semaphore

CI/CD for Monorepos

Effectively building, testing, and deploying code with monorepos.

CI/CD for Monorepos

Effectively building, testing, and deploying code with monorepos

Semaphore

Contents

This book is open source: https://github.com/semaphoreci/book-monorepo-cicd

Published on the Semaphore website: https://semaphoreci.com

Sep 2022: First edition v1.0 (revision c5d9951)

Share this book:

> *I've just started reading "CI/CD for Monorepos", a free ebook by @semaphoreci: https://bit.ly/3yopUT2 (Tweet this!)*

Preface

A monorepo is a new name for an old idea — placing a bunch of software projects into the same code repository. Organizations that overcome the technical challenges associated with adopting monorepos enjoy significant benefits:

- **Cultural** — increased bandwidth of knowledge transfer and a higher level of collaboration among teams.
- **Technical** — common coding and tooling standards, simplified dependency management, and configuration reuse.

Big companies like Google, Facebook, Twitter, and Airbnb have been using monorepos for years. Today, there are a growing number of smaller teams adopting monorepos.

Why the change now? On the frontend side, the proliferation of JavaScript-based tools is such that it is possible to develop very complex applications in a single programming language. Architects of frontend projects now face the problems of separating concerns and avoiding code duplication — and they have a good set of tools to solve these problems by working in a monorepo.

On the backend side, serverless and microservices-based architectures drive developers to logically isolate their code into small units. Most of these services are written with the same set of tools and coding standards, and built, configured, and deployed in the same way. Placing them into a monorepo is an efficient way of avoiding duplicate configurations and processes.

At Semaphore, we have observed this trend in a growing number of teams and have solved the technical challenge of running effective CI/CD pipelines for monorepos.

Using traditional CI/CD tools in the monorepo context, developers essentially need to build, test, and deploy all services all the time. Using Semaphore, developers run dynamic CI/CD workflows that run the right pipelines at the right time. This gives product teams more time to focus on building the next great feature.

Who Is This Book for, and What Does It Cover?

This book is intended for software engineers who are either exploring using a monorepo for software development or looking to optimize the CI/CD process for their monorepo.

By showing what it takes to build a monorepo-first CI/CD pipeline that saves time and speeds up software development cycles, we hope that CTOs and other engineering leaders will be able to determine if monorepos are the way forward for their companies and teams.

Chapter 1, "Introduction to Monorepo", introduces the basics and relates stories about other companies that have successfully migrated to a monorepo. This chapter will help you decide if a monorepo is right for you.

Chapter 2, "Continuous Integration", explains what you need to know about setting up a CI pipeline that builds and tests only the code that changes.

In chapter 3, "Continuous Integration Demo", we apply the knowledge gained so far into building and testing a demo monorepo with working microservices.

Chapter 4, "Continuous Deployment", describes how to expand the CI pipeline with continuous deployments. We'll learn how to implement a continuous deployment pipeline on top of a working project.

How to Contact Us

We would very much love to hear your feedback after reading this book. What did you like and learn? What could be improved? Is there something we could explain further?

A benefit of publishing an ebook is that we can continuously improve it. And that's exactly what we intend to do based on your feedback.

You can send us feedback by sending an email to learn@semaphoreci.com.

Find us on Twitter: https://twitter.com/semaphoreci

Find us on Facebook: https://facebook.com/SemaphoreCI

Find us on LinkedIn: https://www.linkedin.com/company/rendered-text

About the Author

Pablo Tomas Fernandez Zavalia is an electronic engineer and writer. He started out developing for the City Hall of Buenos Aires (buenosaires.gob.ar). After graduating, he joined British Telecom as head of the Web Services department in Argentina. He then worked for IBM as a database administrator, where he also did tutoring, DevOps, and cloud migrations. In his free time, he enjoys writing, sailing, and board games. Follow Tomas on Twitter at @tomfernblog.

About the Editor

Marko Anastasov is a software engineer, author, and entrepreneur. Marko co-founded Rendered Text, the software company behind the Semaphore CI/CD service. He worked on building and scaling Semaphore from an idea to a cloud-based platform used by some of the world's best engineering teams. Follow Marko on Twitter at @markoa.

1 Introduction to Monorepos

Monorepos can be a great force for fostering rapid development workflows. But, are they the right fit for you, your team, and your company?

1.1 What Is a Monorepo?

Not everyone agrees on a single definition for *monorepo*. Some may only accept the term when it applies to companies hosting *all* their code in a single repository. Google is the most famous example of this; their monorepo is theorized to be the largest code repository in the world, which has thousands of commits per day and exceeds 80 TBs in size.

More relaxed definitions will say that *a monorepo is a version-controlled code repository holding a number of independently-deployable projects*. While these projects may be related, they are often separate, logically-independent, and run by different teams. For instance, Airbnb has two monorepos: one for the frontend code and one for the backend code. In this way, a company or organization can utilize multiple monorepos.

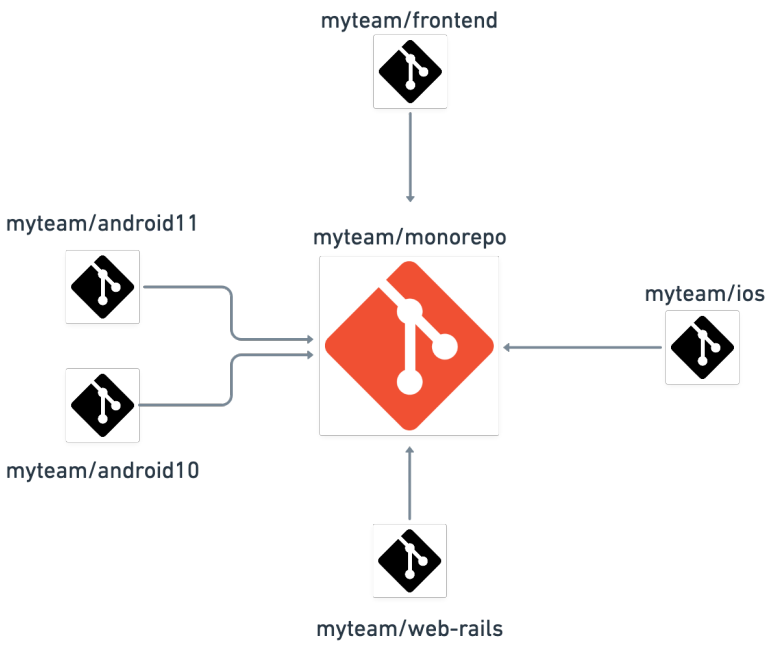
Monorepos are sometimes also called *monolithic repositories*, but they should not be confused with *monolithic architecture*, a software development practice for writing centralized applications using a single codebase. To give one example to these kinds of architectures, think of a Ruby on Rails application handling websites, API endpoints, and background jobs all at once.

1.2 Monorepos vs. Multirepos

The opposite of the monorepo is a *multirepo*, or simply *repos*, where each project is held on a completely separate, version-controlled software repository. Multirepos come naturally — it's what we do when starting a new project. After all, who doesn't like starting fresh?

Moving from multi to monorepo is merely a matter of moving all your projects into a single repository.

```
$ mkdir monorepo
$ git init
$ mv ~/src/app-android10 ~/src/app-android11 ~/src/app-ios .
$ git add -A
$ git commit -m "My first monorepo"
```



Of course, this is just to get started. The hard work comes later, when we get into refactoring and consolidation. To enjoy the full benefits of a monorepo, all shareable code should be moved outside of each project folder and into a common location.

Multirepos are not a synonym for *microservices*. In fact, having one does not require using the other. Later, we'll discuss companies using monorepos *with* microservices. A monorepo can host any number of microservices as long as you carefully set up your Continuous Integration and Delivery (CI/CD) pipeline[1] for deployment.

1.3 What Monorepos Bring to the Table

At first glance, the choice between monorepos and multirepos might not seem like a big deal. On closer inspection, however, it's a decision that deeply influences how you and your team interact.

Monorepos have the following benefits:

- **Visibility**: everyone can see everyone else's code, leading to better collaboration and cross-team contributions. Any developer can fix a bug

[1]CI/CD Pipeline, A Gentle Introduction - *https://semaphoreci.com/blog/cicd-pipeline*

in your code before you even notice it.

- **Simpler dependency management**: sharing dependencies is trivial. There's little need for a complex package manager setups as all modules are hosted in the same repository.
- **Single source of truth**: one version of every dependency means there are no versioning conflicts and no dependency hell.
- **Consistency**: enforcing code quality standards and a unified style is straightforward when you have your entire codebase in one place.
- **Shared timeline**: breaking changes in APIs or shared libraries are immediately exposed, forcing different teams to communicate and join forces. Monorepos keep everyone invested in keeping up with changes.
- **Atomic commits**: atomic commits make large-scale refactoring possible. In theory, a developer can update several packages or projects at once in a single commit. In practice, these types of changes are usually rolled out in stages, not all at once.
- **Implicit CI**: continuous integration is guaranteed as all the code is already integrated into one place.
- **Unified CI/CD process**: you can use the same CI/CD deployment process for every project in the repo.

1.4 Technical Challenges

As monorepos grow, we reach design limits in version control tools, build, systems, and continuous integration solutions. These problems can make a company go the multirepo route:

- **Bad performance**: monorepos can be difficult to scale up. Commands like `git blame` take unreasonably long, IDEs begin to lag, and testing the whole repo for every change becomes infeasible.
- **Broken main/master**: a broken master affects everyone working in the monorepo. This can be seen as either disastrous or as a good motivation to keep tests clean and up to date.
- **Learning curve**: the learning curve for new developers is steeper if the repository spans many tightly-coupled projects. Keep in mind, however, that the same can be the case with multi-repos.
- **Large volumes of storage**: monorepos can reach unwieldy sizes and very large quantities of commits per day.
- **Ownership**: maintaining ownership of files is more challenging. Systems like Git or Mercurial don't feature built-in directory-level permissions.
- **Code reviews**: notifications can get very noisy. For instance, GitHub

sends notifications about PRs to every developer in the repository.

You may have noticed that these problems are mostly technical. Some of them can be mitigated by adopting the *trunk-based development* model, which encourages engineers to collaborate in a single branch — the trunk — and proposes limiting the lifespan of topic branches to a minimum.

1.5 It's Not (Only) about Technology

Choosing a repository strategy is not only a technical matter but also about how people communicate. As stated by Conway's Law, communication is essential for building great products:

> Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.
>
> — Melvin E. Conway

While multirepos allow each team to manage their projects independently, they also put up communications barriers. In that way, they can act as blinders, making developers focus only on the part they own, forgetting the overall picture.

A monorepo, on the other hand, works as a central hub, a market square of sorts where developers, engineers, testers, and business analysts meet and talk. Monorepos encourage conversations while helping bring silos down.

1.6 Notable Monorepo Adopters

Open-source projects, by their nature, have more freedom to experiment and feel greater pressure to self-organize. For three decades, FreeBSD has used CVS and later subversion monorepos for development and package distribution. Other notable projects with monorepo support or that are monorepos themselves are Babel, Google's Angular, Facebook's React and Jest, and Gatsby.

Comercial companies have also posted about their journey towards monorepos. Besides the big ones like Google, Facebook, or Twitter, we find some interesting cases such as:

- Segment.com[2]: a company offering an event collection and forwarding service. Initially, they used one repository per customer. As the number of customers increased, they moved their 140 repositories into a single one.

[2]Goodbye Microservices - *https://segment.com/blog/goodbye-microservices/*

They migrated all the services and dependencies into their monorepo. While the transition was successful, it was very taxing as they had to reconcile shared libraries and test everything each time. Still, the end result was reduced complexity and increased maintainability.

- Airbnb[3]: initially ran on Ruby on Rails. Their "monorail" accompanied the company's exponential growth, until it didn't. Eventually, it was obvious that the rate of changes and number of commits was too much for a single repository. After some debate, they chose to split development into two monorepos: one for the frontend and one for the backend. Both comprise hundreds of services, the documentation, Terraform and Kubernetes resources for deployment, and all the maintenance tools.
- Pinterest[4]: has an ongoing three-year-long migration. The plan is to move more than 1300 repositories into only four monorepos and then consolidate hundreds of dependencies into a monolithic web application. The objective is to get a more uniform build process and higher quality standard. Automation, simplification, and standardization of release practices allowed them to cut down on boilerplate and let developers focus on writing code.
- Uber[5]: their build system used to be a combination of the Golang toolchain and Make. As they moved their mobile development to the monorepo and the number of files reached the 70 thousand mark, Make no longer fulfilled their needs. They elected to adopt Bazel, an offshoot of Google's build system, designed for scalability and featuring incremental builds, to which they ended contributing several patches and improvements. According to Uber, their monorepo is likely one of the largest Go repositories running on Bazel.

