Building a scalable development cluster at Adyen

Building a scalable development cluster at Adyen

Thesis

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gijs WETERINGS

Student number: 4272587

born in Dorst, the Netherlands

Thesis committee:	Prof. dr. A.E. Zaidman,	Technische Universiteit Delft
	Dr. M. Aniche,	Technische Universiteit Delft
	Dr. P. Pawełczak,	Technische Universiteit Delft
	B. Wolters,	Adyen

Keywords: Software engineering, developer tools, scalability, containerization, orchestration

Style: TU Delft House Style, with modifications by Moritz Beller

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

An electronic version of this thesis is available at
`http://repository.tudelft.nl/`.

I have been impressed with the urgency of doing. Knowing is not enough; we must apply. Being willing is not enough; we must do.

Leonardo da Vinci

Contents

Summary

Software systems today are growing to incredible proportions. Software impacts everything in our society today, and its impact on the world keeps growing every day. However, developing large software systems is becoming an increasingly complex task, due to the immense complexity and size. For a software engineer to stay productive, it is vital they can work effectively on a system, being able to focus on the problem at hand. However, large software systems throw up a lot of roadblocks on the way, with complex and slow build processes impacting the developer's productivity to higher and higher degrees as the software system grows. To help developers stay productive, we need new, more powerful ways of assisting them during their activities.

In this thesis, we present our development cluster as a part of the solution to developing software at scale. The cluster provides a high-performance infrastructure that can be used by developers to build and deploy their applications during development. By moving these build and deploy processes to a cluster during development, we can benefit from more powerful computing resources which help developers work more effectively. We evaluate our development cluster in a number of different categories, comparing build speed, system startup and general developer workflows. Additionally, we evaluate how well our solution scales and what the impact on network load is for a company integrating with this system.

This move to cloud-based development brings along new challenges, but also many new possibilities in terms of tooling, developer collaboration and software engineering research. We are convinced our cluster can help scale software development efforts in industry, as well as bring new ways of doing research on software engineering.

Preface

Before you lies the result of 9 months of hard work as part of the degree of Master of Science in Computer Science at the Delft University of Technology. This thesis has been a huge challenge, one I definitely underestimated back in October. I set a personal goal for this thesis, to work on a problem that has both scientific and industry relevance. This has resulted in a thesis full of new experiences. I have discovered a fascinating field of Computer Science that balances both intense technical challenges and the human aspect of having to interact with these technical systems. In fact, I have gained a lot of respect for developer tools, and especially the engineers that work on them. I have managed to deliver a product to Adyen that has sparked discussions and future plans. Along the way, we have ran into some challenges and tackled them as best as we could

There are a couple of people I would like to thank, without whom this thesis wouldn't be possible. First of all, I want to thank Maurício Aniche for his supervision during this thesis. Thank you for trusting me in starting such an ambitious project, giving me the space to explore where the thesis needed to go, and offering your advice that helped lift this thesis to the next level. Also thank you to Andy Zaidman and Przemysław Pawełczak for being part of my thesis committee and helping me defend the work. Next, a big thank you to Bert Wolters, for taking the gamble on me and take on such an ambitious project. I'm confident Adyen can build on this work and hopefully roll it out to all her developers soon. I hope you are happy with the end result, and thank you for the enthusiasm you brought every time I could show you a new piece of the puzzle that fell into place. Next, a big thank you for everyone in the Development and Testing Tools stream at Adyen. Every last one of you has been a major help in my thesis, by discussing the work itself, but also by bringing such positive energy every day! A special thanks to Daan for pushing me back on track a few times when I got distracted by other problems, and to Adriaan for taking on the fight to grow support for this work inside Adyen.

In general, I'd also like to extend my gratitude to everybody at TU Delft. This university has been an amazing place to go from a high school student interested in making websites to an engineer ready for the big leagues. Not just the education, but the staff, everyone at the study association W.I.S.V. 'Christiaan Huygens' and all other students have all been a delight to be around and to learn together.

Last but not least, a major thank you to my family and friends for their support. Being able to discuss problems in the thesis, but also being able to step away from it and relax for a bit really kept me motivated to deliver the best thesis I could. Thank you for providing that balance and for all the wise words that helped me on the way. I really couldn't have done this without you, thank you for being there and pushing me forward!

Gijs
Delft, July 2019

1

Introduction

The processes for software development and deployment are fast-changing landscapes of best practices, standards and ideologies. Different methods have gained popularity over time, mostly conforming themselves to the technological abilities and the business requirements at a company.

In the early days of software development, the development process was constrained by computing time availability. Big, shared mainframes were the only machines powerful enough to do the hard work of building software for developers[1]. Because re-engineering was a costly practice, methodologies such as the Waterfall model[2] rose to popularity. The Waterfall model consists of a number of sequential phases. The idea behind this is to finish a phase before moving on, in order to prevent re-engineering costs due to oversight and rushed decisions afterwards. However, this had the disadvantages of a longer time to market and the inability to adapt during the process.

Following Moore's Law[3], computation power has become significantly more available over time, and has become cheaper. This additional budget of computing power has allowed the formation of new software methodologies. Many of these new methodologies fall under the umbrella of Agile software development[4]. Agile methodologies focus on getting smaller feedback loops while continuously re-evaluating requirements to deal with initial oversights or changing requirements.

This shift to small, feedback-driven loops can also be seen during software development work itself. This process is called Feedback-Driven Development (FDD)[5]. FDD relies from all sorts of tools, from compiler output to peer reviews, from static analysis to production monitoring[6] and from Continuous Integration (CI)[7][8] to manual debugging. Developers have built all these tools around themselves to get feedback on their work as quickly as possible. Some engineering methodologies even set feedback budgets, such as eXtreme Programming[9]. These methodologies focus in particular the "inner-loop" of software development, which is described by one of Microsoft's DevOps engineers Mitch Denny[10] as "the iterative process that a developer performs when they write, build and debug code".

1

1.1 Issues for large-scale software systems

Because developers rely so heavily on feedback from their environment, a developer's productivity is strongly coupled with the time needed for the developer to receive feedback. However, as software systems grow, getting timely feedback can become a bigger and bigger challenge. Major software companies struggle with this loss of productivity, particularly in their build process:

A recent internal study showed that our largest source of wasted engineering time comes from builds that take 2-10 minutes

Dan Lorenc, DevOps engineer at Google[11]

With sufficiently large software systems consisting of many different services, it can even be infeasible to get a full version of their platform running locally. This means developers need to either run mocked services, or accept that parts of the system failing to connect with some of their communication targets. This can cause big discrepancies between a developer's development environment and production, risking unforeseen runtime issues.

1.2 Our industry partner: Adyen

Adyen is a Payment Service Provider (PSP) that was founded in 2006 in the Netherlands, and has since taken the worldwide payments industry by storm. In 2018, Adyen passed €159 billion in processed volume across the platform. Adyen identifies itself by acting as a "unified commerce" platform, meaning many different forms of processing payments are all consolidated to a single platform. For example, Adyen offers over 250 types of local payment methods, such as iDeal and Boleto. With employees spread over 20 offices, Adyen is a worldwide 24/7 operation, and development on the platform does not stop.

1.2.1 Development challenges at Adyen

Adyen has a *Service-Oriented Architecture (SOA)* written primarily in Java (for the backend) and Javascript (for the frontend). With more than 60 unique services, such as risk engines, accounting services and banking applications.

This system has grown in size quickly over the past few years, with Adyen offering more services to their merchants. Graphs on how backend and frontend code have grown over the years are shown in Figure 1.1 and Figure 1.2 respectively.

To illustrate how Adyen developers work on the platform, we've collected build data from a subset of Adyen developers. We did this using Gradle Enterprise's build scans over the course of 1 month (06-05-2019 until 06-06-2019, containing 23 working days of 8 hours). An overview of the most time-consuming tasks is displayed in Table 1.1. Note here, that the tasks named are the entry points to task execution graphs, and the execution time includes any downstream work that needs to be done. We quickly highlight Adyen-specific tasks in the next paragraph.

