



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

KUBERNETES CANARY DEPLOYMENT CONTROLLER

ŘADIČ POSTUPNÉHO NASAZENÍ SOFTWARE NAD PLATFORMOU KUBERNETES

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

Bc. PETER MALINA

SUPERVISOR

VEDOUCÍ PRÁCE

RNDr. MAREK RYCHLÝ, Ph.D.

BRNO 2019

Abstract

The need to deliver value to users sooner rises each year in the competitive IT market. Agility and DevOps are becoming critical aspects of software engineering, seeking tools to enable and improve agile culture. Software projects in such culture tend to deal with new deployment strategies to reduce the risk of new changes breaking the existing system. However, staging and test environments almost always differ from the production. Using the appropriate deployment strategy, such as the canary, improves the overall stability of the system by first testing the new changes on a small subset of production traffic. Multiple experiments were made to prove that canaries can positively enhance deployment stability and reduce the risk that new changes bring.

Abstrakt

Potreba dodania hodnoty užívateľom každoročne rastie na kompetitívnom trhu IT. Agilita a DevOps sa stávajú kritickými aspektami pre vývoj software, vyhľadávajúci nástroje ktoré podporujú agilnú kultúru. Softwarové projekty v agilnej kultúre majú častú tendenciu zaoberať sa stratégiami nasadenia ktoré redukujú risk nasadenia nových zmien do existujúceho systému. A však, prostredia určené pre vývoj a testovanie sa takmer vždy odlišujú od produkčných. Využitie primeranej stratégie nasadenie ako canary zlepšuje celkovú stabilitu systému testovaním nových zmien na malej vzorke produkčnej prevádzky. Bolo vykonaných niekoľko experimentov pre dôkaz, že stratégia canary môže pozitívne ovplyvniť stabilitu nasadení a redukovať risk ktorý prinášajú nové zmeny.

Keywords

Agile, Kubernetes, Istio, Continuous Deployment, Canary

Kľúčové slová

Agile, Kubernetes, Istio, Kontinuálne Nasadzovanie, Canary

Reference

MALINA, Peter. *Kubernetes Canary Deployment Controller*. Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Kubernetes Canary Deployment Controller

Declaration

Hereby I declare that this semestral project was prepared as an original author's work under the supervision of RNDr. Marek Rychlý, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....

Peter Malina

May 20, 2019

Acknowledgements

I would first like to thank my advisor, RNDr. Marek Rychlý, Ph.D. for his guidance and help to steer this thesis in the right direction while allowing it to be my work. I would also like to thank many experts, namely: Mike Nikles (Google) for helping me better understand the needs of companies and reviewing my concept, Adam Glick (Google) for reviewing multiple designs and pointing me into right directions, Ross Boyd (Dashly) for making it possible to review and test the outcomes on an extensive production system and Jakub Kulich (FlowUp) for providing an additional perspective on operating and managing deployments in multiple systems.

Contents

1	Introduction	4
2	Agile in Software Development	5
2.1	The “Why” of Agile	5
2.2	Incremental Iterative Development	6
2.3	Product Scale	6
2.4	Microservice Architecture	7
2.5	Continuous Integration and Delivery	8
3	Continuous Integration and Delivery	10
3.1	Continuous Integration	10
3.2	Continuous Delivery	10
3.3	Infrastructure as a Code	12
3.4	GitOps	13
3.5	Deployment Strategies	14
3.5.1	Recreate	14
3.5.2	Blue-Green	16
3.5.3	Shadowing	16
3.5.4	A/B Testing	16
3.5.5	Canary Deployments	16
3.6	Existing Tools for Continuous Delivery	18
3.6.1	Spinnaker	20
3.6.2	Kayenta	20
4	Kubernetes	21
4.1	Containers	21
4.2	Kubernetes Container Management	22
4.3	Pods and Services	22
4.4	Custom Resource Definitions	23
4.5	API Machinery and Clients	23
4.6	Kubernetes Operators	24
5	Istio	25
5.1	Traffic Management Model	25
5.1.1	Gateway	25
5.1.2	Virtual Service	26
5.1.3	Destination Rule	26
5.2	Envoy Sidecar Proxy	27

5.3	Telemetry	27
5.3.1	Prometheus	28
5.3.2	Kiali	28
5.3.3	Grafana	29
6	Prometheus	31
6.1	Metrics Collection	31
6.2	PromQL	31
7	Design	34
7.1	Runtime Platform	34
7.2	Components of Canary Setup	35
7.2.1	Krane Operator	36
7.2.2	Canary Policy Resource	39
7.2.3	Canary Resource	39
7.2.4	Test Job	40
7.2.5	Judge	41
7.3	Using from CLI	42
8	Implementation	44
8.1	Manipulating Kubernetes and Istio Resources	44
8.2	Deployments Reconciliation Controller	45
8.2.1	Deployments Predicate	45
8.2.2	Canary Reconciler	45
8.2.3	Baseline Reconciler	45
8.2.4	Service Reconciler	46
8.3	Canary Reconciliation Controller	47
8.3.1	Test Reconciler	47
8.3.2	Judge Reconciler	47
8.3.3	Virtual Service Reconciler	47
8.3.4	Reporting Reconciler	48
8.3.5	Cleanup Reconciler	48
8.3.6	Operator Access Rights	48
8.3.7	Packaging with Helm	49
8.3.8	Development and Deployment with Skaffold	50
9	Experiments	52
9.1	Aggregator and Storage Microservices	52
9.1.1	Common System Setup	53
9.1.2	Initialization and Testing Stage	54
9.1.3	Canary Stage	54
9.2	Production-Grade Search Engine	57
9.2.1	Common System Setup	57
9.2.2	Production Usage and Results	57
10	Conclusion	60
	Bibliography	62

Chapter 1

Introduction

Software engineering methodologies have rapidly evolved towards agility and need for quick development of products. Adopting agile concepts and embracing the change, however, also needs to come with the adoption of necessary tooling that supports such culture. While the waterfall and similar approaches tend to deploy a project at the end, the agile approach creates smaller increments that are delivered multiple times across the project timeline.

Tooling has a crucial role in the agile world, supporting teams in incremental building and testing of the product. All the more in the production environment, where the team needs to create product increments with confidence, avoiding errors that could create a potential loss in business or lousy user experience. Testing and deployment strategies are evolving to address this issue, mostly by using staging or development environments.

However, one environment tends to be unique — the production, the holy grail of every product. The production environment tends to serve a higher number of users than staging or dev, and policies or configurations that apply to it are more strict. Thus, they are not equal. Testing changes that may affect the production environment in staging requires considerable effort. Moreover, the changes may also affect product dependencies or other dependent products. Staging environments tend to use sandboxes or mocks of dependencies, which makes it even harder to test changes affecting third parties.

Canary deployments are a strategy that uses the production environment to test the new version of the software in the production safely — the canaries test on a small percentage of production traffic. The deployment approach uses multiple trusted sources, such as metrics exposed by agents, or status of the underlying scheduler, to carefully watch for anomalies and health of deployed service. In case of failure, the system operating the canary performs rollback and report to the development team. Testing changes in the environment they will end up running in can radically decrease a chance of error when deploying new versions.

Chapter 2

Agile in Software Development

2.1 The “Why” of Agile

Agile methodologies were invented to allow product teams to deliver products sooner. This is accomplished by prioritizing product features in a way that allows the team to deliver items with highest value first. Moreover, flexible approach to development may greatly reduce a risk of delivery - e.g. delivering a feature nobody wants.

Scrum, as the most popular agile framework will be discussed in the next chapters, because Agile itself comes with many vague definitions, that are better described when using a particular framework. [7]

Using a framework such as Scrum often comes with a number of activities such as standups, plannings, refinement meetings or sprint reviews. These activities are also called ceremonies. Each ceremony is timeboxed (has a limited time) to allow the team to focus on a single task. [7]

Moreover, Scrum encourages usage of iterative approach using sprints. Sprint is a timeboxed space that commonly spans one, two or four weeks. The longer the sprint is, the more risk it involves in an unstable environment (e.g. when starting the project). This is due to nature of unchangeable sprints - once decided, sprint backlog cannot be moved.

Scrum team consists of 3 main parts:

- Scrum Master - a person responsible for teams compliance with the Scrum framework. The main role of such person is to make sure the team speed is always highest possible by removing obstacles and reducing waste (anything that does not add value).
- Product Owner - a person responsible for prioritizing the tasks and deeply understanding a project the team is working on. The main role of a product owner is to always understand where the value is.
- Development Team - a cross-functional team that has all the skills needed to deliver the project spanning across multiple people.

Next sections are mostly focused on the development team, as it's responsible for the technological decisions when creating the product. However, it is essential that development and deployment strategies are created in a contribution of all team members complying with their goals: speed, value and quality.

2.2 Incremental Iterative Development

The Scrum framework comes with incremental and iterative development approaches. Incremental development states, that features should be added in smaller chunks of work, while iterative defines, that this should happen repeatedly - in sprints. [7]

Scrum also states, that a potentially releasable product should be available at the end of each sprint (it does not state that one needs to release it). However, this approach puts a pressure on the development team, as the focus is no longer on the feature releases, where one could simply postpone the release if the feature was still in progress, but rather on iterations. [7]

This approach comes with many good trade-offs. Because iterations are stable, the team can maintain its steady pace, while stakeholders always schedule their meetings after the same time from the last meeting. Iterations also help the team to maintain contact with stakeholders, syncing the product vision across the company.

When executed correctly, the team produces increments in each sprint - thus, the product becomes larger. Because Scrum recommends maximum number of 9 people in the team (commonly reduced to 7 in many organizations) or 3-5 for research teams, it's usually a good idea to scale the number of teams to maintain low communication overheads instead of scaling a single team further.

2.3 Product Scale

There is a plenty of positive product metrics, such as active users, number of new successful features or customer happiness. User-scale related metrics may be mostly connected to the infrastructure maintenance costs, however, other metrics (such as number of features) often come with an increased complexity of the underlying software.

The steady pace of the Scrum team usually slowly decreases with a rising level of software complexity. That is the time, where teams commonly split into *multiple product teams*, as the product has become too large for a single team to handle. Nonetheless, this brings more complexity on the side of communication and makes releases harder, as both teams may act on their own, having *their own release cycles*. This puts the continuous value delivery at risk, especially from an increased number of dependencies that become to raise between the teams (e.g. one team building a product the other team needs to build on further).

Software complexity is frequently improved by abstractions such as APIs (Application Programmable Interfaces). These help teams communicate responsibilities for the parts of the product and consume the APIs accordingly. Moreover, testing becomes more error-prone and must be supported by extended tooling as connections arise between the product teams implementations.

Furthermore, product releases now retain *multiple artifacts* from multiple teams, such as their own implementations, databases, caches or other systems. This creates a brand-new world for the organization, as the number of components may increase with each new team. [1]

A common practice is to introduce the microservice architecture, which comes with its own set of guidelines and best-practices for the team, that allows organizations to decrease the time to scale teams.

That's where automation comes in handy. Tooling allows teams to keep up the speed they had, automating and decreasing the toil that comes with everyday processes such as integration of code, deployment or monitoring. The process of continuous value delivery

builds on two more processes: Continuous Integration (CI) and Continuous Delivery (CD) which will be discussed in the next chapters.

2.4 Microservice Architecture

Microservice architectures are known for their loosely coupled components that allow flexibility and horizontal scaling, both for teams and the code they write. They are a common pattern for designing distributed systems. [5] Moreover, every microservice should act as an atomic unit in the system complying with the *single responsibility principle*, which makes both communication between teams and product maintenance easier. [1]

As flexibility and horizontal scaling have major impact on Agile process and velocity (value that the team brings each sprint), these two metrics often relate to product success or failure (assuming the product is verified and has a good market fit). Thus, microservices are frequently picked up as a solution for bigger-to-large distributed product teams. It is also common for a single team to maintain multiple microservices, but this decision often comes from the software performance perspective, instead of the team scale one.

The microservice architecture allows teams to decouple their features from the rest of the product using APIs. [1] Each microservice usually exposes a set of methods that represent individual features that the team is maintaining (e.g. user authentication or transaction handling).

The domain of a microservice acts as an abstract model (or context) of the real world. It is a result of determining boundaries and decomposing the larger model of the project. The decomposition should be a model-centric view of the software we are creating, which leads to creation of reflections of the reality of our users - e.g. there may be a difference for a user of train company website who wants to buy a ticket and for an employee behind the desk, even though it's a software selling tickets for both. [1]

A single microservice generally handles a single domain, also known as autonomous business domain (e.g. loans in finance or project management in software engineering). It's crucial for the team to understand the domain as it should reflect the real world as much as it can - making it easier for everybody in the team to understand, as it's not usual for software engineers to have a broad knowledge of other domains such as finance or health. [1]

To better illustrate the implications that come from introducing many services, diagrams shown on figure 2.1 show Amazon and Netflix microservice architectures - also called Death Star Compositions. Decision if it's hard to maintain such architecture is left on the reader.

There is no single answer when it comes to deciding, what should be a microservice, as there are certain drawbacks with every microservice that is created:

- Distributed systems are complex (this complexity comes from network hardware failures, modeling transactions or latency) [5]
- Each service is an independent unit, which comes with management costs (CI/CD, monitoring, communication between teams and their APIs)
- Testing microservices means spinning up multiple servers or creating mocks for each service (making it hard to track down semantic changes)
- Each microservice should be deployed independently. However, microservices connect to each other, so outage of one can mean outage of a whole platform without a proper plan. [1]

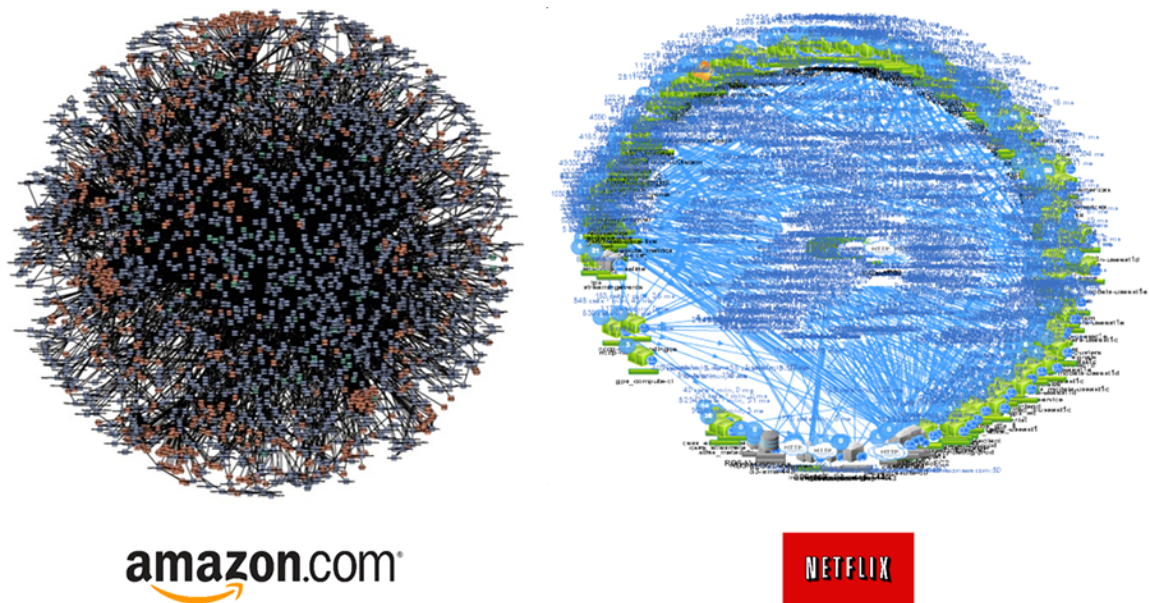


Figure 2.1: Death Star compositions of microservices by Amazon (left) and Netflix (right) (AppCentrica 2016 [2])

- Recovering tens or hundreds of microservices with their databases, caches, proxies, etc. is more difficult than recovering a single replicated instance.

These drawbacks make it a hard decision for teams to decide whether to introduce a new microservice to the mesh, or extend the one that is already allocated. Tooling introduced by Continuous Integration and Continuous Delivery helps to cross these boundaries and automate the extreme amount of work that comes with introducing new components.

2.5 Continuous Integration and Delivery

Feedback cycle and time to feedback recently became one of the most important metrics in software engineering. The faster the time to feedback is, the faster engineers are able to create new features or fix existing bugs. As opposite, it becomes hard to fix bugs that were introduced by code changes month, or even a week ago - this is due to constant work that is being done and needs to be processed when finding the bug.

Continuous Integration (CI) and Continuous Delivery (CD) are two processes in the popular DevOps culture. These processes enable teams to securely make changes to the code that is then tested and deployed to serve users. They also allow fast feedback to developers working on a project. (Note: when talking about fast feedback, it's common to identify process that takes up to 4 hours as fast, as engineers are still able to make multiple changes within a single day).

