. When it's all ready to go, merge the feature back into main

Jan's next step is to start working on implementing the skeleton of the new streaming service and associated unit tests. After a couple days of work, Sandy checks in:



## 8.12 Committing work in progress code

So far Jan has some initial methods for the class `AllCatsAllTheTime`:

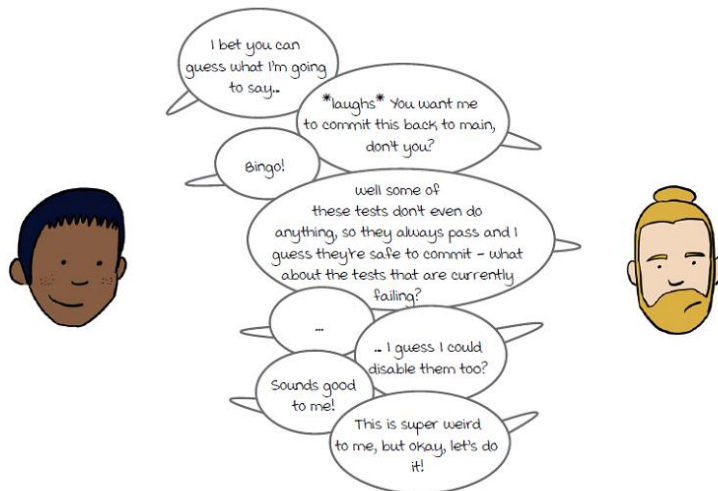
```
class AllCatsAllTheTime(StreamingService):
    def __init__(self, user):
        super().__init__(user)

    def GetCurrentlyWatching(self):
        """Get shows/movies AllCatsAllTheTime considers self.user to be watching"""
        return []

    def GetWatchHistory(self, time_period):
        """Get shows/movies AllCatsAllTheTime recorded self.user to have watched"""
        return []

    def GetDetails(self, show_or_movie):
        """Get all attributes of the show/movie as stored by AllCatsAllTheTime"""
        return {}
```

He's also created unit tests for `GetDetails` (which fail because nothing is implemented yet) and he has some initial unit tests for the other functions which are totally empty and always pass. He shows this work to Sandy and she has some feedback:



### *Shouldn't Jan have more to show for several days of work?*

Maybe (also maybe not, creating mocks and getting unit tests working can be a lot of work). But the real reason we're keeping these examples short is so we can fit them into the chapter - and the idea being demonstrated holds true even for these small examples, i.e. to get used to making small frequent commits, even commits as small as the ones Jan will be making.

## 8.13 Reviewing work in progress code

Jan opens a pull request with his changes: the empty skeleton of the new class, a disabled failing unit test, and several unit tests that do nothing but pass. However by this point Melissa understands why so much of the PR is in progress and isn't phased. She immediately comes back with some feedback:



**Melissa**

Can we include some more documentation? The auto generated docs are going to pick up this new class and all the docstrings are pretty much empty.

Jan is pleasantly surprised that a pull request with so little content can get useful feedback. He starts filling in docstrings for the empty functions, describing what they are intended to do, and what they currently do, for example he adds this docstring for the method `GetWatchHistory` in the new class `AllCatsAllTheTime`:

```
def GetWatchHistory(self, time_period):
    """
    Get shows/movies AllCatsAllTheTime recorded self.user to have watched

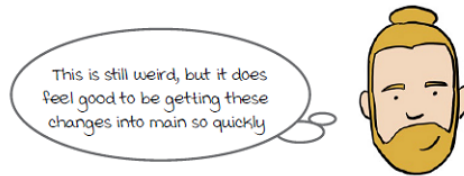
    AllCatsAllTime will hold the complete history of all shows and movies
    watched by a user from the time they sign up until the current time,
    so this function can return anywhere from 0 results to a list of
    unbounded length.

    The AllCatsAllTheTime integration is a work in progress (#2387) so
    currently this function does nothing and always returns an empty list.

    :param time_period: Either a value of ALL_TIME to return the complete
        watch history or an instance of TimePeriod which specifies the start
        and end datetimes to retrieve the history for
    :returns: A list of Show objects, one for each currently being watched
    """
    return []
```

At Watch Me Watch,  
docstrings are in  
reStructuredText  
format.

Once Jan updates the PR with the docstrings, Melissa approves it and it's merged into main.