The *resetAll* task is the container task that builds and deploys a full development environment to the developer's local machine. This system is built from scratch. This means compiling all modules, and packaging it into wars that are deployed to a TomCat

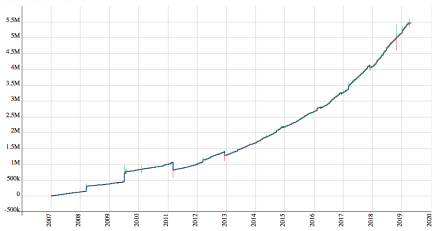

Figure 1.1: Lines of backend code in the Adyen system over time

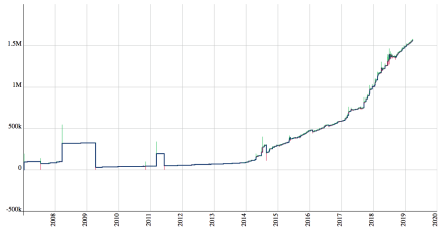

Figure 1.2: Lines of frontend code in the Adyen system over time

Task name	invocations	$\sum t_{invocation}$	mean t	median t
resetAll	1310	29 days, 4 hours	32.06 min	23.34 min
deployWar	5490	9 days, 20 hours	2.57 min	0.76 min
updateModel	911	4 days, 6 hours	6.70 min	1.29 min
deployAllWars	656	3 days, 16 hours	8.01 min	5.31 min
reloadWar	3190	3 days, 8 hours	1.50 min	1.02 min
compileJava	1610	2 days, 3 hours	1.88 min	2.34 min
dbChanges	1070	2 days, 9 hours	3.20 min	1.79 min
test	717	1 day, 13 hours	3.07 min	0.53 min
ensureStaticData	910	1 day, 9 hours	2.16 min	1.26 min
createAllDbs	43	8 hours	11.27 min	9.96 min
Total gradle commands	**22600**	**61 days, 22 hours**	**3.93 min**	**0.71 min**

Table 1.1: An overview of the most used gradle tasks during local development at Adyen (between 06-05-2019 and 06-06-2019)

instance, as well as building all databases including minimal (dummy) data. Together, this gets the developer up and running with a local system.

The *deployWar* and *reloadWar* tasks both build and deploy a single war for a single service, usually the one the developer is working on at the time (*deployAllWars* does the same but for all services at once). *ensureStaticData* and *updateModel* are tasks that check and update the database for various values, such as enum mappings and business logic rule configuration. Finally *dbChanges* attempts to do an in-place migration of the local databases to the latest schema, where *createAllDbs* builds all databases from scratch.

1.3 Towards a scalable development environment

Building and running local development environments for all developers can be a time-consuming activity. In this thesis, we explore how we can make the effort of local development more scalable, in order to increase the productivity of developers. We focus on DevOps principles and industry best-practices to design a development workflow that's scalable both for a growing number of users, and for a growing system.

1

To achieve such a developer experience, we define our main problem statement as:

How can we create a scalable development platform for large-scale software systems?

To support this problem statement, we define a number of challenges we need to solve:

Challenge I: Local build time As depicted in Table 1.1, developers spent almost 62 days of wall time on local build tasks in a single month. Considering the number of developers in this dataset and the amount of working hours, that is a mean of about 36 minutes per day waiting for build tools on the local machine. According to the research by Meyer et al, these moments are a common cause of context switching, which is detrimental to the productivity of developers [12]. Note that this time does not include any continuous integration jobs, those spent another 101 days and 12 hours (on considerably faster hardware) running various builds, test suites and other quality control checks.

Challenge II: Running a local environment A second issue arises after a developer is done building. Due to the size of the system and the limitations of developers' machines, it is currently not feasible for a developer to run a full instance of the Adyen platform on their laptop. By analysing shared team configurations, we see that most developers only run about a third of the services in the Adyen platform locally, before getting in trouble with the runtime environment.

Challenge III: Decreasing build avoidance Whilst build times are already a problem for Adyen developers, developers also seem to avoid rebuilding their local system quite a bit. In the same time frame of Table 1.1, a total of 4295 commits have been merged into master. Assuming an even spread of *resetAll* invocations this would mean an average of roughly 354 changesets have been applied to the platform as a whole before a developer rebuilds his full local setup using the *resetAll* command.

Push early, pull often.

—Adyen way of developing #4

Ideally, developers would be exposed earlier and more frequently to the changes in the platform, both in the services they actively work on and other services. This early exposure helps with early product feedback, and catching issues that were not covered by continuous integration builds. Therefore, it's important that our solution helps with any steps needed to upgrade an existing session to a newer version, particularly for the services the developer is not actively working on.

Challenge IV: Design for growth As a software system scales up, the previous challenges will become larger, both due to an increased number of services, as well as an increase in developers working on these systems. Therefore, we see many software companies experiencing quadratic growth[13] in size of their system. Any system supporting that growth should be designed to grow with the same pace.

We design for 20x

—Adyen way of developing #2

During the project, some additional challenges were defined based on our early experiences and decisions:

Challenge V: Pre-commit work As we formulated the strategy of moving build and run work from the developer's machine, we defined the explicit challenge of being able to work with dirty worktrees. This means that the system needs to be able to work with uncommitted changes to the developer's version of the codebase. While an obvious requirement for active development, this did pose an additional challenge we'll cover in Section 5.3.

Challenge VI: Network load As a result of **Challenge V**, we defined an explicit challenge of keeping the network load between the developer machine and the workspace as low as possible, to prevent building both a solution and a new scalability problem. We will give an overview of the impact on network load in Section 6.6.

All these challenges are based in scalability problems, where a larger system, or more developers working with this system can quickly worsen the problem.

In this thesis, we propose a solution that helps developers in their work on large software systems. We show how software systems with a service-based architectures benefit from our solution during software development. The solution helps developers by providing a framework for source-to-url deployments on horizontally scalable hardware clusters. We show how an existing software system can use this framework to their advantage with minimal modifications.

We demonstrate the usefulness of the system during inner-loop development through a series of experiments. Using historical change sets from Adyen's codebase, we show that our tool provides a scalable workflow. We show scalability for 1) growth in the codebase and 2) growth in the amount of developers. Additionally, we show how this framework adds new opportunities for tools to aid developers even further.

1.4 Thesis outline

In Chapter 2, we will discuss some related concepts, and look how other companies are tackling this problem in industry. In Chapter 3 we will go more in depth on classic virtualization, containerization and orchestration concepts, to give some in-depth background information to help understand the contribution. In Chapter 4 we analyse a classic developer workflow, and identify bottlenecks at scale for this workflow that our platform can focus on. Then, in Chapter 5 we will go over the components of our development platform in detail, describe their purpose and how they will help solve our developer issues at scale. We also present a few added benefits of this type of infrastructure. Next, in Chapter 6 we evaluate how well our platform performs with respect to the challenges we have defined earlier in this chapter. In Chapter 7 we go through some additional learnings on the way, and see how our system integrates with other tools. We summarize our findings in Chapter 8 and discuss future work, both in research and engineering.

2

Background And Related Work

In this chapter, we discuss some topics related to our problem statement, in order to set this thesis in the correct context.

2.1 Developer productivity, interruptions and task switches

Software development is an inherently human-based, intellectual activity[14]. Therefore, a developer's productivity is inherently tied to how positive of an experience the developer has during development.

A big impact to developer productivity is the frequency of interruptions and context switches. Margaret-Anne Storey and Alexey Zagalsky analyzed the effect of chatbots on developers, and noted that chatbots frequently interrupt a developer's work[15]. In a study on developer work habits by LaToza et al[16], 62% of developers mention that switching tasks due to interruptions is a major problem for their productivity.

Interruptions and task switching is not only frequent, it is also very time consuming. Recovering from a major interruption can take an average of 23 minutes[17].

Apart from external interruptions such as emails, messages or co-workers, there are also more intrinsic interruptions. One of these interruptions is a developer having to wait on the build process. If a build process takes too long, a developer will switch tasks and will lose the mental model of the problem he was originally working on. So, as part of constructing a productive environment for developers, we want to minimize the time developers have to wait before the developer gets feedback from his build tooling.

2.2 Feedback-Driven Development

A common theme throughout the evolution of software development methodologies is the forming and tightening of feedback loops. These feedback loops have become prevalent throughout the entire software development process, and continue to grow as the driving

force of developer's behaviour. This process of developing software based on feedback from many different sources is called Feedback-Driven Development (FDD)[5].

FDD tools are often split in two classes, predictive and analytic FDD[18]. Predictive FDD tools try to provide feedback based on static analysis. Analytic FDD tools evaluate the code dynamically to provide feedback on the runtime behavior of the code.

