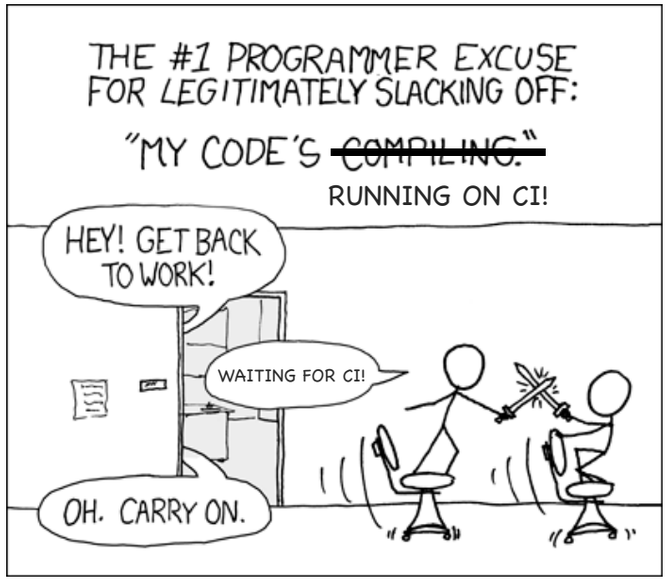
Speed up your test suite by throwing computers at it

RailsConf 2021
April 12-15

Daniel Magliola
@dmagliola

It was the best of times, it was the worst of times. It was the age of "Waiting for CI".



https://xkcd.com/303/

@dmagliola

which takes an age... and is of my least favourite things to do.

Partly because i'm impatient and bad at multi-tasking, but mainly, because if you think about how much time you spend waiting on CI every day, and you multiply that by all the people in your team... That's a lot of time we could be using better, enough that it makes sense to put in some real work to make it faster, and today I'd like to tell you about a few techniques I've used in the past that have given me great results.

First of all though, hi.

My name is Daniel and I learned all of this working for GoCardless, a payments company based in London.
As you probably noticed though, I'm not originally from London, I come from Argentina, so in case you were wondering, that's the accent.

So, I want to help you make your CI finish faster. And as the title of the talk says, I'm proposing that you do this by throwing lots of computers at the problem.

So I'm not going to talk about how to make INDIVIDUAL tests run faster. There are lots of resources out there already, you know, using fixtures instead of factories, mocking stuff, lots of techniques shared by people that can explain them way better than me. The problem with these techniques is that they normally involve rewriting your tests, and they normally take an amount of work that grows linearly with the number of tests you have. And your test suite is probably huge, or you wouldn't be looking at my face right now, so that's not very fun to do.

What I want to talk about is how to reduce the *total* runtime of your CI *test suite*, with a focus on getting the most impact for the time you invest.

And we do that not by making your tests faster...

But by running them on a lot of computers at the same time.

What I want to focus on is in making some systemic changes that'll let your tests still run slowly, and let your team still write tests the way they're used to, but when you're running in CI, you'll run massively in parallel, so you can still finish quickly.

Now, of course, what I'm advocating for here is throwing money at the problem in exchange for saving developer time. This is not always the appropriate way, but for a lot of companies out there, when you have an engineer, with a typical engineer's salary, waiting on CI and getting distracted by HackerNews, there are lots of situations where it makes economic sense to

spend as much money as your CI provider will take from you

And to be honest it's not even that much, we're using tons of machines, and spending on the order of less than $100/developer/month in total.

Now before I start, I want to make a couple notes.

Not sponsored by any CI provider

First of all, you're going to see a bunch of CircleCI on this talk. Mostly on all my screenshots, and I'll also mention a tool or two they provide. This is because we happened to used CircleCI at GoCardless.

This is not an endorsement in any way, I just happen to have most experience with them because of my day-to-day work, and it was the easiest way to get screenshots of complex CI setups. And I also have my fair share of frustrations with them, I'm not trying to recommend them particularly... I just happen to have used them a lot.

Not sponsored by any CI provider

Ideas not specific to any particular CI provider

@dmagliola

And importantly, this not a Circle-specific talk. Everything that I'm going to talk about is around optimizing things that *any* CI provider will have to do, so it will be applicable to pretty much all platforms.

The same thing goes every time I say "rspec". I'm using rspec as an example out of habit, but **almost** everything i'll be talking about will still work with minitest, or any other test framework, in Ruby or in other languages.

I've actually used a lot of this same advice for a PHP project we had. The specifics of the projects could not have been more different, but the thinking behind what i'll share applies to pretty much all languages, and that's what will help you speed things up.

Another thing to keep in mind is that I'm going to show a bunch of code to explain these techniques. This code will be oversimplified to explain the concepts and some of it will move pretty fast. But the talk comes with an supplementary Github repo, where you can find fuller code examples, more documentation on how they work, and you can grab them from there as a starting point for your project.

Unfortunately, most of them you won't be able to just grab and use, you'll probably have to adapt them to your needs, but I've tried to document where you need to adapt them.

Not a step-by-step roadmap

Your mileage WILL vary. Unfortunately :(

More importantly, this is not a "do these 3 things and you'll get this exact result" roadmap kind of talk. Your mileage WILL vary.
Your CI setup will be different from mine. The specific things that are slow for you are different from everyone else's.

I'm going to share a bunch of techniques, some of which will be super relevant, some not as much. This talk is mostly a way to think about the problem of CI times, and a bunch of tools and techniques to help you improve it. But you will need to test these, see which ones help, which don't, and tailor them to your specific situations.

I mentioned a PHP project we had in addition to our Ruby ones. We drastically improved runtimes on both using these ideas, but some things that made a massive difference to one, didn't move the needle at all on the other one, and vice-versa. You'll have to adapt these to your needs.

It'll take some work, but it'll be worth it when you no longer need to wait on CI for ages.
So, measure, experiment, see what works for you!

How do we parallelism?

Alright, with that out of the way, let's get to it...
We want to make our CI suite finish faster, and we're going to do that by running things in parallel.

First things first, how do we even do this? How do we parallelize our tests?

How to parallelize your CI suite

- Manually, along logical boundaries
- Automatically, within each boundary

@dmagliola

There are 2 main ways:

You can manually split off sections of your test suite into "chunks" that make logical sense, and have each chunk run in parallel.
And you can then take each chunk and automatically split it into many machines to also run it in parallel.

This is not an either/or choice, you'll most likely want to do both.

How to parallelize your CI suite

- Manually, along logical boundaries
- Automatically, within each boundary

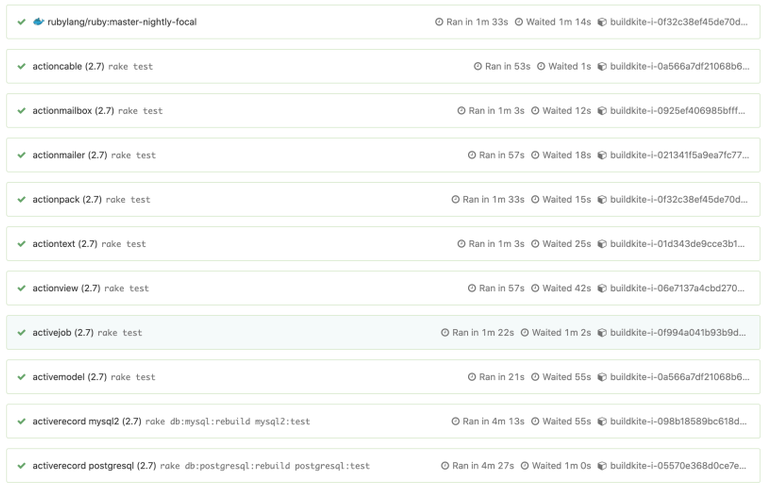
@dmagliola

Let's start with the simplest one. And it's very likely that you're already doing this, but it's worth talking about.

If you can separate your test suite into different pieces that make sense, which will likely be different sub-directories within your "specs" directory, you can create different "CI jobs", calling rspec on each of the different directories for those tests.

How to parallelize your CI suite

First, Manually



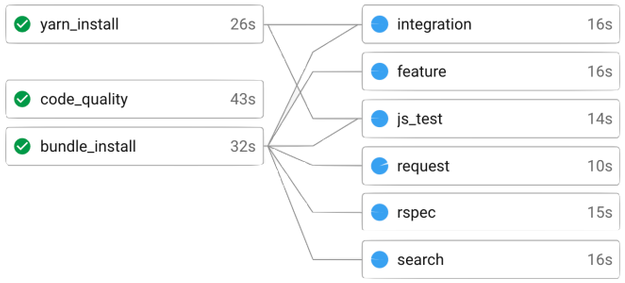
@dmagliola

Rails does this beautifully, for example, if you look at their test suite, you'll see that they have separate jobs for activemodel, activerecord, actioncable, etc, etc. Each of these is a coherent logical unit, easy to understand when looking at the CI setup, and they all run in parallel. If your app is super modular like this, this is a great win. Very easy starting point.

In most Rails apps, you'll have "integration", "models", "controllers", etc. This is not as granular, and typically one of these will take a lot longer to run than the others, but it's still probably worth separating, and it's a good starting point.

How to parallelize your CI suite

First, Manually



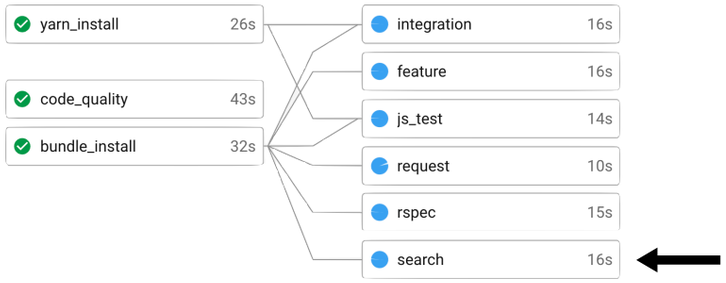
Two things that you should keep in mind:

One advantage of separating these is that you get more granular control over the running of each job. For example, some CI providers will let you choose between different instance sizes, which obviously cost different amounts of money per minute. Sometimes, for certain types of tests, particularly integration ones, you may need a bigger instance to fit all your dependencies in RAM.

If you separate these tests out in their own job, you can pay for larger machines only for the parts of the test suite that needs them, without making the rest more expensive.

How to parallelize your CI suite

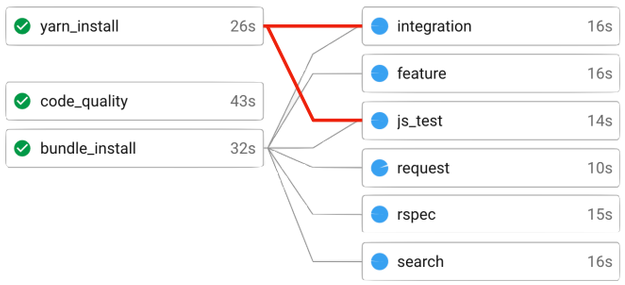
First, Manually



This is our setup for example, and you can see we separated Search tests. These would normally live under Integration, but they had more dependencies than the others, and needed bigger machines so we split them out.

How to parallelize your CI suite

First, Manually



The other thing you can do is control dependencies between jobs better. For example: Your Javascript tests will need to wait until your Node modules get installed... But your Model and Unit tests probably don't.

So if you separate your JS tests from your Unit tests, you don't need to wait for YARN to finish before your UNIT tests start, and they can start sooner. Same goes for our Search example. Most of our tests don't need ElasticSearch to be running, so they don't need to wait for it to boot up.

How to parallelize your CI suite

First, Manually

Ideally don't do this:

```
- rspec spec/models
- rspec spec/controllers
- rspec spec/integration
```

@dmagliola

The second thing you want to do is a bit less obvious.

You want to have a "catchall" job. If you go very granular on this splitting approach, it's very easy to later add a new tests sub-directory, and forget to add the new CI jobs for it.

If you do this, and you later add a new directory "spec/new_tests", you'll likely forget to add a new CI job for it, your new tests won't run in CI, and you will never notice. Which is pretty sad. And dangerous.

How to parallelize your CI suite

First, Manually

Do this instead:

```
- rspec spec/models
- rspec spec/controllers
- ~~rspec spec/integration~~
- rspec $(find spec/**/* | grep -v "spec/models" | grep -v "spec/controllers")
```

@dmagliola

What you want to do instead is have a final "catchall" job. Instead of targeting a specific sub-directory to run, you want to find *all* of your tests, and filter out the ones that are already run by other jobs. Using "find" and "grep -v" lets you do this easily.

Now, i'm not going to lie, this is ugly. I get it. But it fails safe.

The caveat is obviously that need to remember to add an exception here if you later add a new job for some sub-directory...
But now the penalty for forgetting that is you that run *some* tests twice, which is way better than skipping an entire chunk of them! So I think that safety justifies the yuckiness.

How to parallelize your CI suite

- Manually, along logical boundaries ✅
- Automatically, within each boundary

@dmagliola

So that's splitting up your test suite manually, which again, I bet many of you are already doing, but it's worth keeping these caveats in mind.

How to parallelize your CI suite

- Manually, along logical boundaries
- Automatically, within each boundary

@dmagliola

The other way of parallelizing, and the one we're going to focus the most on today, is having multiple machines run a single set of tests, by automatically splitting the files between them.

How to parallelize your CI suite

Then, Automatically

Instead of:

```
rspec spec/unit
```

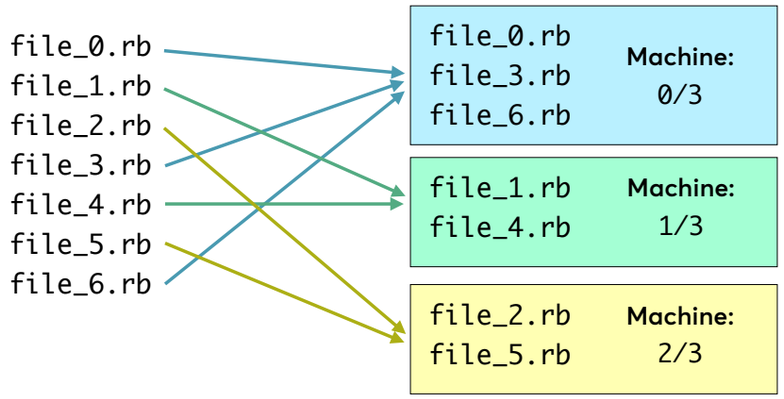
Do:

```
rspec $(find spec/unit -name "*_spec.rb")
```


The key part of how you do this is instead of giving your test runner a directory to run, you give it a list of all the files in that directory. You do this because once you have a list of many files, you can split it into chunks with a little magic. So you do this in many machines, each machine picks a different chunk of that list of files and runs only those, and in aggregate you've run all of your tests, but each of those chunks ran in parallel.

