CONTINUOUS DEPLOYMENT/DELIVERY

CI/CD Pipelines for Microservices

8 min read

Kostis Kapelonis · May 13, 2019

Moving from monoliths to microservices and from Virtual Machines to Kubernetes clusters are two closely related migration paths. Using Kubernetes clusters enables the easy deployment of microservices, and adopting microservices becomes a lot easier when containers are used for the packaged delivery.

If you look at our features page, it is very easy to understand why Codefresh is the first (and only) CI/CD platform for containers and Kuberentes. Currently, Codefresh is the only solution that includes:

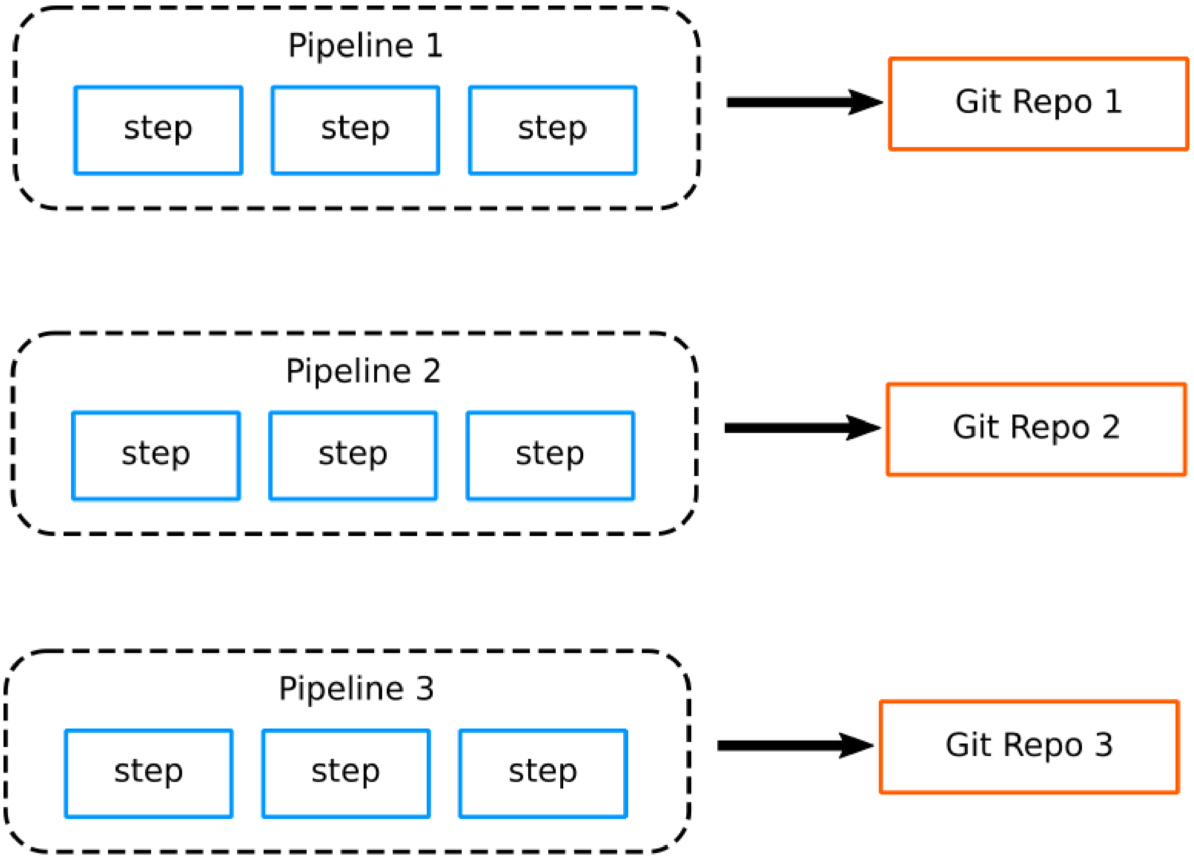
- A system-wide Docker registry
- A system-wide Helm repository
- Custom Docker annotations
- Dedicated Kubernetes dashboard
- A Helm releases dashboard
- A Helm deployments dashboard
- A Helm environments dashboard.

All these features show clearly that Codefresh is the perfect solution for CI/CD if you have already adopted containers and Kubernetes. However, they don't tell the story of the monolith-to-microservice adoption.

In this article, we will see how microservices change the way CI/CD pipelines are organized and why Codefresh is specifically designed with microservices in mind.

Organizing pipelines for monolithic applications

In the past, pipelines for monolithic applications tended to share the same characteristics of the application they were building. Each project had a single pipeline which was fairly complex and different projects had completely different pipelines. Each pipeline was almost always connected to a single GIT repository.



Pipelines for monolithic applications

The complexity of each pipeline was detrimental to easy maintenance. Pipelines were typically controlled by a small team of gurus who are familiar with both the internals of the application as well as the deployment environment.

For each software project, operators are taking care of the pipeline structure while the developers are only working with the source code (going against the DevOps paradigm where all teams should share responsibility for common infrastructure and collaborate on shared problems).

Pipeline size and complexity is often a huge pain point. Even though several tools exist for the continuous integration part of a monolithic application, continuous deployment is a different matter completely which forced a lot of companies to create their own custom in-house scripts for taking care of deployment.

As the number of applications evolved within an organization, the management of pipelines became a bigger issue as well. In the end, most pipelines end as a spaghetti mix of python scripts, configuration management tools (e.g. puppet and chef) sprinkled with some bash scripting all over the place.