1.7 Investing in Tooling

If we have to take only one lesson from all these stories, it is that proper tooling is key for effective monorepos. Building and testing need to be rethought: instead of rebuilding the entire repo on each update, we can use smart build systems that understand the structure of the projects and work only on the parts that change.

[3]From Monorail to Monorepo, Airbnb's journey into Microservices - *https://www.youtube.com/watch?v=sakGeE4xVZs*

[4]Pinterest's journey to a Bazel monorepo - *https://www.youtube.com/watch?v=r5KHQnS6uP8*

[5]Building Uber's Go Monorepo with Bazel - *https://eng.uber.com/go-monorepo-bazel/*

On a high level, a smart build system would need to:

1. Determine which files changed due to commits since the last build.
2. Find all the projects and their dependencies affected by the changes.
3. Build these projects, ideally using some form of caching.
4. Run tests based on affected code.
5. Deploy the projects that have changed into staging or production.

Most of us, however, don't have Airbnb's resources. So, what can we do? Fortunately, many larger companies have open-sourced their build systems:

- Bazel: released by Google and based partly on their homegrown build system (Blaze). Bazel supports many languages and is capable of building and testing at scale.
- Buck: Facebook's open-source fast build system. Supports differential builds on many languages and platforms.
- Pants: The Pants build system was created in collaboration with Twitter and Foursquare. For the moment, it supports only Python, but more languages are on the way.
- RushJS: Microsoft's scalable monorepo manager for JavaScript.

Monorepos seem to be getting more attention, particularly in JavaScript, as shown by these projects:

- Lerna: monorepo manager for JavaScript. Integrates with popular frameworks like React, Angular, or Babel.
- Yarn Workspaces: installs and updates dependencies for Node.js in multiple places with a single command.
- ultra-runner: scripts for JavaScripts monorepo management. Works with Yarn, pnpm, and Lerna. Supports parallel building.
- Monorepo builder: installs and updates packages across PHP monorepos.
- NPM: since version 7, has support for workspace.

1.8 Scaling up Repositories

Source control is another sticking point for monorepos. These tools can help you scale up repositories:

- Virtual Filesystem for Git (VFS): adds streaming support for Git. VFS downloads objects from Git repositories as needed. This project was originally created to manage the Windows codebase (which is the largest Git repository). Works only in Windows, but MacOS support has been announced.

- Large File Storage: an open-source extension for Git that adds better support for large files. Once installed, you can track any type of file and seamlessly upload it into cloud storage, freeing up your repository and making pushing and pulling much faster.
- Mercurial: an alternative to Git, Mercurial is a distributed version control tool that focuses on speed. Facebook uses Mercurial and has contributed many speed-enhancing patches over the years.
- CODEOWNERS: lets you define which team owns subdirectories in the repository. Code owners are automatically requested to review when someone opens a pull request or pushes into a protected branch. This feature is supported by GitHub and GitLab.

1.9 Best Practices for Monorepo Management

Based on what we have learned about monorepos, we can define a minimum set of best practices:

- Define a unified directory organization for easy discovery.
- Maintain branch hygiene. Keep branches small, consider adopting trunk-based development practices.
- Use pinned dependencies for every project. Upgrade dependencies all at once, force every project to keep up with the dependencies. Reserve exceptions for truly exceptional cases.
- If you're using Git, learn how to use shallow clone[6] and filter-branch[7] to handle large-volume repositories.
- Pick a smart build system like Bazel or Buck to speed building and testing.
- Use CODEOWNERS when you need to restrict access to certain projects.
- Use a cloud CI/CD platform like Semaphore to test and deploy your applications at any scale.

[6] Get up to speed with shallow clone - *https://github.blog/2020-12-21-get-up-to-speed-with-partial-clone-and-shallow-clone/*

[7] git-filter-branch reference page - *https://git-scm.com/docs/git-filter-branch*

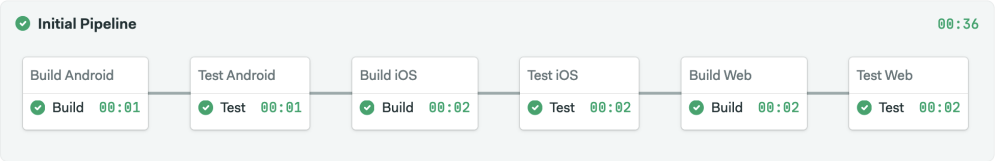
2 Continuous Integration for Monorepos

Monorepos are highly-active code repositories. The default behavior of continuous integration systems, which is to build, test, and deploy everything all the time, is suboptimal in the context of a monorepo. In this chapter you will learn how to use Semaphore's out-of-the-box support for monorepo CI/CD workflows.

2.1 The Challenge of CI/CD with Monorepos

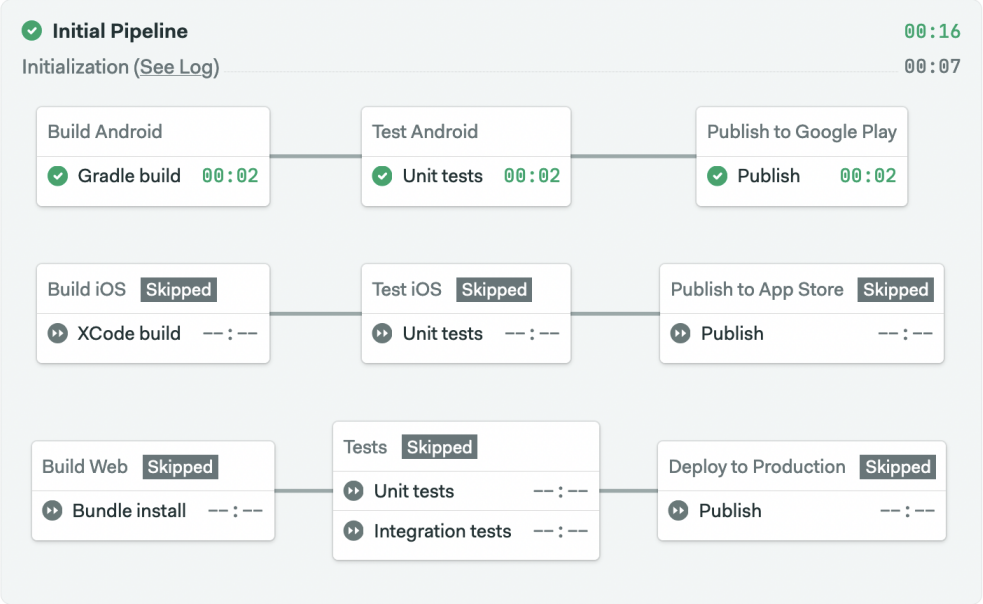
Properly implementing a CI/CD workflow with a monorepo presents its own set of challenges. By default, a CI/CD pipeline will run from beginning to end on every commit. This is expected. After all, that's what the "continuous" in continuous integration stands for.

A classic CI pipeline will run every job in sequence every time a new commit is pushed into the repository.



Running every job in the pipeline is perfectly fine on single-project repositories. But monorepos see a lot more activity. Even the smallest change will re-run the entire pipeline — **this is time-consuming and needlessly expensive**.

Semaphore is a CI/CD platform with native monorepo support. Its change-based, parallel execution feature lets you skip jobs when the relevant code has not changed. This will let you ignore parts of the pipeline you're not interested in re-running.



2.2 Hello World Monorepo with Semaphore

If you're new to Semaphore, spend 10 minutes reading the **getting started guide** to learn the basics of creating a pipeline. You'll find the guide here:

https://docs.semaphoreci.com/guided-tour/getting-started

Back? OK, let's walk through creating a monorepo pipeline.

To follow this guide, you'll need:

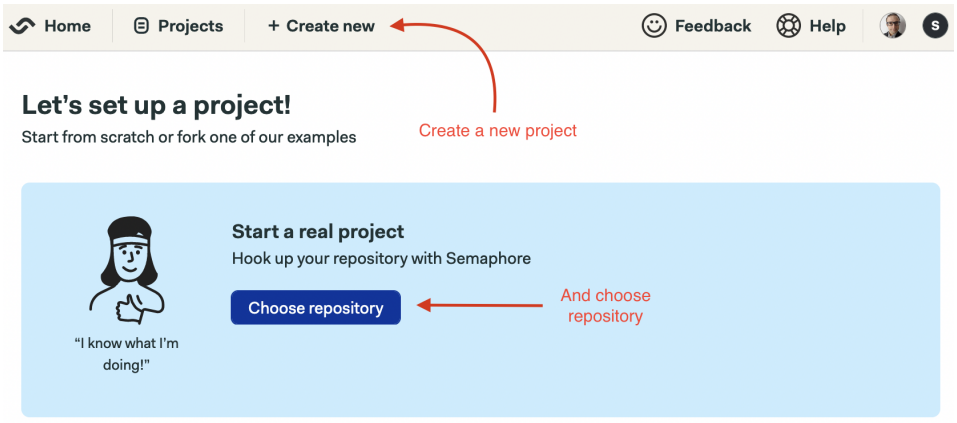
- A GitHub account.
- A Semaphore (*https://semaphoreci.com*) account. Click on *Sign up with GitHub* for a free trial or open source account.

Get started by creating a new repository on GitHub and cloning it to your machine. We'll call the repository "hello-semaphore".

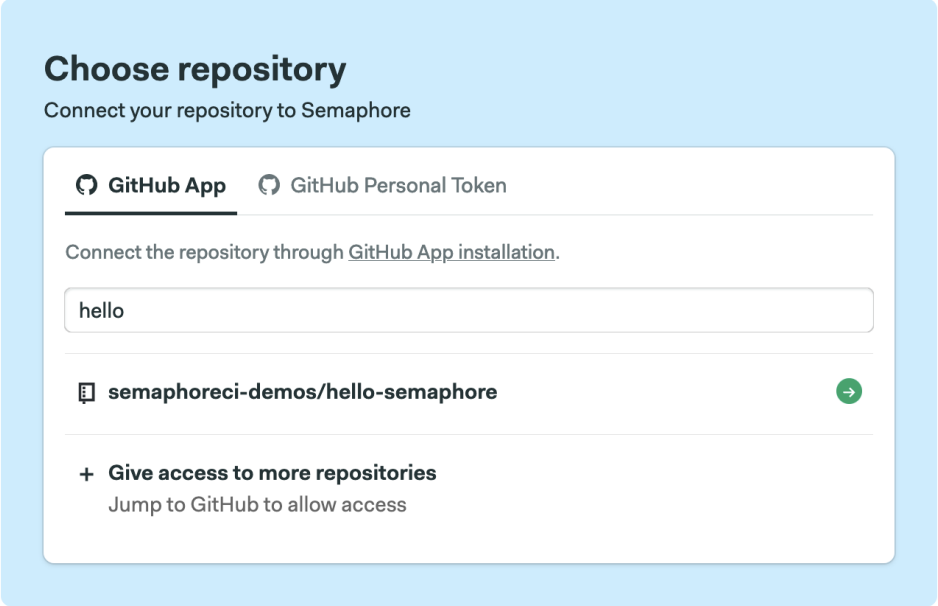
Create a couple of folders in the repository in order to try out change-based detection. Let's call them `service1` and `service2`:

```
$ mkdir service1 service2
$ touch service1/README.md service2/README.md
$ git add .
$ git commit -m "create dummy services"
$ git push
```

Next, log in with your Semaphore account and click on *Create New* in the upper left corner.

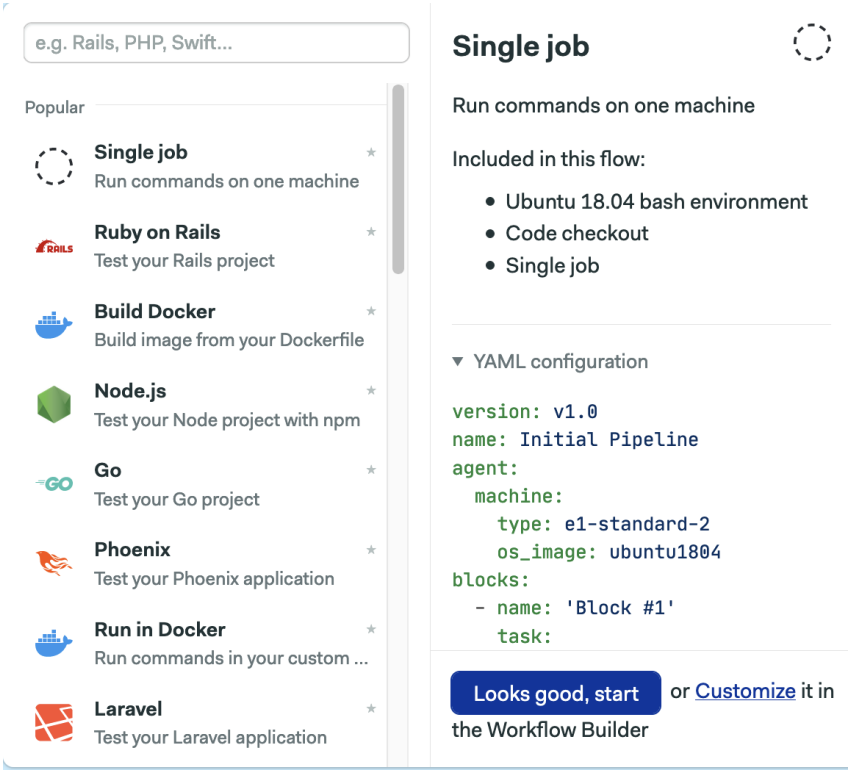


After choosing the "hello-semaphore" repository, wait a few seconds for Semaphore to initialize the project.