How to parallelize your CI suite

Then, Automatically



@dmagliola

In order to split this so that no two machines run the same file, and that all files get run, each machine needs to know two things:

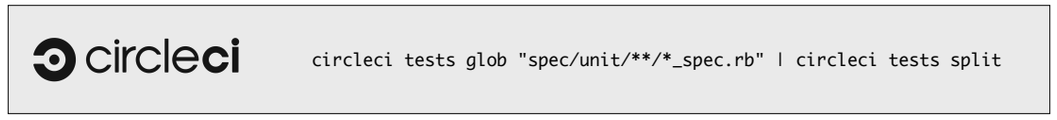
- Which machine it is
- How many total machines there are.

We can have each machine know these through 2 environment variables. And knowing this, each machine can take "every Nth file", with an offset at the beginning, and that basically does it.

Now of all the things i'm going to talk about today, this is the one where different CI providers vary the most.

How to parallelize your CI suite

Helpful CI Providers



CircleCI, for example, have this CLI tool that will help you find your test files and split them for you.
You specify a "parallelism" value for how many boxes to run, you run a command kind of like that, and it pretty much "just works"
And with a bit of work, it can also do smarter allocation based on historical times of each file, it's pretty cool.

How to parallelize your CI suite

Helpful CI Providers

circleci

```
circleci tests glob "spec/unit/**/*_spec.rb" | circleci tests split
```

CODESHIP

```
- type: parallel
  steps:
  - command: BOX_COUNT=2 BOX_INDEX=0 rspec $(...i'll explain soon...)
  - command: BOX_COUNT=2 BOX_INDEX=1 rspec $(...i'll explain soon...)
```


Codeship has a less ideal approach, it lets you specify a number of steps to run in parallel, and you can embed environment variables right there. This needs a bit more work from you, I'll show you in a minute how to use these.

How to parallelize your CI suite

Helpful CI Providers

circleci

```
circleci tests glob "spec/unit/**/*_spec.rb" | circleci tests split
```

CODESHIP

```
- type: parallel
  steps:
  - command: BOX_COUNT=2 BOX_INDEX=0 rspec $(...i'll explain soon...)
  - command: BOX_COUNT=2 BOX_INDEX=1 rspec $(...i'll explain soon...)
```

Buddy

```
rspec $BUDDY_SPLIT_1
```


Buddy has something similar to Circle, where they'll split the files for you ahead of time, and put the list of files in these env variables called "BUDDY SPLIT" that you can use directly.

Sadly, most providers don't have any tools to do this directly, but you can do it yourself by slightly abusing a feature they almost all have: the Build matrix

Most providers will give you a way to run the same job over and over, with slightly different parameters. They generally call it a Build Matrix. And the idea is that you can run the same suite of tests over and over with different versions of Ruby and different Gemfiles for different versions of Rails, for example, to make sure you're compatible with all of them.

How to parallelize your CI suite

Abusing the Build Matrix

```yaml
jobs:
  rspec_unit:
    strategy:
      matrix:
        os: ["ubuntu-latest", "macos-latest", "windows-latest"]
        ruby: [2.6, 2.7, 3.0]
        gemfile: ["Gemfile.rails-5.2.0", "Gemfile.rails-6.1.0"]
    runs-on: ${{ matrix.os }}
    env:
      GEMFILE:  ${{ matrix.gemfile }}

  steps:
    ...
```


This code above is from Github Actions, but they're all very similar.
Now some of those keywords mean something specific for each provider, like "os", or "rvm", etc.
But you can make up your own keywords, and then use those to set environment variables.

And THAT means we can abuse this feature, and bend it to our purposes.

How to parallelize your CI suite

Abusing the Build Matrix

```yaml
jobs:
  rspec_unit:
    strategy:
      matrix:
       box_index: [0, 1, 2, 3]
    env:
      BOX_INDEX:  ${{ matrix.box_index }}
      BOX_COUNT: 4
```


We can make up a "box index" in the matrix, and we use it to number all our boxes, and then we set these 2 environment variables, the ones I was talking about earlier. And if we do this, we'll now get 4 boxes, each box knows which one it is, and knows how many there are.

So with a little command line hackery...

How to parallelize your CI suite

Abusing the Build Matrix

```
find . | sort | awk "NR % $BOX_COUNT == $BOX_INDEX"
```

@dmagliola

you can get each box to pick up their part of the split as I was showing earlier. This script there is a simplified example, but what we're doing is taking all the respective files, and passing them to AWK line by line.

In AWK, that variable NR, tells you what row of the input you are in right now. We modulo that by the total count, compare it to our box number, and decide whether to proceed with this file, or discard it. And that gives you the split you need. TADA!

Now pay attention to that sort there. That's important to keep things consistent!

How to parallelize your CI suite

Abusing the Build Matrix

```
rspec $(find spec/unit -name "*_spec.rb" | sort | awk "NR % $BOX_COUNT == $BOX_INDEX")
```

@dmagliola

Again, that code is very simplified. In reality it looks a bit more like this. But doing this, you can split your tests between as many machines as you want, and parallelize like crazy.

Now again, I admit... This is a bit ugly...
But if your CI platform doesn't help you split your tests, this works!
And if you give it enough boxes, it'll speed up your tests massively!

Startup and Setup Times

@dmagliola

Ok, so everything so far has been pretty much an "introduction". I needed to explain *how* we run things in parallel, so we could get into the main part of this talk. Because now is when things get hard. And interesting...

Because in theory, there's no limit to how many boxes you could have. Give it 1000 boxes and your tests run almost instantly, right?

The truth is, if you do just this, you can improve your CI times a lot, but it won't be ideal. There's a wall you're going to hit pretty quickly, which is imposed by your startup times.

You see, when you run one of these boxes, your tests don't start running instantly. For most CI providers, these boxes are a container that has to be downloaded, started, then you need to run a bunch of setup tasks, and only *then* you're going to run your tests.

If you're not careful, you can easily spend 5 minutes doing this setup, and then you start getting very sharp diminishing returns for each extra box. Think about it, even with *infinite* machines, it'd still take 5 minutes to run you tests.

And also while I'm all for throwing money at the problem, if you are wasting 5 minutes for each of those boxes, you'll want a lot of boxes, and that's a pretty big money bonfire for you.

So you want to focus on these setup times, and make them as small as you possibly can.

Startup and Setup Times

A typical CI setup script

```
rspec:
  docker:
    - image: ruby:2.7.2
    - image: postgres
    - image: redis
  steps:
    - git checkout
    - bundle install
    - yarn install
    - rails db:create db:structure:load
    - rspec
```


Now, a typical CI config looks a bit like this.

That last step I highlighted over there, that's where we *actually* run our tests. But there's a lot of stuff that needs to happen beforehand!

Now, at the beginning of the talk I said I wouldn't talk about how to make your individual tests run faster. All of that work would focus exclusively on that last step. This is what we normally look at to try and make faster. Which makes sense, because it's what takes the longest, and it feels like we can control it.

But here's the thing. Here's how I see this list of steps...

Startup and Setup Times

A typical CI setup script

```
rspec:
  docker:
    - image: ruby:2.7.2
    - image: postgres
    - image: redis
  steps:
    - git checkout
    - bundle install
    - yarn install
    - rails db:create db:structure:load
    - rspec
```

Waste of time and money
🔥💸🔥

Actual Useful Work

@dmagliola

That last bit is what's actually doing the work we want. AND it's the part we can parallelize! So if it's taking longer, we can throw more computers at it.

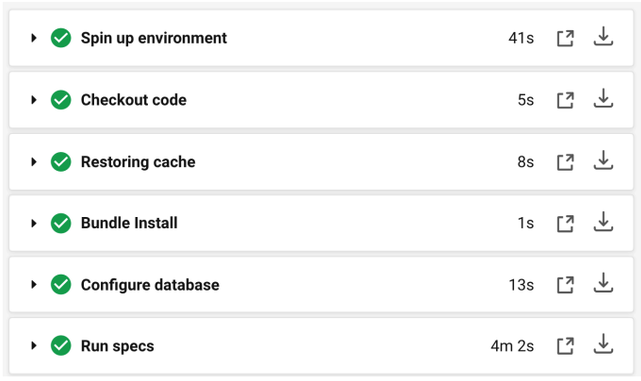
All the stuff that comes before it, that's a necessary evil, but it's waste. And if you have more computers, ALL of them need to do those steps anyway, so it doesn't parallelize at all.

Now the problem with those is that they don't LOOK like you can do anything about them.
Your tests... That's your code. You can change it. You can optimize it.
But a container will take as long to start as it takes to start. Bundle will just take as long as it takes, and you need those gems, right? And all those steps are necessary. It really doesn't LOOK like you can do much about it.

But in reality, we can get clever here, and there's A LOT we can do to start chipping away at those startup times to make this a WHOLE LOT better.

Startup and Setup Times

A typical CI setup script



@dmagliola

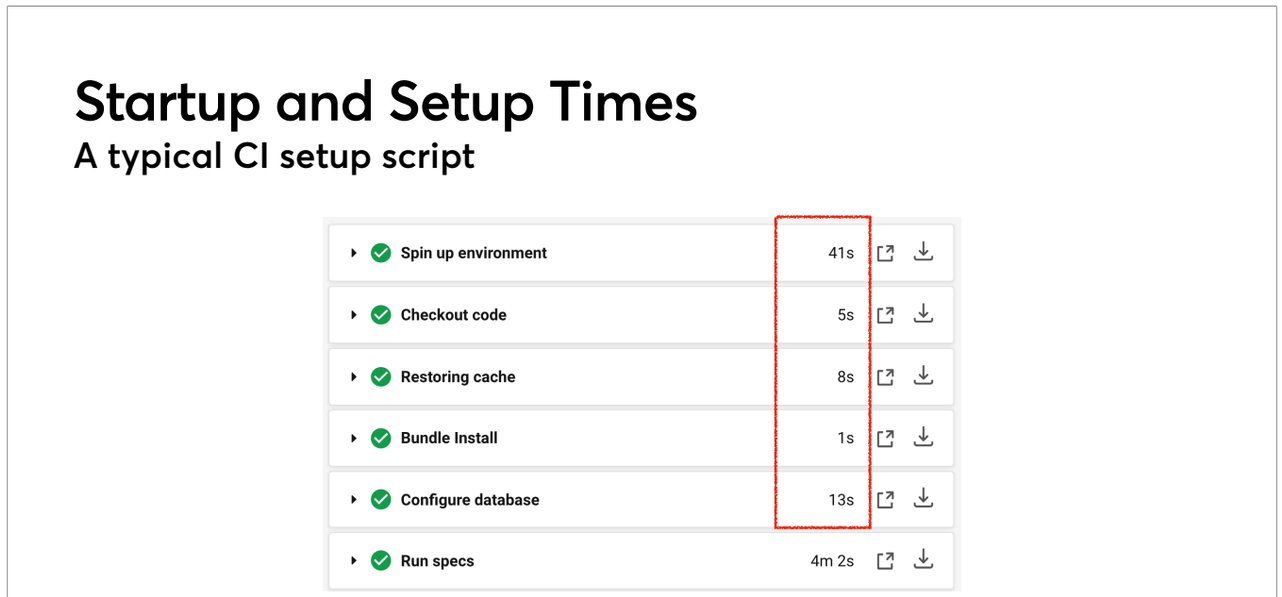
So here's what you want to do... Your CI will probably show you something like this, and again...

Startup and Setup Times

A typical CI setup script

@dmagliola

So what you want to do is look at all the steps that run in your CI job *before* and *after* your actual tests, and look at their running times...



And you want to focus on the ones that are taking a long time, and try to start chipping away at them. Now, these times are not intended as a flex, I just don't have a "before" screenshot... But before we optimizated them, these times are A LOT worse.

However, the important point is, you only care about the slow ones. If one of the things i'm going to talk about next, only takes 5 seconds for you... You don't care, move right along.

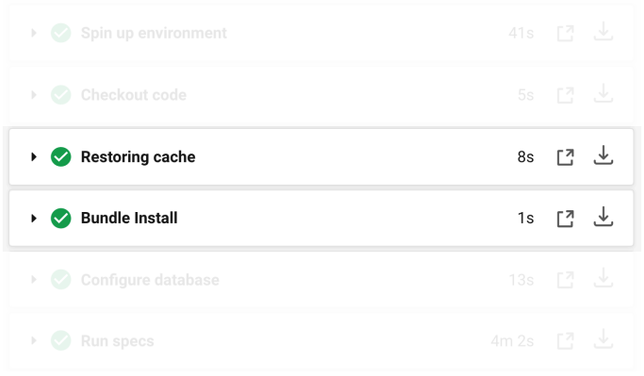
If your git checkout is 2 seconds, then there's not much point doing trickery to optimize it. If it's taking 30... Then it may be worth it.

At this point you might want to pause this talk, and take a look at your ACTUAL CI setup steps and their runtimes, which should put the rest of this talk into context and highlight what you want to pay more attention to.

And with that in mind, I'm going to talk about 3 things:

Startup and Setup Times

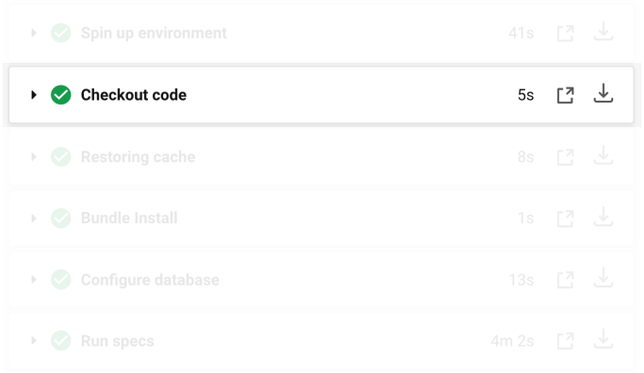
Things we can improve



Installing your gems

Startup and Setup Times

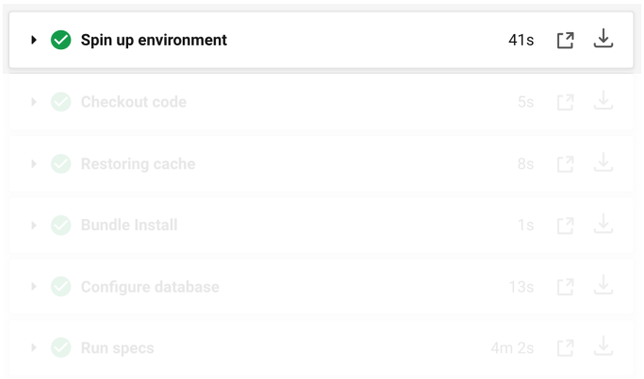
Things we can improve



Checking out your code

Startup and Setup Times

Things we can improve



@dmagliola

and Container Spin up time

Startup and Setup Times

Things we can improve

1. Installing your gems (`bundle install` and gem caches)

2. Checking out your code (`git clone`)

3. Container spin up time

@dmagliola

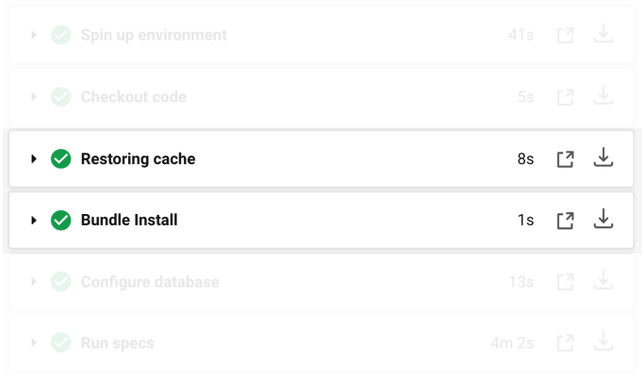
Now, depending on what CI provider you use, some of these bits may involve swimming against the current a bit.