While Continuous Integration fits mostly into Build and Test parts of the framework shown on the figure 2.2, the Continuous Deployment has its space in Release and Deploy parts. Other parts on the picture above, such as operations or monitoring are done by the development team or automated via other tools. The plan and code parts heavily rely on the development team itself.

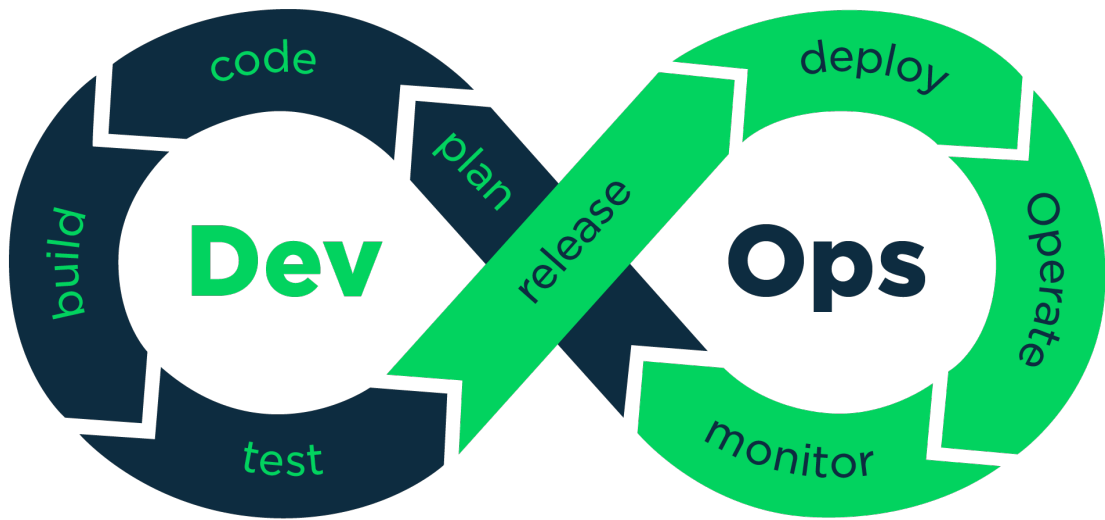


Figure 2.2: Stages of the DevOps cycle (Irma Kornilova 2017 [9])

Agile builds on ability of teams to react and change directions (pivot) quickly within a single goal and domain (meaning, if the goal changes as whole, the process can't apply). However, to enable great agility and reaction times, every team needs to ship with confidence. The confidence comes from both, the scrum master and product owner abilities to guide the product, as well as the level of automation within the product as whole, down to single microservices. As team grows its automation framework, it can gradually rely on tools to catch unwanted behavior, bugs or performance changes. [7]

Chapter 3

Continuous Integration and Delivery

As mentioned in the previous chapter, CI and CD are large parts of a process commonly known as continuous value delivery, that enables continuous end-to-end supply of product features, fixes and optimizations. Moreover, this process heavily relies on the time to feedback which helps engineers to understand consequences of their work. This chapter will talk about implementation of CI and CD, while mentioning challenges and their potential solutions for agile teams.

3.1 Continuous Integration

Continuous Integration is a simple process (compared to Continuous Delivery) of integrating code change as soon as possible. This process helps teams to integrate their code multiple times per day, while validating that the implementations are correct using a number of checks. As simple as the process looks, it relies heavily on robustness of lint, build and test pipelines to catch unwanted changes. This process is even more valuable when making large changes to the codebase, catching possibly unwanted changes in code not owned by the contributor. [\[13\]](#)

The figure [3.1](#) shows a commonly used CI pipeline. The pipeline triggers on every commit of a developer to the repository. Series of steps are executed afterwards, which can be customized by the team, but these steps generally consist of linting (making sure the code complies with the style standard of the team), build (compiling the application to create an artifact usable by the team during deployment) and testing (running whole test suite to prove integrity of the committed code).

Integration of CI into the development process of an Agile teams helps the team build more trust in changes they are doing, as risky changes may be oftentimes caught by the tooling before they are released.

3.2 Continuous Delivery

Continuous Delivery allows teams to build the product in a fast, short cycles, ensuring that the software they built can be reliably released manually at any time afterwards.

It connects on the continuous integration pipeline. The pipeline either accepts artifacts from its predecessor, CI, or builds its own artifacts that are deployable. It then performs

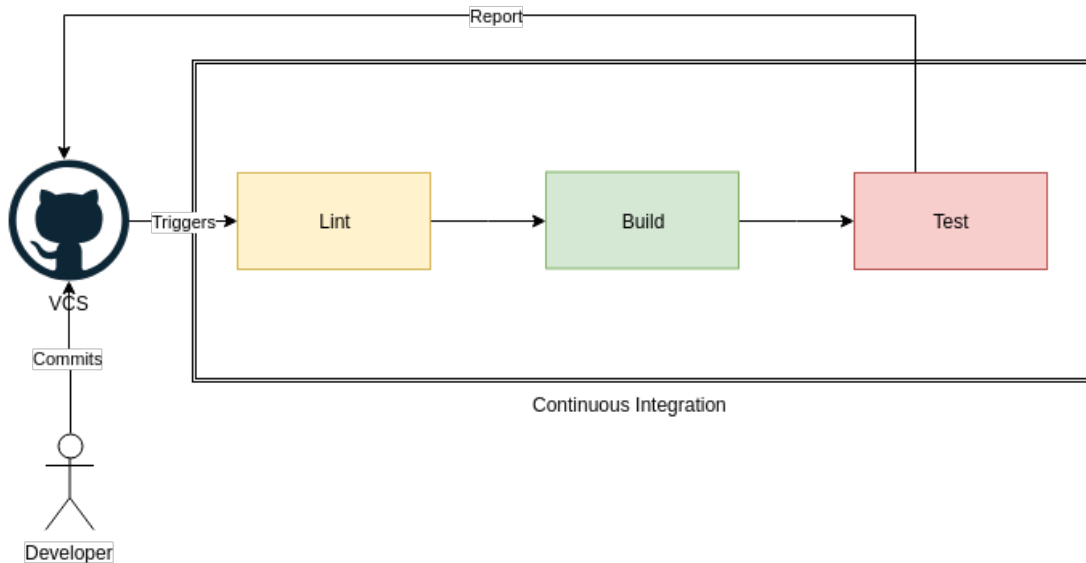


Figure 3.1: Sample CI pipeline containing 3 stages: lint, build and test

application acceptance tests and waits for manual judgment to release. Application acceptance tests are commonly deployed into a staging area, which provides a near-production environment for testing. [13]

Moving it a step further, continuous delivery pipeline can be fully automated, avoiding the manual judgement at its end. This process is then called Continuous Deployment, and requires excessive robustness of the test pipelines.

Integration testing pipeline often becomes robust with the number of tests that need to be performed before feedback is ready for developers. While CI mostly runs unit and integration tests, application acceptance tests are commonly run as end-to-end tests on already running infrastructure.

This creates multiple issues for the development team:

- Infrastructure must become automated, as creating it manually for each test would not be feasible. This becomes even more important if the infrastructure is damaged by the running test, making it unusable for other tests. It's common to recreate the testing environment in these cases.
- A single testing environment may become the bottleneck for the development team. Tests running on the staging environments generally take longer than unit tests. This speed limit comes from multiple sources, such as latency or database migrations.
- It soon becomes expensive to launch multiple environments, both, from the perspective of operators and costs of running the infrastructure.
- There is a number of dependencies and variables that need to be connected to each other for the infrastructure to run correctly. This number often scales rapidly with the development time. These dependencies may also include tooling needed to successfully deploy the original microservices, such as package managers or registries.

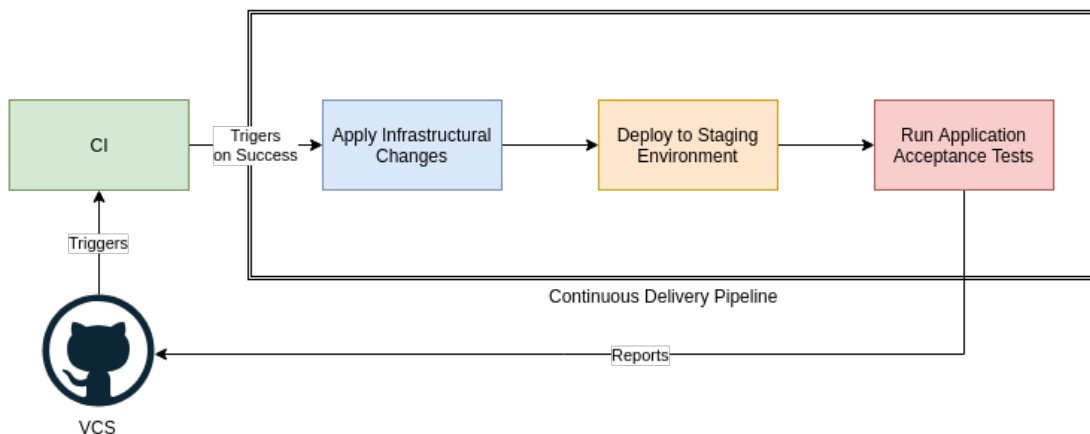


Figure 3.2: Sample CD pipeline triggered by a CI, that applies infrastructural changes, deploys to the staging area and runs the application acceptance tests.

Tackling such problems oftentimes ends up by using Infrastructure as a Code (IaaC). IaaC is a configuration management technique that allows operators to declare the desired state of the target (being machine, cluster, networking, etc.) instead of specifying every step that needs to be achieved to get to the desired state [10].

This pipeline on figure 3.2 shows an example CD pipeline, that is triggered by a success of the CI pipeline. The pipeline first applies any infrastructural changes, making different between current state and the desired state. New artifacts are deployed to the environment afterwards. Application acceptance tests are run as a the last part of the pipeline, as they need to wait for the infrastructure and artifacts to become stable. Results of all parts are then reported to the VCS for the team to investigate if anything failed.

3.3 Infrastructure as a Code

User Interfaces (UIs) and graphical configurators are a great way to lower the barriers for entering the operations world. However, Graphical interfaces often come with a caveat - once you break your configuration, its gone and one can't revert it. These configurations are almost never repeatable nor can be automated by tools.

IaaC, on the other hand, seems to be the reliable approach for system configurations. Even though this approach has reached only single machines in the past, it's now common to build whole infrastructures - including their networking and disks via IaaC solutions such as Terraform. Storing the configuration as a code, e.g. within a VCS (Version Control System) makes the configuration safe. Moreover, it comes from the nature of IaaC that these configurations are repeatable. [10]

Configurations of IaaC also commonly store a so called state. This state provides a last visible snapshot of the infrastructure that was applied. For example, when deploying a VM (Virtual Machine), we would first see the state empty and the VM would be written to the state once it was deployed. This state is often one-way only, so making manual changes to the VM would not overwrite the change in the state and confuse the tool on our next infrastructure application. [6]

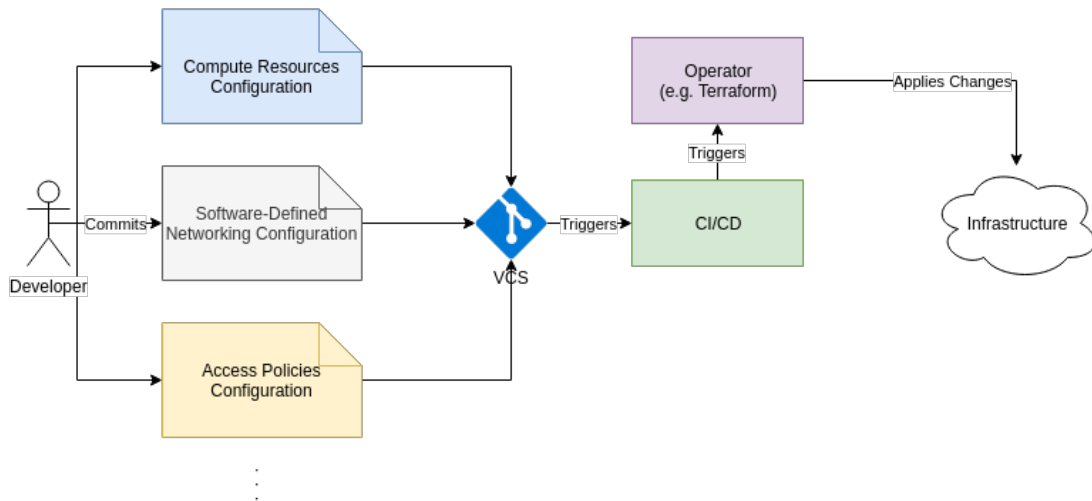


Figure 3.3: An example flow of committing infrastructural configurations and their application

Infrastructure provisioning is a critical part for the process of continuous delivery. IaaS tooling such as Terraform help teams to create infrastructure configurations that are able to spin up the infrastructure whenever they need to. Nonetheless, this process may still become cumbersome, as debugging live infrastructure is more than unpleasant job. However, the state snapshots make this process slightly better - development team is able to reproduce last applied state of the infrastructure to a sandbox area, where the debugging may occur in a more isolated environment, while still having the infrastructure properties that caused the issue.

The diagram displayed by figure 3.3 shows an example IaaS flow, where a developer creates a new compute resources (e.g. VMs), defines networking in another configuration (e.g. load balancing, ip forwarding or DNS) and configuration for access policies (e.g. IAM configuration). Upon committed, the configurations are validated by the CI, while the CD triggers an operator to perform these changes on the target infrastructure. This allows transparent changes to the infrastructure, visible within the VCS.

3.4 GitOps

GitOps extends IaaS further into the world of automation, introducing source control as a single source of truth for runtime configurations. While IaaS allows us to define static configurations of tools, policies or templates used to perform additional tasks, GitOps fully reflects the state that can be observed in the production environment (thus, not only last applied state). [16]

This strategy enables a higher level of observability, which allows the development team to watch for changes within the infrastructure as they come, having better visibility on possible relations between broken environment and even slight infrastructural changes. All of these changes are commonly stored in VCS such as Git, however, it's good to have a separate infrastructure repository, as infrastructural logs may become bloated once operators begin to debug or optimize the infrastructure.

Fully integrated GitOps allows operators to make changes on the fly right in production deployments, while being reflected back into git. However, this process may sometimes be unwanted as changes instantly skip any CI and CD process, which may break the target environment. Thus, it's common to use this technique solely for the purpose of debugging or on-fly optimizations where the state can't be observed on carbon copies of the environment.

However, GitOps principles can still be used without introducing this both-way synced repository by making sure that:

- VCS always holds a snapshot of configuration that can be applied.
- The snapshot has passed all tests and can be renewed if runtime crashes

This enables the team to focus on verifying that things really work before released, instead of making ad-hoc changes afterwards. It also becomes easier for teams to grasp, as any error-state can be snapshotted from runtime, while being automatically replaced with the stable configuration. This enables further debugging and fixing.

To summarize, IaaS dramatically improves visibility of the process done by operators. It allows infrastructure code to also pass checks such as lints or dry runs to validate their integrity. GitOps then allows to define runtime workloads in a Version Control System (VCS) as well, making VCS the source of truth whenever the deployment of the latest stable version of the system is needed.

3.5 Deployment Strategies

Deployment strategies make an important part of the Continuous Delivery process. The process only applies infrastructural changes and updates artifacts from the simple perspective. However, there are multiple scenarios within this process that should be handled differently, such as:

- Database migrations, as all components will need to work with a different model.
- Zero-downtime roll-outs, because we the workloads may be serving life traffic and downtime would be disturbing to the users.
- Risky or hard-to-benchmark changes that may put the whole infrastructure down, such as reconfiguration of network.
- Hard-to-test changes that depend on a large amount of dependencies and can be tested reliably only in production-like environment.

Because different scenarios have different requirements and sometimes conflict (e.g. database migration and zero-downtime rollout), there are multiple common strategies that can be applied to the CD process, based on their goal.

3.5.1 Recreate

Recreate strategy is generally used when operator needs to tear down the infrastructure and spin it up again. There may be multiple use-cases for this:

- Running database migrations that are backwards-incompatible.
- Deploying workloads that are not-backward compatible communication-wise (APIs).