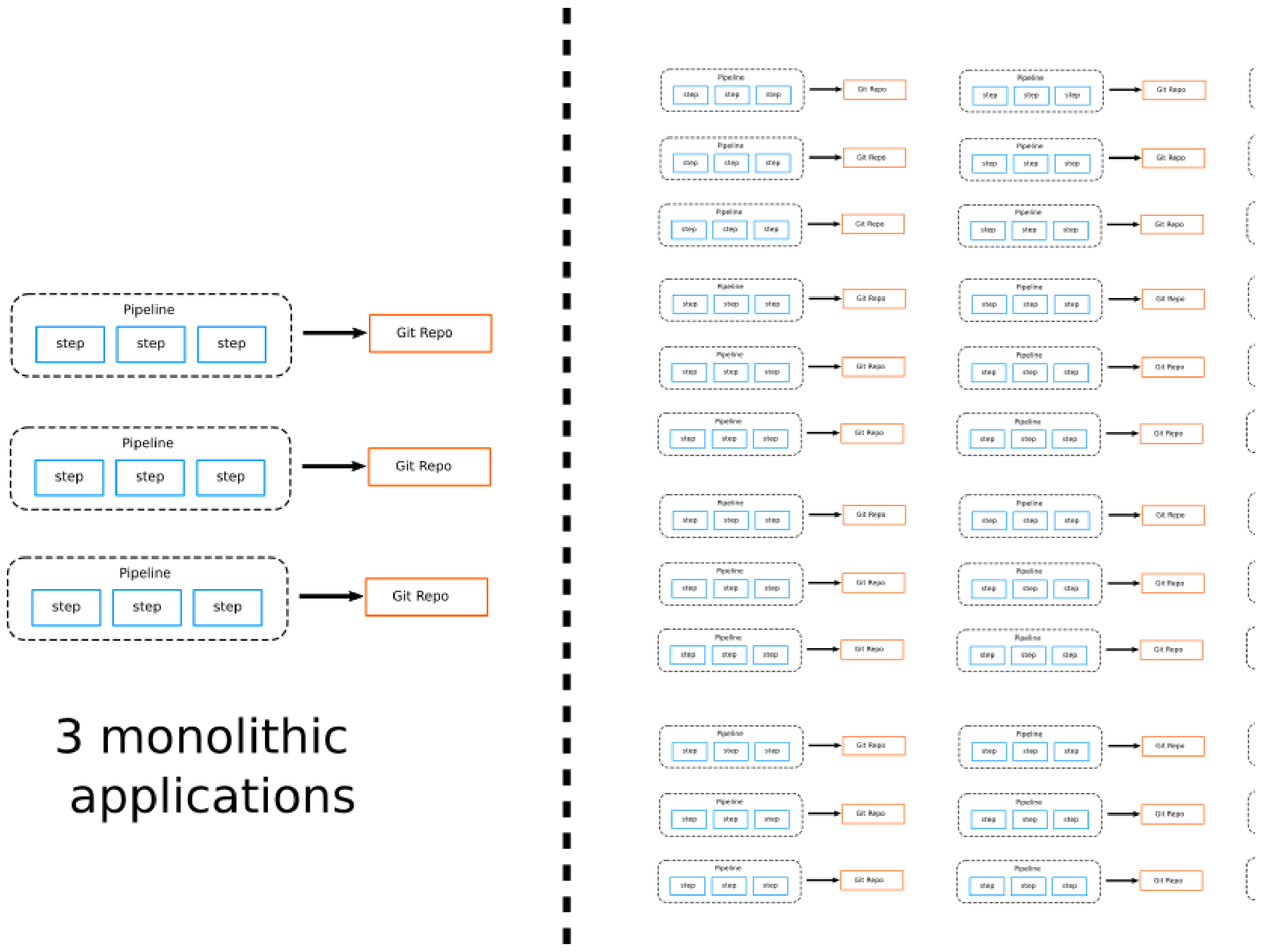
This messy situation is very common within organizations that have not adopted microservices yet and the only reason that it endures is that most developers are still working on a single application so they never feel the pain of pipeline maintenance. That job falls on the operators who are typically spending most of their time fixing pipelines (and creating new ones for new projects) assuming that this is the status quo.

Scalability issues with microservice pipelines

Microservices have of course several advantages regarding deployment and development, but they also come with their own challenges. Management of microservice repositories and pipelines becomes much harder as the number of applications grows.

While a company might have to deal with 1-5 pipelines in the case of monolith applications (assuming 1-5 projects), the number quickly jumps to 25 if each monolith is divided into 5 microservices.