A good example of the usage of analytic FDD tools is Test-Driven Development (TDD)[19]. For TDD the development workflow is to first write tests defining the desired behavior, and after those tests are defined, implement the actual business logic while continuously running the tests against it. The goal of these tests is getting immediate feedback on exactly which parts of the business logic have been implemented successfully and which parts still need attention. This way, developers track their progress based on the automated feedback they receive. Sadly, in practice Test-Driven Development is only done by a small subset of developers[20], despite the process causing less defects compared to ad-hoc unit testing[21]. According to Adnan Causevic et al, reasons for not adopting TDD despite these benefits include development speed and lack of testing experience[22].

2.3 DevOps

In recent years, expectations on software teams to release faster have risen, such that they can keep up with shifting requirements[23]. From this business need, the industry started working more agile, tightening deployment loops to short sprints, often one or two weeks in length. However, once again the business needs have changed to be even more demanding. From this demand, workflows again changed, this time relying on automation to be able to tighten the development cycle even further. This change is where the term DevOps originates from, because to be able to make this possible, Development and Operations need to work closer together, to make sure the company can deliver high quality software at high speed.

To ensure this is possible, the term DevOps rests on five pillars of success:

Reduce organizational silos In order to be able to deliver software continuously, communication between different parts of a company is necessary. Shared responsibility and shared awareness enable the speed needed to deliver software with confidence.

Accept failure as normal With a high velocity of deployments, failure is a fact of life. There will invariantly be failures, so having a plan to mitigate these failures quick, is crucial to success.

Implement gradual changes To be able to deliver more quickly, moving fast is a hard requirement. Reducing cost of failure by shipping small changes that can be validated quickly gives developers the confidence to do so. This practice is known as "Shifting Left Development"[24], and the goals is to catch potential issues as early as possible. In the 2018 State of DevOps report[25], the DORA team finds "large-batch, less frequent software deployments lead to bigger failures that take longer to fix".

Leverage tooling and automation By using the right tools to your advantage, repetitive and time-intensive tasks can be automated to enable quick software delivery. This brings long-term value to the system, as manual work can focus more on bringing direct business value.

Measure everything Measuring the impact of changes to the system is vital to be able to detect and mitigate failures, validate the gradual changes and automate parts of the workflow.

If you can't measure it, you can't improve it

Peter Drucker

In this thesis, we will base ourselves on the definition **"A set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality"** for DevOps.

2.4 Industry solutions to software development at scale

The scalability of software development is a problem for many big companies. Interestingly, many of these companies have developed extremely specialized tools to solve the scalability issues they encounter. In this section, we go over a number of different companies, see their challenges of development at scale, and how they have developed solutions for these issues. In Section 7.4, we'll reflect back on these solutions and compare them to our own framework.

2.4.1 Google

Over the years, Google has encountered many scalability issues, both for development and for production. Many of Google's products have been built individually, so Google's codebase contains a lot of products, in many different programming languages. This means Google needed to solve their build scalability issues in a way, where they could support multiple programming languages. The way they have achieved this, (for most products within Google) is by building their own build system, Blaze[26], which is partly open-sourced as Bazel[1]. By configuring their build tool to behave like a Directed Acyclic Graph (DAG), the system can re-use downstream build artefacts without having to rebuild them.

To further scale the build process, Blaze has the option to defer individual build steps to remote machines. The artefacts generated by these build steps are cached globally and are shareable between developers, so that code that has not changed does not need to be rebuilt, even between developers. This concept is very strong in helping developers with scaling up their local development productivity.

For production, Google has been working on their automated systems for years. Their best practices from all generations of production deployment and orchestration have been consolidated in the Kubernetes project, which is the open source base of Google's internal systems as well[27].

2.4.2 Facebook

Facebook versions their code as a monorepo. Their code is written primarily in Hack and Javascript. This highlights the first step in getting a more scalable developer workflow.

[1] `https://bazel.build`

Over the years, they have migrated their (originally) PHP codebase to Hack. Hack is Facebook's own programming language, which acts as a statically typed superset of PHP. This has helped Facebook migrate incrementally by enabling static analysis, typechecking and other types of feedback to the migrated parts of the codebase.

As Facebook's code grew further, they ran into different scalability issues, this time through long local build times, slowing developers down while waiting for their local builds to complete. To solve this, Facebook created HHVM, a just-in-time (JIT) compiler for Hack. This decreased build times a lot by only having to rebuild classes that have changed.

Facebook also built Buck[2], their build tool which is similar to Google's Blaze. They use this primarily for their Hack and Java codebases.

Later, Facebook has tackled the scale issue even further with Nuclide[3], their own editor based on Atom. This editor spins up a remote development server that can offload any compilation or static analysis work from the developer's machine. These machines reside in Sandcastle, Facebook's own continuous integration solution, so once a developer is done with his work, he or she can transfer the entire development machine to the CI server, saving more time in recompilation and getting CI feedback even faster.

This year, Facebook has even made its first attempts at fully automated bug fixing. Their Getafix[28] platform can automatically detect bugs, and apply mutations to propose a fix to developers. This means that developers don't even have to go through the debugging and build stages for some work, but get a ready to go bugfix to review and send in.

Do note that these solutions are all heavily focused on how Facebook develops their code, so while components are open source, their true power is difficult to recreate.

For production deployments at scale, Facebook has built Tupperware[29], their own orchestration service aimed to provision and deploy new releases of their products many times a day. The volume of code changes at Facebook has grown quadratically[13] over the years, so keeping their development workflow fast and efficient is a main priority to "move fast".

2.4.3 Amazon

Amazon these days is obviously known as a big cloud-based company. Today, the AWS cloud is a major player on cloud-based computing. In an interview with AWS CEO Andy Jassy, he revealed the reason the first version of this platform has been built was to aid internal development processes at scale[30].

Interestingly, Amazon approaches scalability challenges from multiple angles. As indicated by the success of AWS, automation and developer tools help AWS stay effective.

In a more organizational sense, Amazon also recognizes the effects of Conway's law[31] on its products. Conway's Law suggests that organizations designing systems will be constrained to designs that reflect the organizational structure in the organization. In other words, a company will always mirror the design of its software to the design of their organization. Amazon plays into this theory by employing a "scaling by mitosis" strategy[32] for their product teams. This means that it splits up teams (and therefore services) whenever they grow too big. By keeping teams and products small, they individually stay agile within a large ecosystem.

[2] https://buck.build/
[3] https://nuclide.io/

2.4.4 Lessons learned

From these large software companies, their methodologies and their solutions to these scaling problems we can define a few key lessons. We use these lessons as inspiration for our platform architecture.

Companies run into similar scalabilty issues

Automation is key Both in development and in production, these companies all show a heavy focus on automation. While Facebook's Getafix platform is an extreme example, it is indicative of the length these companies go through in automation, to allow their developers to focus on bringing value and less on keeping their system running. In short, automation helps developers prevent interruptions. Therefore a developer is more likely to get in a state of flow, which helps developers stay productive[12].

"Build only once"-mentality This mentality was most prevalent in Google's Blaze system. The idea is to only build a service once, when it changes, and store the built artefact. If the same or another developer then needs the same artefact built again, they can simply download it instead of having to rebuild.

Distribution of work The other interesting lesson from Blaze is the distribution of work. By distributing the work of building multiple services to different machines, build systems can scale out pretty well. This means that given enough machines to distribute the work over, a full build only takes as long as the longest individual service.

Deeper integration with existing tooling helps with effectiveness This is a lesson taken from Facebook's Nuclide, where directly integrating Facebook's development servers into the IDE lowers the barrier of using the development servers' features to a single command or button click.

Learn from production deployment tooling All of the above mentioned companies use some system for orchestration in their production environment. With the 12-Factor App design in mind, we know that it is desirable to have a development environment which resembles production as close as possible. If we look at Adyen's production system, we see services distributed over many different servers, communicating over a network connection. Meanwhile, development happens on a single machine, and a single TomCat instance running all webapps. This difference in runtime environment is quite big, and has the potential to cause issues. The automated orchestration solutions therefore seem like an interesting option to consider.

3

Virtualization

This section gives a brief overview of virtualization. This thesis is mostly focused on virtualization of computer resources with the purpose of sharing these resources among developers, but this section describes virtualization in a general sense in order to get the correct context. Virtualization can be (loosly) defined as:

> Virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others[33].