CI providers try to make it very easy to get started with them, by giving you a sane default that's easy to set up, and does reasonable things. This is great for getting started quickly, but it's not so great for squeezing out every last drop of performance.
For that, you will want to customize things.

And to what extent you can do that will depend on your particular CI platform, but in my experience, most of them allow doing most of these things, they just sometimes involve going outside the beaten path.

Startup and Setup Times

Installing your gems



@dmagliola

Let's talk about bundle first.

In order to run your tests, you're going to need your gems installed. And you either let your CI provider do this magically for you, or you install your gems doing something like this.

Startup and Setup Times

Installing your gems

```
bundle install --jobs=4 --retry=3
```


But as you know, installing all your gems from scratch takes forever. So in order to prevent that, most CI providers give you caching capabilities. Some of them do it automatically for you, some let you do it yourself.

Basically what happens is this

Startup and Setup Times

Installing your gems

```
- restore gems cache
- bundle config set path 'vendor/bundle'
- bundle install --jobs=4 --retry=3
- save gems cache
```

@dmagliola

Before you install your gems, you control where they will get stored with "set path", after you install your gems, you save that directory to your CI's cache and next run, before installing, you restore that cache, which makes "bundle install" run instantly unless your Gemfile changed.

You are probably already doing this caching.

And when you first set this up, this works great. The first run takes a long time, but then your gems are cached, and future runs actually go through this pretty quick. Every time you change the Gemfile, you only need to install a gem or two, that gets cached, and things continue to be fast.

HOWEVER, over time, you may find that restoring the cache starts taking a long time. In our case, I've seen a restore time of about 2 minutes at its worst. This happens because as you upgrade gem versions, the old versions are left behind in the cache, and they bloat the size of the file that you're saving and restoring.

Startup and Setup Times

Installing your gems

```
- restore gems cache
- bundle config set path 'vendor/bundle'
- bundle config set clean 'true'
- bundle install --jobs=4 --retry=3
- save gems cache
```


To prevent this, you want to tell Bundler to automatically clean outdated versions. When you do this, bundle will delete old versions of gems that your Gemfile no longer uses, and that's going to make the CI cache smaller and faster to move around.

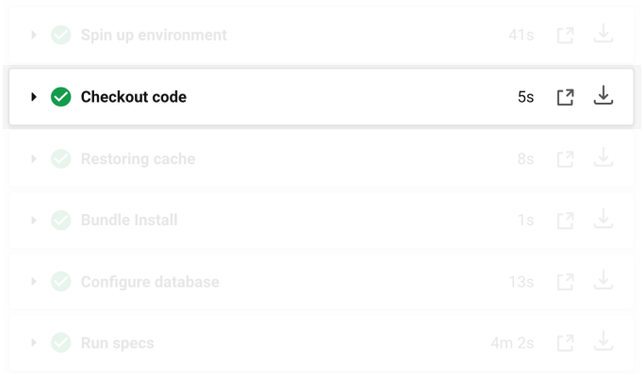
Now unfortunately, the specifics of *how* you do this depend A LOT on your version of Bundler. In older versions you pass "--clean" to bundle install instead of doing this, so you want to look at the documentation for your specific version of Bundler. And while you're there, look for other flags that look like they can save you space and experiment with them.

I'm going to also talk about manually deleting useless files later, which can save you even more space, but it's a bit harder to do.

And finally, look at the docs for your CI provider *in detail*. They all have slightly different features around this that you may be able to use. So keep an eye on those bundling and cache restoring times, and if they get long, this may help.

Startup and Setup Times

Speeding up Git Checkout



@dmagliola

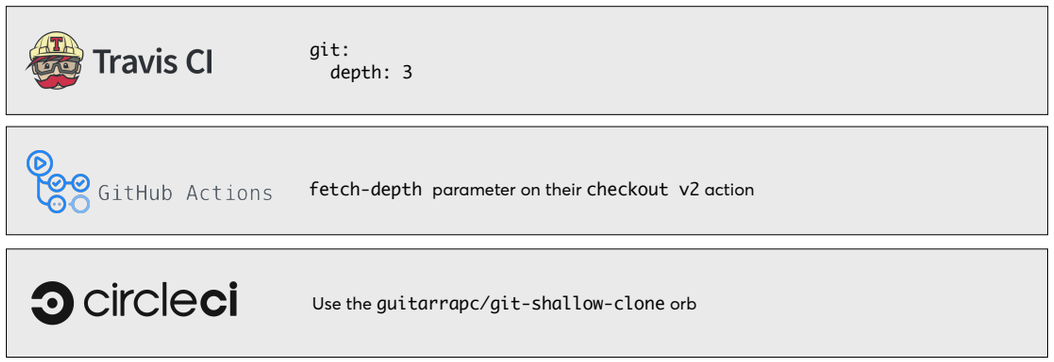
Our next step is "git checkout" or "clone".

This is one where you should only spend time on it if you see the step take a while. For a lot of projects, this is going to be pretty quick.

But if your repo is big and you're taking more than 20, 30 seconds... It may be worth trying to do a "shallow" checkout, where you get only the last few commits, rather than the entire history. This may or may not be faster depending on a number of things, but it's worth trying as a first approach.

Startup and Setup Times

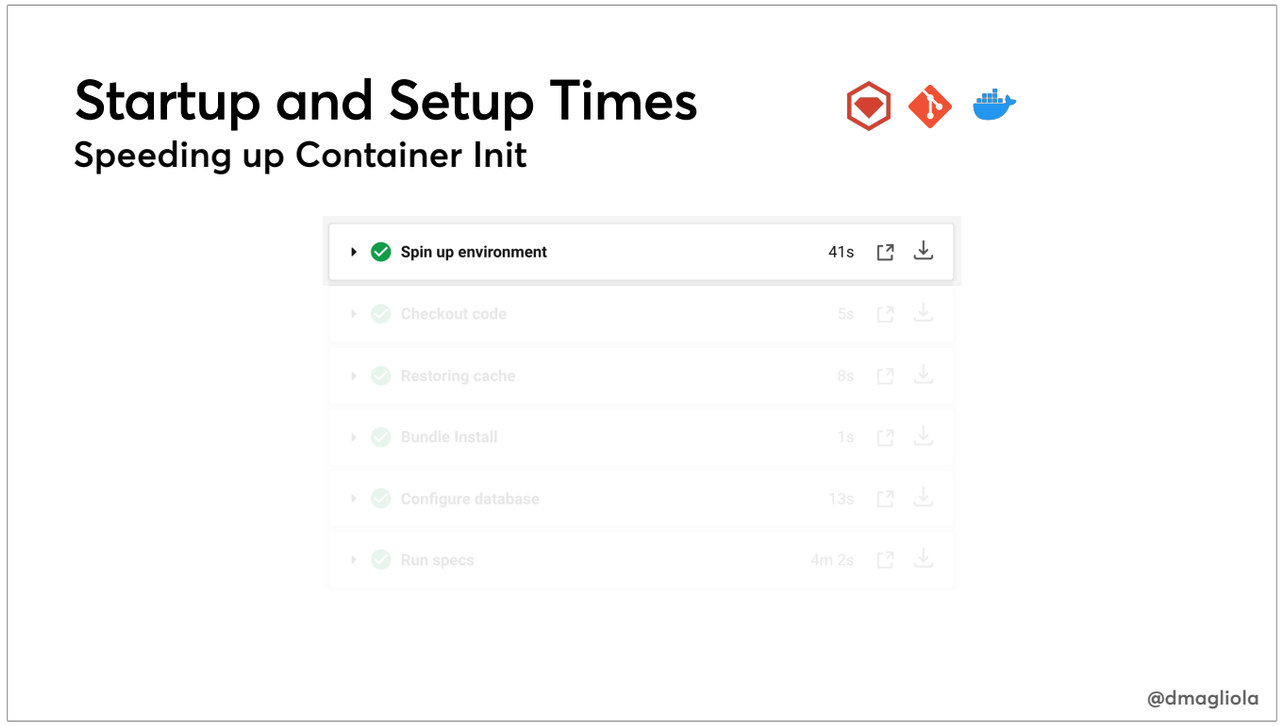
Speeding up Git Checkout



Unfortunately I can't really tell you how you do this in your situation...

This is another one CI platforms give you different options, either directly, or using some "library" that someone has made, so check their docs for how to do this...

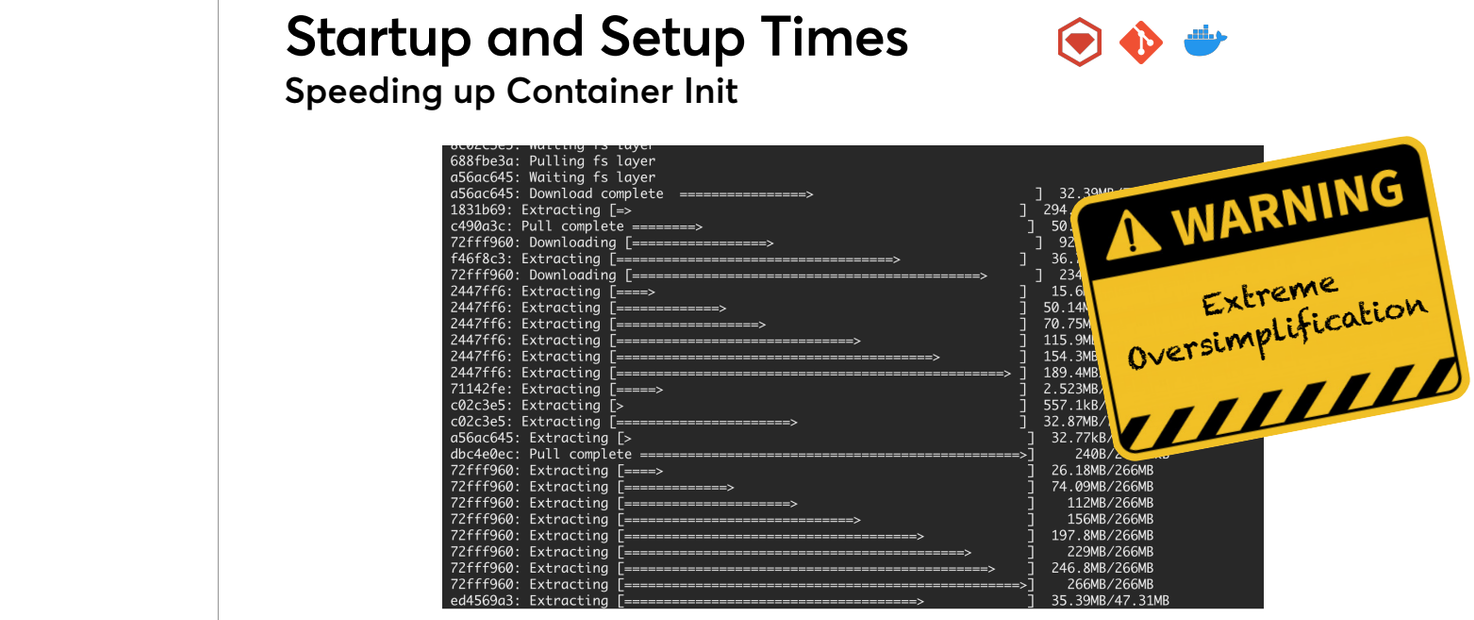
I mostly just wanted to call out that this is one to keep an eye on and that a shallow checkout can help. And if that doesn't help that much, I'll be talking more about this in a few minutes, with another approach.



Alright, so we've talked about 2 of the biggest usual time wasters, now let's look at probably the worst one: Container initialization time.

Now, this section is probably the bizarrest one in my talk, I know it will sound like really weird advice, but it can pay off *massively* if you are having this problem, so give this a shot.

Generally, all CI providers will run your stuff on Docker containers. And if you've ever used Docker, you're probably familiar with this sight



Startup and Setup Times

Speeding up Container Init

@dmagliola

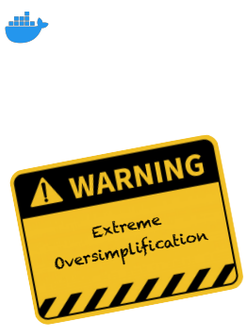
Now unfortunately, for this part to make sense, I need to give you a brief introduction to how Docker works.

Docker Containers are a sort of separate space in your machine, with their own little filesystem and processes that can't touch each other directly. Sort of "lightweight" virtual machines. And just like a virtual machine, they get booted up from "an image", which is basically a giant collection of files.

Now these images, they get created from a set of instructions that you put in a "Dockerfile", which looks a bit like this.

Startup and Setup Times

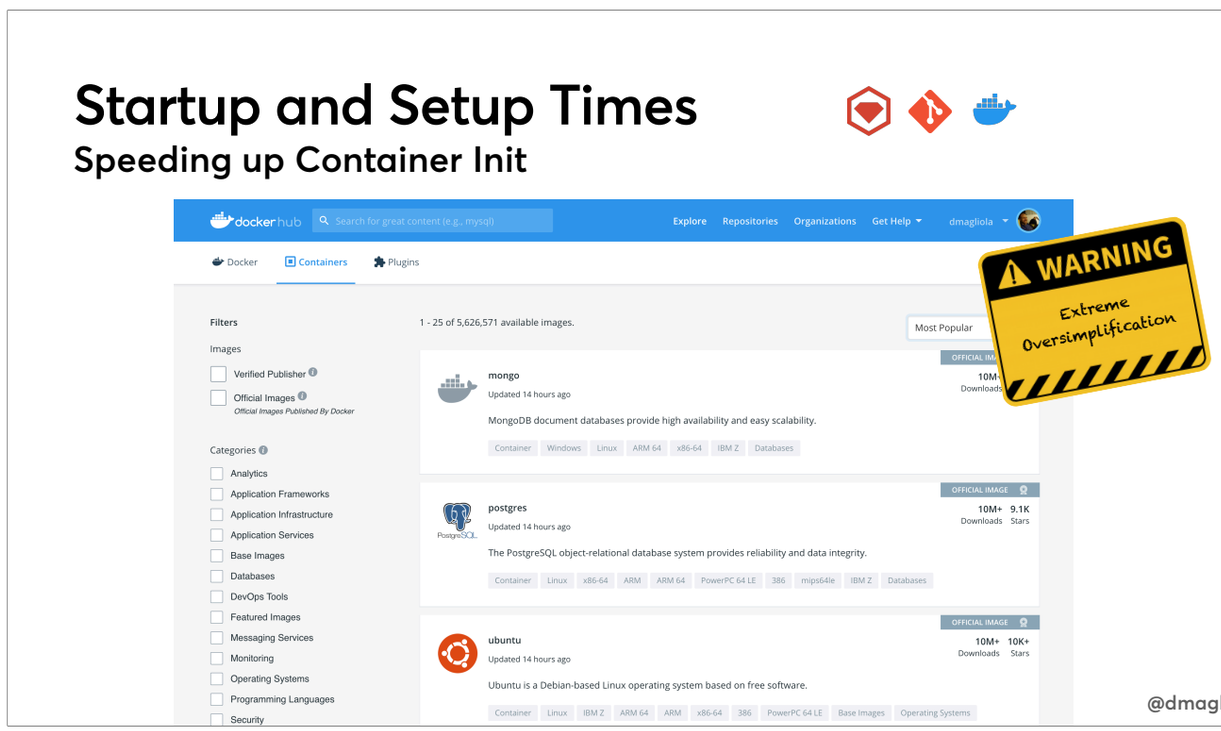
Speeding up Container Init



```
# Start from a tiny Linux
FROM alpine:3.12

# Install Ruby
RUN apt-get install (a bunch of necessary dependencies)
RUN (magic incantantions to install or compile Ruby)
RUN gem install bundler --version 2.2.15

WORKDIR /app

# Install our Gems
COPY Gemfile Gemfile.lock /app/
RUN bundle install --jobs=4 --retry=3

# Install our app
COPY . /app

# Run
ENV RAILS_ENV=production
EXPOSE 80
CMD ["bundle", "exec", "unicorn", "-c", "config/unicorn.rb"]
```


You may say "start from a plain Linux image, install Ruby on it, copy my Gemfile into it, run bundle install, then copy all the files from my app, and take the result of all that and that's my image."