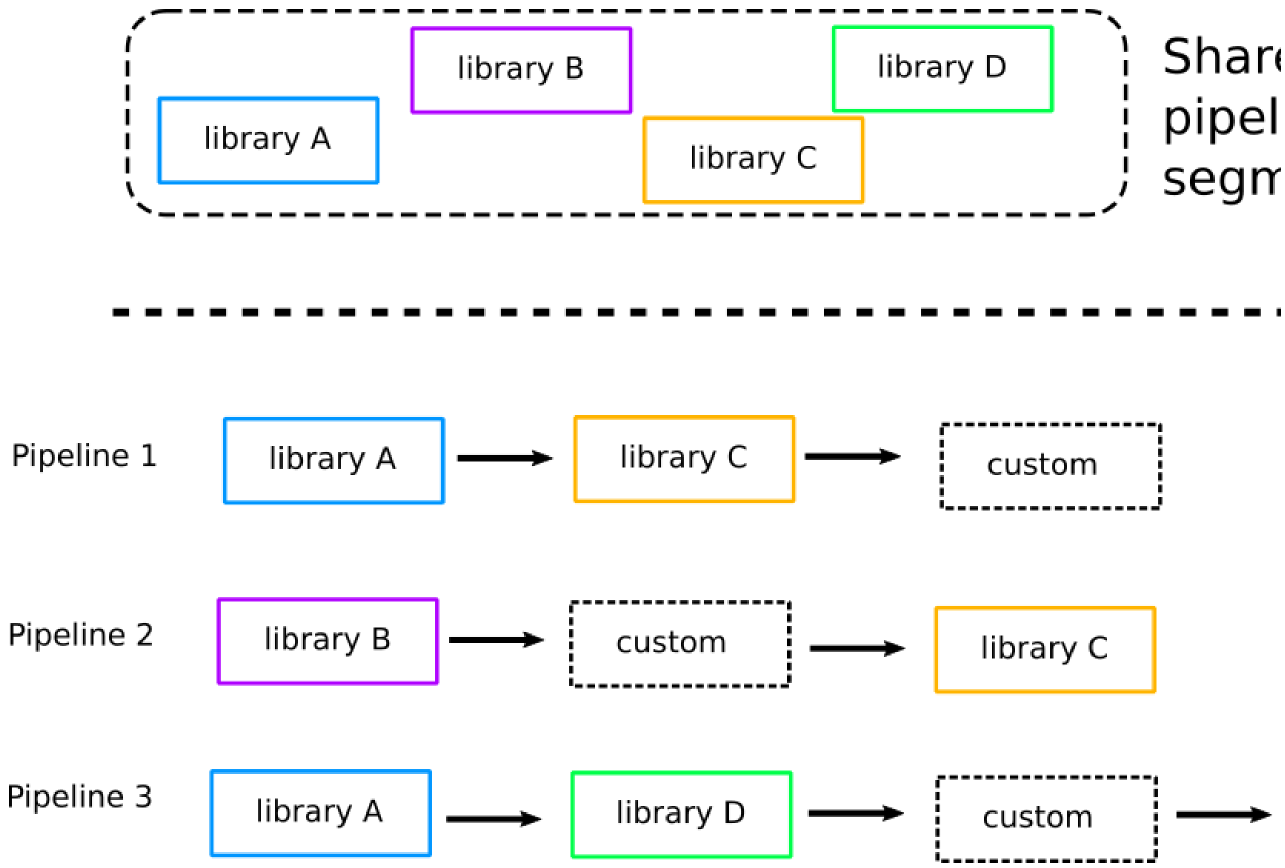
These numbers are different per organization. It is perfectly normal for an application to have 10 microservices. So at a big organization that has 50 applications, the operator team is suddenly tasked with the management of 500+ pipelines.



Moving to microservices

This sudden explosion in numbers prohibits working manually with pipelines anymore. Several CI solutions are not even prepared to work with such a high number of pipelines.

Here is where we reach the biggest pitfall regarding pipeline management in the era of microservices. Several companies tried to solve the problem of microservice pipelines using shared pipeline segments.



Shared pipelines are complex

In theory, this sounds like a good idea:

1. Operators are locating the common parts of pipelines with applications
2. A shared pipeline segment registry is created that holds all those common parts
3. Pipelines in existing projects are re-engineered to depend on the common segments
4. New projects must first examine the library of common pipeline segments and choose what is already there

The final result is that a single pipeline is actually composed of two types of steps, those common to other pipelines and those that are specific to that project only.

This has lead to the development of several solutions which attempt to centralized common pipeline parts and re-use them in the form of “libraries” within software projects. The issue here is that this approach requires a very large time investment as well as a disciplined team that can communicate and cooperates on the following factors:

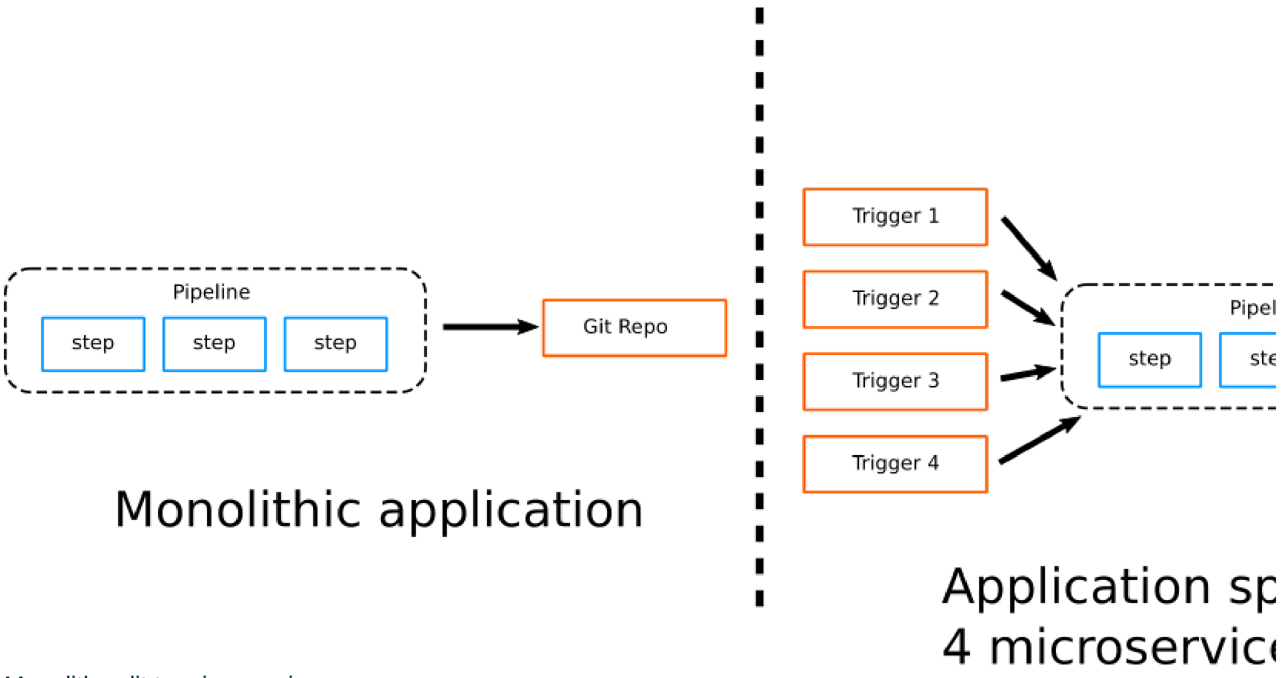
1. Detecting which pipeline segments are indeed common,
2. Keeping the library of common pipeline segments up-to-date,
3. Disallowing copy-pasting of pipelines,
4. Development of brand new pipelines when needed,
5. Initial setup and pipeline bootstrap for each new project created.