Virtualization plays a major role in helping to reduce operational cost of running large software systems[34]. By splitting physical resources between multiple virtualized services or applications, the resources can be used more effectively. This chapter provides insights into various forms of virtualization of services.

3.1 Hypervisor-based virtualization

A hypervisor is a mechanism that allows control over physical resources to be shared across multiple virtual machines. These virtual machines are isolated from each other by only allowing access to physical resources (such as storage or memory) via the hypervisor. Hypervisors can be categorized in two groups[35]:

Type I This type of hypervisor runs directly on the hardware, without any host operating system. This approach is relatively efficient in terms of performance, as it eliminates as many layers of overhead as possible. However, this type of hypervisors are significantly harder to develop, maintain and operate, as it is solely responsible for both the hardware support and the management of the virtual machines on top. This means that any hardware used in the system must be directly supported by the hypervisor, which generally means that hardware support is limited in comparison. The main usecase for type I hypervisors therefore lies in datacenter computing, and other usecases where direct hardware access is not required or desired.

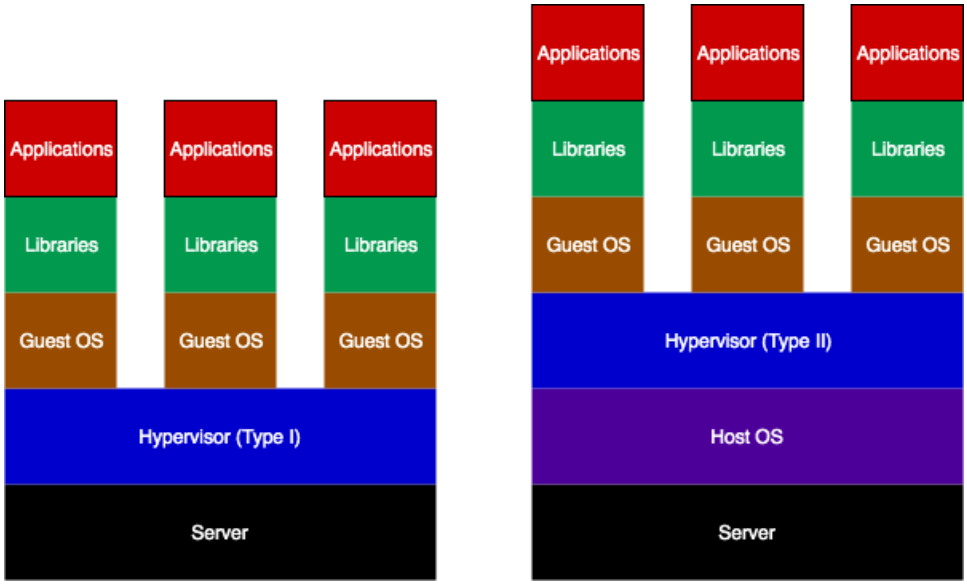

Figure 3.1: A visual comparison between hypervisor types

Type II Hypervisors of type II run on top of a host operating system. This has the advantage of easier operation, as a full operating system is available to provide management services in conjunction with the actual hypervisor software. These types of hypervisors can benefit from existing drivers in the host's OS to talk to the hardware. The downsides of this type of hypervisor are the added overhead of a full operating system running between the hardware and the virtual machines, as well as the additional security risk of virtual machine escapes. These VM escapes, when they happen, result in a complete breakdown of the security model of the system[36]. The main usecase for type II hypervisors is client usage, such as developers running virtual machines on their machines to test software in isolation.

3.2 Container Virtualization

Container Virtualization (more commonly called containerization) is a special type of virtualization, where lightweight environments (called containers) can be deployed, isolated and versioned easily.

In these containers, the focus is generally on one primary application running in the system. A container essentially manages an application including its configuration. Once they are built, they can be moved around easily between hardware, without having any worries about configuration drift between servers.

What makes these containers so lightweight is the fact that they share the host's kernel, separated by kernel-level namespaces.

Containers benefit from some key values in terms of usability[37]:

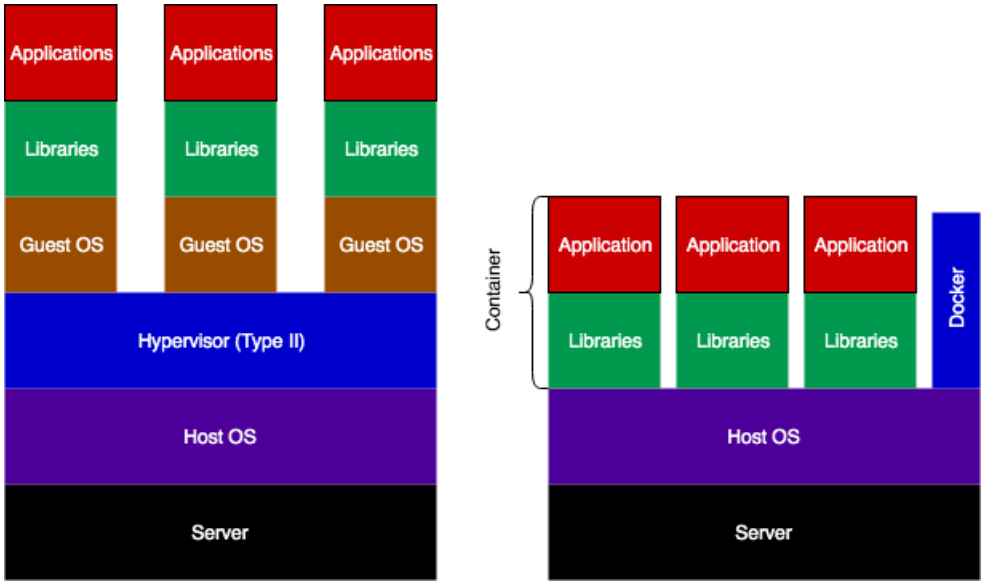

Figure 3.2: A visual comparison between a type II hypervisor and a system running containers

Portable Deployments Containers are made to be easy to move around. A deployment using a container can be moved around between different servers without being affected. This is due to the fact that containers have all dependencies needed to execute, including the runtime environment.

Rapid delivery Once containers are built and tested, they can be deployed quickly and multiple times, allowing for repeat deployments and horizontal scaling.

Scalability Containers can be deployed to a collection of different hardware platforms, from public clouds to laptop computers. This allows developers to easily scale their deployments up and down.

Faster build times Containers are generally small, containing just one service. This means a container can be built quicker by scoping the build execution to a single service.

Higher density with better performance Containers suffer from very little overhead, especially in comparison with virtual machines running on hypervisors. Because containers do not encapsulate their own operating system, they require significantly fewer resources. This means more containers can run with better performance on the same hardware virtual machines could run on.

Runtime consistency Containers provide mechanisms to decouple runtime application-layer software from the host operating system of the physical machine, to isolate and finely control these dependencies. This allows applications to update their

dependencies without worrying about the compatibility of the dependency for all other applications.

3.2.1 System versus Application containers

An important distinction to make when talking about containerization is the difference between System and Application containers. We identify the differences and clarify what we mean in the remainder of this thesis.

Application containers are meant to package and deploy applications without running a full operating system for every application. Their main advantage is easy and light distribution of applications with better control over the runtime dependencies. Application containers generally run only a single process, and therefore are single-purpose.

System containers can be seen as a lightweight replacement for virtual machines, allowing for a full environment to run in a single environment, where multiple applications can run together in a single container. System containers generally consist of a full operating system image.

While containerization technologies can be used for both application and system containers, there is generally a focus on one of the two. In this thesis we limit the "container" definition to application containers.

3.2.2 Enabling the Immutable Infrastructure paradigm

An advantage of using containers is that it enables the use of the Immutable Infrastructure paradigm[38]. This term was coined in 2013 by Chad Fowler[39]. It promises stable, efficient and version controlled infrastructure, by following the rule that once a component has been started, it may not be manually changed. Any change in behaviour therefore requires a new deployment. This makes configuration changes and -differences easier to keep track of and replicate for debugging.