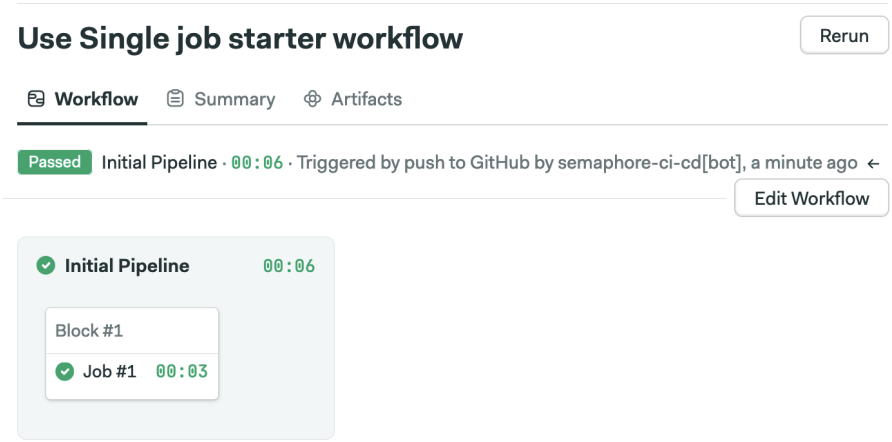


The next screen lets you add people to the project, which we don't need to do for now. Go ahead and click *Continue to Workflow Setup* to proceed.

Finally, you'll reach the template selection screen, select *Single job*, then click *Looks good,* followed by *start*.

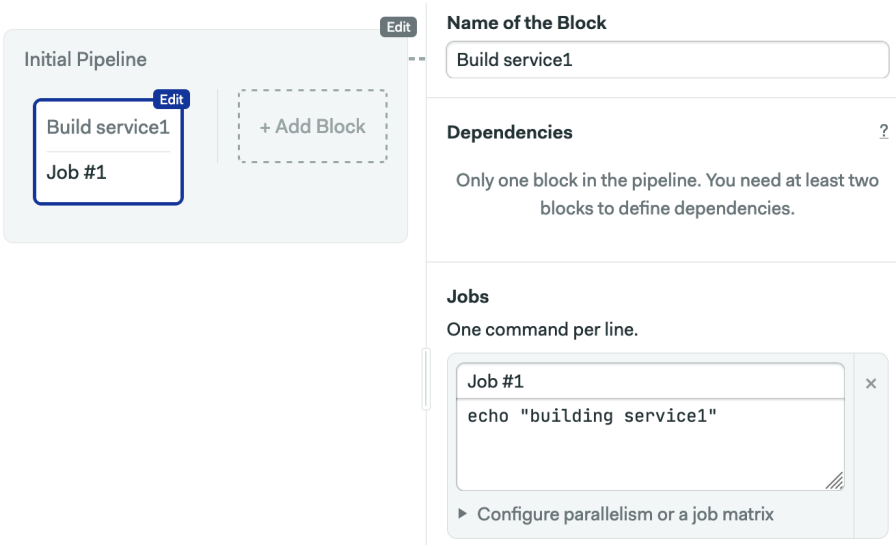


The initial workflow should start immediately.

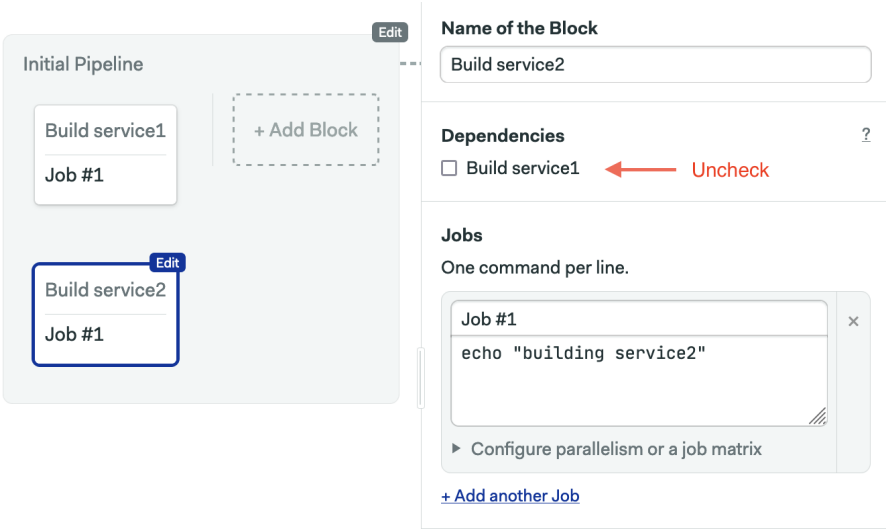


Now click on *Edit Workflow* to edit the pipeline.

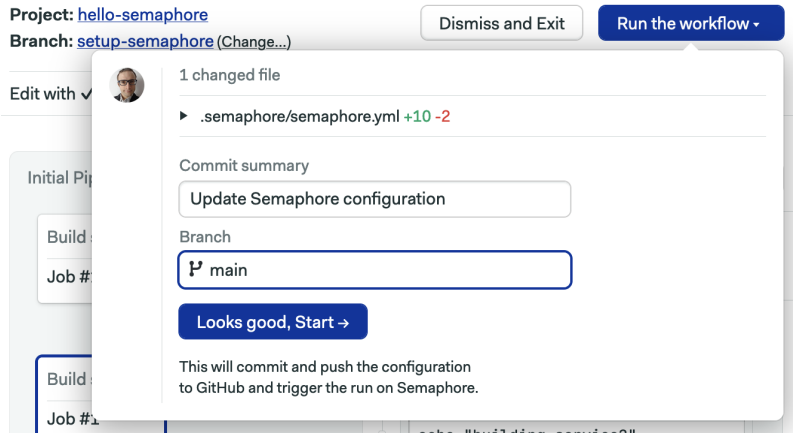
In this screen you can modify and create new blocks in the pipeline. Rename the block to "Build service1" and add the following command: `echo "building service1"`.



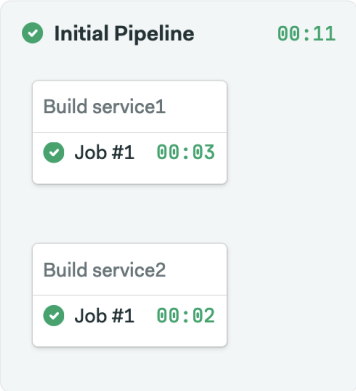
Click on *Add Block*, the new block is called "Build service2". Uncheck the Build service1 in dependencies. This causes both blocks to run in parallel. For the command, type `echo "building service2"`.



Click on *Run this Workflow*, change the branch to the default branch your repository uses (usually, it's called `main`) and click on *Start*.



Both blocks should run in parallel.



2.3 Change-Based Execution

Let's pause for a moment to learn about `change_in`. The change_in[8] function calculates if recent commits have changed code in a given file or folder. This function must be called at the block level. If it detects changes, then all the jobs in the block will be executed. Otherwise, the whole block is skipped. The end result is that this function allows us to tie a specific block to parts of the repository.

The basic usage of the function is:

```
change_in('/web/')
```

[8]Function change_in reference page - *https://docs.semaphoreci.com/reference/conditions-reference/#change_in*

This will run the block if any files inside the `web` folder have changed. Absolute paths start with `/` and reference the root of the repository. Relative paths don't start with a slash, they are relative to the pipeline file, which is located inside `/.semaphore` by default.

We can also target a specific file:

```
change_in('../package-lock.json')
```

Wildcards are supported too:

```
change_in('/**/package.json')
```

Also, you're not limited to monitoring one path, you may define lists of files or folders. The following statement, for instance, will run when the `/web/` folder **or** the `/manifests/kubernetes.yml` file changes (both simultaneously changing work too):

```
change_in(['/web/', '/manifests/kubernetes.yml'])
```

The function can take a second optional argument to further configure its behavior. For instance, if your repository default branch is `main` instead of `master` (GitHub's new default), you'll need to add `default_branch: 'main'`:

```
change_in('/web/', { default_branch: 'main' })
```

Semaphore will re-run all jobs when we update the pipeline, even if no other files have changed. We can disable this behavior with `pipeline_file: 'ignore'`:

```
change_in('/web/', { pipeline_file: 'ignore' })
```

Another useful option is `exclude`, which lets us ignore files or folders.For example, we can ignore all Markdown files with:

```
change_in('/web/', { exclude: '/web/**/*.md' })
```

To see the rest of the options, check the conditions YAML reference[9].

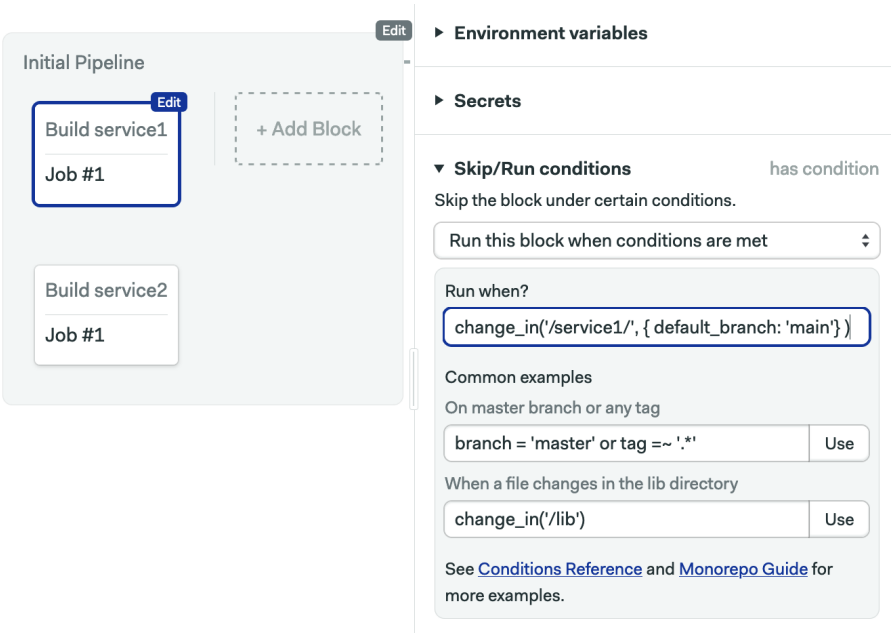
2.4 Using change_in to Speed up Pipelines

In our CI pipeline there is no change detection yet; we'll remedy that now. Click on *Edit Workflow* to re-open the Workflow Builder.

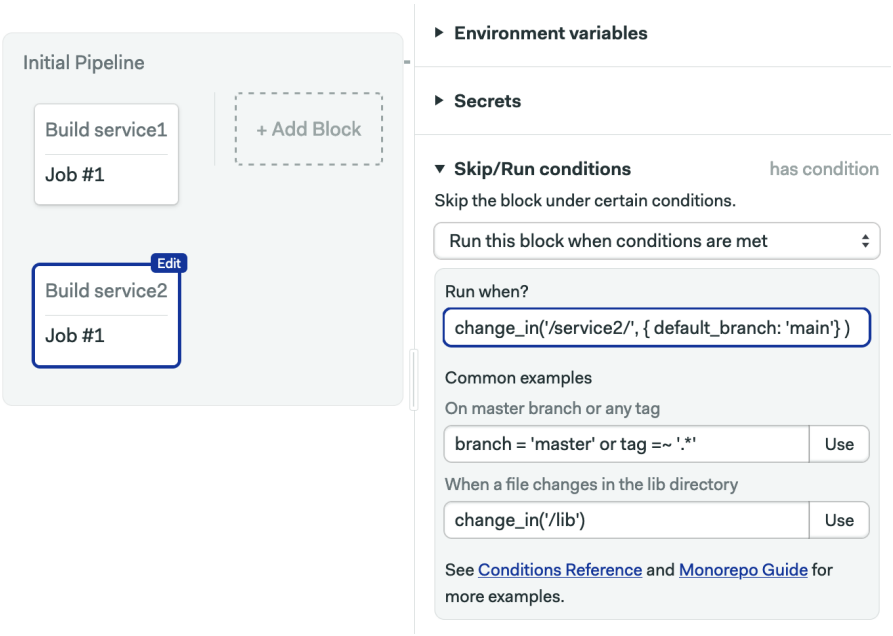
[9]Conditions reference page - *https://docs.semaphoreci.com/reference/conditions-reference/*

On the first block, scroll down until you reach the section *Run/skip Conditions* and enable the option: "Run this block when conditions are met".