Unfortunately, in practice, as the number of microservice applications grows, teams find it very hard to keep all these principles in mind when creating new projects.

Reusing pipelines for Microservice applications

Codefresh is the first CI/CD solution for microservices and containers. Because we are not burdened with any legacy decisions, we are free to define a new model for Codefresh pipelines which is focused on microservices.

The basic idea is that all microservices of a single application have almost always the same life- cycle. They are compiled, packaged and deployed in a similar manner. Once this realization is in place we can see that instead of having multiple pipelines for each microservice (where each one is tied to a GIT repository), we have instead a single pipeline shared by all microservices.



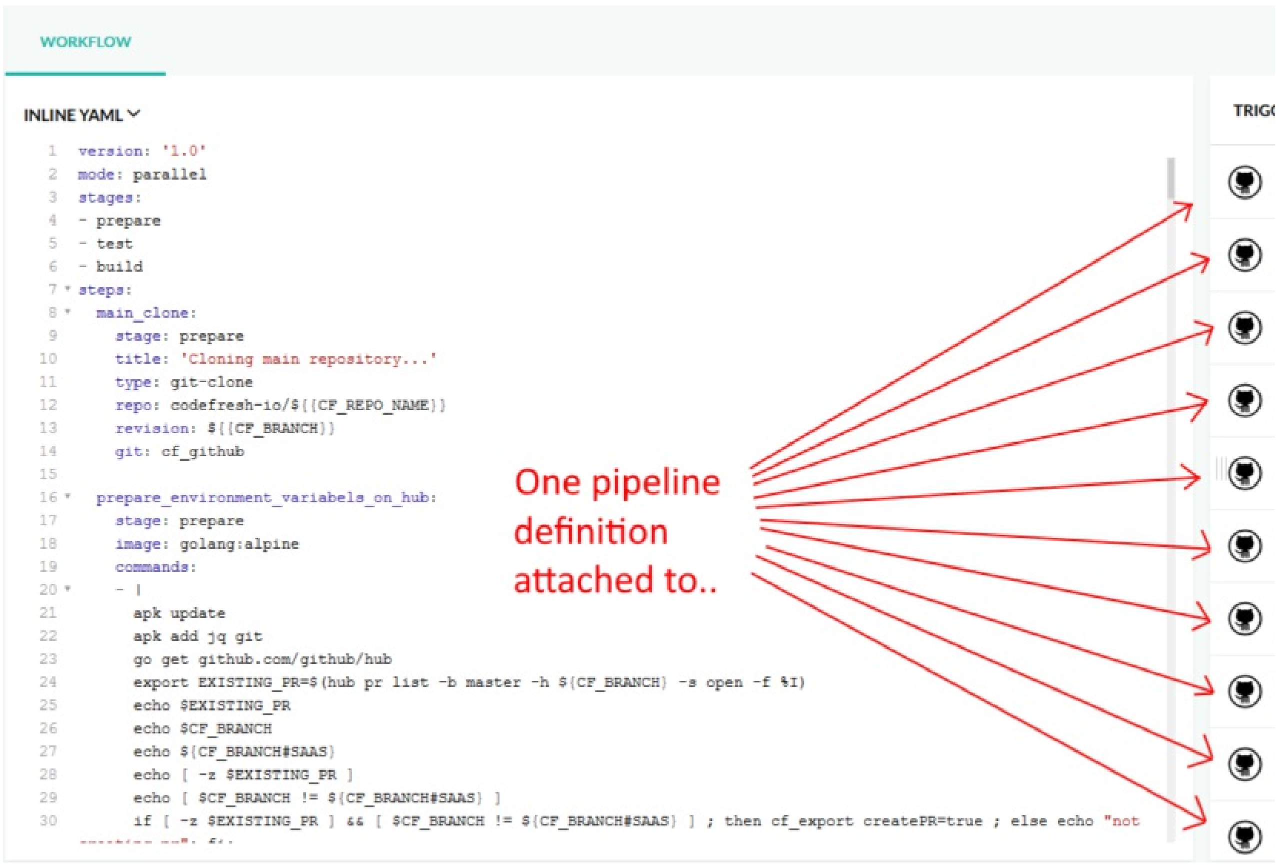
Monolith split to microservices

The impact of this design cannot be understated. First of all, it should be clear that there is no need for sharing pipeline segments anymore. The whole pipeline is essentially the re-usable unit.

This makes pipeline construction very simple.

The biggest advantage, however, is the way new projects are created. When a new microservice is added in an application, the pipeline is already there and only a new trigger is added for that microservice. Notice that the pipeline is not connected to any specific git repository anymore. All information for a repository is coming from the git trigger that started this pipeline.