***Isn't this a waste of time for Melissa? Reviewing all these incomplete changes?***

Short answer: no! It's much easier for Melissa to review these tiny pull requests than it is to review a giant feature branch! Also she can spend more time reviewing the interfaces (e.g. method signatures) and give feedback on them early, before they're full fleshed out. Making changes to code before it is written is easier than making the changes after!

## 8.14 Meanwhile, back at the end to end tests

Meanwhile, unbeknownst to Jan, Sandy and Melissa, other code changes are brewing in the repo!

Jan creates a new branch to start on his next phase of work, and when he opens the end to end tests, and the skeleton service he's been working on so far, he's surprised to see new changes to the code that he's already committed - changes made by someone else!

In the end to end test, he notices the call to `AllCatsAllTheTime.GetWatchHistory` has some new arguments:

```
def GetWatchHistory(self, time_period, max, index):      #A
    ...
    :param time_period: Either a value of ALL_TIME to return the complete
        watch history or an instance of TimePeriod which specifies the start
        and end datetimes to retrieve the history for
    :param max: The maximum number of results to return
    :param index: The index into the total number of results from which to
        return up to max results
    ...
```

#A Arguments have been added to `GetWatchHistory` to support paginating the results

These new arguments have been added to the skeleton service as well:

```
def GetWatchHistory(self, time_period, max, index):
    return []
```

And there are even a couple of new unit tests:

```
def test_get_watch_history_paginated_first_page(self):
    service = AllCatsAllTheTime(ACATT_TEST_USER)
    history = service.GetWatchHistory(ALL_TIME, 2, 0)
    # TODO(#2387) assert that the first page of results is returned      #A

def test_get_watch_history_paginated_last_page(self):
    service = AllCatsAllTheTime(ACATT_TEST_USER)
    history = service.GetWatchHistory(ALL_TIME, 2, 1)
    # TODO(#2387) assert that the first page of results is returned
```

#A These tests always pass because their bodies haven't been filled in, but the author has indicated what needs to be done

Looking at the history of the changes, Jan sees that Louis merged a PR the day before that added pagination to **GetWatchHistory** for all streaming services - and he notices he has a chat message from Louis as well:

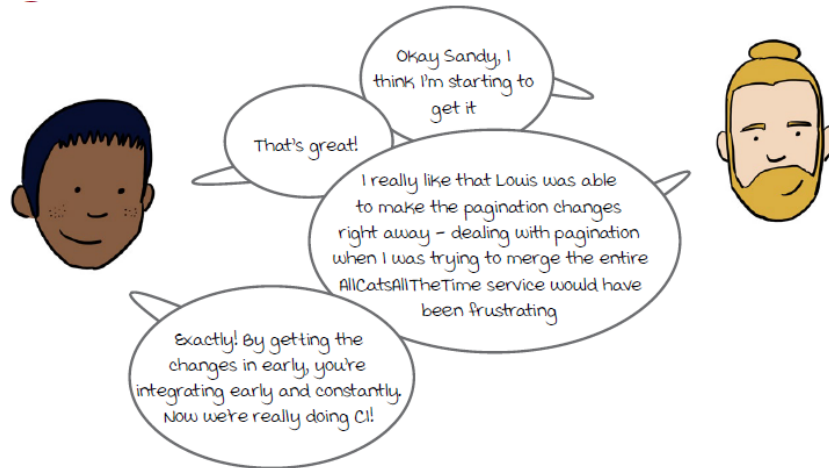
**Louis**

Hey thanks for merging `AllCatsAllTheTime` early! I was worried about how I was going to make sure that any in progress integrations were updated for pagination as well, I didn't want to cause problems for you at merge time. It's great to be able to get these updates in right away

Because Jan merged his code early, Louis was able to contribute to it right away. If Jan had kept this code in a feature branch, Louis wouldn't have known about `AllCatsAllTheTime`, and Jan wouldn't have known about the pagination changes. When he finally went to merge those changes in, weeks or even months later, he'd have to deal with the conflict with Louis's changes. But this way, Louis dealt with them right away!



## 8.15 Seeing the benefits



In this chapter we're starting to move beyond Continuous Integration (CI) to the processes that happen after the fact (i.e. the rest of Continuous Delivery), but the truth is that the line is blurry, and choices your team makes in CI processes have downstream ripple impacts on the entire Continuous Delivery process.

Although Sandy's overall goal is to improve velocity, as they just pointed out to Jan, taking the incremental approach Sandy means that their CI processes are now much closer to the ideal. What is that? Let's look briefly back on the definition of **Continuous Integration (CI)**:

The process of combining code changes frequently, where each change is verified on check in.