Type the following condition: `change_in('/service1/', { default_branch: 'main'} )`. If your repository's default branch is `master` you can skip the `default_branch` option altogether.



Go to the second block and type this condition: `change_in('/service2/', { default_branch: 'main'} )`.

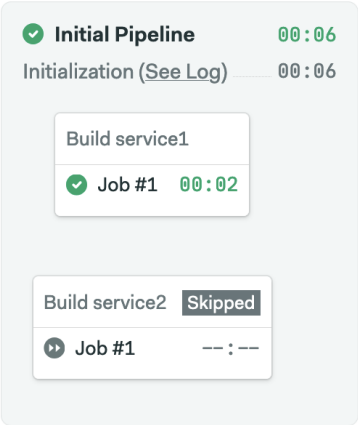


Click on *Run the Workflow > Start* to save the pipeline. Next, run the pipeline again. The first thing you'll notice is that there's a new initialization step. Here, Semaphore is calculating differences in order to decide which blocks should run. You can check the log to see what is happening behind the scenes.

Once the workflow is ready, Semaphore will start running all jobs one more time (this happens because we didn't set `pipeline_file: 'ignore'`). The interesting bit comes later, when we change a file in one of the applications.

```
$ git pull
$ echo "modify service1" >> service1/README.md
$ git add service1
$ git commit -m "modify service1"
$ git push
```

This is what we get:



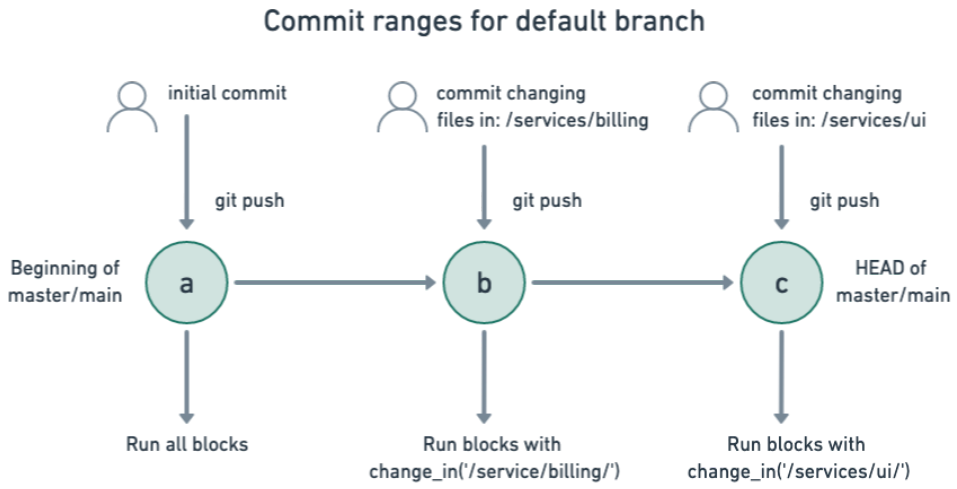
Two things have happened now that change-detection is enabled on the pipeline:

- A new initialization log is shown in the pipeline. The log is the output of Semaphore's initialization job, which reveals what folders or files have been marked as changed.
- Semaphore has detected that some parts of the monorepo have not changed and has skipped the related block. The improved pipeline can now selectively build the monorepo.

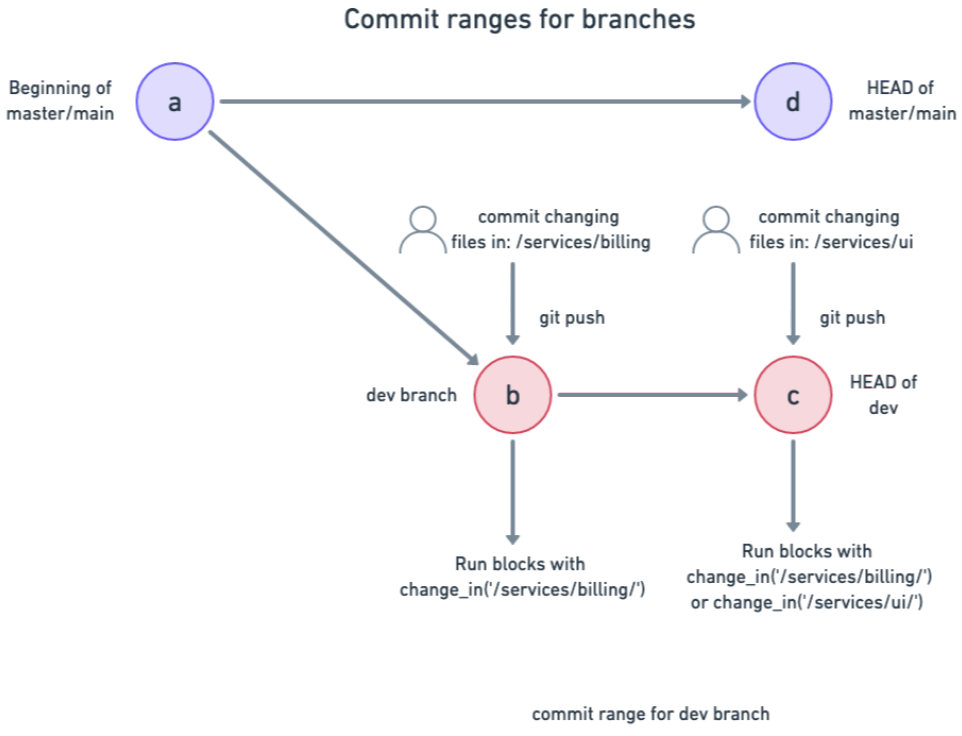
2.5 How Semaphore Identifies Changes

To understand what blocks will run each time, we must examine how `change_in` calculates the changed files in recent commits. The commit range varies depending on whether you're working on `main/master` or a topic branch.

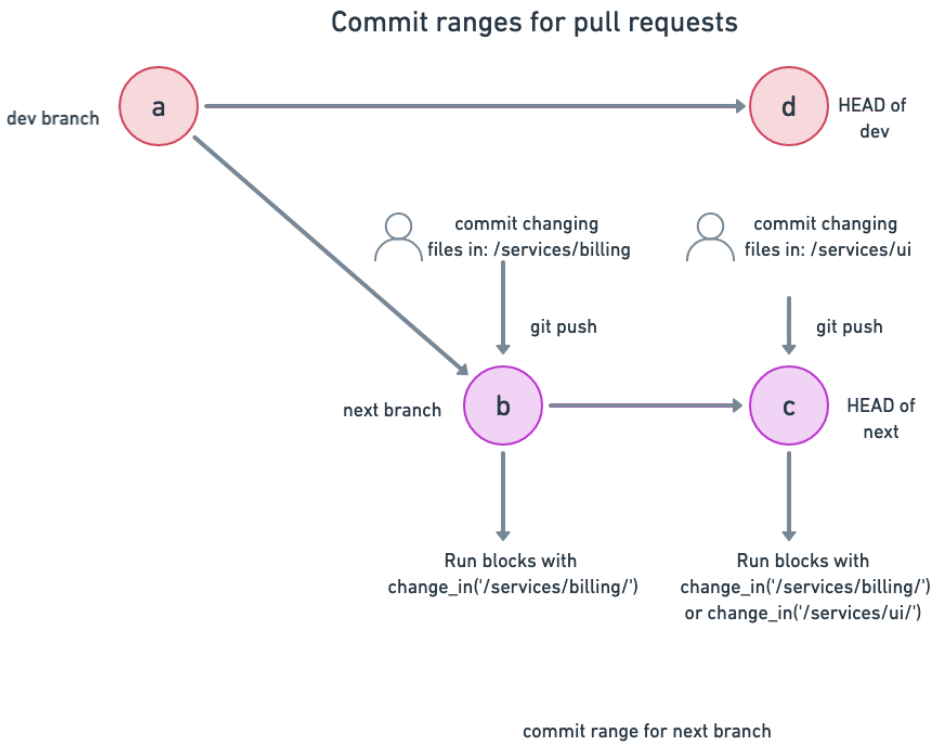
For the main branch, Semaphore compares the changes in all the commits for the push, then skips the `change_in` blocks that do not have at least one match.



Semaphore takes a broader criteria for branches. The commit range goes from the point of the first commit that branched off the mainline to the branch's head. This explains why Semaphore may choose to re-run blocks even on commits that seemingly don't match the change criteria.



Pull requests behave similarly.



The commit range is defined from the first commit that branched off the branch targeted for merging to the head of the branch.

3 Continuous Integration Demo

In the previous chapter, we learned the basics of creating monorepo pipelines using Semaphore. Our CI build was limited to conditionally running essentially empty jobs. We will now expand our knowledge by building a realistic pipeline that deals with dependency management and caching, compiling code, and running tests.

To make it easy to follow along, we have prepared a demo made of three microservices. It works well to show how everything we've seen thus far fits together. And it will act as a springboard that takes us into continuous delivery in the next chapter.

3.1 Monorepo Demo

The demo project we're going to use contains three microservices. The code is located in the `services` folder:

- `/services/user`: a Ruby-based user registration service. Exposes a HTTP REST endpoint.
- `/services/billing`: written in Go. Stores payment details.
- `/services/ui`: is the frontend and is written in Elixir.

All these parts are meant to work together, but each one can be maintained by a separate team and written in a different language.

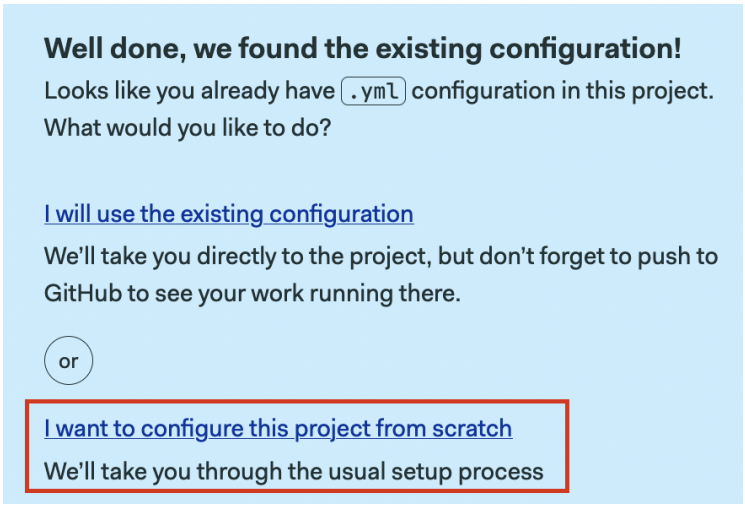
Before moving on, go ahead and fork the repository and clone it into your machine:

https://github.com/semaphoreci-demos/semaphore-demo-monorepo

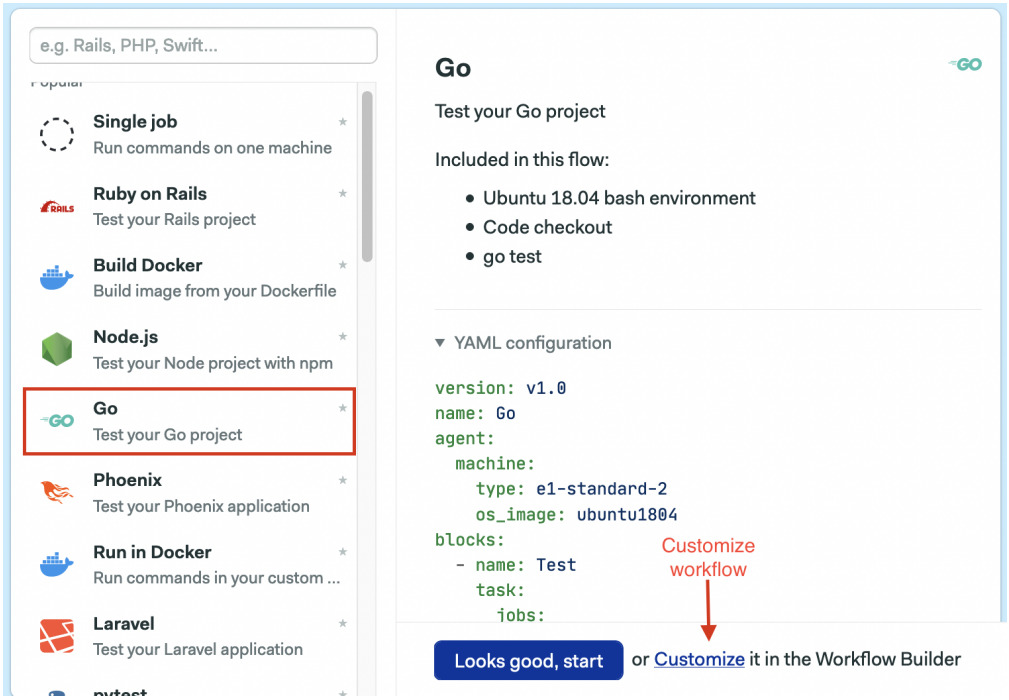
3.2 Setting up the Pipeline

To begin, create a new project in Semaphore and select the demo repository. Alternatively, if you prefer to jump directly to the final state, find the monorepo example and click the *Fork & Run* button.