3.3 Orchestration

Container virtualization is great way of creating runnable, portable applications. However, with these benefits also comes an additional layer of complexity compared to a local deployment on a single physical machine. Containers need to be deployed on one or multiple machines, and need to be networked together for them to be able to communicate. This process can be fairly complex and tedious if it has to be done manually, particularly at scale. To automate and standardize this process, the concept of *container orchestration* has grown in popularity.

> *Orchestration is the automated configuration, coordination, and management of computer systems and software.*
>
> —Thomas Erl[40]

This defintion means that all configuration needed to deploy a given system needs to be managed by the orchestration middleware. It is responsible for scheduling and balancing workloads on the computing nodes (usually either bare-metal servers or (virtual) servers in a public cloud). Orchestration tools are especially useful in combination with container

virtualization, as the images that have to be managed by the orchestration middleware are relatively lightweight and therefore easier to distribute between nodes in the cluster. In this thesis, we will use Kubernetes[1] as our orchestration system.

Note that orchestration tooling very much coincide with the DevOps concepts we discussed in Section 2.3. By automating repetitive and time-intensive tasks such as deploying services and configuring the network connections between these services, we can deliver software faster. The use of these forms of automation will help the "shift left" of deployment, making it possible to do deployments more often and in smaller batches. This is beneficial for production, but we can even use this during the development process.

3.4 Introduction to external tools used

To help us develop a complete scalable development server, we base ourselves on a few external tools. For every tool, we give a brief introduction into what it is and go over its purpose in our system.

We embrace new technology when it has clear benefits

—Adyen way of developing #12

3.4.1 Kubernetes

Kubernetes is an orchestration tool based off of 15 years of DevOps experience at Google[27]. Kubernetes leans heavily on the concept of Infrastructure-as-code[41]. Resources are defined in a declarative way using yaml configuration files. This means that the resource configuration can be checked in to version control systems alongside the code it is running. We use the command-line tool `kubectl`[2] to interact with the Kubernetes system. Using these tools, Kubernetes' strength is to give developers a very low barrier of entry in DevOps operations, giving control over deployment, scaling and networking of various workloads directly to the developer. For our development platform, we are interested in the elasticity Kubernetes can provide, as well as a lot of the automation we can use as a base to build on.

The major benefit of this elasticity is in the fact that it fits our developers need to have very powerful machines to do their build work quickly, but do not need this capacity during other parts of their work. This way, we can use our powerful hardware more effectively, by sharing it between developers as they need it. This cluster can handle the work of a large number of developers, and can increase its capacity in the future by growing the number of nodes in the cluster, either with physical servers in personal datacenters, entirely in a cloud environment such as AWS[3], or with some modifications even in a hybrid of the two. This means that as the cluster requires more capacity, be it due to the number of services growing, due to the number of developers growing or due to a different growth in complexity, adding more capacity is trivial.

All in all, we believe Kubernetes' orchestration services make for a good base platform for our implementation. It provides us with a solid foundation of tools and resources as a distributed cluster of computing power. Upon this foundation we will build our own

[1] `https://kubernetes.io/`
[2] `https://github.com/kubernetes/kubectl`
[3] `https://aws.amazon.com/`

extensions to support development work more specifically. To give a bit more background on Kubernetes, we go into a few core concepts.

Kubernetes architecture

Everything that happens in a Kubernetes cluster goes through the *API server*. This service is the core of Kubernetes and it instructs all other components to create, change or destroy other resources in the system. Clients such as kubectl send their requests directly to the API server, which lives on the *master node(s)* of the cluster. Besides the API server, the master node also has a scheduler and a controller manager component. The *scheduler* holds the rulesets that determines the best place to deploy new resources, and the *controller manager* is responsible for registering and running the various resource controllers in the system.

All non-master nodes are called worker nodes. They have a tiny service called *kubelet*, which receives instructions from the API server whenever any workload on that node needs to be created, modified or destroyed. It also runs a *kube-proxy* service that takes care of the network routing for all workloads on this node.

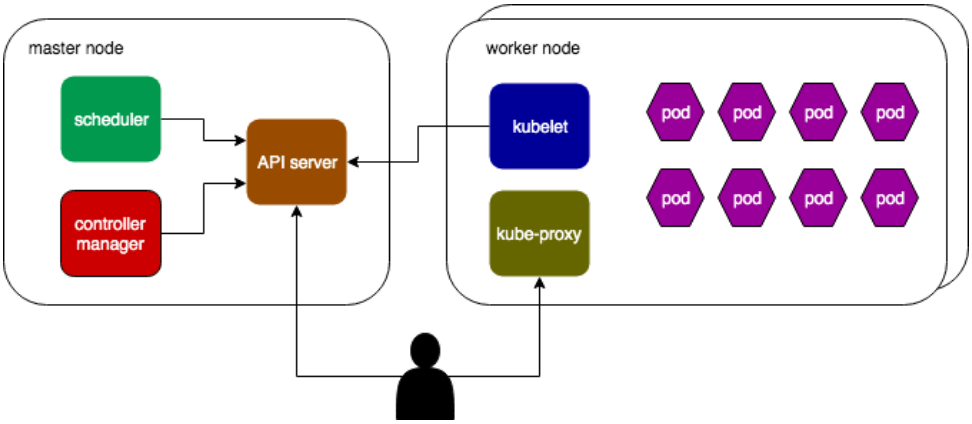

Figure 3.3: A visual overview of the architecture in a Kubernetes cluster

Pods

The smallest unit within Kubernetes is a *pod*. A pod represents a logical set of processes, which are running in one or more containers. The logical set can be defined as the group of processes that would have to run on the same physical server in order to work properly. Generally one container in the pod is considered the "main" container, running the main process. Other containers may be running in the same pod as so called *sidecars*[42], which help the main container with aggregating logs, changing configuration or help out with any other workload that doesn't directly affect a request to the service in the main container. Note that all operations such as pod restarts, redeployments or deletions happen for all containers in a pod, as the Kubernetes tooling operates on a pod level. Example 3.1 shows a minimal pod definition example.

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: myApp
  labels:
    app: myApp
spec:
  containers:
  - name: myApp
    image: myapp:latest
    ports:
      - containerPort: 8080
```

Example 3.1: A sample pod definition

Services

An important property of pods is their ephemeral nature. If for example a new version is available, or pods are scaled horizontally, Pods may be killed or restarted. This means a service running in a Pod can have one or more IP addresses, and these may change over time. In order to provide a more stable endpoint for other services and end-users, Kubernetes has the concept of *Services*. Services route any incoming traffic to a pod that matches its selector criteria.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service # Service will be exposed as my-service
spec:
  selector:
    app: MyApp # Select all pods with label app set to MyApp
  ports:
  - protocol: TCP
    port: 80 # Port forward an outside port to a container
      port
    targetPort: 9376
```

Example 3.2: A sample service definition

Namespaces

Namespaces are Kubernetes' concept of scoping. Namespaces allow us to logically separate workloads, such that we can give every developer a scoped space in the cluster to do their work without getting interfered by work done by other developers. Namespaces also give us the ability to scope permissions for users, such that a developer may only modify resources on their own workspace.

Custom resources

The before mentioned resources are all part of Kubernetes itself. Kubernetes however also supports the extension of its resources in the form of *Custom Resources*. This is a way to define new types of objects and have controllers to help the API server in making the correct alterations to the cluster whenever a change happens to these resources. For a more fundamental understanding of these concepts we refer to the Kubernetes documentation website[43].

3.4.2 Operator SDK

The pipeline that helps the developer with his workflow is based on Kubernetes' concept of *operators*. An *operator* is a service you can build and deploy on a Kubernetes cluster. It consists of two components, an API and a Controller object. We'll cover both here in detail to give a deeper insight of our development operator we describe in this chapter.

The API defines a new type in the form of a *Custom Resource Definition (CRD)*. A *Custom Resource* in Kubernetes is a way to add new types of objects to a Kubernetes cluster. All parts of the Kubernetes ecosystem then can read and interact with objects of these new types. However, on their own these custom resources are simply data objects that won't do anything. A Custom Resource Definition generally consists of three main elements: *Metadata* to enrich tools with additional information about an instance of the custom resource, A *specification* object defining the fields of the object, and a *status* object defining fields that can report back the status of a custom resource. However, on their own any created custom resources of that type will simply store data as it is presented.