And you can specify "when that image runs, run this command to start my app", that's that last Unicorn command



And just like you can create your image, lots of common tools like Postgres or Redis, have pre-made images that you can just use.
These live in a central registry, similar to RubyGems and that means you can tell Docker "hey, run me a Postgres, and a Redis, and this custom image I made which is my app".

And what Docker does is, when you try to run an image, if you don't have it, it'll go to the registry and download it for you, just like Bundler does when you install a gem.

Startup and Setup Times

Speeding up Container Init

```
rspec:
  docker:
    - image: circleci/ruby:2.7.2-node-browsers
    - image: postgres
    - image: redis
  steps:
    - git checkout
    - bundle install
    - yarn install
    - rails db:create db:structure:load
    - rspec
```


And when you're running in CI it's very common to do this. You tell your CI provider "run me a Postgres and a Redis, and also run my tests in a container image that has Ruby 2.7 in it".

And most CI providers give you a bunch of handy images they've pre-made for you. So you may get an image that already has Ruby in it, but also and Node and Chrome in there, so you can easily run your integration tests without having to install everything yourself.

And understanding images is good, but if we want to be able to get those containers to start faster, we need to go one level deeper. We need to talk about layers.

Startup and Setup Times

Speeding up Container Init

```
# Start from a tiny Linux
FROM alpine:3.12

# Install Ruby
RUN apt-get install (a bunch of necessary dependencies)
RUN (magic incantantions to install or compile Ruby)
RUN gem install bundler --version 2.2.15

WORKDIR /app

# Install our Gems
COPY Gemfile Gemfile.lock /app/
RUN bundle install --jobs=4 --retry=3

# Install our app
COPY . /app

# Run
ENV RAILS_ENV=production
EXPOSE 80
CMD ["bundle", "exec", "unicorn", "-c", "config/unicorn.rb"]
```

Takes FOREVER
Never changes

Takes a while
Changes sometimes

Pretty fast
Changes every time

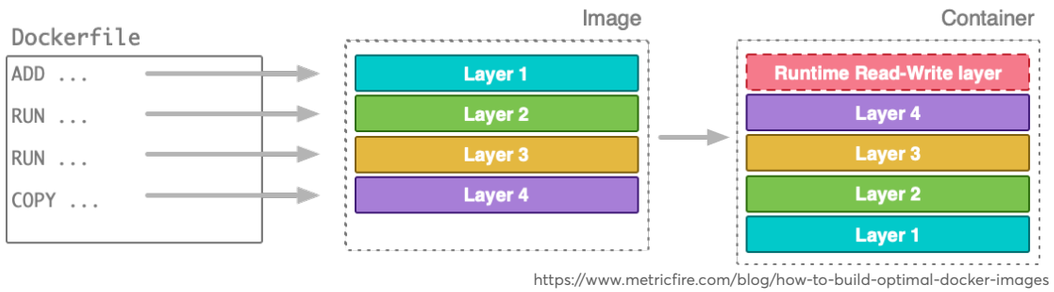
@dmagliola

Because one of the really interesting ideas in Docker, is that when building this image of yours, there are things that take a lot of time to run, but they don't change very often. And there are things that do change very often, but they don't take as long to do.

So what Docker does is, when it's building your image, at each step in this Dockerfile, it'll do what you ask, but all the writing it does to the filesystem in your image gets stored separately from what's already there from the previous steps.

Startup and Setup Times

Speeding up Container Init



https://www.metricfire.com/blog/how-to-build-optimal-docker-images

@dmagliola

It gets put in a "layer", which depends on the previous existing layer and adds or modifies files. And Docker will also take a "hash" of what you're doing, so the next time you try to build the same image, if the step hasn't changed Docker knows that because the hash matches, and goes "ah, i'll just use this layer I've got cached over here", which saves you a lot of time.

Startup and Setup Times

Speeding up Container Init

```
# Start from a tiny Linux
FROM alpine:3.12

# Install Ruby
RUN apt-get install (a bunch of necessary dependencies)
RUN (magic incantantions to install or compile Ruby)
RUN gem install bundler --version 2.2.15

WORKDIR /app

# Install our Gems
COPY Gemfile Gemfile.lock /app/
RUN bundle install --jobs=4 --retry=3

# Install our app
COPY . /app

# Run
ENV RAILS_ENV=production
EXPOSE 80
CMD ["bundle", "exec", "unicorn", "-c", "config/unicorn.rb"]
```

Takes FOREVER
Never changes

Takes a while
Changes sometimes

Pretty fast
Changes every time

@dmagliola

In this sample Dockerfile, you'll be building this bottom part over and over, but this whole chunk at the beginning doesn't change. You're always installing the same versions of the same thing, so Docker just uses the layer it already has. This part *does* change, because your app files probably changed, and so for these it makes new layers. But for everything else, it can save itself a lot of time.

So the result is your image is not a single monolithic file, it's a stack of layers, each of which depends on the previous ones, but they can be downloaded and cached independently. And they can also get shared.

Startup and Setup Times

Speeding up Container Init

```
# Start from a tiny Linux
FROM alpine:3.12

# Install Ruby
RUN apt-get install (a bunch of necessary dependencies)
RUN (magic incantantions to install or compile Ruby)
RUN gem install bundler --version 2.2.15

WORKDIR /app

# Install our Gems
COPY Gemfile Gemfile.lock /app/
RUN bundle install --jobs=4 --retry=3

# Install our app
COPY . /app

# Run
ENV RAILS_ENV=production
EXPOSE 80
CMD ["bundle", "exec", "unicorn", "-c", "config/unicorn.rb"]
```

@dmagliola

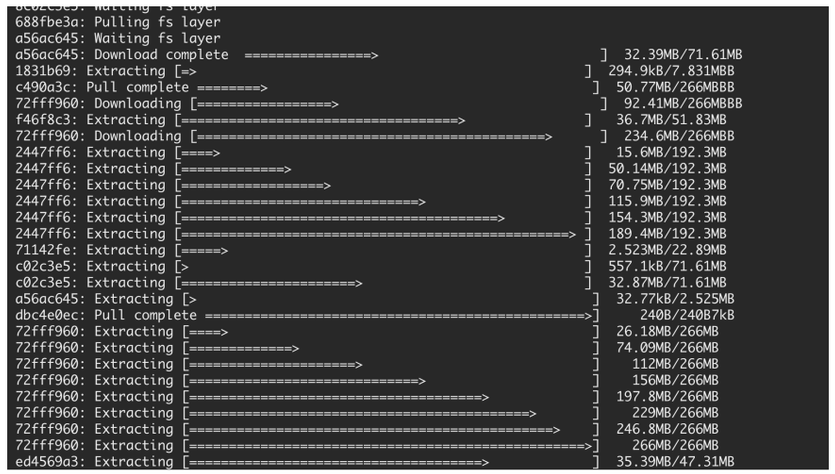
This image you start from, can be anything, it can be your own image if you want, and it can already come with a lot of layers in it.
If you have a bunch of different applications on Ruby 2.7, you might end up making your own base image that includes the installation of Ruby and all the stuff common to them, and then reuse that in the Dockerfile for each app.

If you do that, the layers in that custom Ruby image you made will get shared for all of the apps on a machine. They'll only be built and downloaded once.

And that shared part is also probably the biggest part of your Docker image. Whereas the bottom bit will be different for each app, and for each build of your apps, so they'll get downloaded every time, but the big ones at the top get reused.

Startup and Setup Times

Speeding up Container Init

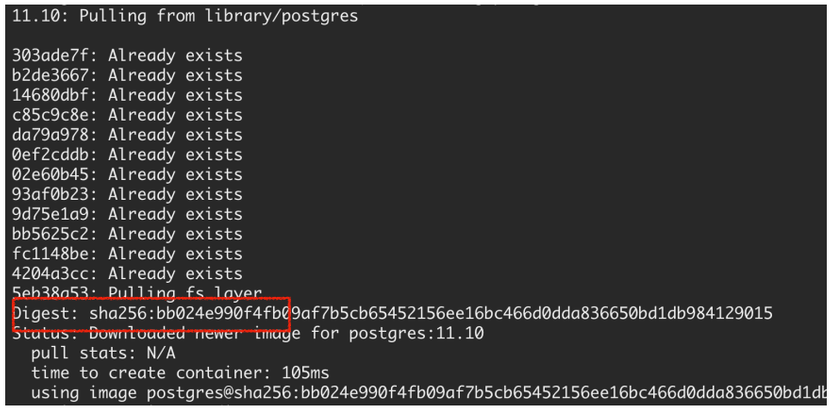


And that is what you're seeing when you see this downloading screen. This is trying to run an image, and it's downloading all of the layers it doesn't have.

And you've probably seen something like this too,

Startup and Setup Times

Speeding up Container Init

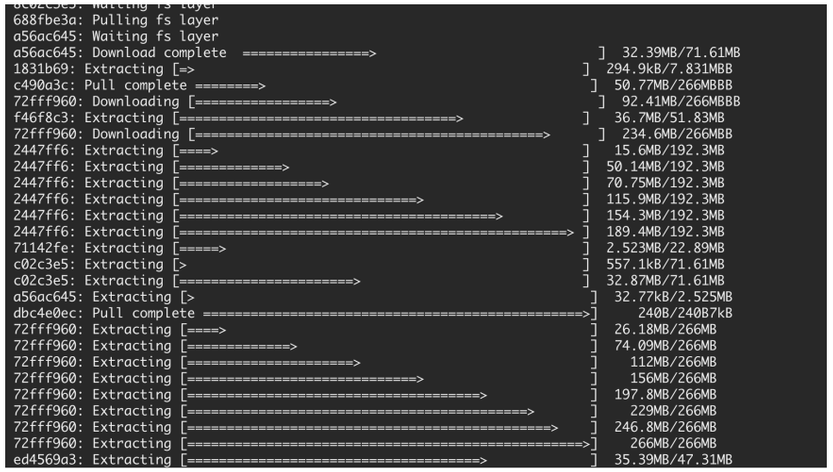


where some layers already exist, and those don't get downloaded.
Here we're only downloading one layer, the others are already cached.

Now, the reason I am talking about all of this, is that depending on whether you get this or this

Startup and Setup Times

Speeding up Container Init



it will make an HUGE difference to how long it takes to start these containers.

You really, really want the machine that runs your tests to already have downloaded the images you're going to use, or at least a lot of its layers, because if it has, your containers will start almost immediately. If it hasn't, it'll first have to download, probably, about a gigabyte, which can take a while, then extract those layers, and only THEN it can start to spin them up.

So you really, really want the machine that's running your tests to already have the layers you're about to use.

Unfortunately, you generally have absolutely no control over this. Your CI provider has a gigantic pile of computers, each running tons of containers for lots of people, and you have absolutely no control over which machine your test will land on, and what layers it will have already cached. Which is why I said this is a somewhat bizarre piece of advice. "Try to make sure the machine you have no control over already has your layers" sounds nuts.

BUT.

What runs in these machines isn't *random*. You can't guarantee what a machine has already downloaded, but you can try to influence the odds in your favour.

There are loads of people running their stuff in these machines. Loads of these people are going to be using Ruby.
So they'll be using a Ruby image. Probably the Ruby image provided by the CI platform.
So if you're running Ruby tests, it's possible that your machine will have the Ruby image cached, because someone probably ran Ruby tests there earlier.

And here is the bit you can control, because not all the images you could use will be equally likely to be used by others. Some versions will be more common than others.