The repository ships with a ready-to-use pipeline, but we'll learn a lot more by setting it up manually. Therefore, when prompted, click on "I want to configure this project from scratch".



We'll start with the Billing application. Find the *Go Workflow* and click on *Customize*:



3.2.1 Billing Service

The first block in the pipeline builds and tests the Billing service. The starter job uses uses two new commands:

- checkout[10]: clones the repository into the Semaphore machine and changes the current directory.
- sem-version[11]: switches the current version of a language.

We have to modify the job in two places before it will work with the project:

1. The app has been tested on Go version 1.14+. So, add this line to the beginning of the job `sem-version go 1.14`.
2. Since the code is located in the `services/billing` folder, add `cd services/billing` after `checkout`.

The full job should look like this:

```
sem-version go 1.14
export GO111MODULE=on
export GOPATH=~/go
export PATH=/home/semaphore/go/bin:$PATH
checkout
cd services/billing
go get ./...
go test ./...
go build -v .
```

The last three commands use Go's built-in toolset to download dependencies, test, and build the microservice.

[10]Checkout reference page - *https://docs.semaphoreci.com/reference/toolbox-reference/#checkout*

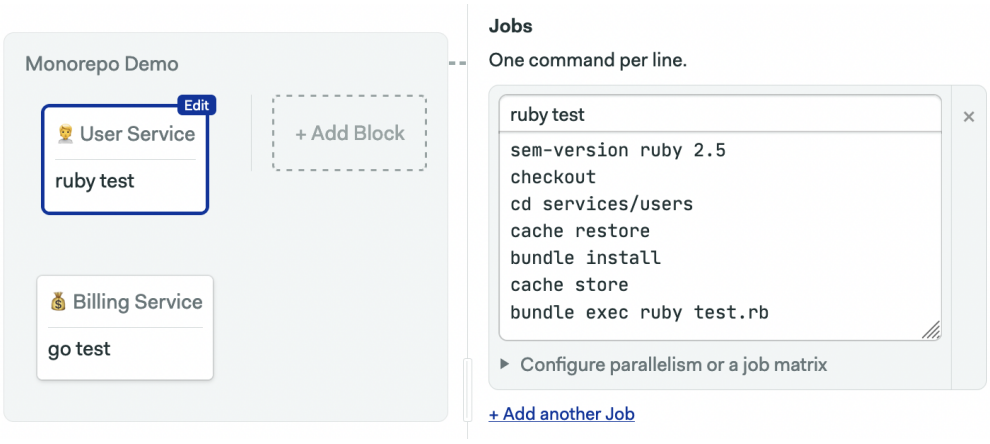
[11]sem-version reference page - *https://docs.semaphoreci.com/ci-cd-environment/sem-version-managing-language-versions-on-linux/*

3.2.2 Users Service

Let's add a second application to the pipeline. Create a new block. Then, add the commands to install and test the Ruby app:

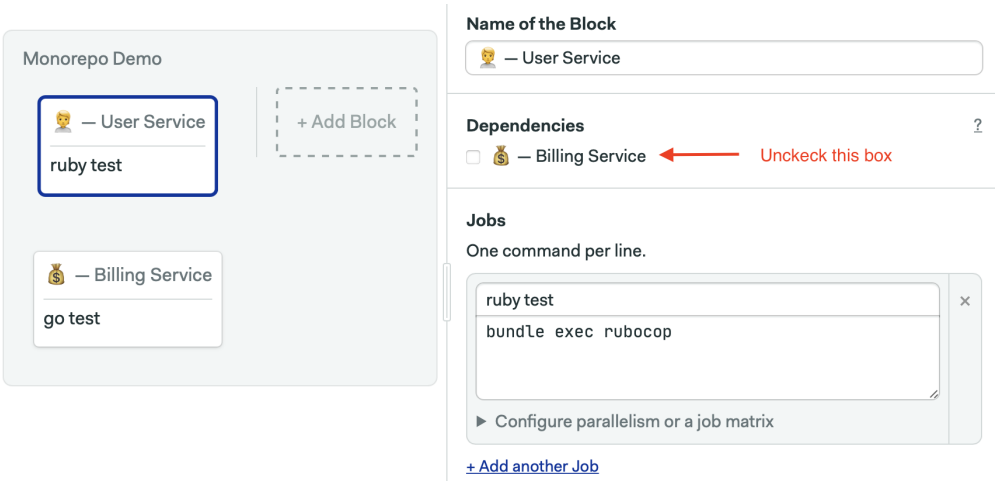
```
sem-version ruby 2.5
checkout
cd services/users
cache restore
bundle install
cache store
bundle exec ruby test.rb
```

If you look closely, there's something new here: cache[12]. This is a built-in command that lets us persist files between jobs and workflows. By itself, `cache store` figures out the project structure *automagically* and saves the relevant files into project-level storage. The cache speeds up this job by saving the downloaded Ruby Gems.



Finally, **uncheck** all the checkboxes under Dependencies.

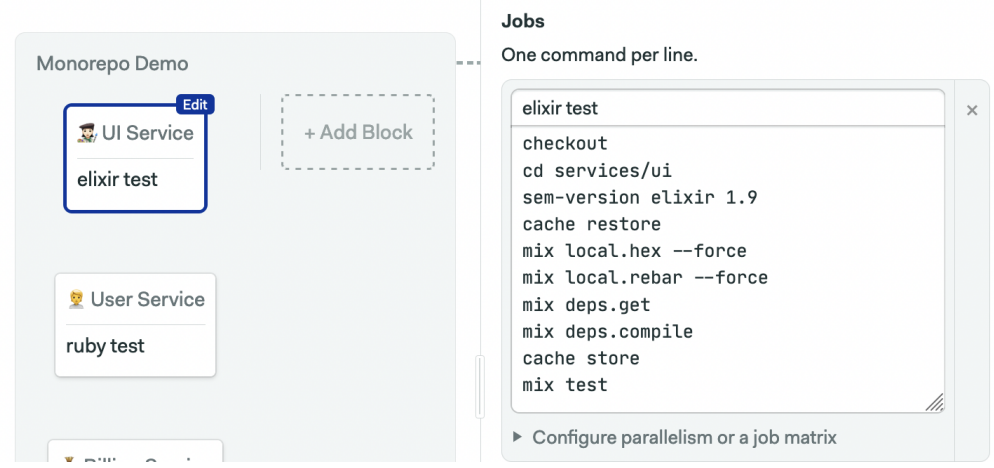
[12]Cache reference page - *https://docs.semaphoreci.com/reference/toolbox-reference/#cache*



3.2.3 UI Service

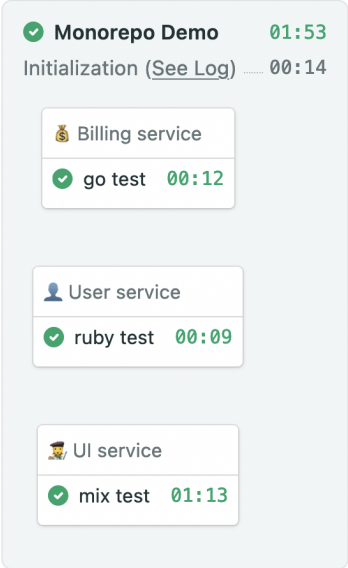
Add a third block to test the UI service. The following installs and tests the app. Remember to **uncheck** all block dependencies.

```
checkout
cd services/ui
sem-version elixir 1.9
cache restore
mix local.hex --force
mix local.rebar --force
mix deps.get
mix deps.compile
cache store
mix test
```



3.3 Configuring Change Detection

You can try running the pipeline now, just to make sure everything is in order. Now, what happens if we change a file inside the `/services/ui` folder?



Yeah, despite the fact that only one of the projects has changed, all the blocks are running. For a big monorepo with hundreds of projects, that's a lot of restless hours of waiting for the build to end. We can do better.

Open the workflow editor again. Pick one of the blocks and open the *Skip/run Conditions* section. Add some change criteria:

```
change_in('/services/billing')
```

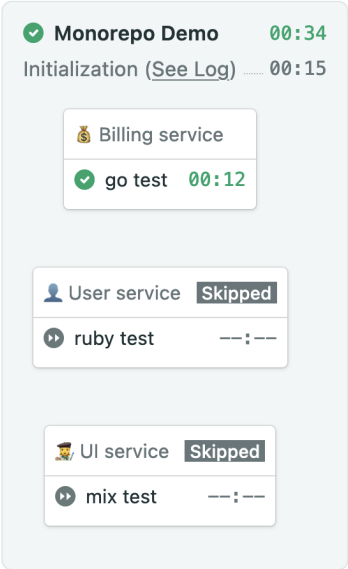
Repeat the procedure for the rest of the blocks.

```
change_in('/services/ui')
```

and

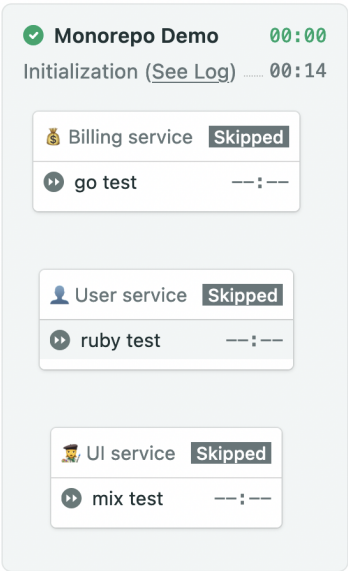
```
change_in('/services/users')
```

With `change_in` in place, Semaphore will only work on microservices that have recently changed.



Can you guess which application we changed? Yes, that's right: it was the Billing app. As a result, thanks to `change_in`, the other two blocks have been skipped because neither met the change conditions.

If we make a change outside any of the monitored folders, then all the blocks are skipped and the pipeline completes in just a few seconds.



3.4 Tips for Using change_in

Tying up a block with a piece of the code results in a smarter pipeline that builds and tests only what has recently been changed.

Scaling up large monorepos with `change_in` is easier if you follow these tips for organizing your code and pipelines:

- Define a unified folder organization, so you can use clean change conditions.
- Design your blocks around project folders.
- When needed, add multiple files and folders to `change_in`. Use this to rebuild all the connected project components within a monorepo.
- Keep branches small, and merge them frequently to cut build times.
- Use `exclude` and wildcards to ignore files that are not relevant, such as documentation or READMEs.
- Use `change_in` in auto-promotions to selectively trigger continuous delivery or deployment pipelines.

In the next section, we'll learn how to apply this principle to continuous delivery. We'll also learn a few more key concepts that play a key role in automatic deployments later on.

4. Continuous Deployment for Monorepos

Chapter three left us with a working CI pipeline. Now that we're through with the basics. Let's focus on the final stage of every CI/CD process: continuous deployment (CD), where we deploy the application services into production systems continually, without human intervention.

The shift from CI to CD is subtle but, in reality, it's a completely different ball game. While everything in CI happens, as it were, within the bounds of Semaphore's systems, a CD pipeline, be it by publishing a package, updating a service, or deploying software, will necessarily interact with the external world. Therefore, extra precautions must be taken to avoid surprises.