To let the cluster act on the custom resource objects, an *operator* also has a *Controller* service. This is a service deployed in the cluster. It subscribes to any events (such as creation, deletion or modification) on the cluster that concern objects with the API's custom resource type. Whenever an event comes in, the Controller *reconciles* the affected object. The process of *reconciliation* is to take the Specification of an object, and make changes to the cluster (such as launching pods, reconfiguring networking or one of many other things) to match it to the desired state defined by the *specification*. Additionally, the *reconciliation* loop also updates the resource's *status* object to give feedback to the users about the observed state of the cluster regarding this object. This way, users can keep track of the state of these custom resources in the cluster, as they can with any other resource.

An important implementation detail to keep in mind is that the reconciliation function may only apply a single change at a time. This can be anything from launching a pod, to modifying another resource's *specification*, to updating the instance's *status* object. After any change to anything in the cluster is triggered, the reconciliation loop can choose to requeue the custom resource instance. If this happens, the instance is put back at the end of the controller's work queue. Whenever it gets back to the front of the queue, it can again apply a single change. This process repeats until the cluster's state (including the instance's reported *status*) is fully up to date with the *specification*. At that point, the resource is not requeued. When any watched objects for that resource change for whatever reason (including nodes going down, changes by the developer or events happening in the system itself), the object is requeued for another loop through the reconciler. This makes sure the object's *specification* still matches the cluster's state.

This pattern of reconciliation with requeueing allows for changes spread widely across

the cluster to get applied in a structured way. All elements of our pipeline therefore use this pattern extensively.

3.4.3 Kaniko

Kaniko is a project by the Google Container Tools team, that allows the building of a container image from a Dockerfile without depending on a Docker daemon. This greatly improves the security of our platform, as we don't have to allow access to a privileged daemon service running on the servers' host operating system.

Kaniko works by emulating all commands in a Dockerfile inside of its own file system. Every step gets executed in a container with a decicated root for the file system of the container image. After every command, a snapshot operation runs to store the current status of the container image. This way, Kaniko can cache intermediate steps and instead of re-running a build process, it can re-use the same output, given no inputs have changed. This massively speeds up the build of a container image for subsequent builds.

We use Kaniko in this project because it can run without this Docker daemon. However, as this project is still under active development, some features were not fully complete yet. To add some visibility to the Kaniko build process for tool integrations, we've done a small contribution to Kaniko, which we'll go into in Chapter 7.

‘

4

Developer workflow

In this chapter, we'll explore the software development workflow for a developer. We go into the steps a developer takes during their daily activities and identify bottlenecks in this process that can impact the developer's productivity. Based on this, we identify the goals for our developer platform.

4.1 Example: Alice fixes a bug

To illustrate a commonly occurring developer workflow, we take the example of fixing a reported bug. Note that this workflow is very general and depending on the developer and/or project, there may be differences in the exact order. However, we will see most of the general principles will still apply.

Our example developer Alice is tasked with fixing a bug in one of the services of a software system. After receiving the report, Alice wants to verify she can reproduce the bug. To do this, she pulls in the latest version of the system from version control. Alice now has to build her local version of the platform. She instructs her build tool to build all services and spin up her local copy of the platform. The build system spends a few minutes building all the different services, databases and other artefacts she needs to run the system. After the build tool is done, Alice can try and reproduce the reported bug, trace the behaviour using her monitoring tools and form her initial hypothesis on what might be wrong.

At this point, Alice enters the Edit-Compile-Test loop. In this loop, Alice will iteratively write some code, compile it and test her solution There are a few different ways this loop can work. Alice could work test-driven: writing a test first that reproduces the bug on a unit level, and then work on a fix in the production code to make the test pass with the expected outcome. Alternatively, Alice could go for a more exploratory approach, running the code through a debugger or reproduce the bug as a user and trace the interaction and outcome. Depending on the nature of the bug, Alice may choose one of these methods, or a mix of both. Any way Alice decides to go, she will go through a number of Edit-Compile-Test loops, until she fixed the issue. She'll then want to clean up her change, verify the bug has indeed been fixed by trying the original reproduction once more, and commit her change.

4.2 Identifying bottlenecks in the software development process

The above scenario is just one of the potential tasks Alice may go through, but the general structure will always be roughly the same:

1. Get a local version of (a revision of) the system going
2. Make changes in a Edit-Compile-Test loop
3. Check in the changes to the system to version control and potentially verify the correctness with the continuous integration system.

In each of these steps, the developer may encounter some bottlenecks

Bottleneck I: Cold start of a system Whenever a developer starts a new task, or has been working for a considerable time on the same task, the developer will want to incorporate intermediate changes done by other developers. This way, the developer is exposed to the other changes of the system. This helps with getting feedback, signalling any overlooked errors early, as well as prevent conflicts with the developers own work. The downside is that the developer spends considerable time waiting for the build tool whenever they pull in changes, as generally multiple services will have been rebuilt.

Bottleneck II: Waiting for compilation When the developer enters their Edit-Compile-Test loop, the time between finishing a change and getting the feedback from compilation, static analysis and automated and/or manual tests can become a serious performance bottleneck. Not only do developers have to wait on their tools, but if it takes a long time to get feedback, developers may get distracted and make a context switch. This negatively impacts the developer's productivity as they have to recover their mental model of the change they were making. The shorter the feedback cycles are, the more effective they are in bringing value to the developer[5].

Bottleneck III: Feedback from Continuous Integration After a change is checked in to source control, the Continuous Integration system will try and verify to the best of its ability if the change does not break anything unexpected. This could be due to merge conflicts with other changes, a failing test or the change not passing the quality gate set up by static analysis. Whatever the reason, the commit will get sent back to the developer for them to fix. However, if getting this feedback takes a long time, the developer will likely have moved on to their next task. The next task may have made significant changes to the system. If the developer has to go back to a previous commit, this may cause multiple build cycles to switch back to the old task, fix the issue and then switch to the new task's setup once more. Besides waiting for the build, this also takes multiple context switches, as the developer may have to recover their mental model for the previous task to understand and fix the issue.

5

Building a scalable developer workflow

As we can see by the bottlenecks we encountered in Section 4.2, a lot of the work that needs to be done to be able to build and test new components, or fix existing ones, relies on tools that build and deploy a local version of whatever software system the developer is working on. However, with large software systems, these operations can be very time consuming due to the large amount of work that needs to happen on the developer's machine during these processes. At the same time, due to the desire of both developers and their employers to be able to work in a flexible environment (work from home, respond to incidents, take computers into meetings), laptops have become industry standard equipment for developers. These laptops are relatively constrained in the computing capacity they can deliver. This creates a shortage in computing power, which slows down these build processes the developer is already waiting on.

We hypothesize that giving developers a way to both keep their flexibility with their laptops, and give them a way of decreasing the impact of these bottlenecks on their daily work, will make these developers more productive. Because of the rapidly growing scale of software systems, a good solution to this problem should be able to grow with a developer over time, instead of simply kicking the can down the road by replacing a developer's machine with a slightly more powerful laptop every time, especially since every developer only needs this computing capacity during these build-and-run tasks.

In this thesis, we present a scalable development cluster that attempts to solve this problem. Our solution is based on the concepts of a cloud-native environments and distribution of work to provide a platform that can scale over time, and that can be shared between developers to make better use of the total computing capacity.

To illustrate how we designed this system and show why it works, we give a high level overview of the full system in Section 5.1. Then, Section 5.2, Section 5.3, Section 5.4, and Section 5.5 highlight the individual parts of the system and explain their role in more detail. To illustrate how users interact with the system we close with examples in Section 5.7 and Section 5.6.

5.1 A scalable developer platform

In order to understand what the developer's workflow looks like, and what happens behind the scenes, we show a simplified, high level overview of the full system in Figure 5.1. In this section, we'll walk through a typical usage scenario from the developer's perspective. The details of any of these components are further described in the next sections.