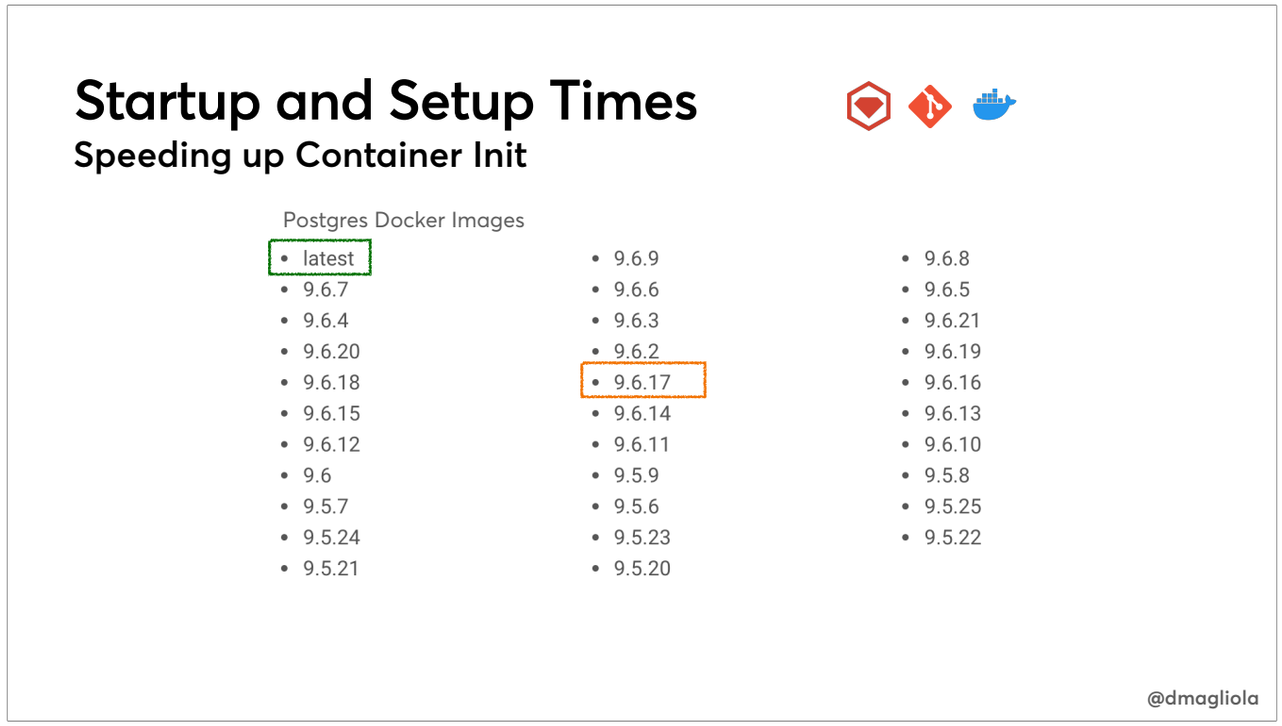
Startup and Setup Times

Speeding up Container Init

Release Version	Release Date
Ruby 3.0.0	2020-12-25
Ruby 2.7.2	2020-10-02
Ruby 2.7.1	2020-03-31
Ruby 2.6.6	2020-03-31
Ruby 2.5.8	2020-03-31
Ruby 2.4.10	2020-03-31
Ruby 2.7.0	2019-12-25
Ruby 2.4.9	2019-10-02
Ruby 2.6.5	2019-10-01
Ruby 2.5.7	2019-10-01
Ruby 2.4.8	2019-10-01
Ruby 2.6.4	2019-08-28
Ruby 2.5.6	2019-08-28
Ruby 2.4.7	2019-08-28

@dmagliola

You're going to have a lot more people using Ruby 2.7.2 that 2.7.0, for example, just because it's a newer version.
And both of them will be *a lot more* common than, say, 2.3.5.
Same goes for Postgres. If you are using `postgres:latest`



it is *way* more likely that someone else has used that recently, than if you're using a random outdated version like, say, `9.6.17`.

This really bit us at one point. We were using a weird version for an image that was actually pretty large to download, and our containers were taking AGES to start. And just switching to a very similar but more popular version saved us about a minute of setup.

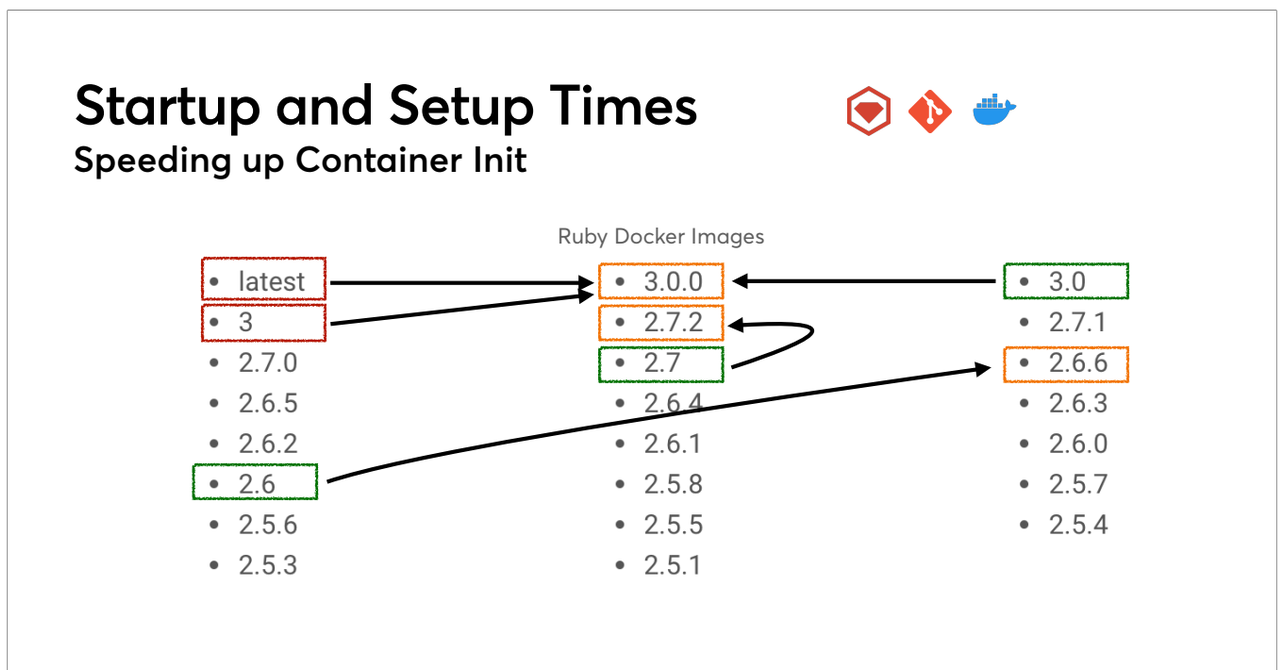
Startup and Setup Times

Speeding up Container Init

@dmagliola

So what you want to do is make sure you are using what looks like the most popular versions of your dependencies, to load the dice in your favour, and increase the chances that the images you use or at least a good chunk of their layers will be cached.

Now again, this is still weird advice, because you have no way of knowing what others are using. But if you see your container startup times are high, you can look inside that "container spin up step" and see whether it is frequently downloading layers instead of using the cache, and if that happens a lot, you can start looking at the other available images and trying different ones to see if they get cached more often.



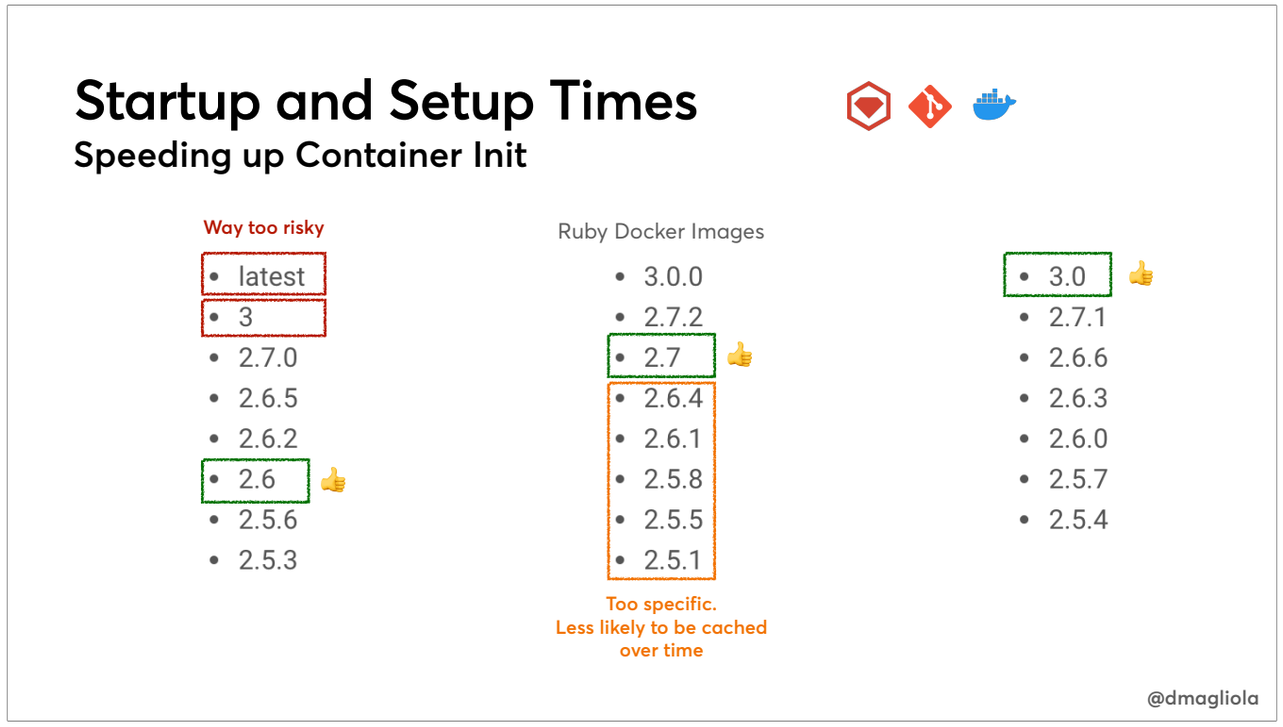
You also want to point to less specific versions, which will be more commonly pointed at.

In Docker, you can have many tags pointing to the same actual image, and these tags can get repointed over time. So, for Ruby, you'll have tags for 2.7.1, 2.7.2, but there's also just 2.7, which points to the latest patch version and probably most people will be using that one, as it repoints itself over time.

The same happens if you have a "latest" tag, like postgres:latest or redis:latest. That will generally be the default most people use, so it'll be more likely to be cached.

Now this advice comes with a big disclaimer. All of the things I'm proposing you do today, like everything in life, have trade-offs. But this one in particular has a massive one. One of the glorious things about Docker, is that you can specify the exact versions of everything, and fully control your environment. So the usual best practice is "choose the exact version of everything in CI that you are running in production". Because then you are testing on the same exact environment as production, or as close as you can get it.

Whereas I'm standing here suggesting that you do the exact opposite, which implies some risk. If you use the latest postgres... now you're testing on a different version than the one you're actually running in prod, and that *could* bite you. The flip side is, you can collectively save *hundreds of hours* of developer's times by doing that. Which one matters most will depend on your particular situation.



And there's some nuance here, and some middle ground. For example, you almost definitely don't want "Ruby latest" because Ruby changes a lot between minor versions; you may get hit by deprecations, backward incompatible changes, etc. But if you're using Ruby 2.7.2, pointing to "the latest 2.7", is probably safe enough and your CI will probably still be trustworthy, and you can get the startup speed benefits of a more common, more cached version.

If you're running an old version of ElasticSearch, or some software that has had major backwards incompatibilities, then yeah, you probably have to still run that. It sucks. But for things like Postgres, which have really good backwards compatibility... Using the latest postgres is probably fine?

Oh, and make sure you're using the Docker images that your CI provider gives you, not the normal "official" ones, as that's what others will likely be using.

Again, it's weird advice, you're making CI less deterministic, which is normally the opposite of what you want... But... speed. You'll have to weigh this one.

The other big disclaimer, of course, is that this is much less deterministic also in terms of timings than anything i've talked about so far. You're always rolling the dice on whether your machine will have your layers. Doing this lets you weigh those dice a little bit to your advantage, but it's still a crapshoot... And because of that it can be harder to gauge whether this helped or not, as the runs you're observing when you're trying it may or may not be representative. I'm going to talk more about this later.

But the general gist for this section is... Try to use common versions of things if you can, it can make your containers boot up a lot faster on average.

Startup and Setup Times

Things we can improve

1. Installing your gems (`bundle install` and gem caches)

2. Checking out your code (`git clone`)

3. Container spin up time

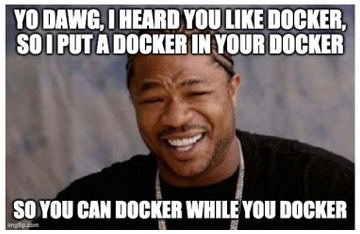
Ok, so, we've covered how to speed up Bundler, Git Checkout and Container Spin Up.

But now that we've talked about Docker, I'd like to take a second look at these same 3 topics but for a more, say, advanced scenario.

Startup and Setup Times

Things we can improve... with Docker

1. Installing your gems... with Docker
2. Checking out your code... with Docker
3. Container spin up time... with Docker?



@dmagliola

If you have the ability to build your own containers and push them to a registry, and your CI platform lets you run your own containers... There's more we can do to gain even more speed.

Now this takes some more work than the previous tips, so maybe it only starts being economical when you start having larger dev teams, which means you're now wasting *a ton* more human time waiting on CI, and you probably have developed some tooling that'll let you do this more easily. So it takes a bit more work, but it was definitely worth it for us, for example.

Startup and Setup Times

Building your own CI docker image

Dockerfile (for production)

```
# Start from a tiny Linux
FROM alpine:3.12

# Install Ruby and other dependencies
RUN (...)

# Install our Gems
COPY Gemfile Gemfile.lock /app/
RUN bundle install --jobs=4 --retry=3


# Install our app
COPY . /app

# Run
ENV RAILS_ENV=production
EXPOSE 80
CMD ["bundle", "exec", "unicorn",(...)]
```

Dockerfile.ci (for CI)

```
# Start from your CI's Ruby image so it's cached
FROM circleci/ruby:2.7-node-browsers




# Install our gems
COPY Gemfile Gemfile.lock /app/
RUN bundle config set path 'vendor/bundle'
RUN bundle install --jobs=4 --retry=3

# Git Clone instead of copy
# (because you'll need to `fetch` on branches)
RUN git clone git@github.com:your_repo

# Nothing to Run, CI takes care of that.
```

@dmagliola

If you can build images easily, one thing that's worked really well for us is having a custom Docker image for your CI environment. This will *not* be the same Docker image that you run in production. It will be very, very different, so you need to have 2 dockerfiles in your repo, one for production, and one for CI.

What you do is start from a Docker Image that your CI provider offers, so it'll be heavily cached, and you start from one that already has most stuff in it, Ruby, Chrome, etc, and you add the stuff you need to run your tests which is the setup you'd normally do in CI, most of it you do it this Dockerfile instead.

The way you'll work with this is you will automatically build this image every time you merge to your main branch, and you always tag it "ci".

Startup and Setup Times

Building your own CI docker image

```yaml
rspec:
  docker:
    - image: circleci/ruby:2.7.2-node-browsers
    - image: postgres
    - image: redis
  steps:
    - git checkout
    - bundle install
    - yarn install
    - rails db:create db:structure:load
    - rspec
```

```yaml
rspec:
  docker:
    - image: your_company/project_repo:ci
    - image: postgres
    - image: redis
  steps:
    - git fetch branch
    - bundle install
    - yarn install
    - rails db:create db:structure:load
    - rspec
```


And in your CI configuration, you point to your registry, and to that "ci" tag, so you're always getting a recent one. Now, importantly, this is not going to always be a reflection of what's in your branch under test right now. These images may take a while to build, and you don't want to wait for them, and this is key. If you had to wait for them to build before you can run your tests, you're actually causing more harm than good. But since you're always using the same tag, if the last merge to main hasn't finished building, that's fine, you'll just be using the one from the previous merge, which is good enough.

Also most of your tests will run in other branches, but using the Docker image from your main branch, so you also need to apply the changes from your branch. So it's important to keep this in mind, because there's a couple of things you'll still have to do in CI AGAIN after the image starts.