This is exactly what our recent feature launch was based on. As an operator you can bootstrap a new project by quickly adding a new trigger on an existing pipeline:



```
version: '1.0'
mode: parallel
stages:
- prepare
- test
- build
steps:
  main_clone:
    stage: prepare
    title: 'Cloning main repository...'
    type: git-clone
    repo: codefresh-io/${{CF_REPO_NAME}}
    revision: ${{CF_BRANCH}}
    git: cf_github

  prepare_environment_variabels_on_hub:
    stage: prepare
    image: golang:alpine
    commands:
    - |
      apk update
      apk add jq git
      go get github.com/github/hub
      export EXISTING_PR=$(hub pr list -b master -h ${CF_BRANCH} -s open -f %I)
      echo $EXISTING_PR
      echo $CF_BRANCH
      echo ${CF_BRANCH#SAAS}
      echo [ -z $EXISTING_PR ]
      echo [ $CF_BRANCH != ${CF_BRANCH#SAAS} ]
      if [ -z $EXISTING_PR ] && [ $CF_BRANCH != ${CF_BRANCH#SAAS} ] ; then cf_export createPR=true ; else echo "not
```

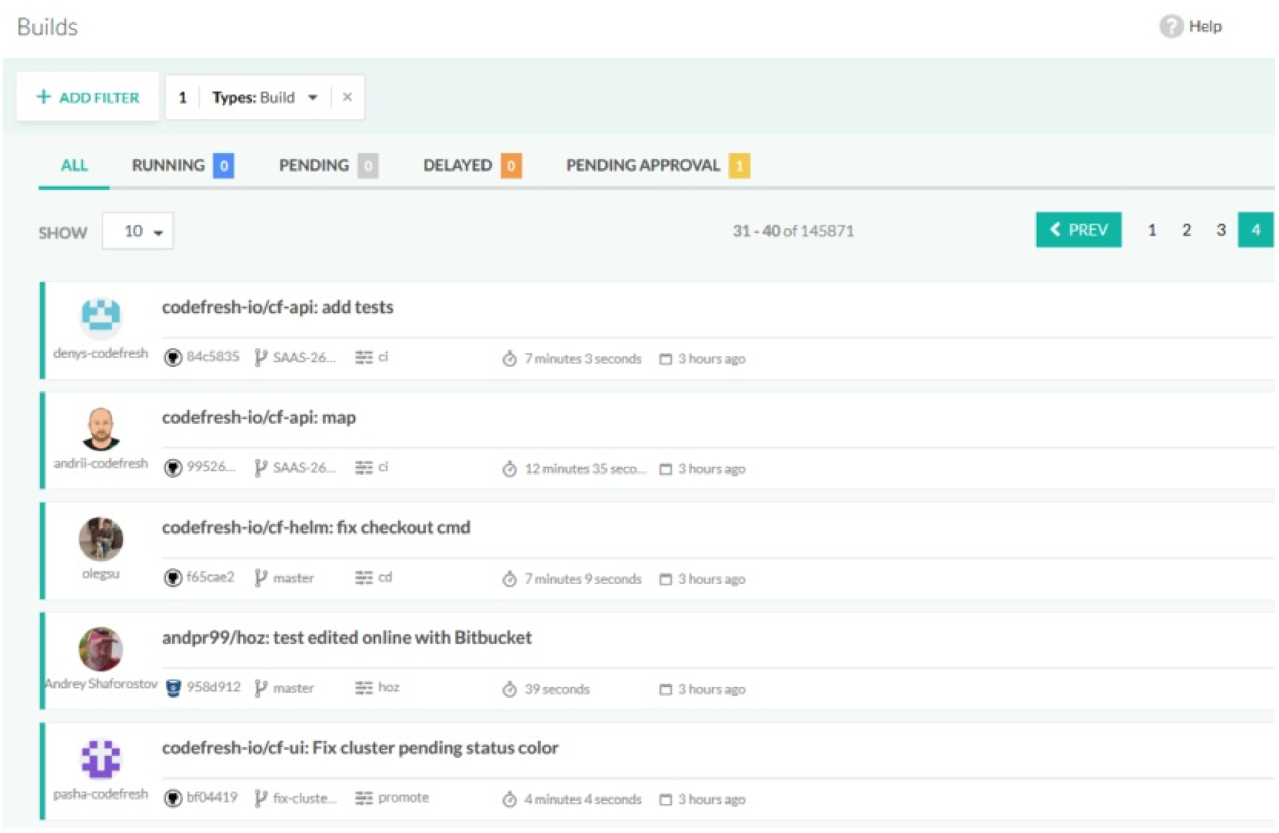
Sharing a single CI/CD pipeline

This is the fastest way possible to bootstrap a new project. As the number of microservices is growing, the only thing that is growing is the list of triggers. All pipelines are exactly the same.

Creating a new pipeline is needed only if the whole deployment process is different for a new project. This is an extreme scenario however, as with containers this will happen very rarely. Once you package an application in a Docker image, the underlying implementation is not really important anymore. Docker images from different programming languages can be handled in the same manner.

Case study: Codefresh on Codefresh

Now that we have explained the theory behind microservice pipelines, we can look at a solid example. And one of the natural examples is Codefresh itself. A very popular question we get from our customers is how our own workflow is structured. It shouldn't be a big surprise that we also use Codefresh internally to manage Codefresh.