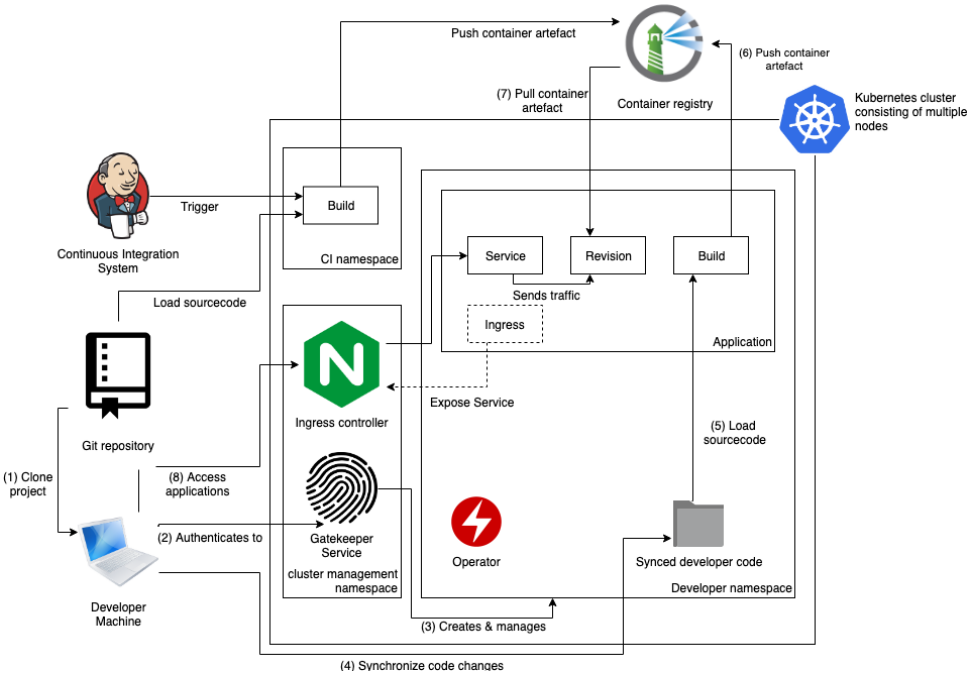

Figure 5.1: The high-level architecture of our development cluster

Suppose a developer is tasked with fixing a bug in one of the system's services. The developer will first pull in the latest changes from source control (Step 1 in the diagram). This way, he makes sure he is on a recent enough revision that he has the bug in his workspace.

The developer's next step is to get the application running, in order to debug and reproduce the bug. In the system, building and running the application all happens on the Kubernetes cluster. To get his personal workspace on the cluster, the developer calls the Gatekeeper service (Step 2) and authenticates himself. The developer is authorized to use the system, so the Gatekeeper service will spin up a Kubernetes namespace including our Operator (Step 3). Anything that is running in this namespace is specific to this particular developer's session. Once the namespace has started, the developer's machine synchronizes any code changes to the cluster, so the cluster has the exact same codebase as the developer (Step 4).

At this point, the namespace also starts all the Application resources. Because the developer hasn't changed anything yet, the namespace can reuse container images it has built before (Skipping step 5 and 6, moving on to step 7). By pulling in these pre-built

images configuring Revision resources with them for every application, the full system is booted very quickly, as no services had to be built. The application controller also configures the necessary Kubernetes resources to connect applications with each other and expose certain applications for external access. The developer can then access the application (Step 8) and start the debugging/reproducing process.

After pinpointing the bug and writing a possible fix, the developer then wishes to test his fix on the running system, to make sure the bug can no longer be reproduced. To do this, the system again visits Step 2-7, this time including Step 5 and 6, which create a new container image for the changed service, and pushing it to the registry, before creating a new Revision resource for the changed application. Once this revision is created, the developer can again interface with his running system and verify the bug has been solved.

In the next sections, we describe the individual parts of the system in more detail.

5.2 Gatekeeper service

The Gatekeeper service is designed to be (as the name suggests) the gatekeeper to the development cluster. It is responsible for authenticating developers and assigning them a workspace on the cluster.

A *workspace* in this concept is a Kubernetes namespace that hosts all services, builds and other tools for a developer. We use this concept to scope the resources of a developer, such that we can control resource usage, do auditing on cluster usage and logically separate out a developer's workloads on the cluster from others.

When a developer calls the Gatekeeper to request a workspace, the gatekeeper returns a namespace identifier. This namespace is created dynamically and provisioned with services that aid the developer, such as the source synchronization daemon (Section 5.3) and the development operator (Section 5.4). The developer then gets the rights they needs to work in this namespace bound to their user in the cluster. Finally, the namespace is annotated with an expiry timestamp, by default 12 hours after creation.

All of this together makes that with a single command, any developer on the cluster can request a namespace, that is provisioned within seconds with all the tools they need to do their work. The Gatekeeper also runs a vacuum job on a set interval to clean up expired namespaces.

Whenever the configuration of a workspace changes, the Gatekeeper will apply the new configurations to all new workspaces that get created after the change. This way, a breaking change to developer tools can be safely deployed without affecting ongoing sessions on the cluster.

5.2.1 Benefits of ephemeral, on-demand workspaces

One of the key benefits of creating these workspaces on demand is that if there is a problem with any of the development tools in a workspace, the user can hand off their workspace to the infrastructure team for debugging and fixing, and in the meantime continue their work by simply requesting a fresh workspace. Additionally, if a developer has to switch between tasks quickly, they can simply create a new workspace, create their fix, and go back to the original workspace afterwards to continue their original task.

Expect failure as normal

DevOps principles

An additional design decision we've made is the implementation of the Gatekeeper vacuum job. This is a process running on a set interval that cleans up old namespaces. The lifespan of a workspace can be configured during workspace creation, and modified later by authorized users. Once a system works smoothly on the platform, we propose a maximum workspace lifespan of 12 hours. This means that developers will get a fresh workspace every day to do their work. This helps with the roll-out of any new configuration, but also pushes developers to automate the configuration of special setups. When a developer automates the setup for their special reproduction case, this will also help them write automated tests for this scenario.

5.3 Developer source synchronisation

Because developers want to be able to test their code before they check in any change to version control, we need a way of shipping code to the developer's workspace without requiring changes to be pushed to a git remote. Therefore, this requires us to directly send source code from developers' laptops to their workspace in the cluster. A major concern for Adyen was the network bandwidth required to push the full codebase to a workspace for every developer. Including git history, this comes down to several gigabytes of source code and assets. Therefore, keeping network load on the system down was a design priority, see also **Challenge VI** in Section 1.3.

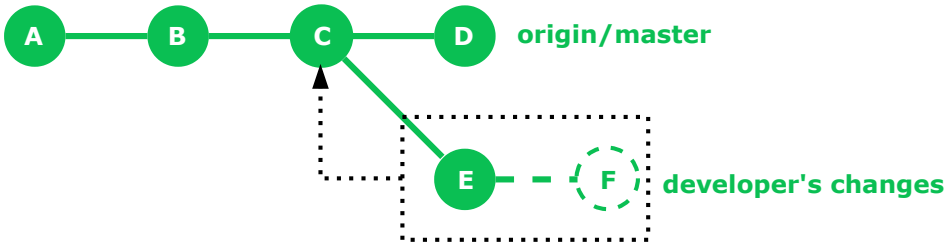

Figure 5.2: An illustration of how we gather the data for synchronization

We minimize the network load of synchronizing changes to the workspace by only sending anything that changed in comparison to a mutually known point. This means that we effectively simulate a (stack of) commit(s), but without the administration of creating a real commit.

To synchronize the codebase on both sides, we find the latest branching point from the master branch on origin. In Figure 5.2, this is commit C. This is the last state of the codebase that is present on the git origin, and therefore will be our common starting point. We then create diff files of any local commits after this point (commit E in the figure), as well as changes and untracked files that have not been formally added to a commit yet (WIP commit F). We send over the branching point and diffs via the `kubectl cp` feature to our source synchronization daemon in the workspace.

This daemon then re-applies the diff files on the same branching point of the codebase, and stores the result in a persistent volume. This volume is the source our build processes can work from. Having this method of source synchronization has a few up- and downsides. The major upside is that we only have to synchronize our changes to the workspace whenever the developer wants to build a new version, minimizing the network load between the developer and the cluster. The major downside to this approach is that we will copy the source code twice in the process of building a service: Once between the developer and the source synchronization daemon, and once between the daemon and the build process of a service.

Note that the cluster does need a clone of the repository to start off with. However, we can make use of two facts to keep this overhead as low as possible still. Firstly, because both the git server and our development cluster live in the same datacenters, we can benefit from datacenter-level bandwidth between these services. Secondly, since we know where in history our starting point is, we can make a fairly shallow clone of the repository, without having to worry about ancient history in this workspace.