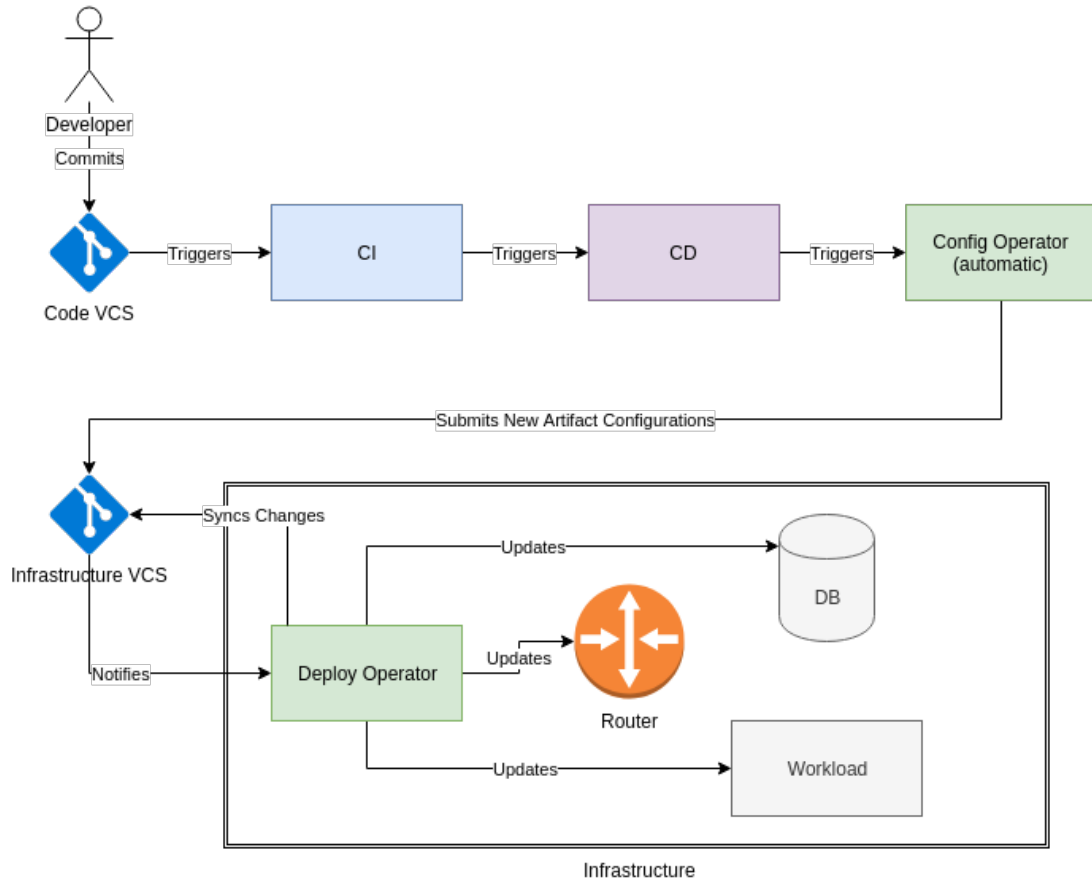


Figure 3.4: A sample GitOps pipeline consisting of two VCS repositories – the product-related code resides in the Code VCS, while the infrastructure configuration in the Infrastructure VCS. Developer making change in the Code VCS triggers both, the CI and CD pipelines, notifying the Config Operator in the end. The Config Operator commits the new artifacts (e.g. container image tags) into the Infrastructure VCS, which then triggers the Deploy Operator. The Deploy Operator updates all resources that were changed by the new configurations to be in accordance with the Infrastructure VCS.

- Using persistent disks that cannot be shared between multiple workloads.

This strategy has a major drawback: downtime between teardown, operation (such as database migration) and spin-up.

3.5.2 Blue-Green

The blue-green deployment is commonly used to reduce the risk of deploying a brand-new version of a product. It reduces the downtime by spinning up a same infrastructure (green) to the production one (blue), releasing the new version and then switching all users to the new (green) version. If anything goes wrong, the traffic is immediately switched back to the previous (blue) version. However, if everything goes well, the old (blue) version is scaled to zero, leaving only the new (green) running. [4] This scenario can be observed on figure 3.5.

Even though this strategy comes with a significant risk reduction, it heavily relies on having the new infrastructure at least the same size as the old one. This can take up days or weeks for big releases, which can negatively impact teams velocity. Another problem comes up when both environments need to either sync their own databases, or databases are not even part of the environments, making workloads connect to another managed environment.

3.5.3 Shadowing

The shadowing approach effectively tests changes that could potentially risk the downtime without affecting the user. Shadowing relies on the same infrastructure as the blue-green deployments. However, there is no replication enabled between DBs of two versions.

Instead of splitting between versions, shadowing copies the traffic, making the same requests to the shadowed version and the production one. All outgoing responses from the shadowed versions are then logged, but not returned to the user. This approach allows feature testing, where two infrastructures are continually compared for changes, making sure there are no unexpected differences.

3.5.4 A/B Testing

A/B testing is a common approach to test new user-facing features such as search algorithms, advanced automation, etc. This approach also requires two different versions of the product to run simultaneously, where traffic is split between requesting users in a previously-specified ratio (e.g., 90:10).

Users routed to the A/B tested version are flagged using tools such as browser cookies or HTTP headers that must be present if the client wants to route to the AB test again. This is done on purpose, so users in A/B test won't be switching between the two versions blindly, which would not be expected from the UX perspective.

However, the infrastructure can be scaled proportionally, instead of running two 100% infrastructures like in the blue-green deployments.

This approach can be often times very effective when testing new version of products, which are backward compatible. Nonetheless, there is no way to test backward-incompatible releases that would require database migrations.

3.5.5 Canary Deployments

The canaries use a different approach to all others, but slightly similar to A/B testing. The difference between A/B testing and Canary Deployments comes in a way they route traffic.

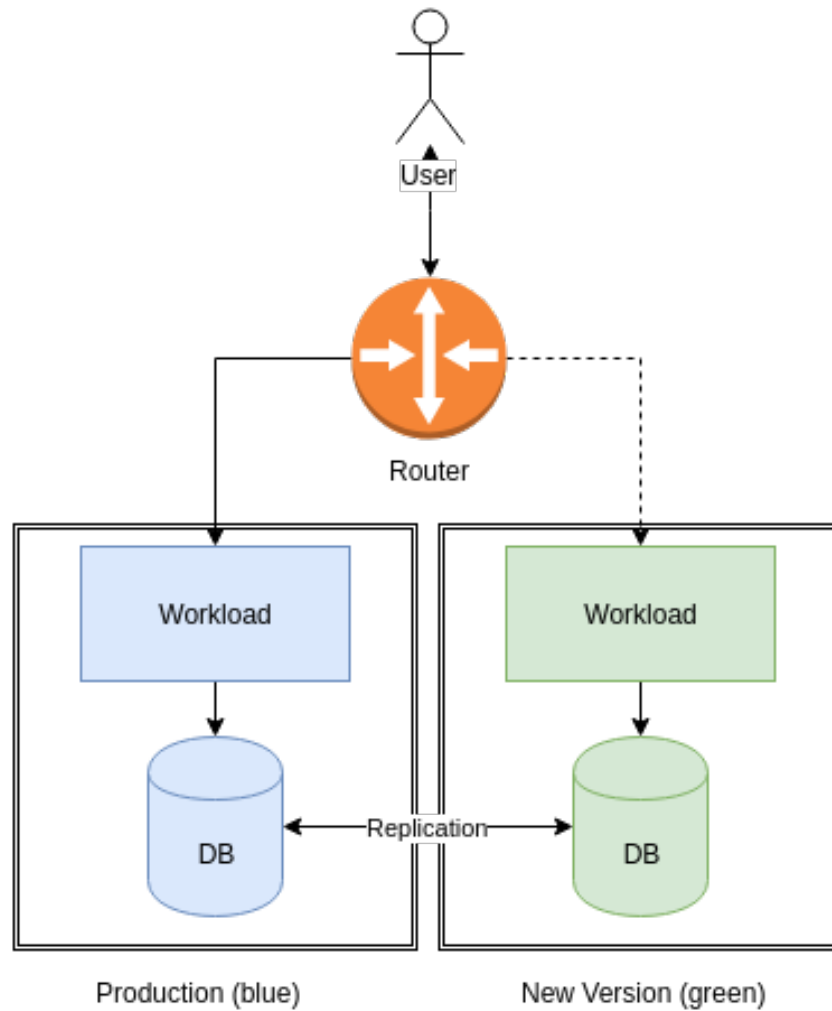


Figure 3.5: An example configuration of Blue-Green deployment strategy that consists of two environments with replicated database and a single router that is being switched to route all traffic first to the Blue deployment and once stable, to the Green deployment. Such configuration requires operators to spin up a whole new infrastructure that is able to handle all production traffic. In case of fialure, all traffic can be safely redirected back to the previous blue version of the system.

While A/B testing tags clients so that further requests can be made to the same version or clients, can freely choose the version, Canary Deployments generally don't care about users requesting the service. Thus, users seeking the service can be blindly redirected to the canary version without notice. This is due to the nature of canaries and A/B tests. While the purpose of A/B testing is to test user's reactions, purpose of the Canary Deployment is the software validity, which may not be visible to the users.

The Canaries generally respond to a small percentage of traffic before they are promoted to all production traffic, making them effectively test the service on a small percentage of users before they reach their full impact.

Even though Canary deployments are a strong approach, it comes with multiple constraints:

- All changes to a database must be backward-compatible (any changes to the model should be avoided in general, but incremental are ok).
- Testing performance requires a baseline component (canary should not be compared with an average of all production workloads) (see figure 3.6).
- Canaries cannot depend on other canaries as the tests may end at different times (the only way is to schedule them together).

Once the Canary is tested and passed all tests, it is gradually promoted to receive more traffic. However, if the Canary fails to integrate with the current system, it is reverted and reported back to the development team.

The figure 3.6 shows a sample canary deployment configuration, where two environments are present. The production environment consists of production workloads (scaled to the production size) and the DB. The canary environment consists of a baseline workload, which is a copy of one of the production workloads, plus the new version. The main router then routes between the Production environment, giving it 95% of the traffic, while 5% is redirected to the Canary router. The canary router then uses simple round-robin to split the traffic between the baseline and the new version. Metrics from both, baseline and the new canary version are compared, while the new version should yield similar or better results than the baseline.

Canary deployments are hard to do in parallel in case one needs to promote each one gradually. To enable parallel canaries, the deployment may be reverted in both cases, whether it succeeds or fails, while its status is being reported to the team. Actual traffic promotion then happens after the configuration is merged into the VCS as described in GitOps.

3.6 Existing Tools for Continuous Delivery

The whole world of continuous delivery is brand-new for most organizations nowadays. There are many emerging CI tools in the open-source community, such as the new version of Jenkins, LambCI that runs entirely on AWS (Amazon Web Services) Lambdas, or container-native Drone. Even though one can set up a simple CD pipeline within any of these solutions, they are not robust enough to call themselves CD solutions, yet.

There is one emerging tool that is becoming popular – Spinnaker. Spinnaker is an open-source collaboration between Netflix and Waze, as both of these companies try to solve large-scale multi-cloud continuous delivery all over the world.

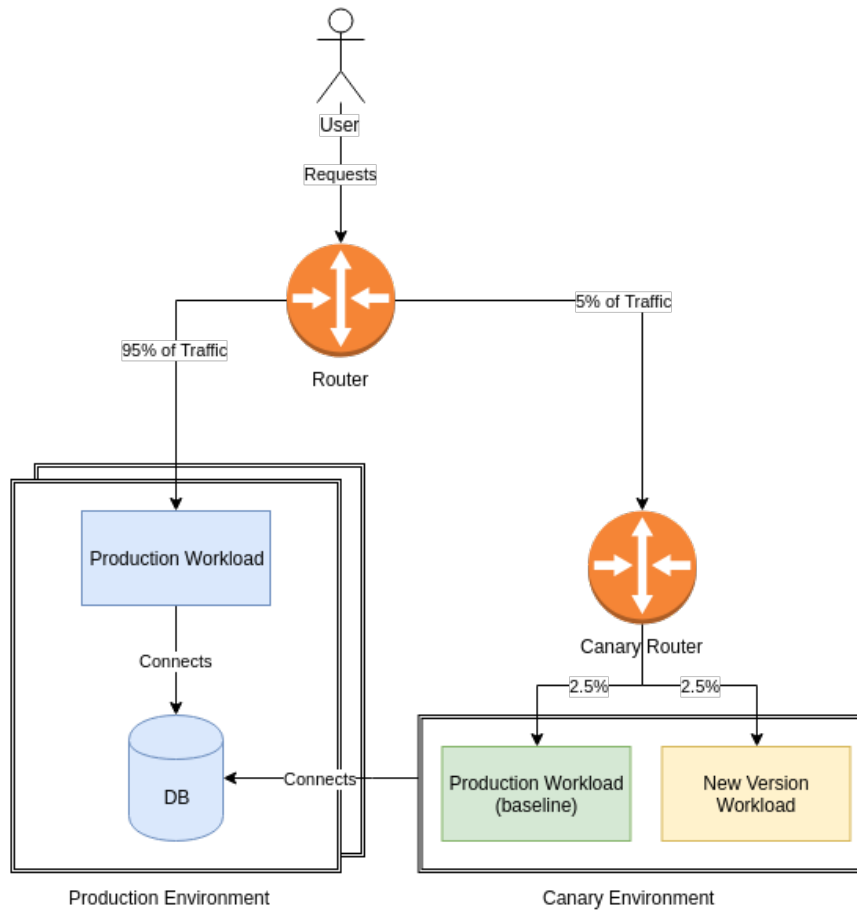


Figure 3.6: A Canary Deployment composition with two routers switching between the stable environment and the canary, using the same production database

3.6.1 Spinnaker

Spinnaker can be defined as a multi-cloud continuous delivery tool, that allows developers to create complex deployment pipelines while providing tools to support multiple deployment strategies. It also supports pipeline triggers from already existing CI systems, such as Jenkins or TravisCI. [14]

There are multiple supported cloud providers, such as AWS, Google Cloud, or Kubernetes as a platform. Spinnaker has a large scale of supported services for deployment, making it a robust multi-cloud solution.

However, as the whole idea of Continuous Delivery is still in its beginnings, Spinnaker also comes with comprehensive documentation on how it works. This can become tough to operate for new or smaller teams, as one must deploy Spinnaker with all its features to start.

Moreover, Spinnaker comes with a sizeable infrastructural cost, consisting of 11 microservices, starting at four cores and 8GB of RAM just to run. Even though this is worth the price for a large company, assuming the cost would be around 105\$ per month for a cloud provider (calculated as a custom machine in the Google Cloud Platform Frankfurt region, 4vCPUs, 8GB RAM), it may be too much for a smaller company to start the system. [15]

Nonetheless, even though Spinnaker can be seen as the most robust CD solution nowadays, its complexity, maintenance cost, and the learning curve can make it hard for smaller organizations to run.

3.6.2 Kayenta

Google and Netflix develop a subsystem for canary analysis called Kayenta - deeply integrated into Spinnaker pipelines, allowing developers to spin-up multiple canary pipelines in parallel, collect metrics, and act upon them.

Kayenta subsystem creates a baseline deployment (typically three instances) as well as the canary deployment (3 instances as well, to match the baseline). It re-configures the target load-balancer to split traffic into these instances, gathering metrics for the judgment. Kayenta then provides a two-step pipeline, where a user first defines which metrics should be collected, and their sources. The second step configures a judge that will evaluate metrics aggregated from 0 to 100 - normalizing the output. The judge then classifies if the canary deployment was a success, marginal or failure. [8]

The system is fully integrated with Spinnaker and resides in its repositories as well. Even though the system itself is of great inspiration, cons for Spinnaker, likewise, apply here.

Chapter 4

Kubernetes

Kubernetes is a platform for creating, deploying, and managing distributed applications. These applications are usually packed as containers, which are fundamental building blocks of our workloads, deriving the underlying kernel and containing runtime, libraries and binaries (or source code) for the application to run. [4]

Istio is a service mesh build on the Kubernetes ecosystem providing advanced features, such as circuit breaking, distributed tracing, fault injection, traffic management, policy enforcement, and more. Istio also comes with built-in tools and dashboards providing better visibility on what's going on in each microservice and the infrastructure overall. [12]

4.1 Containers

Containers are a technology that allows us to package code into self-contained runnable packages. These packages are also called images and contain everything but kernel to run the packaged application. Container images are immutable, and their creation should always yield the same result for the same input. Immutability makes container technology suitable for repeatable deployments and use in a distributed system.

Containers are composed of filesystem layers, where layers stack on each other. Layering makes it easy to reuse some of the data, e.g., when a new version of our source code comes up, we can only build that part, instead of installing the dependencies all over again. [4]

Docker is currently the most popular implementation of containers technology. It comes with its own CLI (Command-Line Interface) and a file format called Dockerfile. Dockerfiles specify all steps (layers) that must be performed to create a container image.

```
# Dockerfile
FROM ubuntu
MAINTAINER Peter Malina <peter.malina@flowup.cz>
CMD [echo, hello world]
```

The Dockerfile implementation above creates a new image from the image of the Ubuntu system (latest version), sets the maintainer and configures the image to run echo once it's run as a container. Once the image is built, it can be found in the local image repository under its name. Once run, the container will print "hello world" to the console and exit.

Container images should always be tagged. The tag defines a name for the container image, under which it can be found in the repository once built. Images can also be pushed into a remote repository. Remote repositories are then used as a central unit for image distribution to the deployment targets (machine).

4.2 Kubernetes Container Management

Kubernetes is a container management system that comes with scaling, resiliency, service discovery, container security, and rolling updates/downgrades out of the box. However, instead of containers, the basic building blocks of Kubernetes are called Pods. Pods group containers into the same network and all containers within a pod always deploy together.