Builds

Help

+ ADD FILTER | 1 Types: Build ×

ALL | RUNNING 0 | PENDING 0 | DELAYED 0 | PENDING APPROVAL 1

SHOW 10 | 31 - 40 of 145871 | PREV 1 2 3 4

User	Build	Commit	Branch	Pipeline	Duration	Started
denys-codefresh	codefresh-io/cf-api: add tests	84c5835	SAAS-26...	ci	7 minutes 3 seconds	3 hours ago
andrii-codefresh	codefresh-io/cf-api: map	99526...	SAAS-26...	ci	12 minutes 35 seco...	3 hours ago
olegsu	codefresh-io/cf-helm: fix checkout cmd	f65cae2	master	cd	7 minutes 9 seconds	3 hours ago
Andrey Shaforostov	andpr99/hoz: test edited online with Bitbucket	958d912	master	hoz	39 seconds	3 hours ago
pasha-codefresh	codefresh-io/cf-ui: Fix cluster pending status color	bf04419	fix-cluste...	promote	4 minutes 4 seconds	3 hours ago

Codefresh running on Codefresh

The Codefresh platform is itself a set of microservices that run on Kubernetes. We have about 20 microservices, but they are easily managed by a handful of pipelines.

In fact, most of the heavy work is done by two pipelines:

- A pipeline called “CI” which packages each microservice.
- A pipeline called “CD” which deploys each microservice.

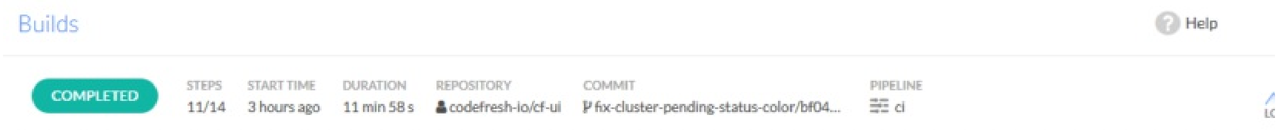
That’s it! Even if tomorrow our platform has 50, 500, or 5000 microservices the number of pipelines will still be 2. Every new microservice will be handled by simply adding a new trigger to those pipelines.

By the way, our pipelines are public, so feel free to adopt them for your own infrastructure (or at least study them and get some ideas on possible workflows).

Let’s see them in order…

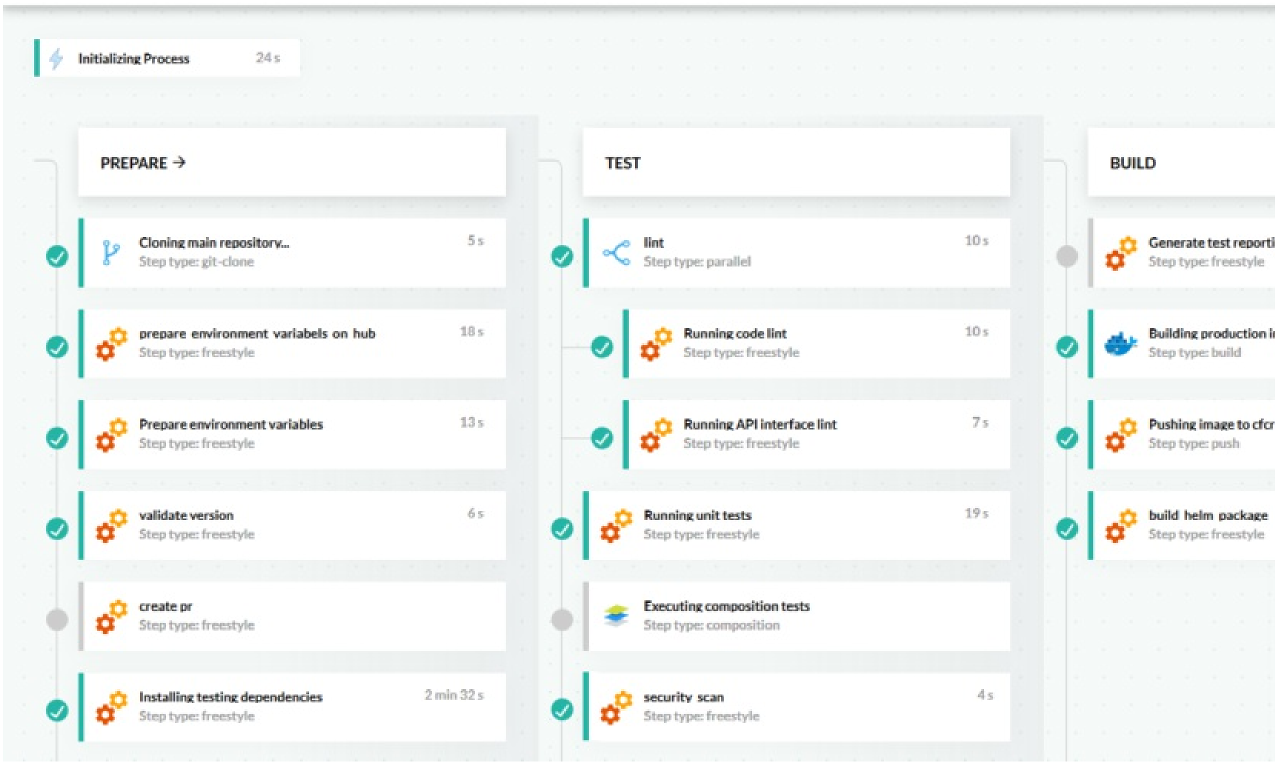
A CI pipeline for microservices

First, we have the Continuous Integration pipeline. This pipeline is connected with git triggers to each of our microservices.