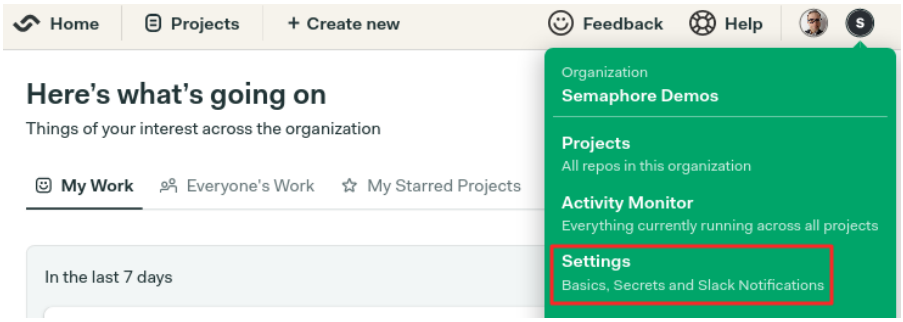
Before we configure an automated deployment, we'll need to master two Semaphore concepts:

- A **secret** holds the access keys required for authentication with external systems.
- **Promotions** connect the CI and CD pipelines together to create complex workflows.

4.1 Secrets

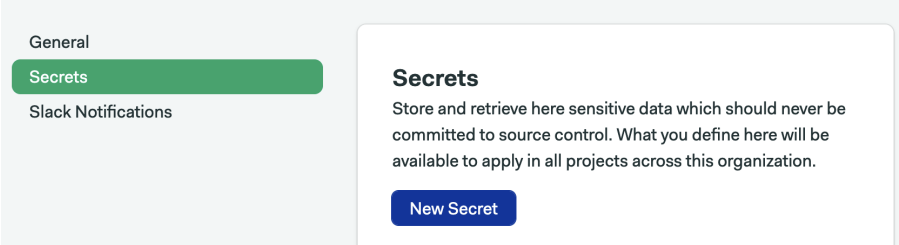
Telling Semaphore how to deploy software typically means storing a password, some API keys, or other sensitive information as a secret. Secrets[13] are encrypted variables and files that are decrypted into jobs on a need-to-know basis in order to keep your data secure.

Secrets can be accessed through the **Settings** option in the organization menu.

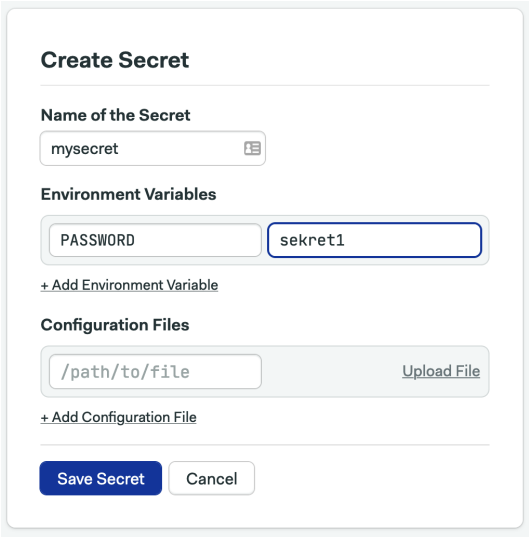


The *Secrets* menu lets you manage all the secrets within the organization.

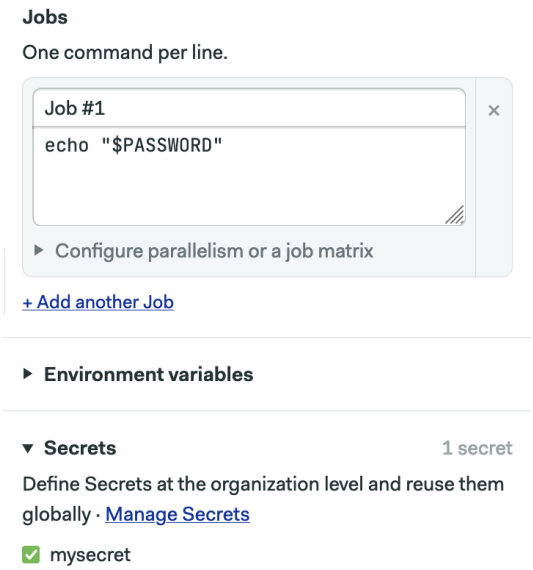
[13] environment variables and secrets - *https://docs.semaphoreci.com/essentials/environment-variables/*



A secret is, in short, one or more variables or files, which are encrypted once you press *Save Secret*.



To use the secret in a job, you need to enable it at the block level. Enabling the secret will make Semaphore decrypt it, import the value as environmental variables or copy attached files into all the jobs in the block.



As you can see in the output of the log, you can access the secret value like any other environment variable.

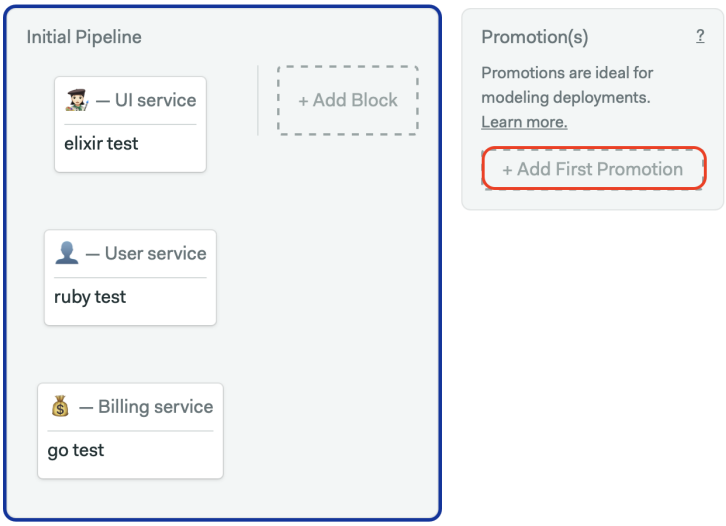
▶	1	Exporting environment variables	Passed in 00:00
▶	43	Injecting Files	Passed in 00:00
▶	46	Setting up the Semaphore toolbox	Passed in 00:01
▶	86	Starting an ssh-agent	Passed in 00:00
▶	89	Connecting to cache	Passed in 00:00
▼	95	echo "$PASSWORD"	Passed in 00:00
	95	sekret1	00:00
	96		00:00
	97	export SEMAPHORE_JOB_RESULT=passed	Passed in 00:00

4.2 Deploying with Promotions

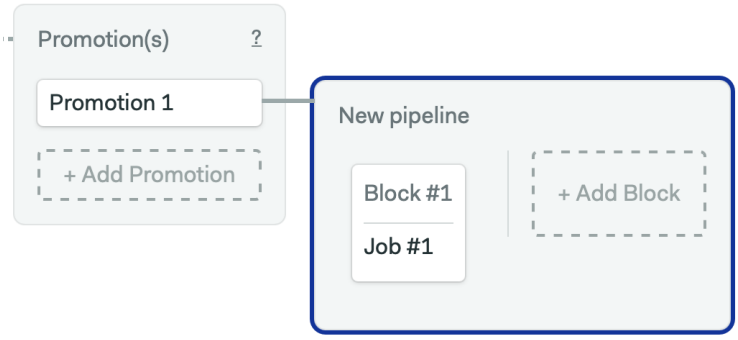
Promotions[14] connect pipelines together. While there are no fixed rules, they are usually placed in the natural "space" that exists between CI and CD.

Promotions are created via the *Add Promotion* button in the workflow editor. This will create a new pipeline.

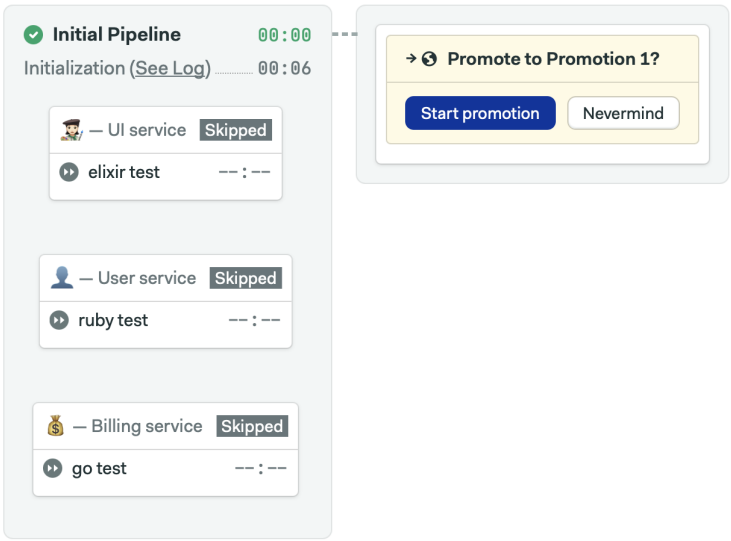
[14] deploying with promotions - *https://docs.semaphoreci.com/essentials/deploying-with-promotions/*



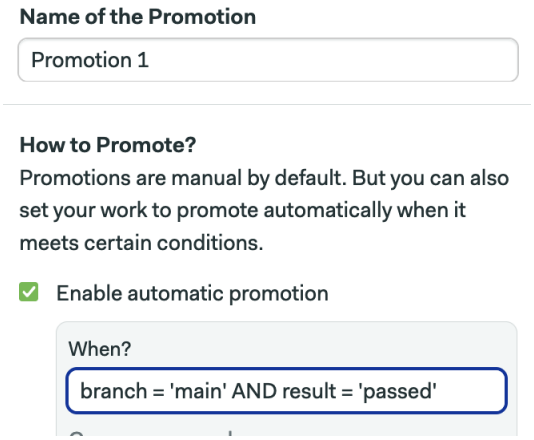
There's nothing special about this pipeline, you can create blocks and jobs as usual.



By default, promotions are not automatic, which means that you need to manually start them by clicking a button once the workflow has started.



Auto-promotions are activated when specific conditions are detected, such as when a commit is pushed into a certain branch. Checking the *Enable automatic promotion* box brings up a field to type the conditions that determine when the next pipeline starts.



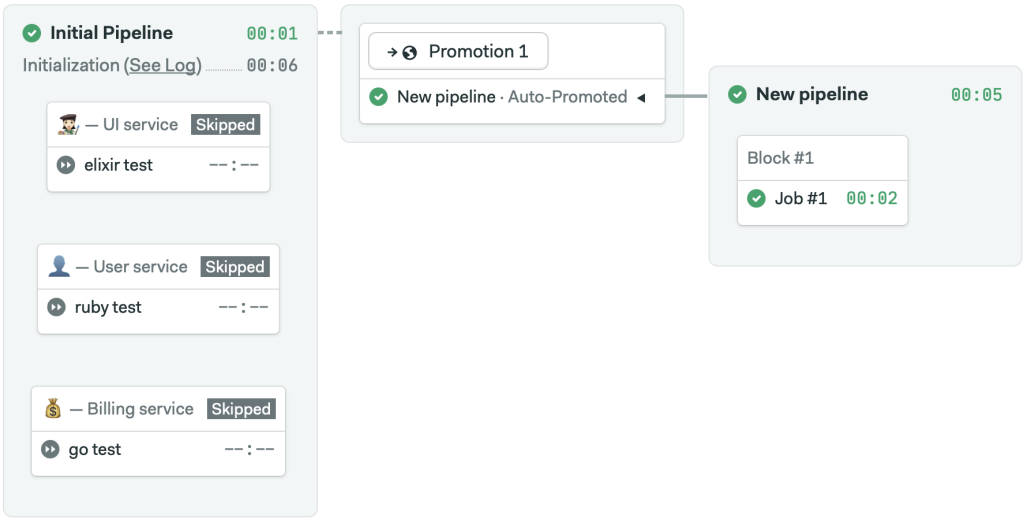
Conditions are specified by mixing one or more of the following:

- **branch**: evaluates to which branches the commit was made.
- **tag**: used to detect a Git-tagged release.
- **pull request**: used when the workflow was triggered by a pull request.
- **change detection**: checks if files have changed in one or more selected folders or files.