With long lived feature branches, code changes are only combined as frequently as the feature branches are brought back to main. But by committing back to main as often as he can, Jan is combining his code changes with the content of main (and enabling other developers to combine their changes with his) frequently instead!



### Takeaway

Improving deployment often means improving CI first.



### Takeaway

Avoiding long lived feature branches and taking a more incremental approach, with frequent merges back to main not only improves Continuous Delivery overall but is better Continuous Integration as well.

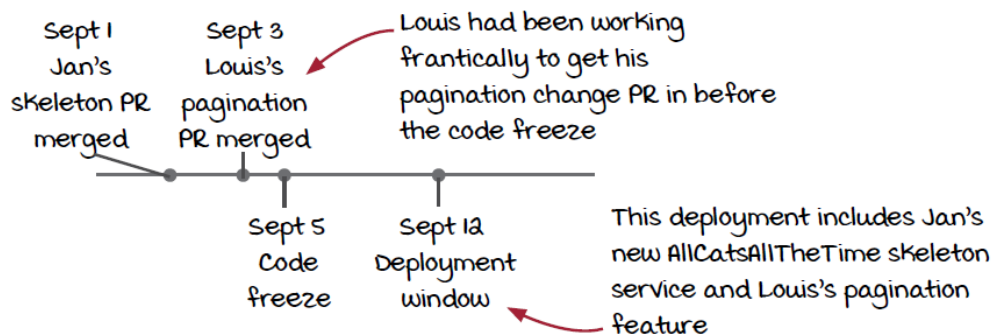
## 8.16 Decreasing lead time for changes

By getting closer to the Continuous Integration ideal, Sandy and Jan are having a direct impact on the entire CD process, and specifically they are having a positive impact on Watch Me Watch's DORA metrics. Remember Sandy's goals

- **Deployment Frequency:** move from being a medium to high performer by going from deploying once every 2 months to deploying at least once a month
- **Lead Time for Changes:** move from being a medium to a high performer by going from an average lead time of 45 days to one week or less

Jan's most recent PR (including a skeleton of the new streaming class and some WIP unit tests) was only a couple of days before a code freeze and the subsequent deployment window. The result is that Jan's new integration code actually made it to production as part of that deployment.

Of course the new integration code doesn't do actually anything yet, but the fact is that the changes Jan is making are making it into production. Sandy takes a look at the lead time for these changes:



Jan merged the skeleton class 4 days before the code freeze. 2 days before the code freeze Louis updated `GetWatchHistory` to take pagination arguments. The code freeze started 2 days later, and 1 week after that there was a deployment.

The entire lead time for the skeleton class change starts when Jan merged on Sept 1 and ends with the deployment on Sept 12, for a total of an 11 day lead time.

Let's compare that to the lead time for the changes Louis was working on. He'd been working in a feature branch since before the last deployment window, which was July 12. He'd started on July 8, so the entire lead time for his changes was from July 8 to Sept 12, or 66 days.

While Jan's changes are incremental (and currently not functional), Jan was able to reduce the lead time for each individual change to 11 days, while Louis's changes had to wait 66 days.

## 8.17 Continuing AllCatsAllTheTime

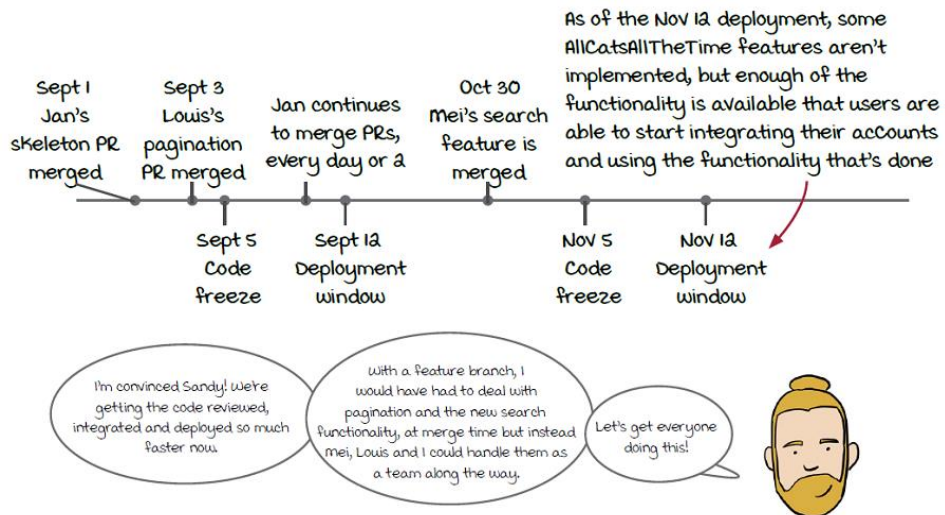
Jan continued to work with Sandy to use an incremental approach to implementing the rest of the AllCatsAllTheTime integration. He worked method by method, implementing the method, fleshing out the unit tests and enabling end to end tests as he went.