Startup and Setup Times

Building your own CI docker image

1. Installing gems in your image

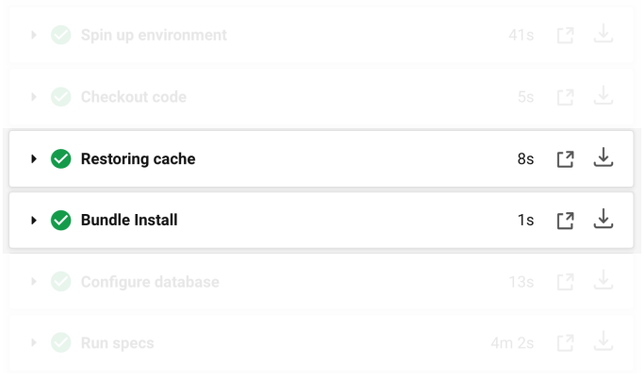
2. Checking out your code in your image

3. Optimizing your layers for startup speed

So I want to talk about how we do the same 3 things again, but with a Docker twist.

Startup and Setup Times

Installing gems in your image



First, Bundler. As we discussed earlier,

Startup and Setup Times

Installing gems in your image

```
- restore gems cache
- bundle config set path 'vendor/bundle'
- bundle install --jobs=4 --retry=3
- save gems cache
```


your CI provider gives you a cache you can use to save and restore your bundle install. And while this cache is really useful, sometimes it can actually be quite slow to restore.

One thing that can sometimes help is running `bundle install` on your Docker build, which means your gems will be there already once the container boots up.

Startup and Setup Times

Installing gems in your image

```
- ~~restore_gems_cache~~
- bundle config set path 'vendor/bundle'
- bundle install --jobs=4 --retry=3
- ~~save_gems_cache~~
```


And then you don't need the cache save and restore running in CI, you already have your gems in your image.

There's 2 things to keep in mind, though:
First, this image is built against your main branch. If you're testing in a branch where you updated the Gemfile, it won't have those latest gems, so you still need to run `bundle` in CI.

Startup and Setup Times

Installing gems in your image

```
- ~~restore_gems_cache~~
- bundle config set path 'vendor/bundle'
- bundle install --jobs=4 --retry=3
- ~~save_gems_cache~~
```

You still need to run
Bundle Install in CI!

@dmagliola

But this bundle will, the vast majority of the time, finish instantly, or install just one gem, it's going to be very quick.

The other thing to note is that you are trying to save time by not needing to do that cache restore, but the Docker layer where you install all your gems still needs to be downloaded, so you still care about how much space these gems take on disk, or the layer will be huge and take forever to download.

Now earlier when we were using the CI cache, the main improvement was deleting old gems.

Startup and Setup Times

Installing gems in your image

```
- bundle config set path 'vendor/bundle'
- bundle config set clean 'true'
- bundle install --jobs=4 --retry=3
```


This doesn't apply here, because you never have old gems. If your Gemfile changes, your container build starts that layer from scratch without any old gems.

However, there is a lot of bloat when you're using "bundle install". Bundle keeps a bunch of caches you don't need, and which can be quite big, so you want to get rid of them. There are a couple parameters that Bundle offers to prevent those caches, but those have changed between versions of Bundler, and I may be doing something wrong, but it seems to still keep caches around.

Startup and Setup Times

Installing gems in your image

```
RUN bundle config set no-cache 'true' \
    && bundle config set path 'vendor/bundle' \
    && bundle install --jobs=4 --retry=3 \
    && rm -rf root/.bundle/cache \
    && rm -rf vendor/bundle/ruby/2.7.0/cache
```


But a thing you can do is explicitly *delete* those caches after installing your gems, which will save you the download time. Your "bundle install" command will end up looking a bit like this. You will need to tweak these paths, however, depending on your particular system.

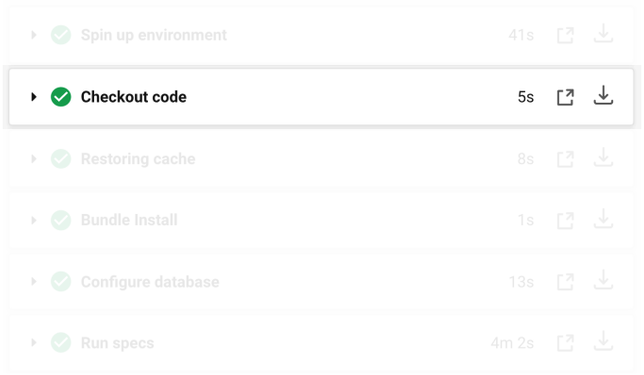
I will talk a bit more in a minute about how to find those paths, and how to figure out how big your layers are and make them smaller in general. But as an example, in our particular Docker build, deleting these files takes this layer from 500 Mb to 400. We save almost 100Mb, which otherwise we'd be downloading over and over and over. It's quite a big reduction on that download time, so it pays off doing it.

And by the way, this is something you can also do if you are not making your own images. If you are doing a normal bundle install in your CI, you can delete these directories before you save your cache, to make it smaller. The only issue is it's not as easy to find what those paths should be.

So that's how we make Bundle faster.

Startup and Setup Times

Cloning your repo in your image



@dmagliola

This same idea applies to Git Checkout / Clone.

If you remember the problem, cloning the whole repo may take a long time. However, you don't care how long it takes during Docker build, so you can do a full `git clone`, and then when the container runs in CI, most of your repo is already there. You just need to do a `git fetch` of the branch you're testing, which is only going to pick up the few commits that are not already in the image, and it's going to be way faster than even a shallow clone.

And finally, I want to talk about optimizing your layers.

As I was mentioning earlier, your container will be composed of many layers, one for each command you run in your Dockerfile, and you want to maximize the likelihood that as many as possible of those layers are cached. This is why we start from a base image that your CI platform gives you.

But there are layers that will almost never be cached, which are the ones you added to that base image. Here there's a balancing act you want to play...

Startup and Setup Times

Optimizing your layers

```
Using default tag: latest
latest: Pulling from library/node
16cf3fa6cb11: Extracting [=========================================>     ]  41.75MB/45.38MB
3ddd031622b3: Download complete
69c3fcab77df: Download complete
a403cc031cae: Download complete
b900c5ffbaf4: Downloading [===========>                                   ]  52.01MB/214.3MB
f877dc3acfca: Download complete
dd90c29890c7: Downloading [=====================>                         ]  15.53MB/33.74MB
5d739305453c: Downloading [==============================>                ]  1.547MB/2.38MB
64b469d11365: Waiting
```


Docker will download a bunch of these in parallel. How many depends on your CI platform's Docker configuration, and you want to try to optimize for this number.

What you are trying to balance here is the bandwidth you get for each parallel download stream, and the latency of roundtrips.
Let's say you have 1Gb of layers to download. If you have only one layer of 1Gb, you won't be parallelizing on that download at all. All the megabytes that need to be downloaded will be done sequentially, and you won't be using as much bandwidth as you could. If you had more layers, that'd happen more in parallel and it'd download faster overall.

However, if you have *too many* layers, a lot of them might be tiny, and you're going to be doing lots of roundtrips between your CI machine and your Docker registry and wasting time there.

Startup and Setup Times

Optimizing your layers

```
RUN apk add --r
  && curl -fsSL
  && curl -fsSL
  && gpg --batc
  && mkdir -p /
  && tar -xzf y
  && ln -s /opt
  && ln -s /opt
  && rm yarn-v$
  && apk del .b
  && yarn --ver
```


So you don't want to have too many layers.

This is why it's common to see the double-ampersand in Dockerfiles. If each of these were a separate RUN command, each would end up as an individual layer.

The double ampersand means you get a single layer with what's left after doing all those things.

Startup and Setup Times

Optimizing your layers

```
RUN bundle config set no-cache 'true' \
    && bundle config set path 'vendor/bundle' \
    && bundle install --jobs=4 --retry=3 \
    && rm -rf root/.bundle/cache \
    && rm -rf vendor/bundle/ruby/2.7.0/cache
```


The other thing you're optimizing for is the size of those layers in the first place. You don't want them to be huge or they'll be slow to download.

You want to look at the size of your layers, and if they are big, see if there is stuff you can delete. Some examples of this are temp files, or `tar.gz` files you may have downloaded and extracted, remember to delete the tarred file if it's big.

It's very important, though, that you delete those files *in the same RUN command* as you create that stuff. Pay attention to those ampersands. If those deletions up there were a separate RUN command, then we'd add a layer with all the useless crap, which has to be downloaded, and then a second layer that deletes it. Which defeats the purpose.

Startup and Setup Times

Optimizing your layers

Docker history

```
$ docker history 4d032a7b0ca9
IMAGE          CREATED      CREATED BY                                      SIZE
4d032a7b0ca9   3 days ago   /bin/sh -c #(nop)  CMD ["node"]                 0B
<missing>      3 days ago   /bin/sh -c #(nop)  ENTRYPOINT ["docker-entry…   0B
<missing>      3 days ago   /bin/sh -c #(nop) COPY file:238737301d473041…   116B
<missing>      3 days ago   /bin/sh -c set -ex   && for key in     6A010…   7.76MB
<missing>      3 days ago   /bin/sh -c #(nop)  ENV YARN_VERSION=1.22.5      0B
<missing>      3 days ago   /bin/sh -c ARCH= && dpkgArch="$(dpkg --print…   92.8MB
<missing>      3 days ago   /bin/sh -c #(nop)  ENV NODE_VERSION=15.12.0     0B
<missing>      9 days ago   /bin/sh -c groupadd --gid 1000 node   && use…   333kB
<missing>      9 days ago   /bin/sh -c set -ex;  apt-get update;  apt-ge…   561MB
<missing>      9 days ago   /bin/sh -c apt-get update && apt-get install…   141MB
<missing>      9 days ago   /bin/sh -c set -ex;  if ! command -v gpg > /…   7.82MB
<missing>      9 days ago   /bin/sh -c set -eux;  apt-get update;  apt-g…   24MB
<missing>      9 days ago   /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>      9 days ago   /bin/sh -c #(nop) ADD file:5f8ab4280b54d9147…   101MB
```

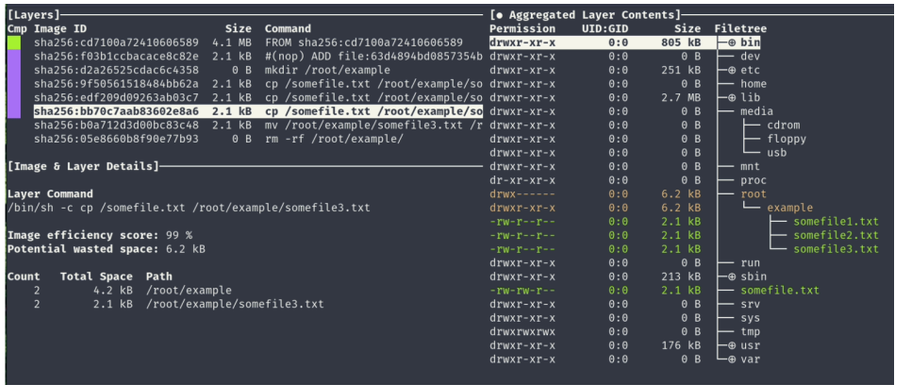

So to do this, you need to see what layers you have in your image and how big they are.

An easy way to inspect this is by using `docker history`, this will show you the layers in an image, and how much space they take.

Startup and Setup Times

Optimizing your layers

Dive: https://github.com/wagoodman/dive



@dmagliola

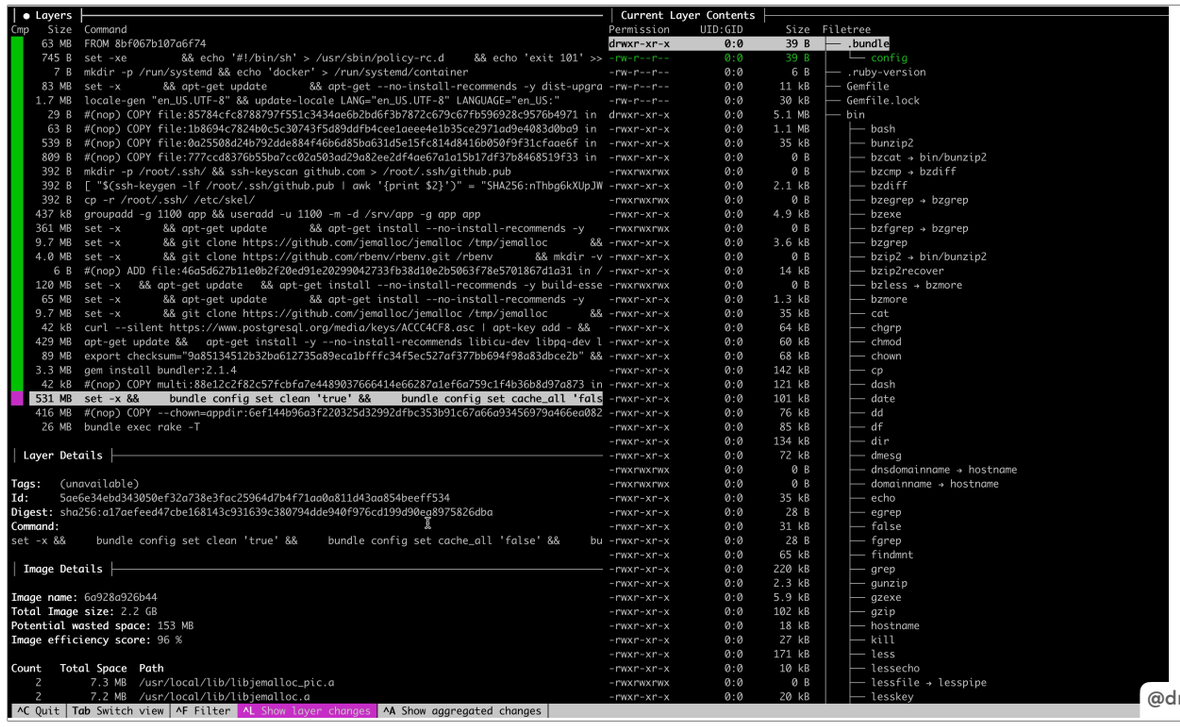
And if you want to look inside these layers, there's a great tool called `dive` that will show you the filesystem of the image, and what each layer has in there. This is really useful to find stuff left behind by build processes that you can delete, like those Bundler caches I mentioned earlier.

Startup and Setup Times

Optimizing your layers

```
RUN bundle config set no-cache 'true' \
    && bundle config set path 'vendor/bundle' \
    && bundle install --jobs=4 --retry=3 \
    && rm -rf root/.bundle/cache \
    && rm -rf vendor/bundle/ruby/2.7.0/cache
```


And this is also how I found those paths to delete.
Here's what that looks like:



(video plays)
This up here is showing us all the layers in the image, and we can pick which one to look at, and over there it's showing us the file system, and in color the stuff that's new on this layer. And you can ask it to see only new stuff, and you can start collapsing directories, and you can clearly see that 19Mb cache directory over there, and in a minute we're going to see... There's the other one. Another 77 Megs. So those are the two paths you saw me delete in that previous command, that's how you find them, and we delete those two and we saved 100 Megs.