The default conditions will make the new pipeline start when all tests pass on

the `master` branch:

```
branch = 'master' AND result = 'passed'
```

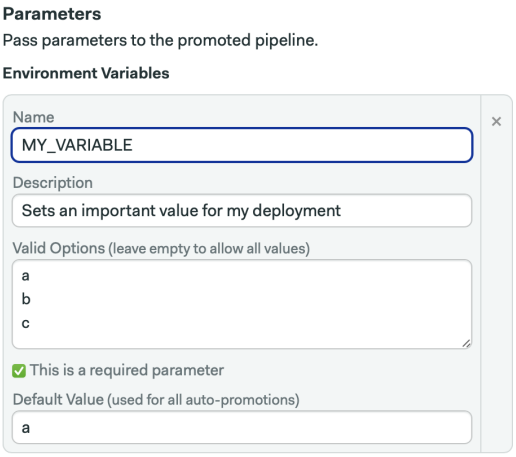


4.3 Parametrized Promotions

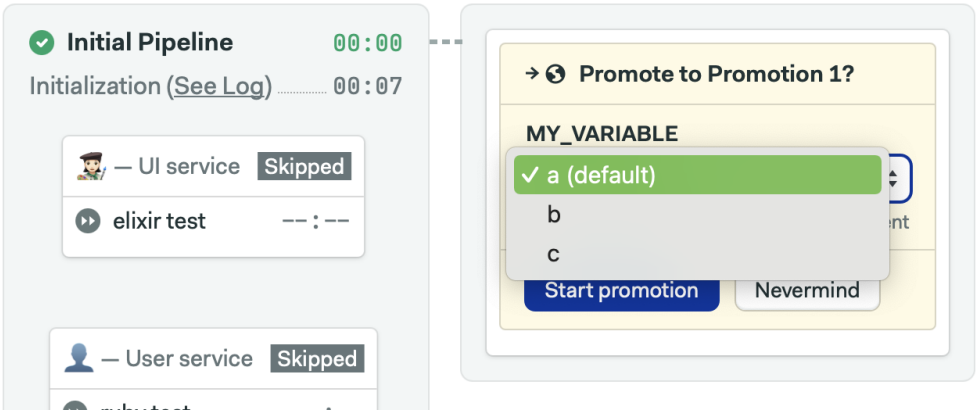
Parametrized promotions let us reuse a pipeline for many tasks. For instance, you can create a deployment pipeline and share it among multiple applications in the monorepo, ensuring you have a unified release process for all the services.

Parametrized promotions work in tandem with environment variables — we define one or more variables and set default values based on the same conditional syntax we use in regular promotions.

To create a parameter, scroll down to the promotion pane and click *+Add Environment Variable*.



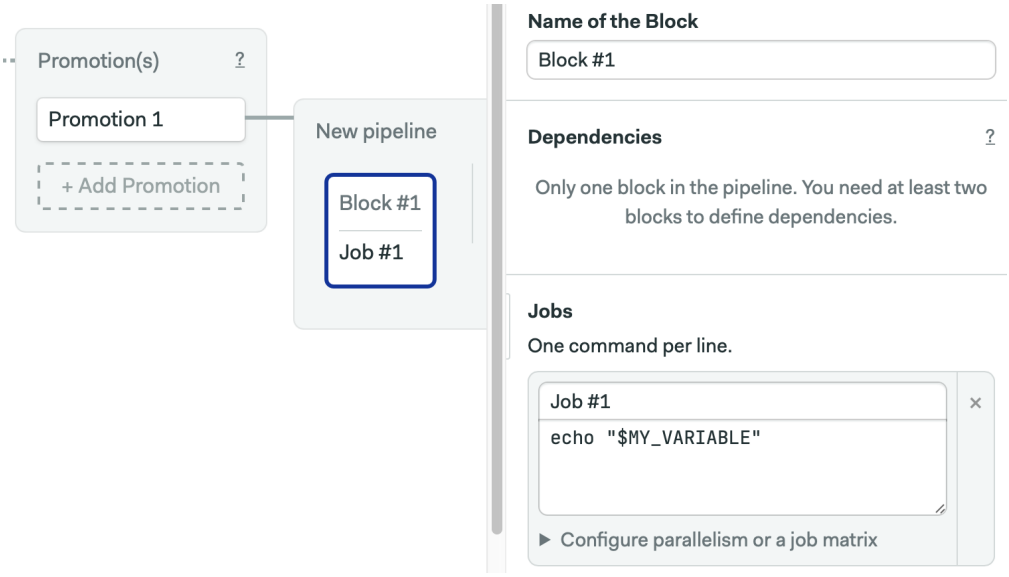
When the promotion is started manually, we can choose a value from the list. With auto-promotions, however, the default value is used.



There are three important things to keep in mind while defining a parameter:

- Leaving the list of allowed values empty lets you type in any value, which opens the possibility for human errors.
- Parameters can be optional or mandatory. Required parameters must have a default value defined. Non-mandatory parameters can be empty.
- You can define multiple parameters in the same promotion.

Parameters define global, per-pipeline environment variables that jobs in it can access directly.



4.4 Staging the Demo

Let's see how to apply what we learned to the deploying the demo.

We want a sturdy CI/CD process. Testing the services in CI is no guarantee of zero errors in production. A considerable degree of extra confidence can be gained by using a staging environment. Consequently, we will need two new pipelines:

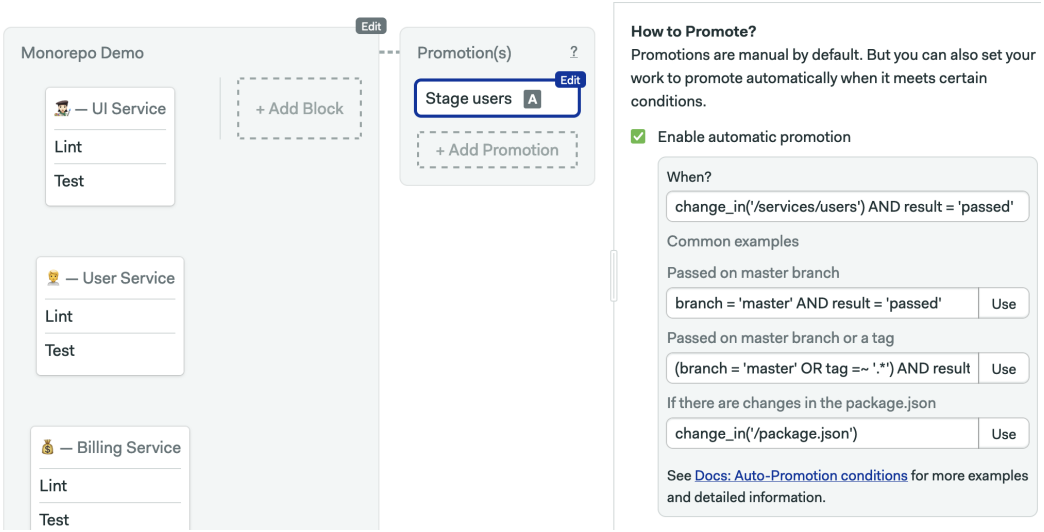
- **Staging**: runs the application in a production-like environment and performs smoke tests.
- **Production**: if tests succeed, it deploys into the production systems.

4.4.1 Staging the Users Service

Begin by creating a new promotion and making it automatic. We'll deploy the User service on every change committed to the `master` branch. The auto-promotion condition will then be:

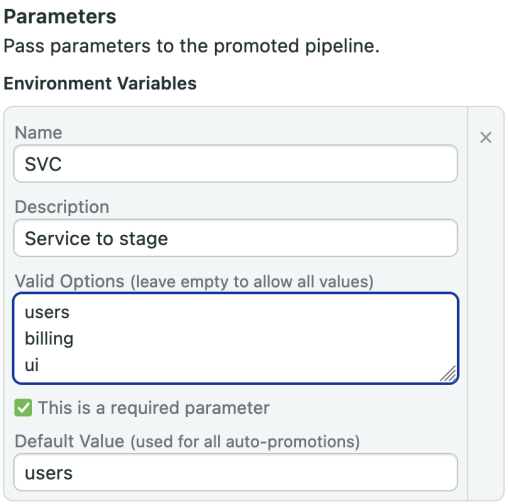
```
change_in('/services/users') AND results = 'passed' AND branch = 'master'
```

Type the condition into the *When?* field.



In the same pane, immediately below, you'll find the parameters section. Click *+Add Environment Variable* and type the following:

- **Name** of the variable: `SVC`
- **Description**: Service to stage
- **Valid options:** `users`, `billing`, `ui` (one per line)
- **Default value**: `users`

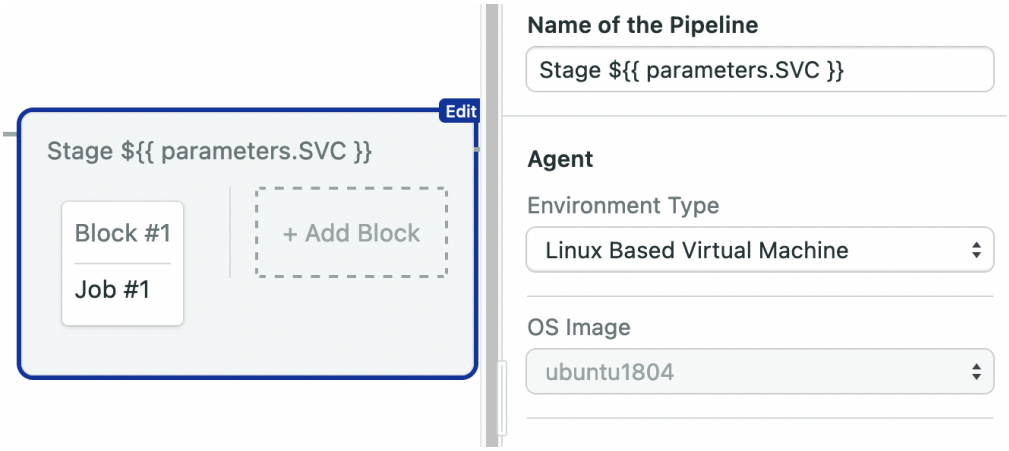


What we're doing here is creating an environmental variable, called `SVC`, that takes one of the three services in the repository.

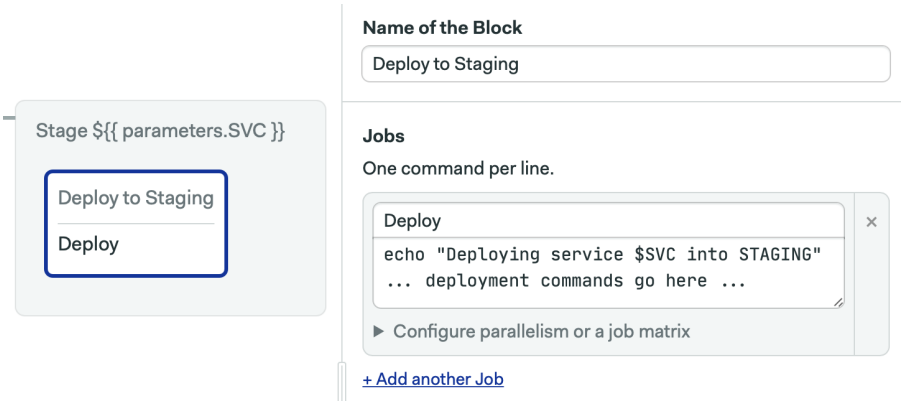
Next, we'll create the staging pipeline. Click on the newly created pipeline and scroll down to the *YAML file path*. Replace the default value with

`.semaphore/stage.yml`

Click on the new pipeline and set its name to: `Stage ${{ parameters.SVC }}`. The special syntax allows the `SVC` variable to be expanded dynamically once the pipeline begins running.



We'll use the first block in the staging pipeline to deploy `SVC`. Type the deployment commands for this service. Add whichever secrets and environmental variables you need to release the new version into the staging environment.



If you need inspiration for setting up the jobs, we've written a lot about this on the Semaphore blog:

- What Is Canary Deployment: *https://semaphoreci.com/blog/what-is-canary-deployment*
- What Is Blue-Green Deployment: *https://semaphoreci.com/blog/blue-green-deployment*
- A Step-by-Step Guide to Continuous Deployment on Kubernetes: *https:*

//semaphoreci.com/blog/guide-continuous-deployment-kubernetes

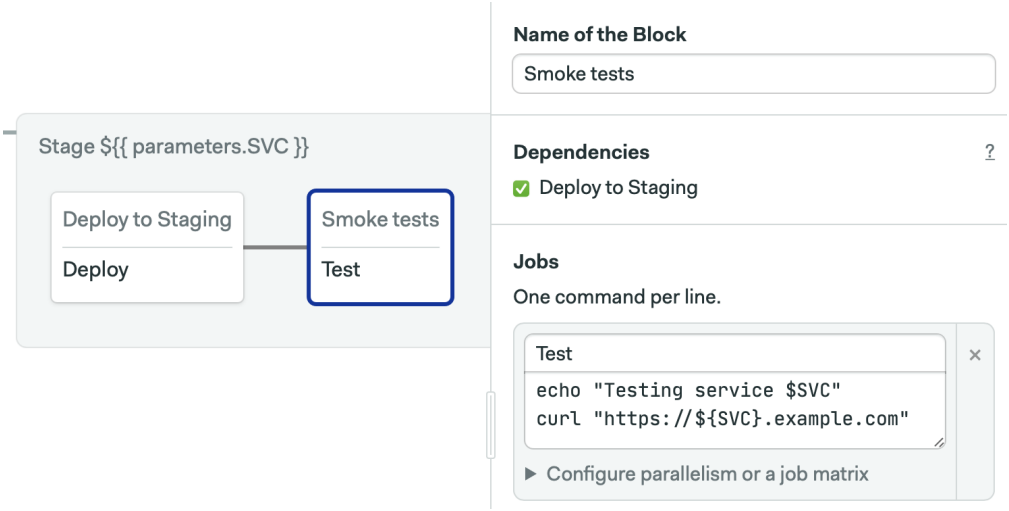
- JavaScript Monorepos with Lerna: *https://semaphoreci.com/blog/javascript-monorepos-lerna*
- Android Continuous Integration and Deployment Tutorial: *https://semaphoreci.com/blog/android-continuous-integration-deployment*
- Python Continuous Integration and Deployment From Scratch: *https://semaphoreci.com/blog/python-continuous-integration-continuous-delivery*

4.4.2 Smoke Testing

Having a production-like environment presents an invaluable opportunity for testing. Let's take a look at how Semaphore enables smoke tests.