Kubernetes also supports service discovery via a component called service. Pods can scale up and down, making their IP (Internet Protocol) addresses volatile. However, services preserve, and their IP addresses and Domain Name System (DNS) names always stay the same (unless reconfigured).

The whole system is glued together using particular metadata, called labels. Labels are key-value pairs, which can be attached to any component inside Kubernetes - e.g., one can label a pod “app=myapp”. Label selectors can then target this pod within service, that will define this same key-value pair for selection. The traffic coming to the service component is then load-balanced between all pods which match selectors labels.

4.3 Pods and Services

Pods are basic building blocks for workloads running in the Kubernetes clusters. They join containers which can’t run without each other, by deploying them together and maintaining a local network within the pod. Containers in the same pod can then find themselves on the localhost network.

Each pod a ClusterIP allocated (IP address unique to the whole cluster) which can be used for pod-to-pod communication. However, using direct ClusterIPs is not a recommended approach, because a pod may be rescheduled at any time, causing it to lose the IP.

Services provide service-discovery capabilities by creating maps of all selected pods with their ClusterIPs. Whenever a workload connects to the service, the request is redirected to one of the underlying IP addresses of an actual pod.

Even though Services with ClusterIPs solve the communication problem inside of the cluster, inbound traffic is blocked by Kubernetes by default. We can configure a Service to use one of NodePort or LoadBalancer types instead of ClusterIP, to allow incoming traffic from the outside world.

NodePort service allocates a single port for the service and opens this port on all Kubernetes machines running in the cluster. All traffic to this port is then redirected to the service.

LoadBalancer Service acts as a NodePort Service. However, Kubernetes comes with multiple cloud-based adapters, which will then allocate a cloud-specific load-balancer and route its traffic to the port of the service. This approach will create a Google Cloud Load Balancer on Google Cloud or Elastic Load Balancer on AWS, but multiple open-cloud solutions are supported as well.

The example figure 5.1 shows a sample scenario for two pods running the same application. The first application runs a single container within the pod, while second runs an application container as well as the cache container. Both of these pods are labeled “app=myapp” which makes them exposed to the MyApp Service label selector “app=myapp”. Inbound traffic to the MyApp Service will thus be load-balanced between these two pods.

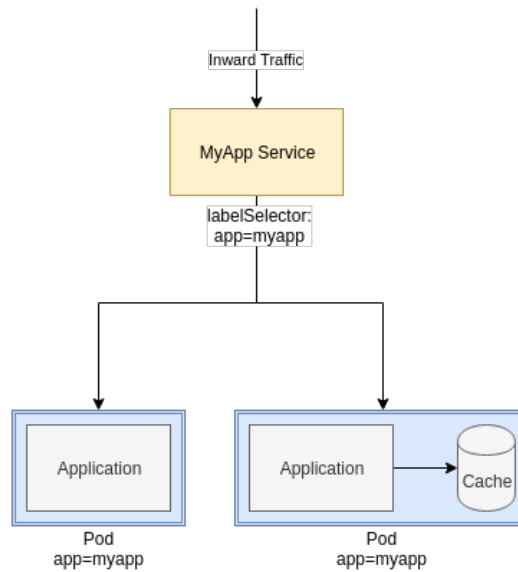


Figure 4.1: A configuration of a single Kubernetes service using the *labelSelector* to target multiple different pods.

4.4 Custom Resource Definitions

Many teams and organizations tackle specific problems where abstractions such as Deployments or Services need further packing into domain-specific configurations for readability and tooling integrations. These configurations often pack together multiple configurations that would have no value with missing resources (e.g., deploying web server Deployment without exposing it to the public via Service).

Custom Resource Definitions (CRDs) allow Kubernetes users to define custom types that can then be versioned and used within the Kubernetes system. CRDs act as first-class citizens and are obtained through the standard tooling. They also preserve the same interface for manipulating as the standard resources such as Deployments.

Because CRDs are also configured via the standard Kubernetes configuration API, they are also defined as JSON/YAML configuration files. Furthermore, CRDs can define an OpenAPIv3 (also called Swagger) properties which are then used for validation and displaying of metadata.

Kubernetes API Machinery client and server implementations then process each configuration. The API Machinery defines an SDK that allows developers to manipulate resources and call the Kubernetes API in multiple different versions.

4.5 API Machinery and Clients

The API Machinery SDK provides a low-level library for encoding/decoding and schemas of basic building blocks of Kubernetes. The library is used as a base for the whole Kubernetes platform as well as its clients and server interface implementations.

The client libraries (such as Golang one) then import and reuse API Machinery to create versioned interfaces for Kubernetes API manipulation. The API supports basic

CRUD (Create, Read, Update, Delete) operations as well as unique metadata and status updaters. Moreover, all resources support watchers - structures used to watch for particular CRUD changes and notify subscribed clients.

Watchers are then heavily used within the Kubernetes implementation itself, as well as in tooling that needs reactively respond to changes made to resources. One of the significant users of watchers are applications called Operators. They watch for changes either in standard definitions or CRDs and make further changes based on their state.

4.6 Kubernetes Operators

Operator as an idea was first released by CoreOS in late 2016, while the first CoreOS operator framework was introduced mid-2018. Operators are supposed to help automate operations on resources that would otherwise be manual, such as creating a new database cluster, or hard-restarting failed resources in case the Kubernetes health-checking system is not sufficient enough. [11]

In the spirit of this thesis, operators are going to be used to fully automate the creation of canary deployment dependencies: baseline and canary deployments, as well as to configure traffic splitting and two worker jobs: test and metric evaluator.

Operators can hook into life-cycle of Kubernetes resources via the Watch API. The API allows them to react to changes in real time. The process inside an operator is also called the reconciliation process. The reconciliation process accepts a change event that defines which resource (by name, namespace, and type) was changed and allows it to modify any other resource within the cluster. However, it's a good practice always to write idempotent implementations as some events may be repeated multiple times (e.g., other resources changing the same resource).

Writing idempotent reconciliation cycles is even more critical in terms of Kubernetes transaction contracts. Because every resource contains a hash of its contents, underlying API Machinery may revoke the request to update a resource specification or its status in cases where the resource has been already changed by another system. Failing the reconciliation cycle will then re-queue the request, firing the reconciliation cycle once more. Any finished actions must thus be skipped and failed repeatedly.

Chapter 5

Istio

Istio is a service mesh built primarily on Kubernetes, that allows configuration of traffic management, security, and telemetry collection via the Kubernetes configuration API. It consists of multiple physical components as well as many CRDs that allow Kubernetes users to define network-level policies.

Two significant parts of Istio will be described further: traffic management and telemetry/observability.

5.1 Traffic Management Model

5.1.1 Gateway

Gateway resource is an L6 router that allows the definition of accepted ports, hosts, protocols, and TLS enforcement. Gateways directly use Kubernetes Ingress resources and bind to them to provide on-edge policy enforcement.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: my-gateway
spec:
  selector:
    app: istio-ingresscontroller
  servers:
    - port:
        number: 443
        name: http
        protocol: HTTP
      hosts:
        - "example.com"
      tls:
        mode: SIMPLE
        serverCertificate: /etc/certs/server.pem
        privateKey: /etc/certs/privatekey.pem
```

The code listing above defines an Istio Gateway called *my-gateway* that listens on an HTTP port 443, only accepting traffic from the host *example.com* and enforces the TLS

connection for the traffic. Istio uses this configuration to open the port and direct the inbound traffic to the set of rules within Virtual Services.

5.1.2 Virtual Service

Virtual Services are L7 routers running behind the on-edge Gateways. They provide an interface that allows Istio users to control HTTP/S traffic accepted by a defined set of gateways directed to a defined set of Kubernetes Services.

Virtual Service (VS) rules compute before the traffic reaches the Kubernetes Service, thus can redirect traffic to a different Service, add/remove headers, enforce Cross-Origin Resource Sharing (CORS), etc. Virtual Services are also configurable for traffic splitting either by weights (e.g., for the Canaries) or using HTTP headers and cookies (e.g., for A/B testing). One can also combine these, creating A/B tested canary environment.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: example-virtual-service
spec:
  gateways:
  - example-gateway
  hosts:
  - example.com
  http:
  - match:
    - uri:
        prefix: /example
    route:
    - destination:
        host: example-service
```

The code example above shows a Virtual Service used to route all traffic from the *example-gateway*. The rules apply to all traffic that is coming from the *example.com* host and the Virtual Service tries to match and direct traffic to a particular service: in this example, all traffic prefixed with */example* is being routed to the *example-service*.

5.1.3 Destination Rule

Destination Rule (DR) is the last piece in the traffic management model of Istio. DRs allow defining post-routing policies that enforce rules once the target of Virtual Service has been decided. Because the resulting policy of VS defines which Kubernetes Service should be targeted, DRs can apply further policies on load balancing, TLS settings or configuration for the connection pool. Furthermore, DRs may split the Kubernetes Service into multiple Subsets, which are then referenced separately (e.g., Service may have multiple versions, all defined under a single Kubernetes Service, while divided by subsets).

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: example-dr
```

```
spec:
  host: "example.com"
  trafficPolicy:
    loadBalancer:
      simple: LEAST_CONN
  subsets:
  - name: canary-version
    labels:
      version: canary
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
```

The Destination Rule example above defines a default rule for all services behind the *example.com* host. While all hosts are load balanced through the *LEAST_CONN* policy, the subsets may override these settings. The example shows a subset targeting workload labels *version=„canary“* which uses the *ROUND_ROBIN* load balancing strategy.

5.2 Envoy Sidecar Proxy

There are multiple configurations, such as Gateways, Virtual Services, or Destination Rules. However, all of these are only logical pieces of the infrastructure.

The main physical component is called the Envoy Proxy. Lyft - the taxi company is maintaining Envoy and drives it by the community as an open-source alternative to Nginx. Istio is configuring Envoy in the process of gathering all configurations, merging them and creating a single Envoy configuration, distributing it across the whole mesh.

These proxies distribute via a pattern called Sidecar. This pattern describes a container added to each Kubernetes Pod for proxying, telemetry collection or other general cases. Envoy, thus, are being distributed by the process of injection into every Pod running in the Istio mesh. All traffic is then proxied via Envoy that enforces all configured policies.

5.3 Telemetry

Istio comes with built-in telemetry mechanisms via its Envoy Proxies. All proxies generate several metrics that can be scraped by systems such as Prometheus or Stackdriver for further analysis.

Istio allows to visualize metrics through 3 main tools:

- **Prometheus** - a system used for metric collection and querying, that supports rich queries via its custom query language.
- **Kiali** - a network traffic visualizer, used to visualize Kubernetes and Istio resources in real-time.
- **Grafana** - a metric visualizer that queries Prometheus in real-time, displaying dashboard of metrics for all mesh services.

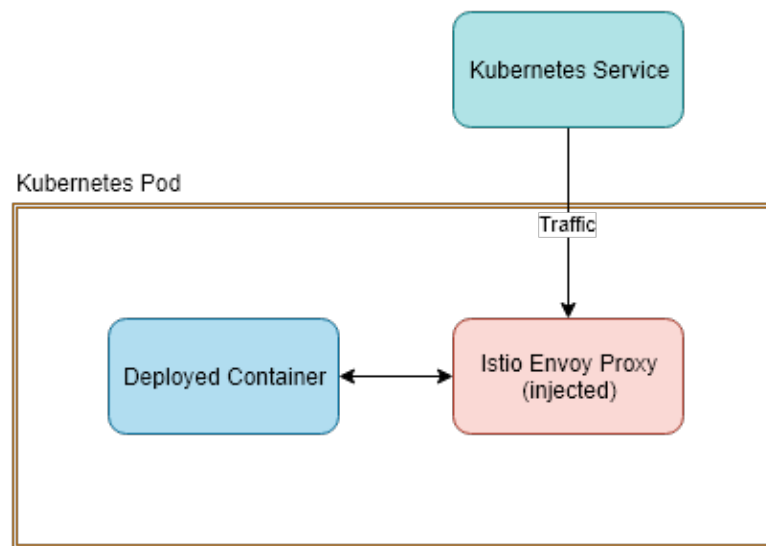


Figure 5.1: The Envoy Sidecar configuration injected into each Pod running within the Istio service mesh. The injector re-maps all ports of the original configuration to go through the proxy for policy enforcement.

5.3.1 Prometheus

Prometheus is a single-purpose metric scraping engine that allows pull-based metric scraping from many targets at once. It comes with a rich PromQL query language that allows users to query and operate on-fly on gathered metrics.

Prometheus is a core part of default Istio configuration as it provides its telemetry capabilities. Because it collects all metrics across the cluster and network mesh, they can then be used to further analyze behaviors of services, such as error rates, cumulative CPU or status codes (and many more) in real time.

A full chapter is dedicated to Prometheus, as it's heavily used further in the implementation.

5.3.2 Kiali

Kiali provides insights through service and workload graphs, automatically visualizing connections between services inside the Istio service mesh. It also displays communication properties, such as throughput, security, or error rates. Kiali dashboards are updated in real-time, making it an excellent companion when debugging or for general monitoring and observability purposes.

There are four main types of graphs available in Kiali:

- **App graph** creates a simple visualization of running applications, aggregating all versions into a single node in a graph.
- **Service graph** visualizes only service resources within a namespace without visualizing any particular workloads or versions.
- **Versioned app graph** creates a separate node for each version of an application, while also visualizing all versions as a part of a single stack (figure 5.3).

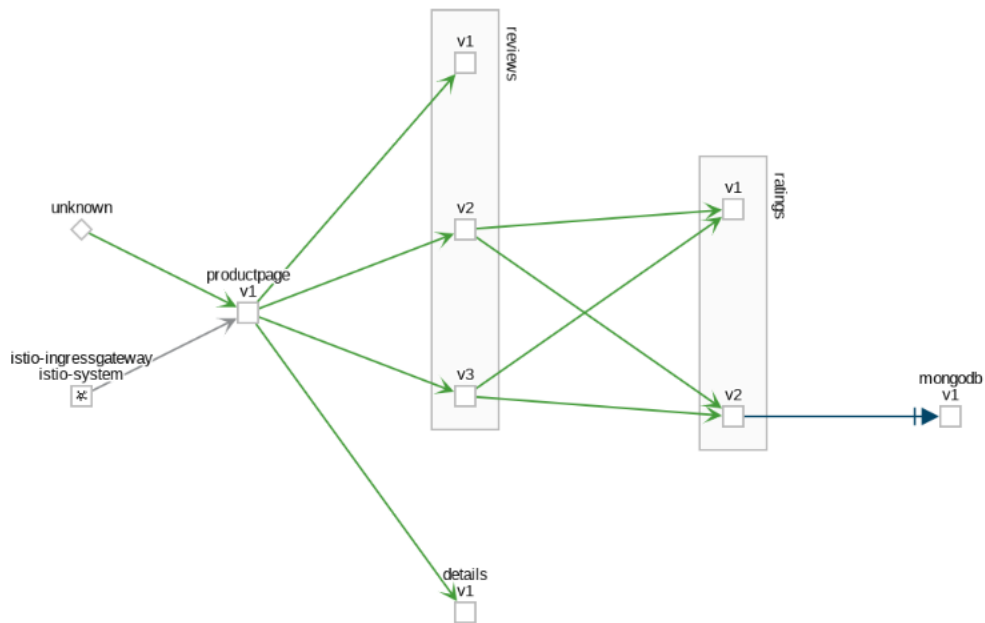


Figure 5.2: A sample Kiali graph visualizing a versioned app graph with workloads for the *productpage*, *details*, *reviews* and *ratings* applications with their versions plus a MongoDB stateful application. The unknown traffic commonly comes from sources not residing in the Istio service mesh such as Kubernetes health checking systems.

- **Workload graph** creates a separate node for each workload without grouping versions or apps into stacks.

5.3.3 Grafana

Grafana is another from the set of open-source tooling available in the Istio package. It provides a -real-time visualization of metrics and allows the creation of full dashboards. The graphs can be directed to fetch data needed for their rendering through custom sources such as PostgreSQL or Graphite, as well as more common sources used by the Istio like Prometheus.

There are multiple dashboards for workloads or services running in the cluster already set up, as well as dashboards for all control components of Istio such as Mixer or Pilot. Default dashboards allow beautiful out of the box observability right from the start. Users can edit or create new dashboards in Grafana as needed.



Figure 5.3: A sample from the Istio Grafana workloads dashboards. The out of the box dashboard shows an average number of incoming requests as well as their response codes. Other graphs show duration, requests size, or response size percentiles. Dashboards like this one can be easily used to set up the first monitoring environment for the Istio service mesh.

Chapter 6

Prometheus

Prometheus is an open-source monitoring and alerting solution that allows collection and storage of metrics, visualizing via graphs, and alerting based on the collected metrics. Because there are plenty of different monitoring systems already in-place, Prometheus also focuses on creating exporters of metrics from already existing systems. It also implements a custom rich query language called PromQL, that allows execution of complex expressions with multiple queries. [3]

6.1 Metrics Collection

Prometheus uses a pull-based metric collection to scrape data from its targets. The pull-based scraping allows users to configure collectors on the Prometheus side, without needing to affect running services in any way. Moreover, Prometheus can seamlessly scale its scrapers as a load of scrapers is predictable based on many targets, metrics in each target and period.

All data scraped by Prometheus are time series. Each data point then has its timestamp, along with labels. Labels allow Prometheus to store and query data in multiple different dimensions (e.g., app=web-server, version=v1/v2/... where version is the next dimension). Labels are inclusive. Thus data with the same minimal set of labels always belongs to a single dimension. PromQL then builds on this abstraction of labels to enable rich queries. It uses three main metric types for metric collection:

- **Counter** is used for cumulative calculations of given metrics (e.g., cumulative CPU time).
- **Gauge** is used to represent a single numerical value that rises or goes down (e.g., memory usage).
- **Histogram** is used as a sampler of metric values into multiple buckets (e.g., number of requests with latency 0-100ms, 101-200ms, etc.).

6.2 PromQL

PromQL is an open-source query language specifically designed to work with Prometheus. It comes with a powerful expression evaluation engine that allows the building of simple queries as well as composite queries containing multiple expressions. This grants access to metrics for other tools leveraging data from Prometheus exporters.

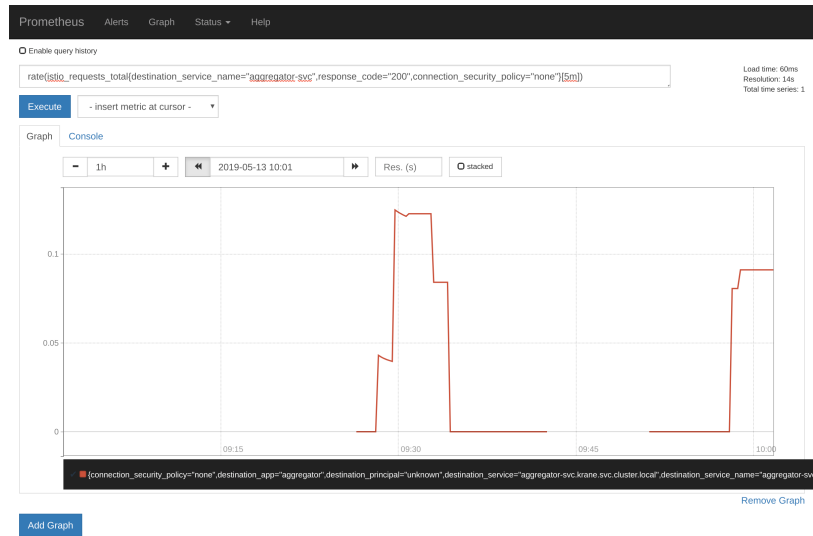


Figure 6.1: The graph visualization of prometheus query used to get all requests that passed with a status code of 200 prorated to an average per second.

In terms of canary deployments, both, canary and baseline deployments should be evaluated on user-defined metrics such as CPU or number of erroneous requests. PromQL enables this with simple metric queries that are evaluable for thresholds, such as the one below:

```
istio_requests_total{
  destination_service=~"application-canary.*", response_code="5xx"
}[5m]
```

The query above returns a range vector of total requests through the last 5 minutes with the response code of 5xx (range of 500 HTTP errors). However, because this is a cumulative metric, it is not recommended to be used for threshold such as a maximum number of errors. PromQL has several utility functions that can be used to normalize such cumulative metrics to per-second average values like the `rate()` function. The example below uses the `rate()` function to calculate the average number of 5xx requests per second. The graphed result of such a query can be seen on the figure 6.1.

```
rate(
  istio_requests_total{
    destination_service=~"application-canary.*",
    response_code="5xx"
  }[5m]
)
```

The example above creates a new range vector that can be used to evaluate the average error rate from the `application-canary.*` services. Moreover, when assessing such metrics for the canary deployment, there might be a maximum target value for such metric, which fails the canary test if the metric reaches the value (e.g., the canary analysis may state, that the maximum number of errors can't go above 20 per second). PromQL allows the creation of such queries, that return simple scalar values such as maximum from the range vector. A

function called *max()* can be called on the above query to determine the maximum average value, which can then be compared to the desired amount:

```
max(
  rate(
    istio_requests_total{
      destination_service=~"application-canary.*",
      response_code="5xx"
    }[5m]
  )
)
```

Nonetheless, differences relative to both, canary and baseline deployments are commonly used to determine the health of canaries. In such cases, PromQL supports composite queries are used to create a difference of a metric between two services through arithmetic operators. Queries such as the one below can then be used to determine differences between errors rates of the canary. The query then returns a single value, which is a maximum difference between rates of canary and baseline for the given metric (HTTP response codes in this case).

```
max(
  rate(
    istio_requests_total{
      destination_service=~"application-canary.*",
      response_code="5xx"
    }[5m]
  )
-
  rate(
    istio_requests_total{
      destination_service=~"application-baseline.*",
      response_code="5xx"
    }[5m]
  )
)
```

Because the Prometheus team fully maintains PromQL, it also has support in the Prometheus UI, which can be used to graph queries for manual testing of queries or alert setup.

Chapter 7

Design

This chapter will focus on the design of the canary controller called Krane (pronounced Crane). Agile and DevOps come with a good guideline on choosing goals for automation and reducing toil (manual work that could be automated). Many tools covering broad topics of CI are already on the market, while there seems to be a small amount of tooling covering the CD process.

The design of the project has multiple high-level goals:

- **Kubernetes-native** experience, using Kubernetes Custom Definitions and configuration metadata to allow standard tooling, such as `kubectl` to work with Krane.
- **Reliable** deployment via Helm (package manager commonly used in the Kubernetes environment).
- **Stateless runtime**, that stores all configurations and results within the cluster, making it reliable and independent of another tooling.
- **Programmable, yet configurable**, as the system should expose a set of APIs to easily define common patterns while allowing developers to use programming language they like to configure specialized checks.
- **Reactive**, so other systems may watch for results published by Krane, and users can use other systems to trigger Krane, e.g. from CLI, or by an HTTP or gRPC call. The reactivity allows other CI systems to call Krane after they did their work.
- **Pluggable** networking and metric collector cores, so even though Krane will come with Istio and Prometheus enabled by default, one can extend this by re-implementing the interfaces.

7.1 Runtime Platform

Krane makes use of multiple systems for container orchestration, network policy enforcement, metrics collection, and metrics evaluation. These form a set of dependencies that need to be present for the Krane to run without problems:

- **Kubernetes** is used as a container orchestrator and scheduler that provides a declarative API for component configuration. It allows workload scheduling and internal exposure.

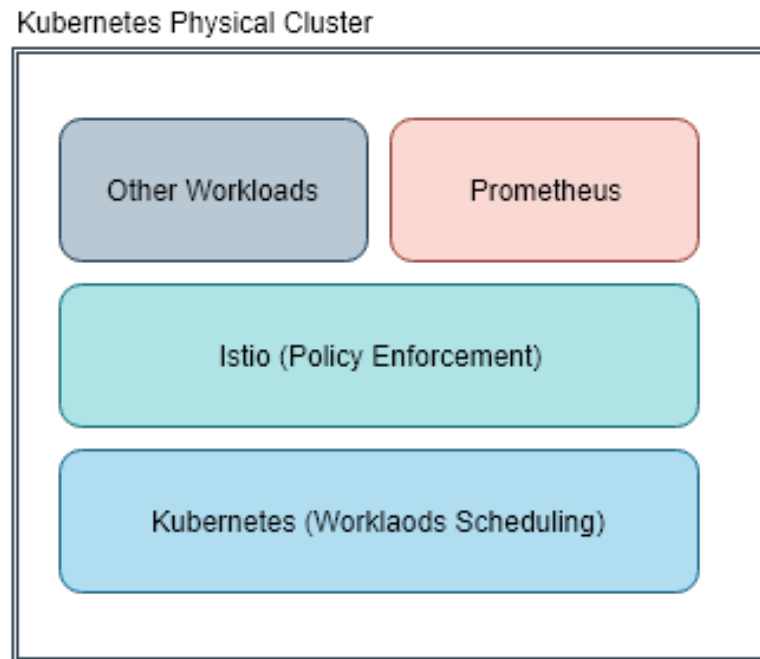


Figure 7.1: The infrastructure of Kubernetes, Istio, and Prometheus that the Krane Operator needs to run on. All workloads are scheduled via Kubernetes, but Istio injects proxies in the process. Prometheus deployment is then done via Istio installation as a standard workload.

- **Istio** is a service mesh used for policy enforcement and advanced routing mechanisms such as L7 traffic splitting, failover mechanisms such as retries or fail injection for testing purposes.
- **Prometheus** is a service used for metric collection and evaluation that scrapes all workloads deployed to Kubernetes via Istio service mesh.

7.2 Components of Canary Setup

Krane consists of 5 components:

- **Krane Operator**, the only always-on component is running within the cluster where its operations are made. The operator is responsible for the creation and maintenance of canary deployments and the reactivity of the system.
- **Krane Policy Configuration** is a CRD component that defines a set of policies that should be applied to a particular set of deployed workloads containing the policy label.
- **Canary Configuration** is a CRD that defines the real-time state of running canaries inside the cluster. The operator automatically updates the resource.

- **Test Job** is a replicable component configured to perform a set of tests on the target service. Users can replace the component by a custom pod complying with the interface definition of the Test Job.
- **Judge** is a replicable component configured to perform checks on pre-defined metrics. Users can replace the component as in case of Test Job, by complying with the Judge interface.

The diagram 7.2 shows a high-level view of components contained in the Krane system. Full lines represent the system without an interception. However, when a Krane Resource is submitted, the Operator spawns the Canary version and Baseline along with the Test Job and the Judge pods. It then re-configures the service to split traffic between the Production Version and the Canary Version after the tests and test judging are successful.

7.2.1 Krane Operator

The Krane Operator is a core part of the system. It provides an always-on service that creates, monitors and updates Canary Resources.

The Operator automatically watches for labels inside newly created Deployment resources in a given namespace. After a label with the *krane.sh/canary-policy* has been detected in a deployment, the Operator automatically starts processing this deployment as a deployment. Detection of these labels brings seamless integration with other existing systems, as no other changes than adding a label are needed.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aggregator-deployment
  namespace: krane
  labels:
    app: aggregator
    version: stable
    krane.sh/tier: stable
    krane.sh/canary-policy: aggregator-policy # <-- The Canary Policy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: aggregator
      version: stable
      krane.sh/tier: stable
  template:
    metadata:
      labels:
        app: aggregator
        version: stable
        krane.sh/tier: stable
    spec:
      containers:
        - name: aggregator
```

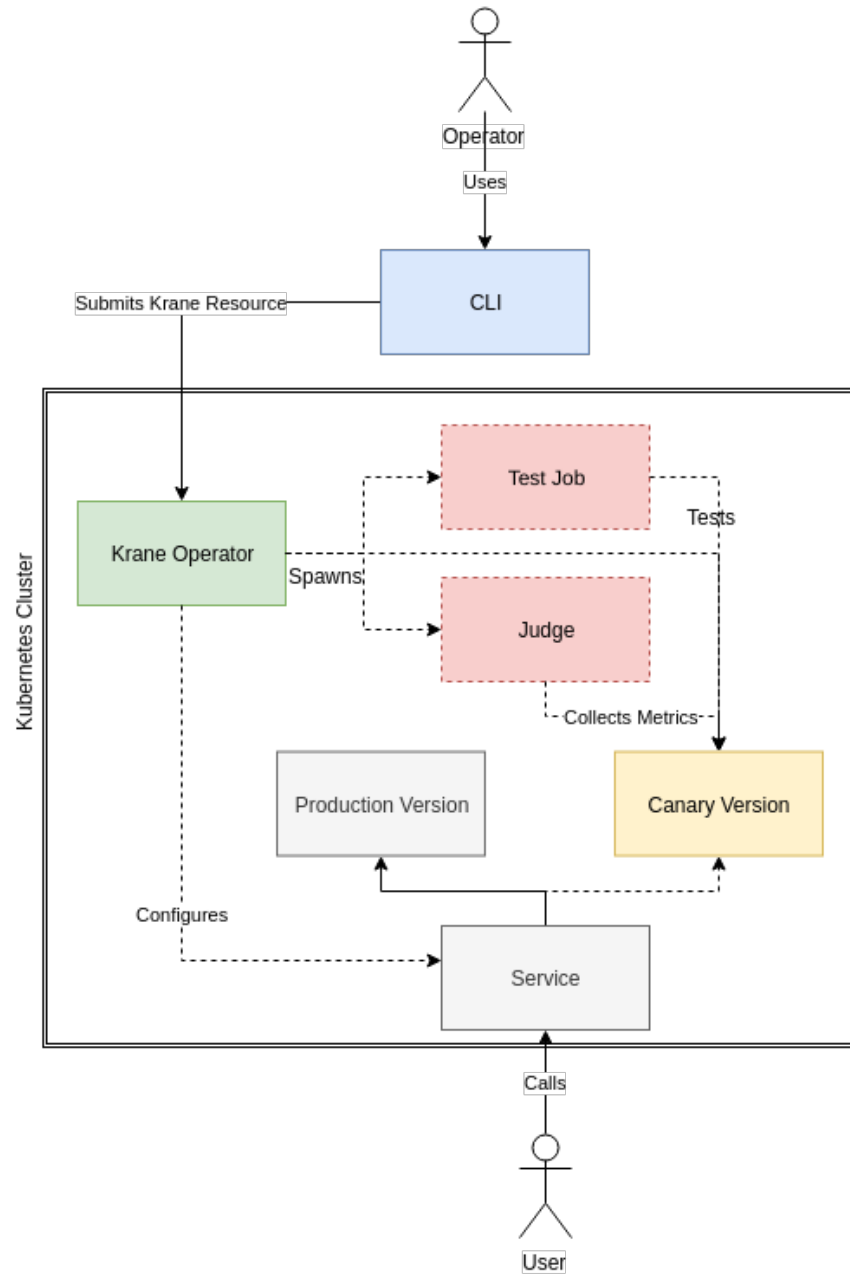


Figure 7.2: A high-level architecture of the system, showing five main components and their connections. The Service and the Production Version are part of the stable system before the Krane system deployment.


```

image: gcr.io/petomalina/aggregator:v1
ports:
  - containerPort: 80
    name: http

```

The code listing above defines a single-replica deployment with the *aggregator-policy* set as the Canary Policy Resource described in the next section. The operator ignores all deployments without this policy that are not part of the broader Canary configuration (also described later).

The Operator makes use of the standard Kubernetes API using API Machinery. It performs iterative checks on Canary Resources that were submitted by Kubernetes users. The iterative checking makes it possible to watch for changes in Canary Resources as well as their children - Test Job, Judge and the Canary Version of the application. Once an observed change occurs, subsequent tasks are performed, and this change is reported into the Kubernetes event hub outside of the Krane system.

Krane Operator has an internal pipeline that executes every time a new Canary Resource is created. This pipeline consists of 5 linear stages and one parallel:

- **Initializing** stage prepares the canary deployment and creates the baseline deployment with a single service directing traffic to both via round-robin.
- **Testing** stage creates the Test Job pod and configures it to target the newly created Canary Version. The Test Job is then left to perform integration and end-to-end testing.
- **Canary** stage configures the Service to split traffic between the production workload and the canary workload, while Judge is collecting metrics.
- **Reporting** stage reports to a configured reporter after testing, and canary stages have passed. Reporting allows third-party systems such as Slack or CI/CD to get notified of finished canaries.
- **Cleanup** stage occurs once all work on the canary deployment is done and everything except metadata can be cleaned up. This stage removes all workloads, services and rolls back the virtual service modification to the original state.
- **Parallel Judging** is a stage that occurs in parallel with Testing and Canary stages. It continuously checks for metric changes and reports failure in case defined metrics rules are broken.

Every stage has its pipeline that can also remain in multiple states, based on the status of reconciliation. The statuses are:

- **Queued** is an intermediary state used to trigger reconciliation loop of other controllers, generally at the end of the previous stage.
- **In Progress** is a state describing active work of components used in the stage. For example, having Testing stage In Progress means Test Job and Judge were successfully deployed and are executing their jobs.
- **Success** indicates that the stage is complete and succeeded. Thus other stages may continue.

- **failure** indicates that the stage is complete and failed. Thus other stages should not continue unless they are the Cleanup stage.

7.2.2 Canary Policy Resource

Canary Policy Resource is a CRD that allows users to configure policies for executing the canary deployments that target a specific policy (krane.sh/canary-policy link). The Operator then uses a particular policy to bootstrap the canary process as well as maintain and evaluate it.