A few weeks into the work, another team member (Mei) who is working on a search feature adds a new method was added to the `StreamingService` interface:

```
class StreamingService:
    ...
    @staticmethod
    def Search(show_or_movie):
        pass
```

This new method will allow users to search for specific movies and shows across streaming providers, and the author of the change adds the new method to every existing streaming service integration. Since Jan has been incrementally committing the AllCatsAllTheTime class as he goes, Mei is able to add the Search method to the existing AllCatsAllTheTime class - she doesn't even need to tell Jan about the change at all! One day Jan creates a new branch to start work on the `GetDetails` method and he sees the code that Mei has added.

That's two major features that have been integrated with Jan's changes as he developed (pagination and search) that normally Jan would have to deal with at merge time with his normal feature branch approach. In addition, after the next deployment (Nov 12), even though the integration isn't complete, enough functionality is present for users to actually start using it and for marketing to start advertising the integration.



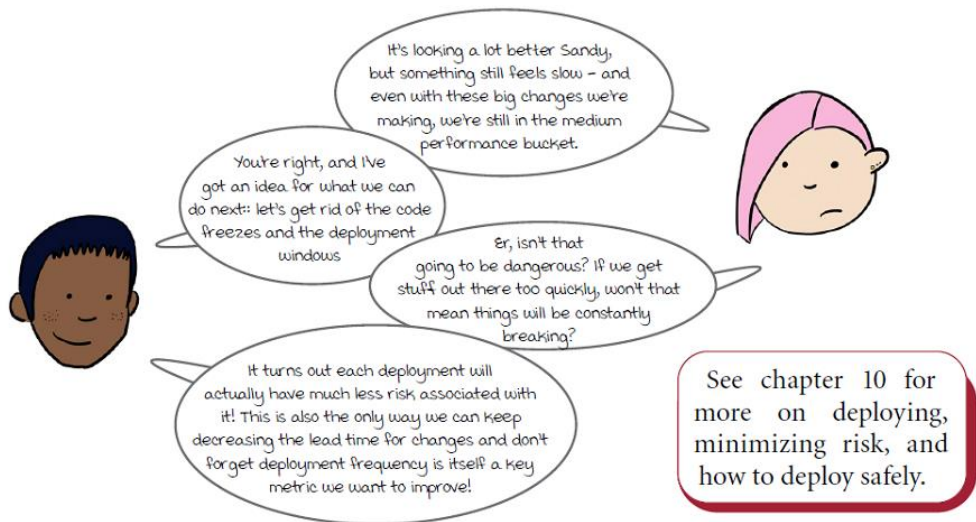
## 8.18 Deployment windows and code freezes

Sandy and Jan present the results of their experiment back to Sasha and Sarah. They show how by avoiding long lived feature branches and merging features incrementally they've encountered multiple benefits:

- The lead time for changes is decreased
- Multiple features can be integrated sooner and more easily
- Users can get access to features earlier

Sasha and Sarah agree to try this policy across the company and see what happens, so Sandy and Jan set about training the rest of the developers in using how to avoid feature branches and use an incremental approach.

A few months later, Sandy revisits the lead time metrics for all the changes to see how they've improved. The average lead time has decreased significantly, from 45 days down to 18 days. Individual changes are making it into main faster, but they still get blocked by the code freeze, and if they are merged soon after a deployment, they have to wait nearly 2 months to make it into the next deployment. While the metric has improved, it still falls short of Sandy's goal to upgrade their lead time for changes from being aligned with DORA medium performers to high performers (1 week or less).