Create a new block and add the commands required to check that the service is healthy. For example:

```
echo "Testing service $SVC"
curl "https://${SVC}.example.com"
```

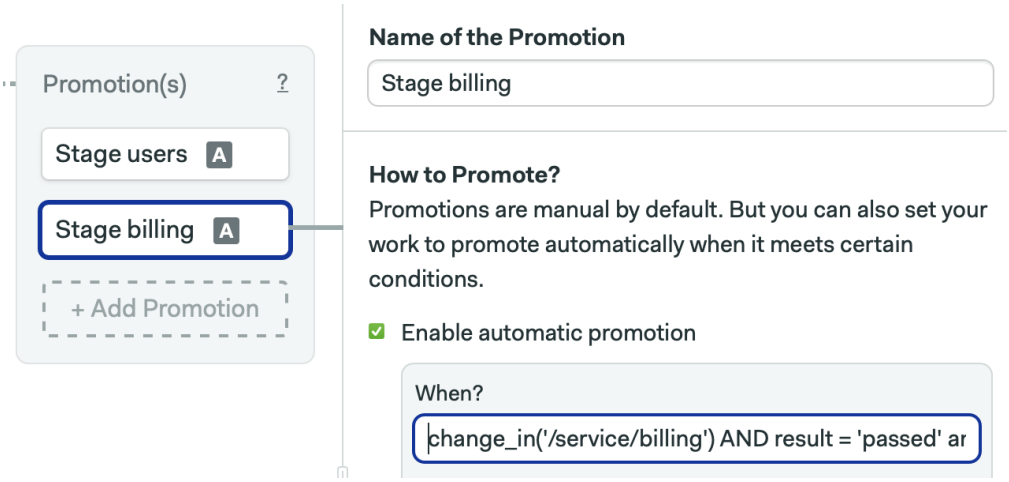


4.4.3 Staging the Rest of the Services

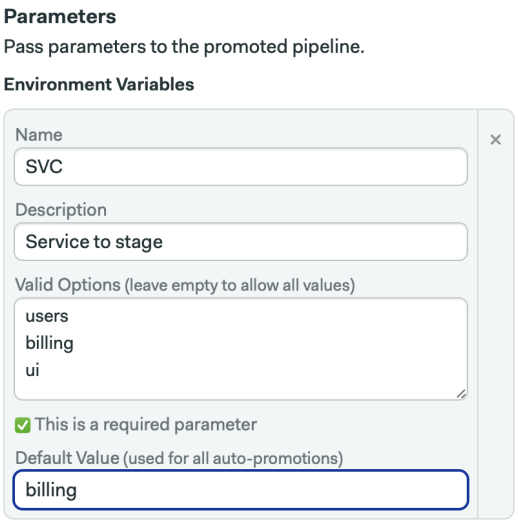
Thanks to parametrization, our staging pipeline is universal. We can reuse it to stage the Billing and UI services.

Create a new promotion below to "Stage users". The criteria for releasing may be different for each service. Let's say that we want to deploy Billing only on Git-tagged releases. Hence, the *When?* field should read:

```
change_in('/service/billing') AND result = 'passed' and tag=~ '.*'
```



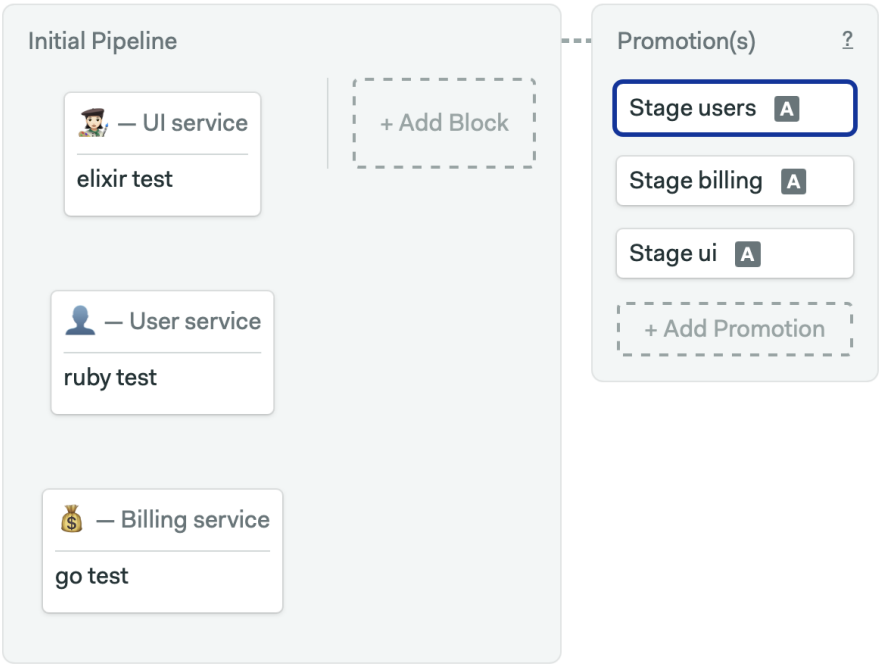
The parameter for this promotion will be almost exactly the same as Users, the only difference is that the default value will be `billing` instead of `users`.



Click on the newly-created pipeline and open the *YAML path* section. Replace the path of the file with `.semaphore/stage.yml`.

Repeat the same procedure with the UI Service:

1. Create a new promotion.
2. Type an auto-promotion condition.
3. Create a SVC parameter with a default value of `ui`.
4. Change the YAML path to `.semaphore/stage.yml`.



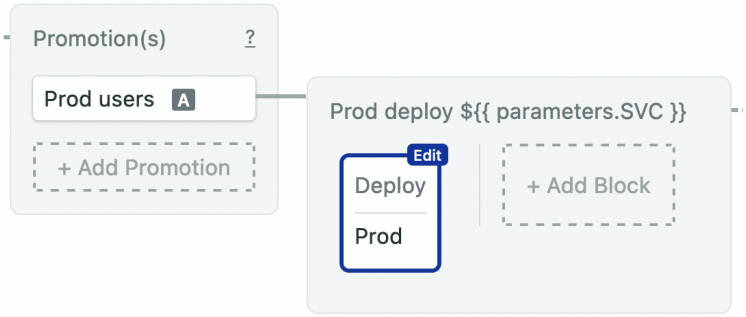
4.5 The Production Pipeline

If testing on staging passes, chances are that it's pretty safe to continue with production.

4.5.1 Promoting the Users Service to Production

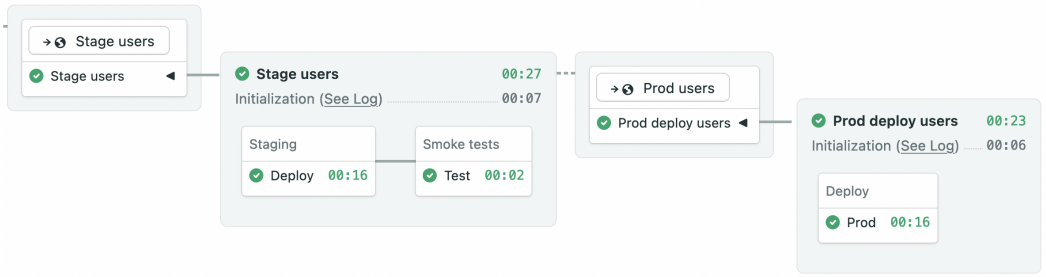
We'll keep things simple by creating a deployment pipeline with one job. The rundown of the steps is:

1. Create a promotion branching off the staging pipeline, using the same auto-promotion and parameters as before.



2. Ensure that `users` is the default value of the parametrized pipeline.
3. Rename the new pipeline to: `.semaphore/deploy.yml`
4. Type the deployment commands (the service to deploy is stored on the `SVC` variable).
5. Activate any required secrets and set environment variables as needed.

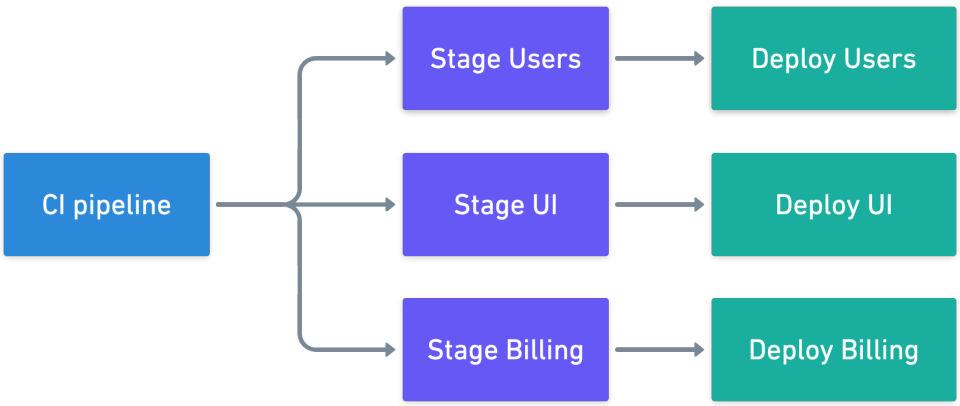
Click on *Run the Workflow* to give it a whirl. You may need to manually start the staging and deployment pipelines. Check that the Users service is deployed to both environments.



4.5.2 Deploying the Billing and UI Services

The deploy to production pipeline can also be reused for the rest of the services. So, repeat the procedure: add two additional promotions branching off the stage pipeline and set the *YAML pipeline* file to `.semaphore/deploy.yml`.

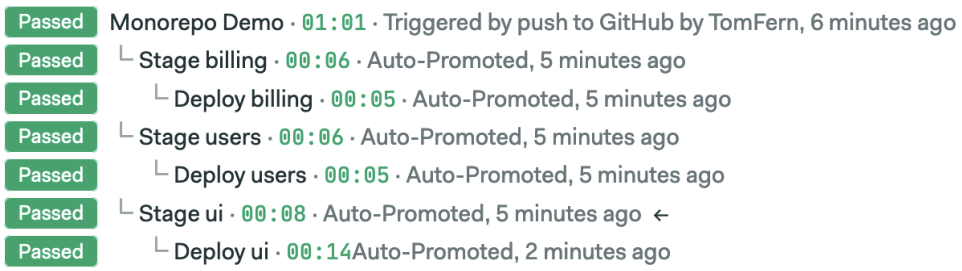
At the end of the setup you will have a total of three pipelines (CI, staging, and production deployment) connected by six promotions.



4.6 Ready to Go

The CI/CD process is 100% configured. The only thing left to do is save it and run it to ensure everything works as expected.

The resulting workflow is too big to see all at once on one page. Still, you can see the overview in the project's dashboard.



The deployment is complete as soon as everything is green. Good job and happy building!

5 Final Words

Nice work! The book may be finished, but there's a lot more to learn and do yet in the world of monorepos. So, go and build something awesome!

5.1 Share This Book With The World

Please share this book with your colleagues, friends, and anyone who you think might benefit from it.

Share the book online: *https://semaphoreci.com/resources/monorepo-cicd*

5.2 Tell Us What You Think

We would absolutely love to hear your feedback. What did you get out of reading this book? How easy/hard was it to follow? Is there something that you'd like to see in a new edition?

This book is open source and available at *https://github.com/semaphoreci/book-monorepo-cicd*.

- Send comments and feedback, ask questions, and report problems by opening a new issue.
- Contribute to the quality of this book by submitting pull requests for improvements to explanations, code snippets, etc.
- Write to us privately at learn@semaphoreci.com.

5.3 About Semaphore

Semaphore https://semaphoreci.com helps developers continuously build, test and deploy code at the push of a button. It provides the fastest, enterprise-grade CI/CD pipelines as a serverless service. Trusted by thousands of organizations around the globe, Semaphore can help your team move faster too.