If your container is taking a while to download, it may be worth inspecting the layers and seeing if there's low hanging fruit there that may help.

Final thoughts

1. Observability
2. Flaky tests
3. Uneven distribution of tests
4. Too much of a good thing
5. Critical Paths

@dmagliola

Alright, we're now done with tricks to optimize your startup times. And helping your tests start faster is one of the most important things you can do that will let you parallelize more aggressively and have faster overall times.

But I want to talk about a few more things that are important to keep in mind:
- Observability
- Flaky tests
- Uneven distribution of tests
- Having Too much of a good thing
- Critical Paths

The first thing i'd like to talk about is Observability. As I mentioned earlier, CI times are quite variable, because there's a lot of factors involved. For starters, how long a container takes to start depends on whether the particular machine you're running on has the layers cached. But also, sometimes, the network is running a bit slow and things take longer. Other times, the machine you're running on is just having a sad day.

Because of this, it's really hard sometimes to know whether you're actually making improvements. You may do a little experiment in your CI config file, put it on a branch, and it runs super fast... And maybe that's because your experiment is successful. But maybe you just got lucky. It's hard to know because it varies so much. It's also hard to stay on top of your setup, which over time will trend towards taking longer and longer, and you probably won't notice that drift unless you're watching it like a hawk. Which you aren't.

Final thoughts

Observability

PIPELINE	STATUS	WORKFLOW	BRANCH / COMMIT	START	DURATION
anony 320	Success	tests	master ae77194	3mo ago	1m 3s
anony 318	Success	tests	master	3mo ago	1m 0s
anony 316	Success	tests	master	3mo ago	28s
anony 313	Success	tests	master	3mo ago	51s
anony 306	Success	tests	master b776496	4mo ago	32s
anony 300	Success	tests	master 5188a48	7mo ago	54s
anony 298	Success	tests	master d2d8dfb	7mo ago	1m 9s
anony 295	Success	tests	master a0165b6	8mo ago	55s
anony 292	Success	tests	master 061d7ee	9mo ago	47s
anony 290	Success	tests	master e92c5d0	9mo ago	41s

@dmagliola

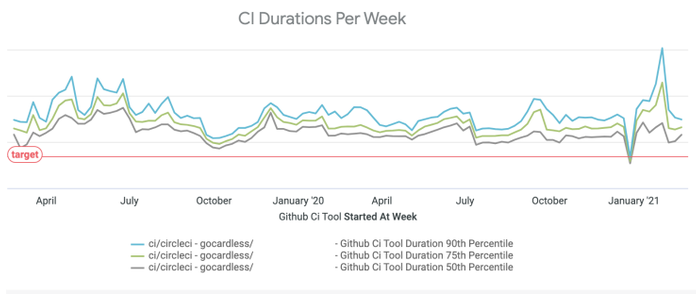
So to combat this, the very least you can do here is sampling when you are experimenting.

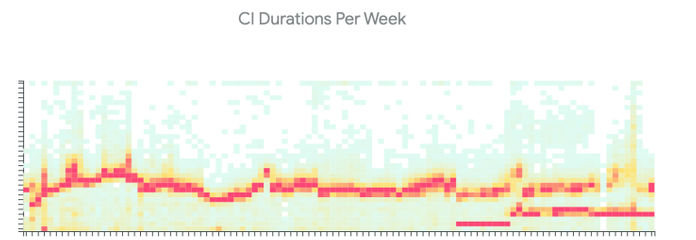
Before you try some change, you're going to want to see how long it's taking now. Get a baseline. To do this, don't just look at the last build in `main`. Look at 5 or 10 builds over the last couple of days. Notice not just the average time different steps take, but also the variance. Get familiar with how things normally perform. What steps are consistent, and which are all over the place. Because those are the ones that lead to sadder builds, and the one you want to focus on if you can.

And when you're running an experiment, re-push to your branch 3 to 5 times, to get more samples. And that'll give you a better idea of whether you're actually changing things.

Final thoughts

Observability





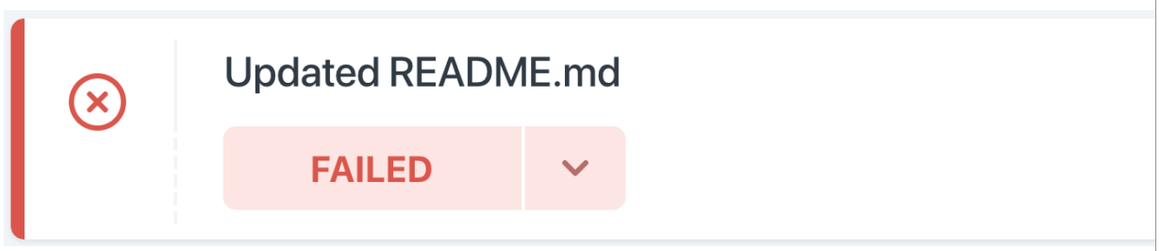
@dmagliola

But ideally, you can build some observability over this. It's hard to be specific on the best way to do this because it will depend a lot on your specific observability stack but as a general pointer, Github will send you webhooks when CI steps complete, and you can use those to build information about timings, and push it to your observability layer of choice.

And then you can make yourself a beautiful dashboard that'll let you see with more precision how your changes affect CI runtimes, and how they evolve over time. I admit this is quite a bit of work, but we've had great results with this because it's an early warning system that things are getting slow, and it also gives us much more confidence on the changes we make.

Final thoughts

Flaky Tests



Flaky tests are the bane of our existence. The worst case scenario, which a lot of us know, is when your tests take forever to run, and then a flaky test fails, so we need to re-run everything *again* and wait forever *again*. It just adds insult to injury.

No matter how fast you make your CI suite run, if you need to run it again often, you're going to have a sad time.

And for this, I have two suggestions:

Final thoughts

Flaky Tests

```ruby
RSpec.configure do |c|
  c.example_status_persistence_file_path = "tmp/rspec_examples.txt"
end
```

```
bundle exec rspec --only-failures
```


First, rspec has a feature where it'll store in a file the tests that failed, and you can then execute it again and run only the failed tests.

The way this works is Rspec will store the test failures in a file that looks like this,

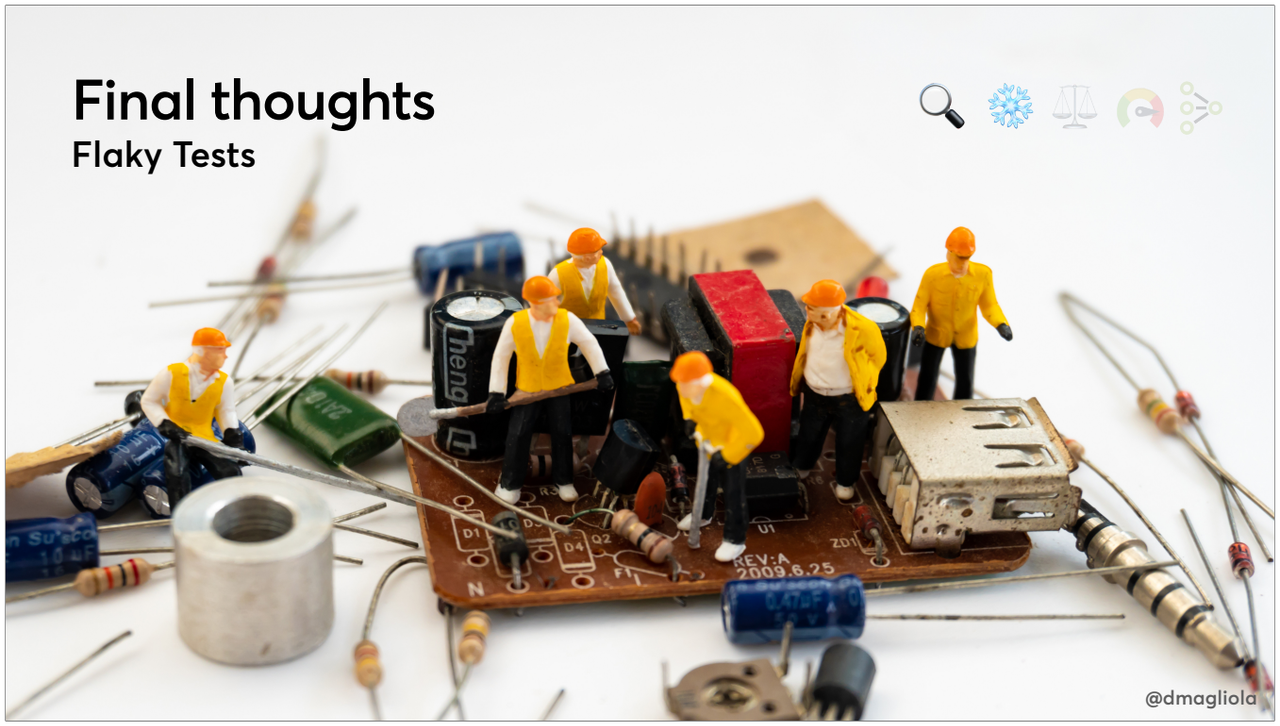
Final thoughts

Flaky Tests

```
example_id                          | status | run_time        |
----------------------------------- | ------ | --------------- |
./spec/lib/something2_spec.rb[1:1] | passed | 0.00087 seconds |
./spec/lib/something2_spec.rb[1:2] | passed | 0.00036 seconds |
./spec/lib/something_spec.rb[1:1]  | passed | 0.00006 seconds |
./spec/lib/something_spec.rb[1:2]  | passed | 0.00004 seconds |
./spec/lib/something_spec.rb[1:3]  | passed | 0.00004 seconds |
./spec/lib/something_spec.rb[1:4]  | passed | 0.00004 seconds |
./spec/lib/something_spec.rb[1:5]  | failed | 0.0105 seconds  |
./spec/lib/something_spec.rb[1:6]  | failed | 0.00026 seconds |
```


and will use that to know what to run next time.

So you can use this in CI, to always re-run your failures, in case they are flaky. This doesn't *fix* the problem, but it makes it less likely it'll make CI red.



The other thing you want to do is fix your flakies, and for this, it's important to think about motivations.

When you get a flaky, you obviously would like to fix it, like the good developer you are. But you were actually trying to get something done, and you need to ship that thing, and the flaky is getting in your way, and the thing it's testing is owned by another team anyway, so you don't really know what's up with it, so you just retry the branch and move on with the actual thing you were trying to achieve.

We all do it. We don't like it, but life, right. And then we do a "bug bash", or a "hackathon", or whatever, but no one is actually keeping track of what tests are flaky, so you can't fix them either. It's hard to actually get around to fixing these.

But we *can* do better, by having robots help us keep track of them, which gives us the right alignment.
What you want to do is automatically detect these flakies, and create a ticket in your bug tracker.

Final thoughts

Flaky Tests

```ruby
RSpec.configure do |c|
  c.example_status_persistence_file_path = "tmp/rspec_examples.txt"
end
```

```
bundle exec rspec
cp tmp/rspec_examples.txt tmp/rspec_examples_first.txt
bundle exec rspec --only-failures
```


The basic idea of how you do this is once you've run and had failures, you make a copy of the failures file, and run again with `--only-failures`.

Final thoughts

Flaky Tests

`tmp/rspec_examples_first.txt`

example_id	status	run_time
./spec/lib/something2_spec.rb[1:1]	passed	0.00087 seconds
./spec/lib/something2_spec.rb[1:2]	passed	0.00036 seconds
./spec/lib/something_spec.rb[1:1]	passed	0.00006 seconds
./spec/lib/something_spec.rb[1:2]	passed	0.00004 seconds
./spec/lib/something_spec.rb[1:3]	passed	0.00004 seconds
./spec/lib/something_spec.rb[1:4]	passed	0.00004 seconds
./spec/lib/something_spec.rb[1:5]	failed	0.0105 seconds
./spec/lib/something_spec.rb[1:6]	failed	0.00026 seconds

`tmp/rspec_examples.txt`

example_id	status	run_time
./spec/lib/something2_spec.rb[1:1]	passed	0.00087 seconds
./spec/lib/something2_spec.rb[1:2]	passed	0.00036 seconds
./spec/lib/something_spec.rb[1:1]	passed	0.00006 seconds
./spec/lib/something_spec.rb[1:2]	passed	0.00004 seconds
./spec/lib/something_spec.rb[1:3]	passed	0.00004 seconds
./spec/lib/something_spec.rb[1:4]	passed	0.00004 seconds
./spec/lib/something_spec.rb[1:5]	failed	0.0105 seconds
./spec/lib/something_spec.rb[1:6]	passed	0.00026 seconds

❄️ Flake!

@dmagliola

And now you have *two* failure files, one for each run, which should be identical. If they aren't, then you have a flaky test.

With a bit of Bash hackery, you can find tests that have failed in the first run but succeeded in the second one.

Final thoughts

Flaky Tests

```
grep "| failed" rspec_examples_first.txt \
| cut -d" " -f1 \
| xargs -I{} grep -F {} rspec_examples.txt \
| grep "| passed" \
| cut -d"[" -f1 \
| uniq \
| xargs -I{} echo "Flaky ${SUITENAME} spec: {}" \
| xargs -I{} jira create (...)
```

Find the whole thing, documented, at:
https://bit.ly/faster-ci

Don't look at that code too hard, you can find the complete thing, explained, in the supplementary repo.

But you can find those flakies and pass their paths to a utility that'll create a ticket in JIRA or your bug tracker of choice to fix that test. Jira has a CLI that will do this for you, which you can install on your CI Dockerfile, other bug trackers also have CLIs you can use, or worst case scenario, you can `curl` into their API.

And now, there's an actual ticket, opened by a robot, so it's not even you being annoying or anything... There's a ticket which someone needs to triage, and assign to the right team, and... it'll get prioritized later, sure. But now, it's visible. And it's trackable. And trackable, means it's fixable.

We've done this and it's massively helped us reduce the number of flakies we have, because now it's a ticket that is someone's problem, even if you batch them and solve them later... it's much more actionable than having something that's gets in your way, at the worst possible time, that you just "retry" to get on with it and move on with your life.

Final thoughts

Uneven distribution of tests

Ok, let's talk about uneven distributions...

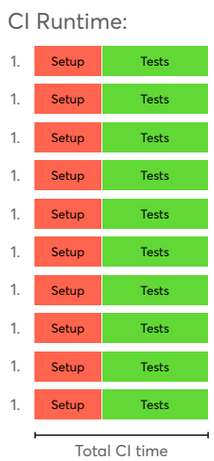
I talked a lot earlier about how setup and startup times are the main barrier to lots of parallelism, because if your tests take a long time to even start, pretty quickly you get to a point where adding more machines isn't very helpful.

But there's also another barrier you can hit: tests distributed unevenly between machines.