The input configuration consists of multiple fields required for the canary deployment to be bootstrapped and evaluated:

- **Name** of the canary policy that is uniquely identifies the policy.
- **Base** defines the initial deployment that the policy should use for Canary deployment. The Base should always a deployment that will be used to create baseline deployment.
- **VirtualService** defines a virtual service resource that should be used to split traffic between the base deployment and the canary setup.
- **Service** is a name of the specific Service within the VirtualService configuration that is used for traffic splitting. The Service is needed as some virtual services may be used to navigate traffic from/to multiple hosts or versions.
- **BaselineMode** of the configuration that defines which canary configuration should be used (baseline, canary). Defaults to baseline.
- **Lifetime** configures if the resource should be temporary or continuous, thus remain running in the cluster even after tests were successfully completed. The continuous mode can be useful when benchmarking multiple versions of the canary, allowing operators to modify-and-run existing canaries.
- **Ports** is a set of ports that are exposed by the newly created canary deployment. This configuration can be used in case there are changes to the ports in the new canary resource. If unused, ports will be copied from the baseline deployment.
- **TestJob** provides a set of fields that configure the Test Job pod. The configuration will be described later in the Test Job section.
- **Judge** provides a set of fields to configure the Judge component. The configuration will also be further described in the Judge section.

7.2.3 Canary Resource

The Canary Resource is a fully-managed resource created and maintained by the Krane Operator. The Operator creates the Canary Resource every time the *krane.shpolicy* label is detected in the deployment. All updates to underlying resources as well as the progress of the canary deployment itself are then reflected in the configuration to allow users and other systems read and act upon canary status changes.

The output configuration has multiple fields, where all six stages have a submodel of status and detail:

- **Current Stage** which acts as an enumerable value of 5 stages described above.

- **Initialization** defines the initialization stage of setting up canary, baseline, and their exposure to the cluster.
- **Testing** represents the testing stage of initializing Test and Judge jobs as well as performing tests.
- **Canary** defines the stage where traffic is being split from the target Virtual Service (production) and routed into the canary setup.
- **Reporting** establishes the step of reporting the results back to a defined system.
- **Cleanup** defines the stage of cleaning up all workloads, resources and modifying virtual Service back to its original state. The only remaining resource then becomes the canary configuration with all statuses.
- **Judging** defines the parallel stage of metric judging process.

7.2.4 Test Job

The Test Job is a custom service that complies with the Test Job input interface. It is configured using environment variables which are injected to the pod by Kubernetes before it starts. Exiting the Test Job with an exit code other than zero fails the pipeline while preserving logs using the Kubernetes logs exporter.

Following environment variables are passed into the pod by the operator:

- **KRANE_TARGET** that contains FQDN (Fully Qualified Domain Name) or an IP and port of the test target.
- **KRANE_BOUNDARY_REQUESTS** is a number of maximum requests that the target Test Job can do. The service should comply with this number and exit once it reaches the number of desired requests.
- **KRANE_BOUNDARY_TIME** is a string time in a Golang Duration parse format (e.g., 5s means five seconds, 1h15m30s means an hour, fifteen minutes and thirty seconds, etc.). Boundary time is a complementary value to the requests. Service should always comply with both if they are defined, making these values upper boundaries of requests and time.
- **KRANE_BOUNDARY_RPS** sets a limit for how many requests can be performed per second by the controller. This value is set to 0 by default, meaning unlimited.
- **KRANE_IGNORE_FAILS** will fail the test on first request failure if set to false. Otherwise, tests will continue to run, and it's up to Judge to evaluate the metrics.

Istio implementation proxies all requests to automatically determine which requests failed and succeeded, reporting them to the metrics collector. No other implementation is required, but custom implementations may use metrics endpoint to allow Prometheus scrape custom Test Job results for judging.

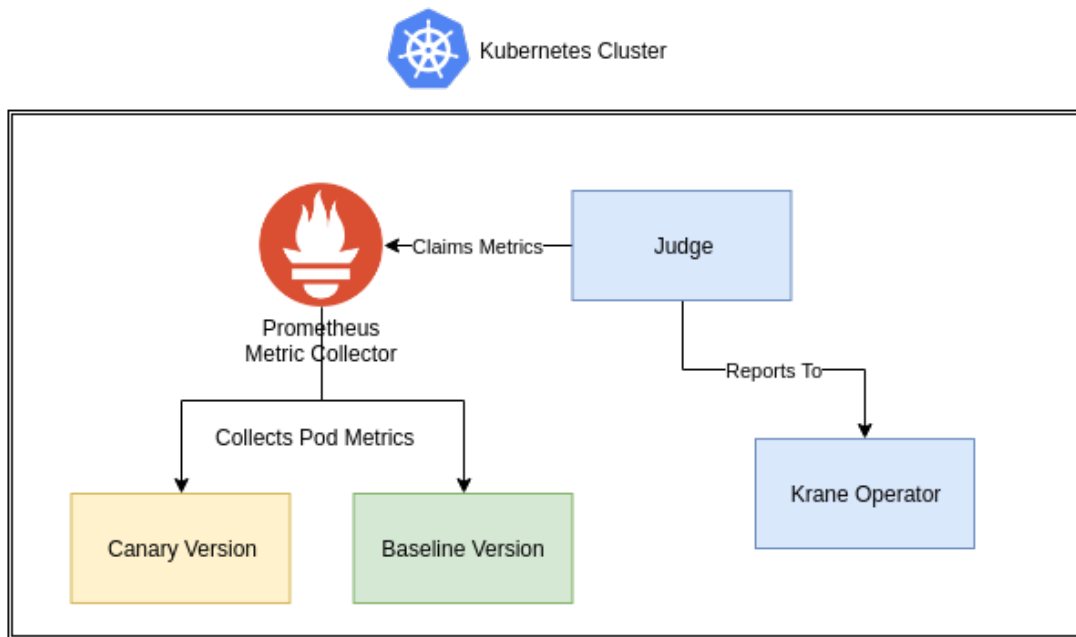


Figure 7.3: A closer look at the metric creation and claiming. Two canary services generate metrics that are collected by the Prometheus collector, which is, in turn, queried by the Judge to claim and evaluate the metrics. The Judge then reports these evaluated metrics back to the KO.

7.2.5 Judge

Judge is the component responsible for metric evaluation. The component accepts a set of metrics and watches them for both, Baseline and the Canary version deployments.

An example Judge configuration in YAML can be defined as:

```

judge:
  image: gcr.io/petomalina/aggregator-judge:v1
  boundary:
    time: 10m
  diffMetrics:
    - metric: container_cpu_system_seconds_total
      container: aggregator
      diff: 0.05

```

This example configures the Judge job to watch the cumulative CPU metric and create a diff between canary and baseline deployments. It establishes a regular CPU check with an accepted difference of 0.05 CPU seconds. If the metric fails to comply with the accepted difference during multiple scraping cycles of Prometheus, the metric is considered to be failed, reporting the whole Judge job as failed.

7.3 Using from CLI

Design of the Krane Operator makes use of the standard Kubernetes CLI tooling via a tool called `kubectl`. All resources, including the CRDs, can be queried and watched through the `kubectl`, which makes all Krane components easily accessible. All CRDs support singular/plural names that allow listing and querying single resources with their full definitions.

```
peter_malina@cloudshell:~ kubectl get canaries
```

```
my-app-aggregator-deployment-ap0cv3-canary 8d
my-app-aggregator-deployment-ah87p9-canary 8d
my-app-aggregator-deployment-cdefgy-canary 5d
my-app-aggregator-deployment-qjk52b-canary 4d
my-app-aggregator-deployment-1l2e9a-canary 4d
my-app-aggregator-deployment-j89azh-canary 4d
```

The code example above shows a listing of all canary configurations available in the default namespace within a currently configured cluster. The `kubectl get` command provides a simple API to get any resource (even CRD) from the Kubernetes cluster. Users may then use one of the listed names to query details of the necessary resource.

A detailed view can be seen in the example below, where the canary configuration named `aggregator-policy-aggregator-deployment-j89azh-canary` is being queried directly with its full YAML output. The metadata object is being stripped as it does not contain relevant data (thus three dots). The example shows a successful canary configuration that passed through all of its stages - initialization, testing, canary, reporting and cleanup as well as the parallel judging. The `status` subobject also contains `podName` for judging and testing sections of the configuration. This name field contains a name of the pod for logging and metrics querying (a user may want to get logs and metrics from these pods even though they were already cleaned up by the successful exit of the entire pipeline).

```
peter_malina@cloudshell:~ kubectl get canary \
    aggregator-policy-aggregator-deployment-j89azh-canary -o yaml
```

```
apiVersion: krane.sh/v1
kind: Canary
metadata:
  name: aggregator-policy-aggregator-deployment-j89azh-canary
  ...
spec:
  deployments:
    base: aggregator-deployment
    baseline: aggregator-deployment-j89azh-baseline
    canary: aggregator-deployment-j89azh-canary
    policy: aggregator-policy
status:
  canary:
    message: Rerouting Configuration In Progress
    status: success
  cleanup:
    message: Rerouting Configuration In Progress
```

```
    status: success
initialization:
  message: Initialized baseline and canary deployments
  status: success
judging:
  message: Job finished successfully
  podName: aggregator-policy-aggregator-deployment-j89azh-canary-judge
  status: success
progress: cleanup
reporting:
  message: Reported
  status: success
testing:
  message: Job finished successfully
  podName: aggregator-policy-aggregator-deployment-j89azh-canary-testjob
  status: success
```

Chapter 8

Implementation

This chapter provides a closer look at implementation details of the Krane Operator and its internal composition of controllers and reconcilers. The implementation is physically divided into two controllers. A controller is a unit responsible for a single resource (or CRD), and its children subtree, triggering reconciliation cycles based on changes to the resource and its children. The primary reconciliation cycle divides into multiple reconcilers responsible for each child in the subtree.

8.1 Manipulating Kubernetes and Istio Resources

Kubernetes is a system exposing declarative API via YAML/JSON configurations. The declarative approach lies in the ability to describe the final state of a particular component instead of defining a set of imperative steps. The same configuration options are then exposed via Kubernetes REST API that also accepts Protocol Buffers serialization format.

Even though both, Kubernetes and Istio are projects close to each other, they both use a slightly different approach. Kubernetes is fully prepared for use in external systems - all types are fully defined and support OpenAPIv3 specification. The implementation allows seamless usage of all Kubernetes types such as Deployments or Services. However, Istio defines all its types like so:

```
type VirtualService struct {
    meta_v1.TypeMeta 'json:',inline"
    meta_v1.ObjectMeta 'json:"metadata"
    Spec map[string]interface{} 'json:"spec"
}
```

This specification, however, is the minimal configuration for any CRD, as the Spec field defines „map of anything (also more maps)“. Nonetheless, this type has a correct schema and thus, can be registered without problems. The type safety is lost when using such type though.

Istio uses one more implementation internally - an implementation that operates on Protocol Buffers serialization instead. The problem with this implementation is, Operator SDK (still alpha in the time of writing) does not have any way to override the default serialization mechanism. Because the Protocol Buffers *jsonpb* uses a different serialization technique for enumerable values, correctly decoded values from JSON may end up being encoded in a different format with no internal change to the serialized data.

As a result, all types from the Istio *v1alpha3* API with backward compatible fields are copied to the resulting project, and the serialization format is corrected for the enumeration values.

8.2 Deployments Reconciliation Controller

The Deployments Reconciliation Controller is responsible for the Initialization stage and watching of Deployment resources such as the one listed in the Krane Operator subsection. Every time a new deployment with *krane.sh/canary-policy* label is created or updated, the controller catches the triggered event and starts the reconciliation cycle.

The reconciliation cycle of the Deployment Reconciliation Controller is responsible for the creation of the Canary configuration resource and ends with setting the Success or Failure flags on the Initialization stage that passes control of the Canary resource to the Canary Reconciliation Controller. All reconcilers are passed the Canary Policy that is being fetched at the start of each reconciliation cycle.

8.2.1 Deployments Predicate

Before any event reaches the reconciliation cycle, the Deployments Predicate is executed to determine if the Deployments Reconciliation Controller should handle the event. This predicate allows filtering based on all four events delivered by the Operator SDK: Create, Update, Delete, and Generic.

There are the following rules applied to each event:

- **Create** events are filtered based on the existing *krane.sh/canary-policy* label and *canary* tier inside the *krane.sh/tier* label as policy may be set for baseline or stable tiers as well.
- **Delete** events are filtered out, as no deletion events should be triggered and deleting a resource may trigger fail of the whole pipeline, and thus cleanup.
- **Update** events must comply with the same rules as the **Create** events.
- **Generic** events must comply with the same rules as the **Create** events as well.

8.2.2 Canary Reconciler

The Canary Reconciler is responsible for updating the Canary Deployment and registering it with the Canary configuration resource. It automatically adds missing labels for the Canary configuration and directly updates the deployment. This reconciliation cycle may run and fail multiple times during the internal bootstrap phase of the deployment, as deployment configuration continuously updates until it stabilizes with some number of available replicas.

8.2.3 Baseline Reconciler

Baseline Reconciler is responsible for building the Baseline Deployment from the existing Base configured within a Canary Policy. Baseline instances are either built from the current production version of the deployment or the newly created canary deployment. The decision bases on the configuration of BaselineMode from the CanaryPolicy:

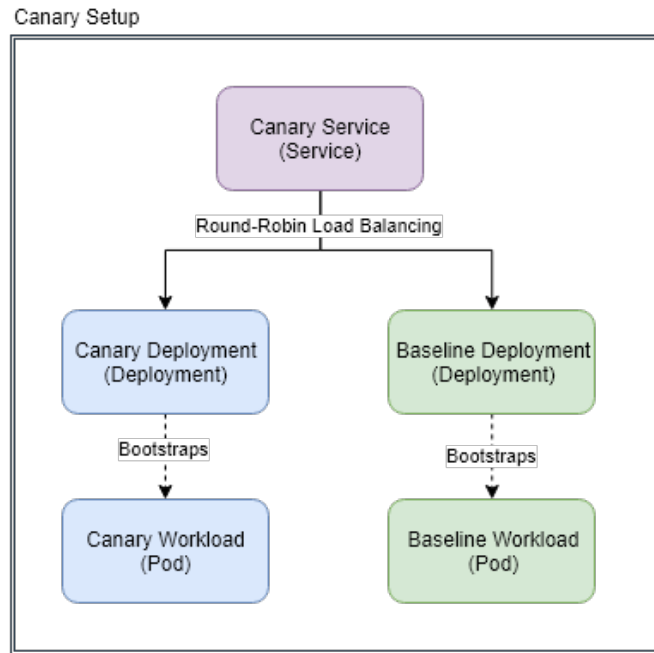


Figure 8.1: Final Canary Setup after all workloads have been deployed and connected with the Kubernetes Service.

- **New** directs Baseline Reconciler to use the Canary Deployment configuration for the Baseline Deployment. This configuration is mainly usable during changes to the deployment configurations, for example, when there is a change in environment variables, secrets, or exposed ports.
- **Old** directs the reconciler to use the Stable Deployment configuration for the Baseline Deployment. *Old* is the default configuration used when there are no changes to the deployment in the canary or in case the baseline should also test backward-compatible changes to the environment.

The reconciler then updates all labels of the newly created Baseline Deployment and binds the Baseline Deployment to the Canary Deployment (making it a garbage-collected child whenever the Canary Deployment is deleted). The Baseline deployments also carry all information needed to find any other components of the canary setup as it brings the Canary Policy and Canary configuration names in its labels.

8.2.4 Service Reconciler

Lastly, the Service Reconciler creates a Kubernetes Service (DNS configuration) that can be used to create round-robin load-balancing rule between the created Baseline and Canary deployments. The service uses the Canary Policy and Canary configuration names to determine which deployments to include in its DNS configuration. The created service is then also bound to the Canary Deployment, applying the same logic for garbage collection as the Baseline Deployment. The final setup can be visualized on figure 8.1.

8.3 Canary Reconciliation Controller

Canary Reconciliation Controller is a unit responsible for taking care of the Canary configuration once its Initialization stage has reached success or failure. The controller assumes that all prerequisites of canary setup are already met, continuing with the reconciliation of Test and Canary jobs until it reaches the final state of Cleanup.

Both Test and Judge reconcilers are using the availability status from Kubernetes to check if the job has finished:

- **Waiting** means the containers are initializing.
- **Running** means the containers are currently running, and another status change will be available once they finish.
- **Terminated** reports if the container completed with an exit code (needs to be determined).

Because all containers are running within the Istio service mesh, Istio injects Envoy Proxy to all deployments that will run until exited by an external process. Because Istio Proxies are running until the whole Pod finishes, the reconciler can't wait until they exit (because they won't). A filter applies for all Istio proxies to avoid matching statuses with their containers.

8.3.1 Test Reconciler

The Test Reconciler bootstraps and maintains the Test Job. It creates the job as a single-replica Pod with a predefined image configured in the Canary Policy. The Test Job binds as a child of the Canary configuration. The reconciler passes all environment variables described in the Design to the newly created container in the Pod.

8.3.2 Judge Reconciler

Judge Reconciler applies the same number of steps as the Test Reconciler. However, Judge gets all configurations from the Canary Policy to its configuration via environment variables. The Judge Pod bootstrapper then assumes the container connects to the Prometheus API via given *KRANE_PROMETHEUS* DNS name. The container is otherwise free to use any other resources of the cluster if it can access them via internal RBAC policies.