Builds

Help

	STEPS	START TIME	DURATION	REPOSITORY	COMMIT	PIPELINE
COMPLETED	11/14	3 hours ago	11 min 58 s	codefresh-io/cf-ui	fix-cluster-pending-status-color/bf04...	ci


Initializing Process 24 s
PREPARE →
Cloning main repository... 5 s Step type: git-clone
prepare environment variabels on hub 18 s Step type: freestyle
Prepare environment variables 13 s Step type: freestyle
validate version 6 s Step type: freestyle
create pr Step type: freestyle
Installing testing dependencies 2 min 32 s Step type: freestyle
TEST
lint 10 s Step type: parallel
Running code lint 10 s Step type: freestyle
Running API interface lint 7 s Step type: freestyle
Running unit tests 19 s Step type: freestyle
Executing composition tests Step type: composition
security scan 4 s Step type: freestyle
BUILD
Generate test reporti Step type: freestyle
Building production i Step type: build
Pushing image to cfcr Step type: push
build helm package Step type: freestyle

CI pipeline for microservices

The pipeline steps do the following:

- Check out the code for each service
- Check the software version and whether this is a Pull request or not
- Download extra dependencies for testing
- Run linting (i.e. syntax checks) on the source code
- Run unit tests
- Run integration tests using Docker compositions
- Generate test reports.
- Build deployment docker images
- Run security scans using Twistlock
- Push images to the internal docker registry
- Create Helm packages

The pipeline definition has several conditionals so not all steps are running at all builds. Some of them are skipped depending on the nature of the commit.

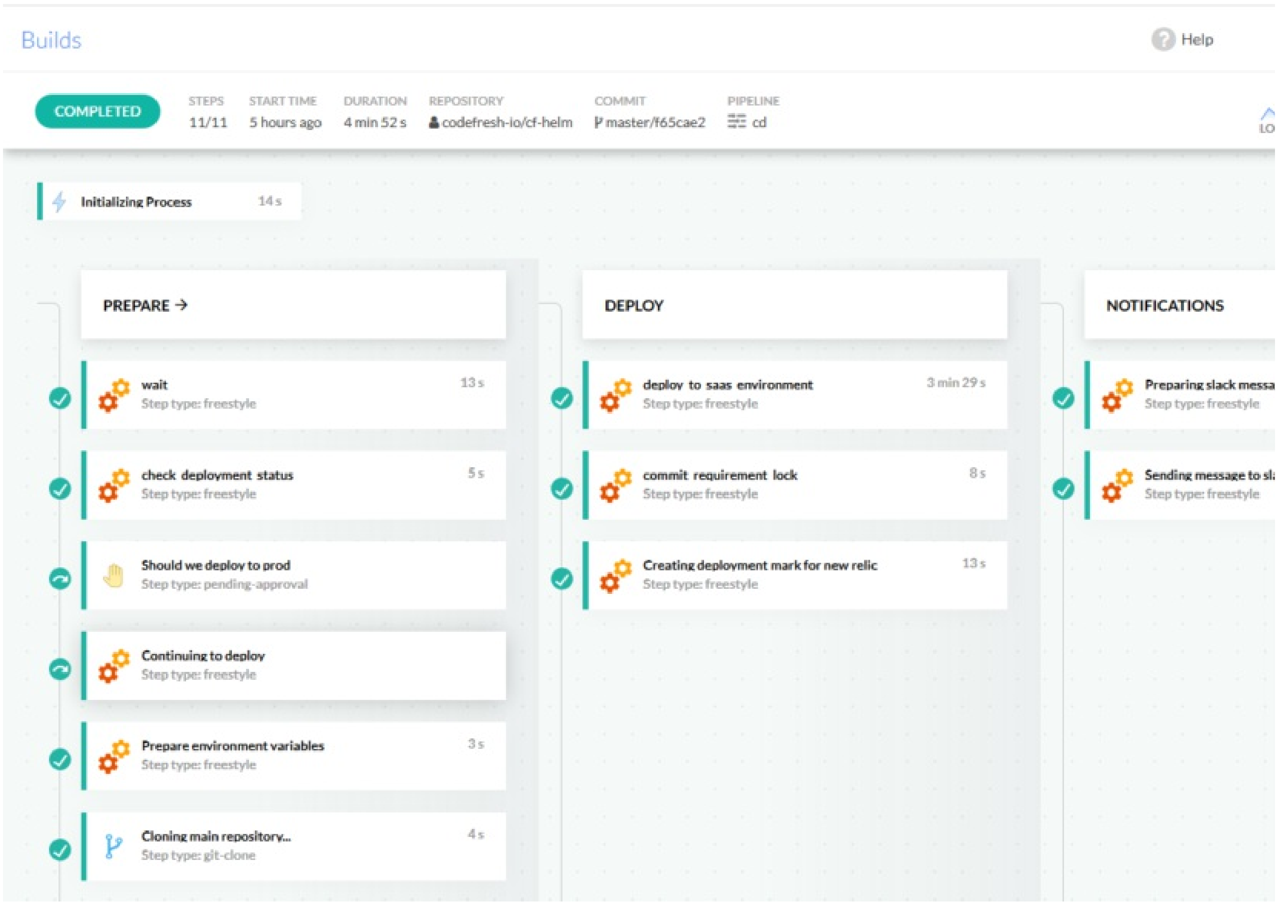
The pipeline is connected via git triggers to all microservices, so whenever a commit happens it runs automatically and prepares both a Docker image and Helm chart for each release candidate.

The important point is that this pipeline does NOT refer to any specific git repository, nor it is connected to one. It is a generic CI pipeline that is reused by all microservices depending on the trigger that calls it.

This makes pipeline maintenance very easy, as there is only a single definition to maintain. It also means that all best practices within a company (e.g. security scanning or quality control) are automatically applied to all projects and teams without any extra effort.