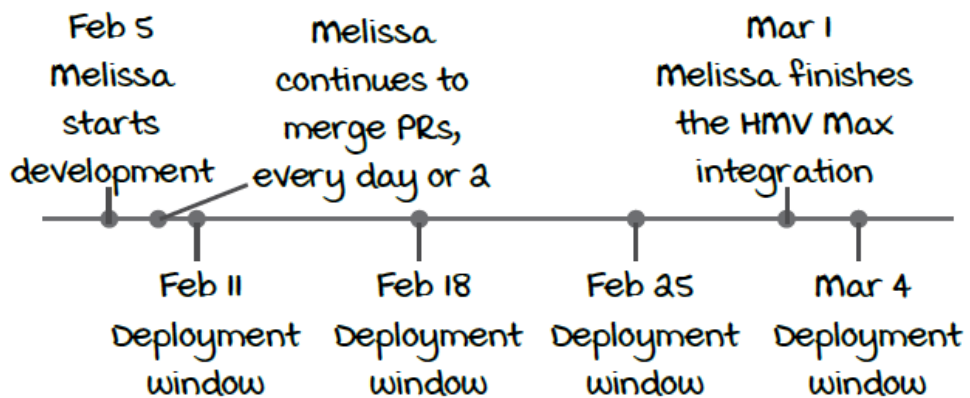
They discuss a plan and agree to try doing weekly deployments and to remove the code freeze entirely.

## 8.19 Increased velocity

Sandy keeps track of metrics for the next few months and observes feature development to see if things are speeding up and where their DORA metrics land without code freezes and with more frequent deployments.

Melissa works on integration with a new streaming provider, HMV Max (Home Movie Theatre Max):

The integration takes her about 5 weeks to completely implement, and during that time there are 4 deployments, each of which includes some of her changes.



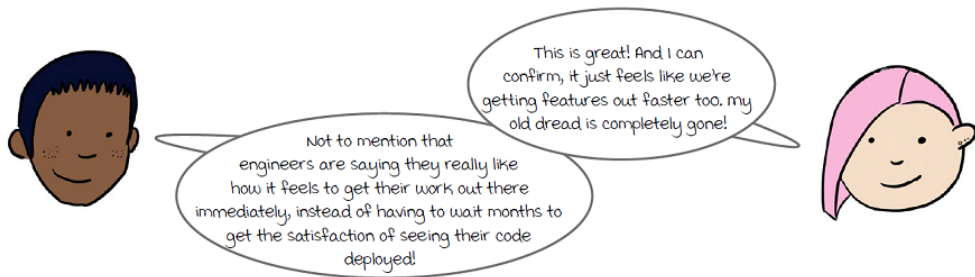
The lead time for Melissa's changes is a maximum of 5 days and some changes are deployed as quickly as 1 day after merge.

Sandy looks at the stats overall and finds that the maximum lead time for changes is 8 days, but this is very rare since most engineers have gotten into the habit of merging back into main every day or two. The averages are:

- **Deployment Frequency:** once a week
- **Lead Time for Changes:** 4 days

Getting to high performance with this metric was as easy as changing the intervals between deployment windows

Sandy has accomplished their goal: as far as velocity is concerned, Watch me Watch is now aligned with the DORA high performers!



Sasha, Sarah and Sandy also wonder how they can move beyond being high performers to being elite performers, but we'll save that for chapter 10!

## 8.20 Conclusion

Watch Me Watch had introduced code freezes and infrequent deployment windows with the hope of making development safer, however it mostly just made development slow. By looking at their processes through the lens of the DORA metrics, specifically the velocity related metrics, they were able to chart a path toward moving more quickly.

Moving away from long lived feature branches, removing code freezes and increasing deployment frequency directly improved their DORA metrics, and rescued the company from the feeling that features were taking longer and longer, allowing their competition to get ahead of them. Not to mention, the engineers realized this was a more satisfying way to work!

## 8.21 Summary

- The DevOps Research and Assessment (DORA) team has identified 4 key metrics to measure software team performance and correlated these with elite, high, medium, and low performance
- Deployment frequency is one of two velocity related DORA metrics which measures how frequently deployments to production occur
- Lead time for changes is the other velocity related DORA metric, measuring the time from which a change has been completed to when it gets to production
- Decreasing lead time for changes requires revisiting and improving Continuous Integration practices. The better your CI, the better your lead time for changes
- Improving the Continuous Delivery practices beyond CI often means revisiting CI as well
- Deployment frequency has a direct impact on lead time for changes; increasing deployment frequency will likely decrease lead time for changes

## 8.22 Up next . . .

In the next chapter we'll examine the main transformation that happens to source code in a CD pipeline: building that source code into the final artifact that will be released and/or deployed.