All metrics are being collected via Prometheus and continuously queried by the container. The reconciler expects the container to report either 0 exit code after all stages are done or other exit code in case of a failure.

The container should be able to consume and execute the defined Prometheus queries. Any other functionality is then left on Krane users to implement. However, the container must always obey time boundaries set by the *KRANE_BOUNDARY_TIME* environment, otherwise can be killed unexpectedly.

8.3.3 Virtual Service Reconciler

Virtual Services are being managed by Istio users (such as operators) instead of Krane. Nonetheless, Krane needs to mutate these configurations to allow the Canary setup to accept production traffic after all tests have passed.

The reconciler is only applied once Initialization and Test stages have succeeded.

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: aggregator-vs # <- VirtualService from the Canary Policy
spec:
  http:
    route:
      - destination:
          host: aggregator-svc # <- Service from the Canary Policy
          subset: stable
          weight: 90 # <- weight split of the production traffic
      - destination # <- new destination rule
          host: aggregator-svc-01hz2-canary # <- newly created Canary Service
          weight: 10 # <- weight split taken from the production traffic

```

8.3.4 Reporting Reconciler

Reporting makes it possible to notify other systems of successful or failed canaries, such as continuous integration or VCS (Github, Bitbucket, etc.). The reporting reconciler acts upon the reporter configuration of the executed Canary Policy. Any failed reports will result in failing the whole Reporting stage of the Canary, but won't be repeated (users can see the failure in Canary status).

8.3.5 Cleanup Reconciler

There are multiple components deployed and configured to form the Canary Setup. While they may stay unchanged in case a user needs to debug them, the default strategy is to clean up all components. There are two main cleanup points:

- **Workloads and Services** are cleaned up by deleting the Canary Deployment, Test Job, and Judge. Because the Baseline Deployment and the Canary Service is bound to the Canary Deployment, they are garbage-collected automatically.
- **Traffic Routing** is cleaned by reverting changes to the target Virtual Service. However, because there may have been multiple changes made to the Virtual Service (e.g., from other canaries or by the user), the Operator maintains the target host in its configuration to delete it and move the traffic split from it back to the production destination.

After reconciler cleans up, there is only the Canary config (and unmanaged Canary Policy) left in the cluster. The cleanup process is exceptionally needed for Test Job and Judge, as both contain the Istio Envoy Proxy that will run forever until explicitly killed. The component tree can be seen in figure [8.2](#)

8.3.6 Operator Access Rights

Global RBAC policies system secures all Kubernetes APIs. The system consists of Roles, Role Bindings and Service Accounts. A role defines a list of rules in the form of *<api-Group>:<resource>:<verb>*, e.g. *apps.deployments.update* that defines the right to update Kubernetes Deployments. The Service Account resource defines a name which is then

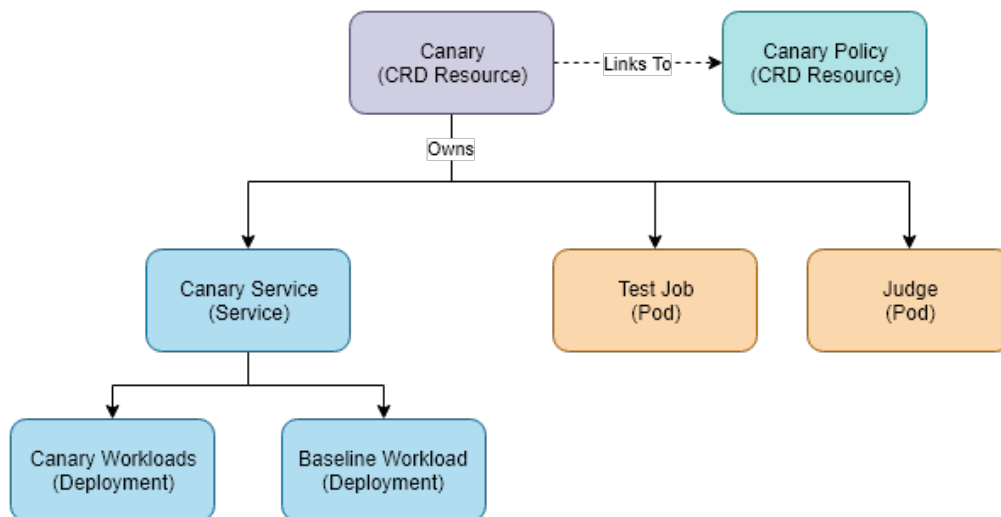


Figure 8.2: The Kubernetes component tree dependencies. Deleting any node will also trigger garbage collection of its dependencies, e.g., deleting the Canary resource will subsequently delete all components within this tree.

used as a reference in workloads such as Deployments to determine what access rights the workloads have.

Role Bindings then define subjects and role references, that are used to glue together a set of rules and the Service Account reference. Updating the Role bound to a Service Account will then automatically update all access rights for workloads using the Service Account.

Krane needs access to the following resources:

- **Pods and Services** from the *core* API to create the Canary Service as well as the Test Job and the Judge.
- **Deployments** from the *apps* API to bootstrap and modify the Canary and Baseline deployments.
- **VirtualServices** from the *networking.istio.io* API to modify the target Virtual Service for traffic split.
- **Canaries and CanaryPolicies** from the *krane.sh* API as these are the core components of Krane.

8.3.7 Packaging with Helm

Helm is a package manager specifically created for the Kubernetes ecosystem. It allows packaging of Kubernetes templates and supports Golang template engine for an ability to replace values inside templates with pre-defined user values. There are six resources deployed with the Operator package:

- **Role, Role Binding, and Service Account** described in the previous section.
- **Canary CRD** that defines a CRD for the Canary configuration resource.

- **Canary Policy CRD** that defines a CRD for the Canary Policy resource.
- **Operator Deployment** that defines the Deployment resource for the operator itself. It binds the service account created in the previous templates to the Deployment, allowing it to use the internal Kubernetes API.

Multiple values are then configurable via *values.yaml* before deployment, such as operator version, reporter webhook or internal path to the Prometheus service:

```
operator:
  version: "v1"
  resources:
    limits:
      memory: 128Mi
      cpu: 500m
    requests:
      memory: 64Mi
      cpu: 250m
reporter:
  hook: https://example.com/notify
prometheus:
  host: prometheus.istio-system.svc.cluster.local
```

8.3.8 Development and Deployment with Skaffold

The development cycle and the deployment process of an application built on Kubernetes tend to be quite different from deploying a binary via SSH. It consists of multiple steps:

- **Building Container Image** is the first step that takes a Dockerfile or other supported open format and creates file system layers based on the commands executed by the user within the Dockerfile. All layers can be cached; however, change to a previous layer will trigger the build of all succeeding layers (e.g., changing dependencies will trigger their download from scratch).
- **Pushing Container Image**, as the image must be in an accessible place for the deployed environment to access - it's common to use Docker Hub or another public repository provider for open-source projects.
- **Templating the Deployment Configuration** via helm and updating the image tags with the newly created and pushed tags takes place next, and configurations apply to a cluster via the Kubernetes configuration API.
- **Applying the Final Configuration** is the last step that happens after the templating. The templating submits the configurations via the configuration API, while Kubernetes calculates all steps to get to the new final states.

The configuration below specifies the *skaffold.yaml* used to configure Skaffold to build the Dockerfile within the context of the repository root and then apply the helm configurations that reside in the *deployment/krane* directory:

```
apiVersion: skaffold/v1beta8
kind: Config
build:
  artifacts:
    - image: gcr.io/petomalina/krane
      context: .
      docker:
        dockerfile: ./build/package/Dockerfile
deploy:
  kubectl:
    manifests:
      - ./deployment/krane/*
```

The sample above shows an actual Skaffold configuration that creates the Krane Operator artifact, pushes it to the *gcr.io* image repository and then applies the Helm configuration while replacing the image name with the newly created image.

Chapter 9

Experiments

The experiments chapter contains two real-world scenarios tested on a development environment, while one also being tested in a near-production environment (staging). All situations have attached figures from Kiali, Prometheus, and Grafana for better understanding of system state and its implications.

9.1 Aggregator and Storage Microservices

The first experiment contains two microservices collecting, aggregating, and storing pure metrics data (e.g., clicks, page loads, etc.). A service called Aggregator serves as an entry point to the system, aggregating metrics when new data points arrive while saving the metrics to the Storage service for persistence.

```
apiVersion: krane.sh/v1
kind: CanaryPolicy
metadata:
  name: aggregator-policy
spec:
  base: aggregator-deployment
  virtualService: aggregator-vs
  service: aggregator-svc
  baselineMode: NEW
  ports:
    - port: 80
      name: default
      protocol: TCP
  test:
    image: gcr.io/petomalina/aggregator-testjob:v1
    boundary:
      requests: 600
  judge:
    image: gcr.io/petomalina/aggregator-judge:v1
    boundary:
      time: 10m
  diffMetrics:
    - metric: container_cpu_system_seconds_total
```

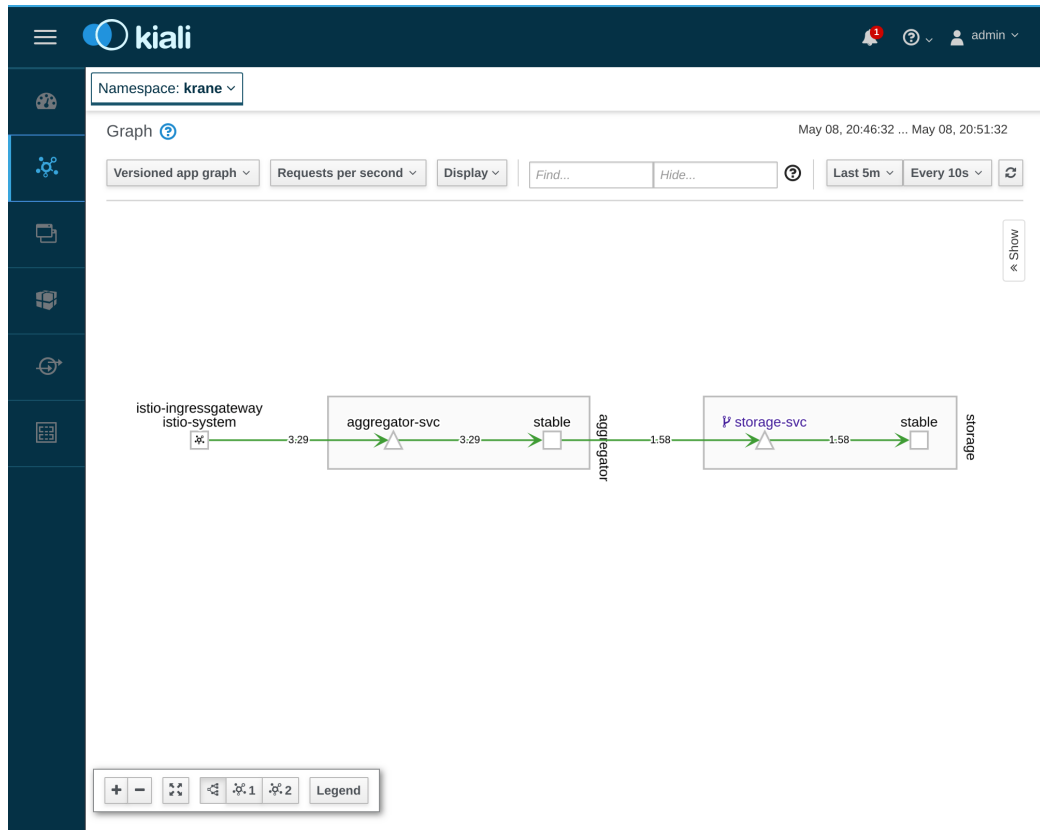


Figure 9.1: The default setup visualized by Kiali. The setup consists of two workloads: aggregator and storage, and two services for these services. Moreover, the istio-ingressgateway provides an entry point to the whole system from which the production traffic flows.