A CD pipeline for microservices

The next step is to run the Continuous Deployment pipeline for each microservice:



Builds
Help

COMPLETED
STEPS 11/11
START TIME 5 hours ago
DURATION 4 min 52 s
REPOSITORY codefresh-io/cf-helm
COMMIT master/f65cae2
PIPELINE cd
LO

Initializing Process 14 s

PREPARE →
wait 13 s
Step type: freestyle
check deployment status 5 s
Step type: freestyle
Should we deploy to prod
Step type: pending-approval
Continuing to deploy
Step type: freestyle
Prepare environment variables 3 s
Step type: freestyle
Cloning main repository... 4 s
Step type: git-clone

DEPLOY
deploy to saas environment 3 min 29 s
Step type: freestyle
commit requirement lock 8 s
Step type: freestyle
Creating deployment mark for new relic 13 s
Step type: freestyle

NOTIFICATIONS
Preparing slack messa
Step type: freestyle
Sending message to sl
Step type: freestyle

CD pipeline for microservices

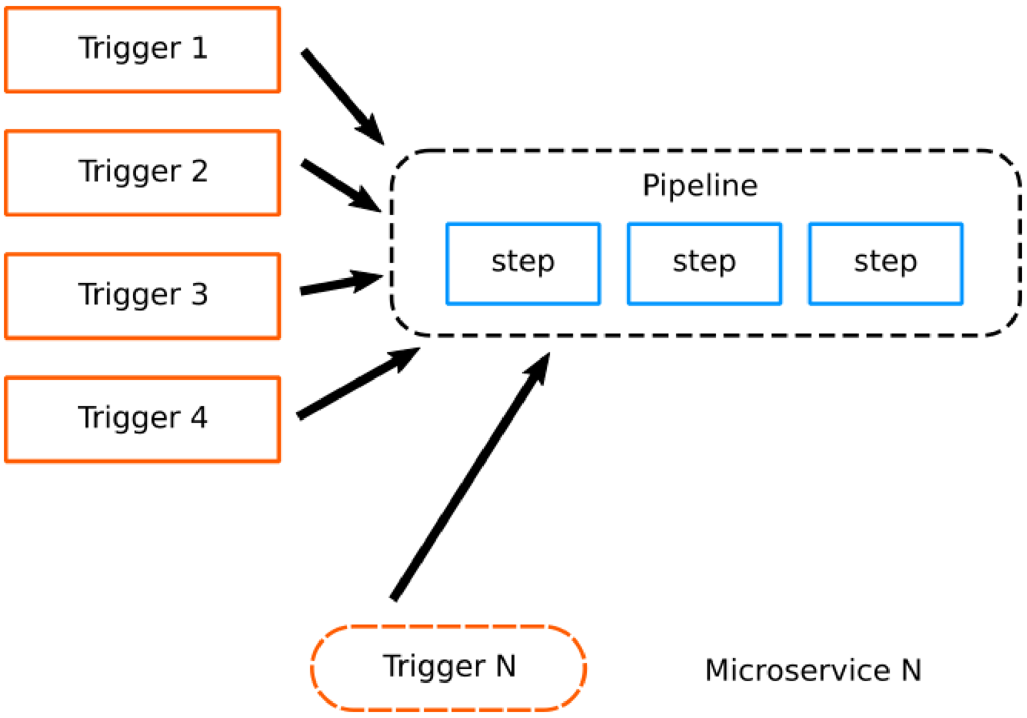
This pipeline is launched automatically as part of a "promotion" which takes place when the Codefresh team is making an official release for a microservice. It does the following:

- Checks that no other deployments are happening in parallel for the same service
- Waits for manual approval
- Installs the previously prepared Helm package (created in CI pipeline)
- Commits back to the git repo the dependencies list of the app
- Creates a mark in New Relic for monitoring purposes
- Sends a message to our Slack channel to show the status of the deployment

Again, if you read the pipeline definition you will see that it is not tied to any specific git repository. Instead, it is reused for all microservices that need it.

The advantages of using CI/CD pipelines for microservices in this pattern should now be clear to you. Instead of spending time with a mess of pipelines and a big number of "reusable" pipeline segments or shared libraries, with Codefresh you can centralize all the CI/CD logic in just two pipelines.

The icing on the cake is that if tomorrow we need to add a new microservice in the mix we only need to add a new GIT trigger for it in the existing pipelines. It doesn't get any easier than this!



Adding a new CI/CD pipeline for a microservice

Ready to try Codefresh and start creating your own CI/CD pipelines for microservices? Create Your Free Account Today!]]>

About Kostis Kapelonis

Kostis is a software engineer/technical-writer dual class character. He lives and breathes automation, good testing practices and stress-free deployments.

Enjoy this article? Don't forget to share.

Comments

* All fields are required. Your email address will not be published.

Comment...

Name

Email

Post Comment

Related Posts

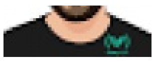
C O N T I N U O U S   D E P L O Y M E N T / D E L I V E R Y

Create Codefresh pipelines with new project grouping, enhanced editor and ready-to-use templates

Kostis
Kapelonis

C O N T I N U O U S   I N T E G R A T I O N

How to integrate your AWS CodeCommit repository to Codefresh



Guy
Salton

CATEGORIES

Helm Tutorial

Deployment Verification Testing

Serverless

Continuous Deployment/Delivery

Docker Tutorial

Kubernetes Tutorial

Containers

Codefresh News

Continuous Integration

DevOps-Tutorial

Docker Registry

Webinars

Security-Testing

How Tos

Get a head start on building better pipelines!

Schedule a demo with a Codefresh expert today.

Request Demo

Product

- Kubernetes Deployment
- Pricing
- Status
- Docker CI
- Continuous Delivery for Kubernetes
- Helm Release Management

Resources

- Codefresh Live Events
- Codefresh Plugins
- Documentation
- Case Studies
- Kubernetes Guides
- Docker Guides
- How-to Videos
- Containers Academy
- GitHub

Company

- About Codefresh
- Contact Us
- Careers
- Blog

Get Stated

Create Account

Schedule Demo

Follow Us

Terms of Service