The idea of parallelizing is you're going to go from this

Final thoughts

Uneven distribution of tests



The dream

@dmagliola

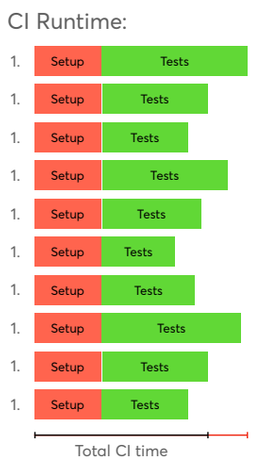
to this, right?

We got 10 machines, so after setup costs, our tests should take a tenth of the time.

However, this is not quite what you get. What you get is a bit more like this...

Final thoughts

Uneven distribution of tests



The reality

Because some test files take longer than others, not all machines finish at the same time.

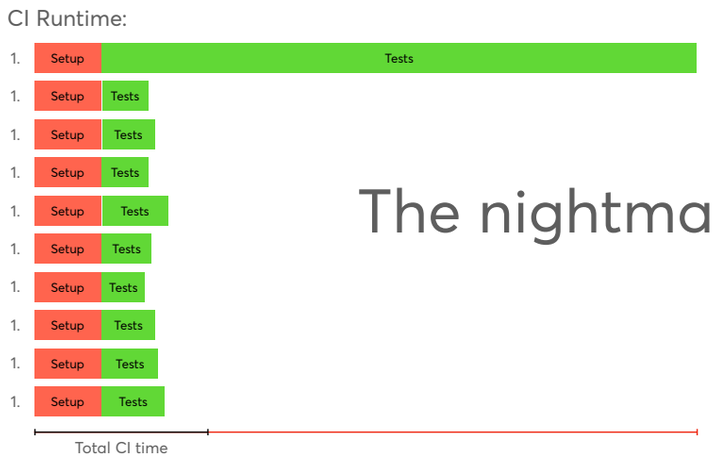
And now your CI time is as long as the longest-running machine.

Now, what you see here is actually quite good. If you are getting a distribution like this, you got pretty lucky. That little red line over there, that's how much extra time you got over the ideal scenario, that's not bad.

However, sometimes you get into a pathological situation where this happens

Final thoughts

Uneven distribution of tests



@dmagliola

Now, this is exaggerated for effect, you would have to be really unlucky to get this... But you kind of see the idea here.

If you have this distribution, adding more machines doesn't really help you that much, unless you get lucky and it rejiggles the files more favourably but hoping for that is not a good situation to be in.

The way to work around it is to distribute your files between your machines such that the timings end up more even. Unfortunately, THIS IS HARD. It's a very annoying problem.

I know of only two solutions, and much to my dismay, they involve mentioning commercial vendors.

Final thoughts

Uneven distribution of tests

```
circleci tests glob "spec/unit/**/*_spec.rb" | circleci tests split
```


One of them I mentioned in passing earlier: CircleCI has this CLI tool that helps you split the tests between machines.
This little tool can split things in many different ways, and one of them involves it storing in files how long each test took, with a file similar to that RSpec one I showed you for the flakies. It stores that file centrally, so it persists between builds, and it uses those timings to try and split things more fairly.

And it works pretty well, to be honest. So if you're on Circle already, you can just do that.

Final thoughts

Uneven distribution of tests

Knapsack

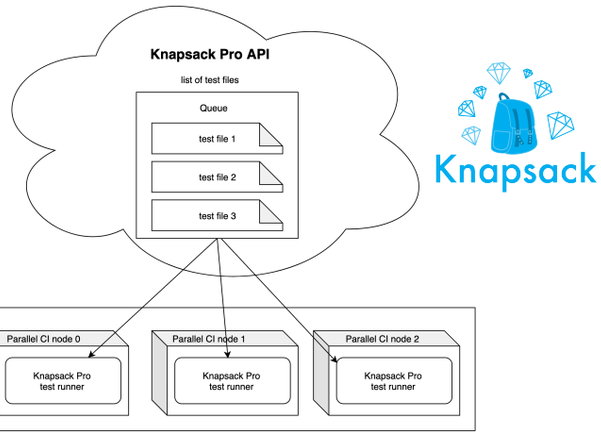
https://knapsackpro.com

@dmagliola

The other solution I know of is Knapsack.
Knapsack is a commercial solution that acts as an external queue, that each of your machines talks to and "pulls" some tests from.

Final thoughts

Uneven distribution of tests



The way it works is they run a server that knows all the files that have to be run, and each machine is repeatedly asking that server for more files to run... So a machine that churns through files faster gets more files, and the ones that are running longer files, basically end up getting fewer files, which evens things out.

Supposedly they also store past test timings and do a bunch of fancy magic to distribute things better. Hence the name, get it?. I'm not sure how much that helps, in my uninformed opinion, just having the central queue and the gradual pull is doing most of the heavy lifting there...

And a side advantage of knapsack is also that, for the CI providers that don't help you do parallelism, you no longer have to keep track of which machine is which, split the files manually, etc. You start as many machines as you want, and they all pull from the queue, which does make your life easier if your CI provider isn't.

It's not a super cheap solution, but it may be worth trying them and seeing if it helps

Now this is something i've thought about, but never actually tried, so take it with a pinch of salt...
But that Rspec failures file stores how long each test took...
After running your tests, you *could* do some pre-processing on it, and make a little file of your own of how long each FILE takes...
And you *could* persist that between builds using your CI's caching mechanism that you normally use for gems.
And you *could* have a little Ruby script in the middle of that command that splits your tests using awk, that sorts the list of files by time they take to run, descending...
And I think if you do that... You've made yourself your own poor man's Knapsack...

Now, again, I haven't tried it, so I may be missing something. And from the perspective of throwing money at the problem to save developer's time this is probably a bad idea... But if uneven tests are killing you, and for some reason you can't use Knapsack... It may be worth trying?
Again, wild thought, but under the right circumstances, maybe?

Ok, so we've turned our containers into lean, mean testing machines, they start up super fast, and we're running them dozens at a time, they finish evenly, our CI times are amazing, we can see that in our beautiful dashboards, and all our flakies are gone.

AWESOME, right?

So now for the bad news...
As I mentioned earlier, execution time can vary a lot depending on how sad the machine you're running in is, and this will always affect your CI times a bit.

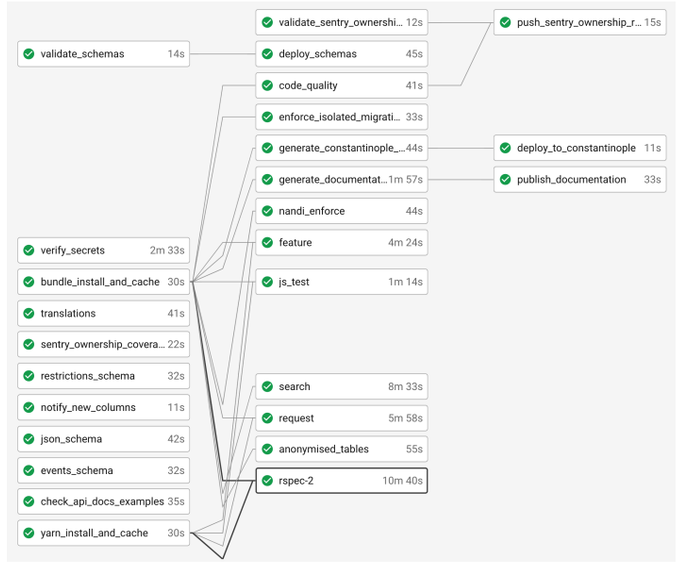
But sometimes, you're going to get a machine that is really, really sad, and it'll just take forever to run your tests. Or the network will fail you a little bit, and a step that should be instant will take 10 minutes. Or, just, literally forever, it may get stuck and never actually finish and you need to manually cancel it. This doesn't happen often, but when it happens, it's really sad, and more importantly, it kind of negates a lot of the improvements we've made.

And so here's the sad part... The more machines you run in parallel, the faster your tests will finish... But also the more likely it is that one of them is a sad one that'll take a really long time. Now, again, this happens very infrequently, but if you're running, say, 64 machines, it'll happen on many more pushes than if you're running in 4. So you need to keep an eye on this. The more machines you add, the faster it'll run, until you'll hit a point where things may start actually taking longer on average because of this problem.

Sadly, there isn't a silver bullet for this, it's a classic trade-off, and the best we can do is have good observability which will let us figure out what is the sweet spot that'll give us, on average, the lowest CI times.

Final thoughts

Critical paths



@dmagliola

Finally, I want to make a quick note about critical paths.

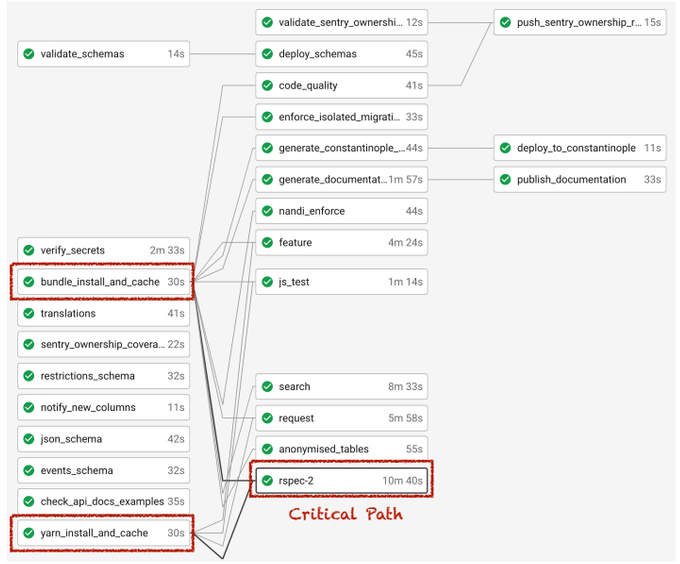
Depending on the complexity of your project, and how much tooling and automation you've developed, you may have a CI workflow that is quite complicated. This is ours.

One thing that is obvious if you think about it, but it's easy to lose sight of, is that the *only* thing you care about is how long it takes to get your branch to green. You don't care how long it takes each individual step to be green, you only care about how long it takes for the *last* one to be green.

So you should focus your efforts on the steps that are in the critical path.

Final thoughts

Critical paths



@dmagliola

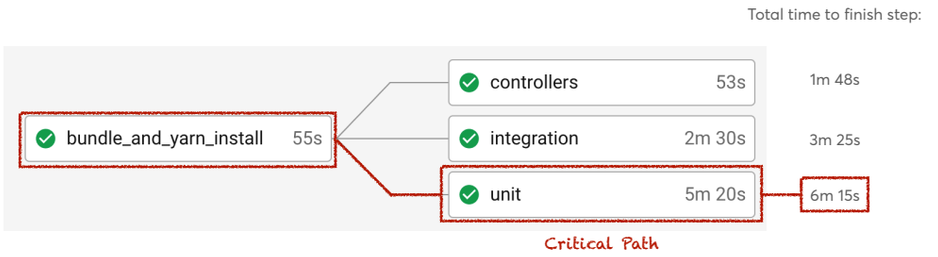
In this particular workflow, these THREE are the only steps that matter. Optimizing anything else is pretty much a waste of effort. It's possible that we could make this "secrets" step go faster, but it's not going to make the overall workflow faster, so you shouldn't bother.

And this is *especially* true if you're making those steps faster by adding parallelism, because it makes it more likely than one of those steps will hit a slow machine as I just mentioned, and now you've shot yourself in the foot. For the steps in the critical path, there's a sweet spot of parallelism, where you balance the time you gain with the risk of a sad machine. But for steps that are *not* in the critical path, if they are using *any* more machines than they need to, you're getting the extra risk with no benefit to offset it. And you're also spending more money on machines, again, without any actual benefit.

So focus on the critical path, and ignore everything else. Of course, as you make a step faster, you may remove it from the critical path. That's great, make it no faster than that, focus on the new critical ones.

Final thoughts

Critical paths



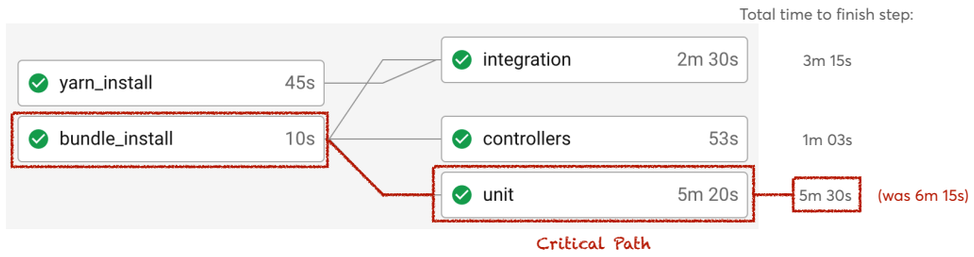
@dmagliola

Importantly, this means not only making those steps faster, but also considering those dependencies. There may be ways of restructuring your tasks so that one of your steps no longer depends on another, and you can gain a huge amount of time with this.

This is very typical if you have a "setup bundle and yarn" job, and a bunch of other steps depend on it... If your "unit tests" step is the slowest, and it's depending on that "setup bundle and yarn", this may be a bad idea. It probably doesn't need yarn,

Final thoughts

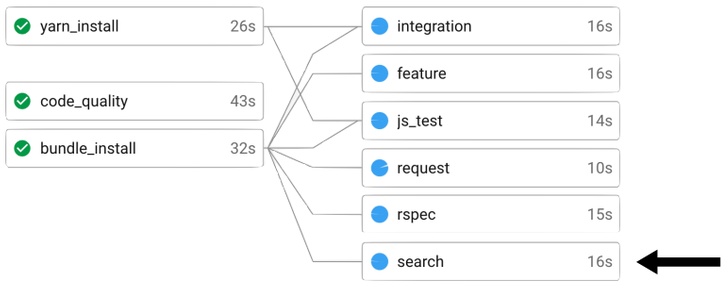
Critical paths



and you can make only the steps that *do* need it depend on it, and save some serious time on your critical path

Final thoughts

Critical paths



@dmagliola

Same goes for jobs that use a lot of different containers for their dependencies.
It's common for only a few tests to need all of those dependencies.

Separate those tests out into their own job, and only add those extra containers on *that* job, like we did with Search at the very beginning of this talk, so the rest of tests don't need to wait for ElasticSearch to boot up.

Recap

- Parallelize a lot, but optimize your startup times
  - Save time on Bundle Install, Git Checkout, and Container Spin up.
- Build your own images for absolute control on 👆
  - Keep those layers tight!
- Optimize your dependencies, improving your critical path
- Keep an eye on your runtimes over time
- Get rid of flakies by putting them in your backlog.

So, to recap...

So that's it from me.

I've covered a lot of different techniques just now, some of those will hopefully help.

Remember, you mileage WILL vary. Some of these will help in your particular scenario, some won't.

But hopefully this will give you some ideas on how to approach this problem, and some combination of these techniques will allow you to reach CI bliss.

Thank you

@dmagliola

Speed up your test suite by throwing computers at it

RailsConf 2021
April 12-15

Details, code, slides and more info at:
https://bit.ly/faster-ci

Daniel Magliola
@dmagliola