```
container: aggregator
diff: 0.05
```

The Canary Policy above defines a configuration for any Canary setup targeting it, automatically exposing the port 80 for both, Canary and Baseline deployments, using the *gcr.io/petomalina/aggregator-testjob:v1* image for testing with a boundary of 600 requests and using the *gcr.io/petomalina/aggregator-judge:v1* to gather the metrics and evaluate them with a limit of 10 minutes. The collected and evaluated metric, which is a cumulative CPU time that is being differentiated between the Baseline and Canary deployments is called *container_cpu_system_seconds_total*. In case the cumulative CPU increases by 0.05 CPU seconds for multiple data points, the Judge will fail on this metric.

9.1.1 Common System Setup

Current system setup consists of these two services, a Gateway that accepts any HTTP requests on port 80 and a Virtual Service that binds to the gateway and directs the traffic to the particular versions of the Aggregator microservice. The visual representation of the system is on figure 9.1

The screenshot shows the Google Cloud Platform interface for the 'DummyProject'. The 'Workloads' section is active, displaying a table of workloads. The table has columns for Name, Status, Type, Pods, Namespace, and Cluster. The workloads are filtered by 'Cluster: canary-test' and 'Namespace: krane'. The table lists several deployments and pods, including 'aggregator-deployment', 'aggregator-deployment-ah87p9-baseline', 'aggregator-deployment-ah87p9-canary', 'my-app-aggregator-deployment-ah87p9-canary-judge', 'my-app-aggregator-deployment-ah87p9-canary-testjob', and 'storage-deployment'. The status of each workload is indicated by a green checkmark (OK) or a yellow triangle (Running).

Name	Status	Type	Pods	Namespace	Cluster
aggregator-deployment	OK	Deployment	1/1	krane	canary-test
aggregator-deployment-ah87p9-baseline	OK	Deployment	1/1	krane	canary-test
aggregator-deployment-ah87p9-canary	OK	Deployment	1/1	krane	canary-test
my-app-aggregator-deployment-ah87p9-canary-judge	PodInitializing and 1 more issue	Pod	0/1	krane	canary-test
my-app-aggregator-deployment-ah87p9-canary-testjob	Running	Pod	1/1	krane	canary-test
storage-deployment	OK	Deployment	1/1	krane	canary-test

Figure 9.2: Google Cloud Platform visualization of initializing workloads of Judge and Test Job as well as already running workloads of all deployments.

9.1.2 Initialization and Testing Stage

The Initialization and Testing stages bootstrap the Canary and Baseline Deployments as well as Test Job and Judge. The Test Job targets the Canary Service, which load-balances between the Canary and Baseline Deployments while Judge queries Prometheus for metrics. This stage remains active until all testing by the Test Job completes and the Canary configuration can proceed to the Canary phase. The progress can be seen on the screenshot showing workloads running on the Google Cloud Platform 9.2. The figure 9.3 then shows Kiali visualization of all workloads and services already running.

9.1.3 Canary Stage

The Operator modifies the Virtual Service of the Aggregator service to split the production traffic 90:10 to test the service on the production traffic, while the Judge keeps querying the Prometheus API for metrics. After the stage is over, a cleanup that will delete all workloads and revert the Virtual Service configuration will occur. The example of successful Judge job can be seen on the figure 9.5 while the failed metric on the figure 9.6.

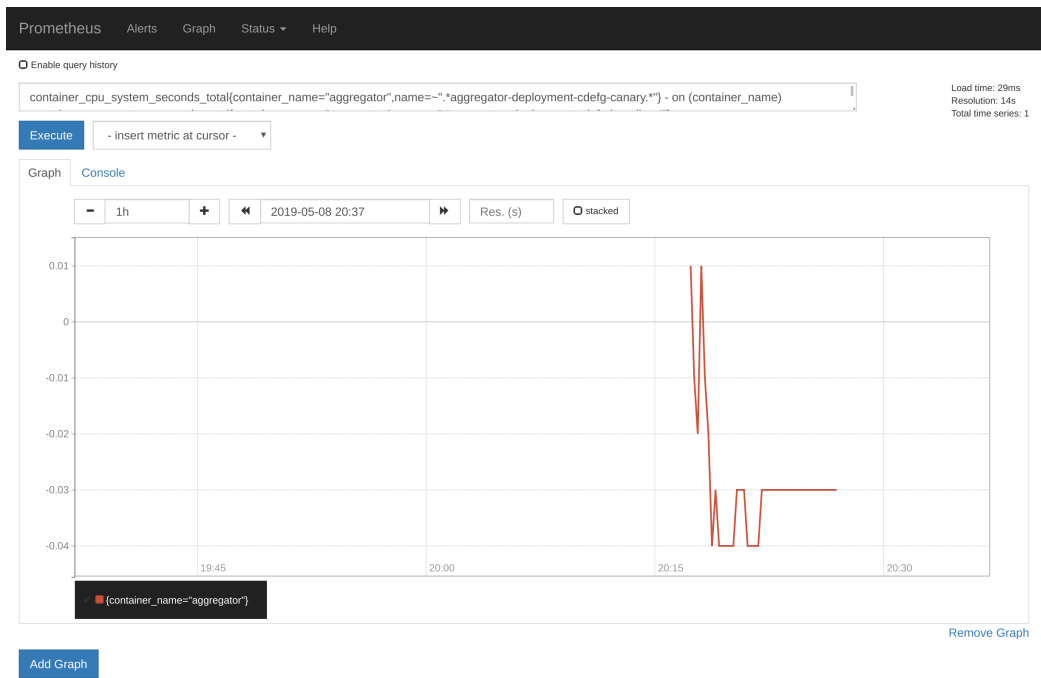


Figure 9.5: Prometheus visualizing a metric called *container_cpu_system_seconds_total* that the Judge uses to determine if the metric canary setup has failed. Because the Judge uses 0.05s allowed margin, the following margin of -0.04s up to 0.01s is allowed.

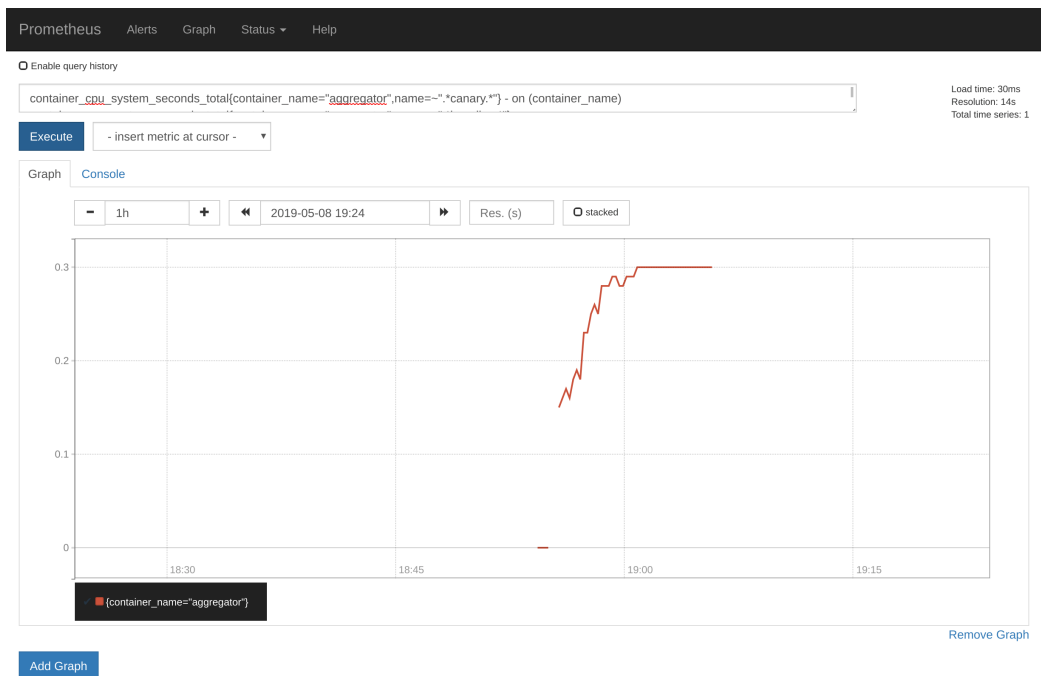


Figure 9.6: Prometheus visualizing the same metric as on the figure 9.5. However, the metric slowly starts increasing which fails the Judge.

9.2 Production-Grade Search Engine

The second experiment uses a production-grade system that searches and sources results of the whole mortgage and property market in the United Kingdom. The service uses mainly Cloud PubSub (a message bus from Google available on Google Cloud) to receive messages and calculate results; however, it also exposes a gRPC and HTTP APIs. Operators and developers use these endpoints for direct calls when testing or benchmarking.

Because this is a production system under strict NDA, performance indicators, API results, or valuable metrics could not be disclosed.

9.2.1 Common System Setup

The system setup consists of 2 microservices responsible for search in mortgage and property markets. Both services are capable of serving traffic on their own. However, their connection makes it possible for users to search for properties on the market while also getting mortgage deals if they are eligible. The services (also shown on figure 9.1 are:

- **Goldfish** is a mortgage sourcing engine that deals with eleven thousand mortgage deals per search on average. Deal filtering is using more than thirty rules with CPU intensive algorithms such as mortgage amortization table calculation and Annual Percentage Rates (APRs). The complexity, however, differs based on search preferences (e.g., some users may not want to get deals from particular banks, which narrows down other filtering rules).
- **Ernie** is a property search engine dealing with one hundred properties per search on average. This number heavily depends on the search criteria. However, when possible, Ernie also queries Goldfish for mortgage proposals so end users can instantly see for what deals they are eligible in case they want to buy a house.

All properties searched through the Ernie trigger a search for best possible deal on Goldfish, which results in fan-out from Ernie to Goldfish every time a search is done. For example, if Ernie finds twenty properties the user is eligible for, it creates 20 searches on the underlying Goldfish engine to resolve best deals for such properties based on the user and property data.

While Goldfish is a stable production-proven implementation of mortgage sourcing, Ernie is a service that is being actively developed and tested. These two cases make an excellent test composition for canaries, as both services are in different development stages, while both are eligible for canary deployments.

Setup of these two services allows seamless integration through the canary process, as shown in figure 9.8. Two services are connected in the production environment via their services, testing, and reporting back to the developers' results of canary analysis.

9.2.2 Production Usage and Results

Krane was first deployed to a production environment on May 6th, 2019. It has been two weeks in the time of writing of this thesis. There is a total number of twenty-three commits from which eight had a canary deployment associated after their CI pipelines have successfully passed. From these eight canary analyses:

- Five has passed all steps successfully and reported back. Further deployment of these versions was stable, and no major bugs or performance errors were found.

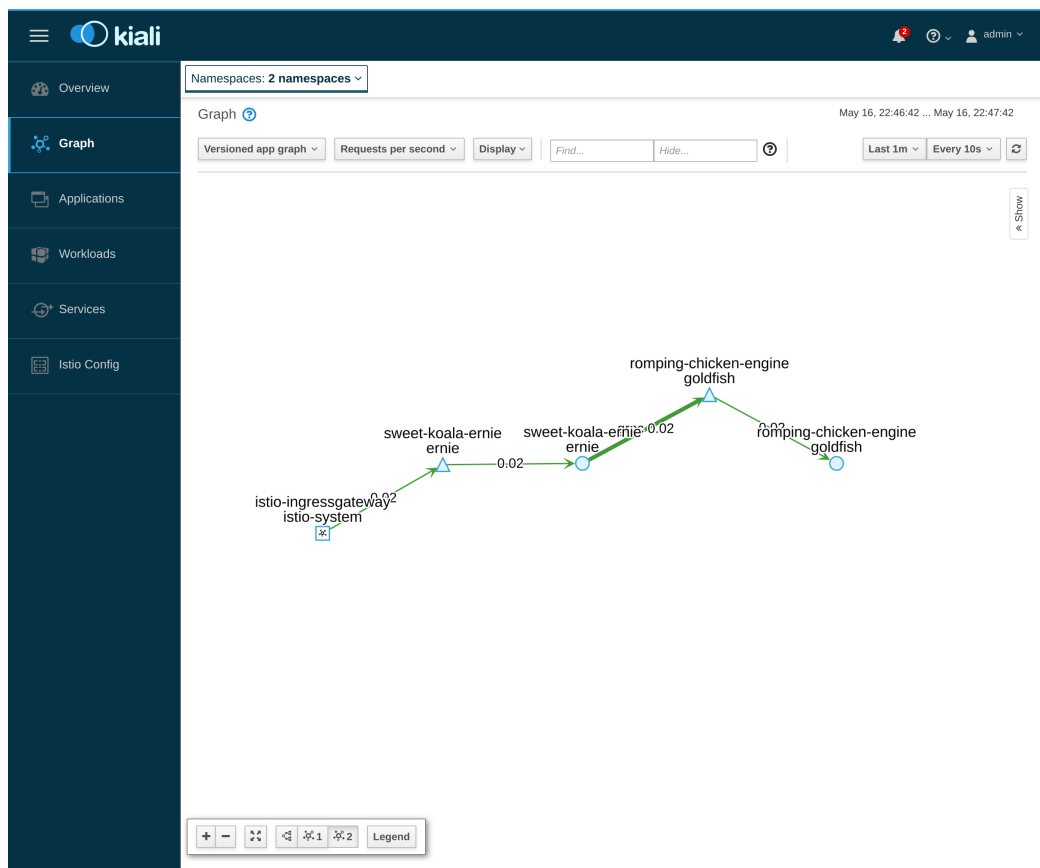


Figure 9.7: The default setup visualized by Kiali. The setup consists of two workloads: Goldfish and Ernie. All other traffic to Goldfish and Ernie is hidden for the purpose of cleanness in this example.

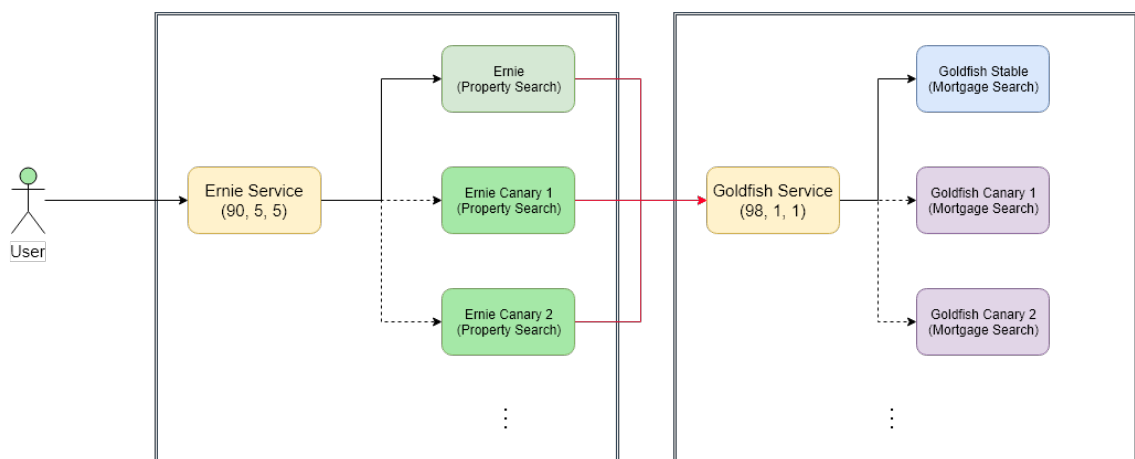


Figure 9.8: The canary configuration of Ernie and Goldfish with two canaries performing analysis on both workloads. While all traffic to Ernie is split 90:10 (production:canaries), Goldfish has its traffic split at 98:2. The configuration differs as Goldfish services tend to serve much more traffic than Ernie.

- Two has failed the Test Job execution and didn't make it to the canary analysis. Because CI was only able to test static behaviors through unit tests and lints (using mocks wherever possible), it did not catch parameter exchange between the third party API that is implemented as an acceptance test. Behaviors were corrected and added into unit tests for further speedup of the feedback cycle.
- One had failed the Canary stage in production, when user input value in such a corner case, it made the whole server restart with a stream of errors. The canary deployment was automatically reverted, and the status reported. Further investigation of logs and traces made it possible to quickly (in under 1 hour) fix the problem and create a canary that is now part of the five successful canaries.

Chapter 10

Conclusion

Continuous integration and delivery fit into the agile world with the concepts of automation and goals to create a viable cycle time for teams. The thesis covers common deployment strategies as well as the canary strategy. Many tools are being developed to solve problems of continuous delivery and canary deployments every day, but only a small percentage have enough impact while also being open-source. One of the tools is called Spinnaker - a joint effort of Google and Netflix to build bulletproof continuous delivery tool for enterprises using multi-platform solutions. Tools such as Spinnaker may not always be acceptable solutions for small teams as they require long term investments into infrastructure and operations team. More minor, more focused tooling is necessary for fully agile, self-contained, cross-functional teams.

Kubernetes, Istio, and Prometheus are discussed as tools for container provisioning, network policy enforcement, and metric collection. Kubernetes has become a de-facto platform for deploying and managing containers in the past years, as it has grown its user base from small companies to enterprises. Istio creates a layer on Kubernetes called service mesh. The service mesh is responsible for policy enforcement across all its services, which allows operators to build service-to-service restrictions as well as software-defined networking. Lastly, Prometheus, used in and outside of the Kubernetes world is an open-source alternative (and standard for many) to collect metrics, query using expressive PromQL and create alerts for metrics that do not comply with the rules.

The second part describes a tool called Krane, implemented to deal with canary deployments on Kubernetes and Istio, released under MIT license on GitHub. The link can be found on the address: <https://github.com/petomalina/krane>. Krane uses Kubernetes resources to configure, deploy, and manage workloads for all parts of the canary setup. It provides a two-stage process of testing and metric collection to assure that the analyzed code runs smoothly in the production environment before any production traffic reaches it. The canary setup is being continuously watched via Prometheus and automatically reverted in case the canary does not comply with the metric assertions. The second stage configures existing Virtual Services (routers) to split traffic to the canary setup, while continuously watching the metrics. After the canary analysis is done, all components excluding the status object are cleaned up, and the status is reported. A simple manual mechanism for cleanup was also implemented, in case a user wants to preserve all components for further debugging.

Multiple experiments across smaller systems were made to prove that Krane is working in a non-disruptive fashion when configuring the underlying system. Two more important tests were made to show the same applies to production-grade systems. The first experiment

was made on a setup of two microservices, held in a near-production environment to observe potential anomalies and succeeded in managing multiple canary scenarios. The second experiment was made on a production environment with two CPU heavy microservices searching property and mortgage markets in the United Kingdom. The experiment is ongoing and was able to execute eight production canaries so far, from which three would cause a production error and were not caught by the CI running unit and integration tests. The canary deployments were in particular good in detecting changes in API contracts or erroneous user inputs.

Bibliography

- [1] Amundsen, I. N. . R. M. . M. M. . M.: *Microservice Architecture*. 2016. ISBN 9781491956250.
- [2] AppCentrica: *THE RISE OF MICROSERVICES*. [Online; visited 08.04.2019]. Retrieved from: <https://www.appcentrica.com/the-rise-of-microservices/>
- [3] Brazil, B.: *Prometheus: Up Running*. O'Reilly Media, Inc.. 2018. ISBN 9781492034131.
- [4] s Brendan Burns Joe Beda, K. H.: *Kubernetes: Up and Running*. O'Reilly. 2017. ISBN 9781491935675.
- [5] Burns, B.: *Designing Distributed Systems*. O'Reilly. 2018. ISBN 9781491983638.
- [6] Hashicorp: *Terraform – State*. [Online; visited 07.01.2019]. Retrieved from: <https://www.terraform.io/docs/state/index.html>
- [7] Jarrell, J.: *Using the Scrum Framework*. [Online; visited 05.01.2019]. Retrieved from: <https://app.pluralsight.com/paths/skill/the-scrum-framework>
- [8] Kaul, N.: *Introducing Kayenta: An open automated canary analysis tool from Google and Netflix*. [Online; visited 10.01.2019]. Retrieved from: <https://cloud.google.com/blog/products/gcp/introducing-kayenta-an-open-automated-canary-analysis-tool-from-google-and-netflix>
- [9] Kornilova, I.: *DevOps is a culture, not a role!* [Online; visited 08.04.2019]. Retrieved from: <https://medium.com/@neonrocket/devops-is-a-culture-not-a-role-be1bed149b0>
- [10] Nova, J. G. . K.: *Cloud Native Infrastructure*. O'Reilly. 2017. ISBN 9781491984253.
- [11] Philips, B.: *Introducing the Operator Framework*. [Online; visited 06.04.2019]. Retrieved from: <https://coreos.com/blog/introducing-operator-framework>
- [12] Posta, C. E.: *Istio in Action*. Manning. 2019. ISBN 9781617295829.
- [13] s Ryn Daniels, J. D.: *Effective DevOps*. O'Reilly. 2016. ISBN 9781491926307.
- [14] Team, S.: *Homepage*. [Online; visited 08.01.2019]. Retrieved from: <https://www.spinnaker.io/>
- [15] Team, S.: *Homepage*. [Online; visited 08.01.2019]. Retrieved from: <https://www.spinnaker.io/setup/install/environment/>

- [16] Weaveworks: *GitOps*. [Online; visited 07.01.2019].
Retrieved from: <https://www.weave.works/technologies/gitops/>

Appendix A

Content of the CD

- /dip.pdf is the master's thesis text
- /dip/src is a directory containing the source text of master's thesis
- /krane is a directory containing the Krane project from the latest commit on May 20th, 2019
 - Makefile contains commands to build, run and deploy the project
 - Gopkg.* files contain locked Golang dependencies for the project
 - build/ folder contains Dockerfile used to containerize the application
 - deploy/ folder contains Kubernetes deployment files and CRDs
 - infra/ folder contains the tested Istio release
 - version/ folder contains a single version file containing the build version
 - pkg/ contains all implementations of controllers, reconcilers and the API
 - cmd/ contains the entrypoint to the Krane application (main)
 - example/ contains an example deployment using the canary deployment strategy