5.4 Build containers on-cluster

Our pipeline offers an on-cluster solution to help developers get quick feedback on their written code. Our build pipeline is flexible enough to allow for a wide range of build setups. The only requirements for a service to be built is a Dockerfile describing how to build the service, and a way of describing the system how to supply the build process with the correct source files. To configure the build setup for a service, we present the 'Build' custom resource. This resource is meant as a cluster configuration of a build pipeline, while leaving the application-specific implementation details to individual containers. Our Build specification has two main elements. The first step is loading the source code to build in our environment. After that, a number of build steps are executed in sequence. We'll go into both phases of a Build more in depth.

Below you see an example of a build resource for a service.

```
apiVersion: adyen.com/v1alpha1
kind: Build
metadata:
  name: hipster-shop-frontend
spec:
  source:
    git:
      repository:
https://github.com/GoogleCloudPlatform/microservices-demo.git
      commit: 55f5061532798b9730b33b46401989c7115f742d
  buildSteps:
    - container:
        name: sleeper
        image: busybox
    - kaniko:
        dockerFile: ./Dockerfile
```

```
        buildContext: /workdir/src/frontend
        destination: hipstershop/frontend:latest
        registrySecretVolume: kaniko-secret
  volumes:
    - name: kaniko-secret
      secret:
        secretName: dockerregistry
        items:
          - key: .dockerconfigjson
            path: .docker/config.json
```

Example 5.1: Sample configuration of a Build resource

Once a Build resource gets created in the cluster, our Operator queues it up for reconciliation. During this reconciliation process, the operator will use the specification provided by the user to configure and spin up a pod. This pod is configured to run everything needed for the Build to execute. This consists of three main elements: (1) A volume that acts as a working directory for the entire build. This is a Kubernetes volume of the type EmptyDir, which means the volume is created when the pod is created without anything in it, and it's lifecycle is directly linked to the pod, so it is cleaned up together with the pod, after the build is completed. (2) Secrets needed to clone git repositories, pull in images and push built images to our container registry. (3) A number of containers that will run in sequence, called InitContainers in the world of Kubernetes. These InitContainers carry out the work in the configured build steps. These three elements in the pod's configuration enable it to execute the full build. The Build controller keeps track of what this Pod is doing during its execution, and reconciles the Build resource whenever something important changes.

To illustrate how this works, we will walk through the lifecycle of the build specified in Example 5.1. Note the lifecycle and the specification of the Build resource are highly parallel. This helps users track better what the system is doing, and makes it easier to configure a Build resource.

5.4.1 Loading source files in the working directory

The building and deployment of a service in the developer's workspace starts with the source code. Our Build resource offers two ways of fetching the source code for a service. Which one to use depends on the usecase. If the user wants to build an artefact for a revision that has been checked in to source control, they can configure the source block for their Build resource with a git block. An example of this can be seen in Example 5.1.

When the Build controller reconciles a Build resource with this block, it configures the build Pod to start off the build with a container designed to pull in this git revision. The source container will always be the first one in the pod, as it doesn't make sense to build anything when there is no source code available yet.

```
  source:
    fromVolume:
      volumeName: codebase
```

Example 5.2: Sample configuration getting source code from an existing volume on the cluster

As an alternative to building a service from a checked in git commit, another scenario we are especially interested in is the concept of building services before the git commit is done. For this, we have seen our source synchronization workflow in Section 5.3. This workflow copied any code changes from the developer's machine to a volume in the cluster's workspace. However, when we start a build with the `fromVolume` source, this makes a copy of the data in the volume the developer synchronizes their source code to. This is done for two reasons: (1) To keep the source code isolated while we're running our build process, and (2) to get a copy of the source code on the node in the cluster where our build is running. If we keep it in the original volume we synchronized to, we run into performance issues due to the build process continuously having to go back and forth to a different node's copy of the data. Because of these reasons, our `fromVolume` starts off with a single copy operation to get all the source code close and ready for use.

5.4.2 Perform build steps

Once code has been loaded into the working directory, the Build resource can kick off the build. To specify what happens next, the user of the pipeline specifies any number of build steps, as shown in our Build resource example in Section 5.4. These steps can be a few different types, for our prototype we only have two implemented.

The most basic build step is the definition of an almost regular container. The only difference is that the controller will inject the working directory volume automatically, so it's always ready to be used. This configuration can be used for many different special usecases, such as uploading build reports to a CI server, running tests, or anything else you need to do during a build, but that doesn't need to be included in the final image or is preprocessing for the final image build. An example of this can be seen in Example 5.1.

In addition to a bare container option there is also the option for a Kaniko block. This takes a few arguments, and builds a configuration with opinionated defaults for the Kaniko container. Kaniko takes the role of a Docker daemon's local build process. The difference is that it doesn't use the privileged daemon, but executes all Docker commands in userspace to build an image.

Once configured, the build Controller takes a build resource including the source and the to be executed build steps, and creates a pod that executes these steps in the correct sequence.

5.4.3 Reporting build result

In order to keep the user informed on what is happening during the build, the controller tracks the pod during its run, and updates the build resource's Status field whenever there is important information. In our case, the Build resource keeps track of the current phase of the build (Pending, Succeeded or Failed), when the Build pod started its work, when it was done, how long the build process took. Once the build is done, one more Status update takes place. The controller looks through the containers in the build pod, and looks for the last container that published a `sha256` digest in its termination message. This is the digest it can use automatically for the next phase, which is running a service. Users can monitor their build progress with `kubectl get builds.adyen.com`.

```
1 status:
2   duration: 3m11s
```

```
3   endtime: "2019-05-14T13:42:12Z"
4   imagedigest:
      hipstershop/frontend@sha256:7544b0722b8bd2616720e8a22e...
5   phase: Succeeded
6   starttime: "2019-05-14T13:39:01Z"
```

Example 5.3: Example of a Build's status at the end of a build

5.5 Managing the full Application pipeline

At the heart of our framework is the Application controller. This is the glue of our Operator, and allows for automatic management of a source-to-url deployment. The Application Custom Resource Definition holds a configuration for a build pipeline and a template for Revisions. This is a YAML based configuration of a service, in order to pass in the configuration of how to build and run a particular service. Examples of these configurations can be found in Appendix A. These configurations need to be written once whenever a new service is added, we will go over that process in Section 5.6.

If the Application does not get a digest pre-configured to run, it will start a Build resource that is responsible of building a container image of the service, based on its own configuration. Once the Build resource reports back successfully, it provides the digest of the just built image. The Application resource is then requeued, so the Reconcile loop can configure a new Revision resource. This Revision resource actually deploys the built image and manages its life-cycle.

5.5.1 Routing service communications

The Application resource also optionally manages a Kubernetes Service object. This is an abstraction from Kubernetes to provide a consistent endpoint for the services. Generally, if the Application needs to talk to anything else, be it another application or an end user, a Service is probably desired.

A service object uses so called "label selectors" to find pods labelled with a specific key-value tuple. All alive pods with this tuple are considered target endpoints for this service. The service keeps track of the existing pods in the system and (randomly) load balances traffic between the current active nodes. That traffic is all traffic pointed to the Service's IP address. This means that applications can remember the stable IP address of a service, rather than all having to deal with updating, scaling, or even failing pods. This greatly simplifies any communication between user and application, or applications amongst each other.

5.5.2 Routing external users to an application

When the Application controller configures a service for an Application, it gets a stable IP address for the Application inside the cluster. This way, other Applications can send traffic to that IP address to communicate. However, external users such as our developers are not in the cluster directly, so they can not access services with this IP. Therefore, we need an easy, scalable, automated approach to give developers access to their development environment from outside the cluster.

There are a number of different ways of exposing externally facing services. A NodePort service can expose a port on every node in the cluster for your service, but every port for every service across namespaces has to be unique. A Loadbalancer service allows for duplicate ports, but only by giving every service an external IP, which is also not scalable.

An automatic and scalable solution can be found by using Ingress controllers. These run as a single (scaled) deployment for the entire cluster, and effectively run an NGINX proxy. To allow applications across namespaces to expose their services to the outside world, the Application creates an Ingress Object. This is a resource within Kubernetes that is monitored by the NGINX proxy. It defines a structure for virtual host-based routing. Our Application operator has control over the configuration of this resource. This means that once an Application "sampleapplication" in the namespace "johndoe" is configured to expose itself, a user can navigate to `http://sampleapplication.johndoe.clusterdomainname`.

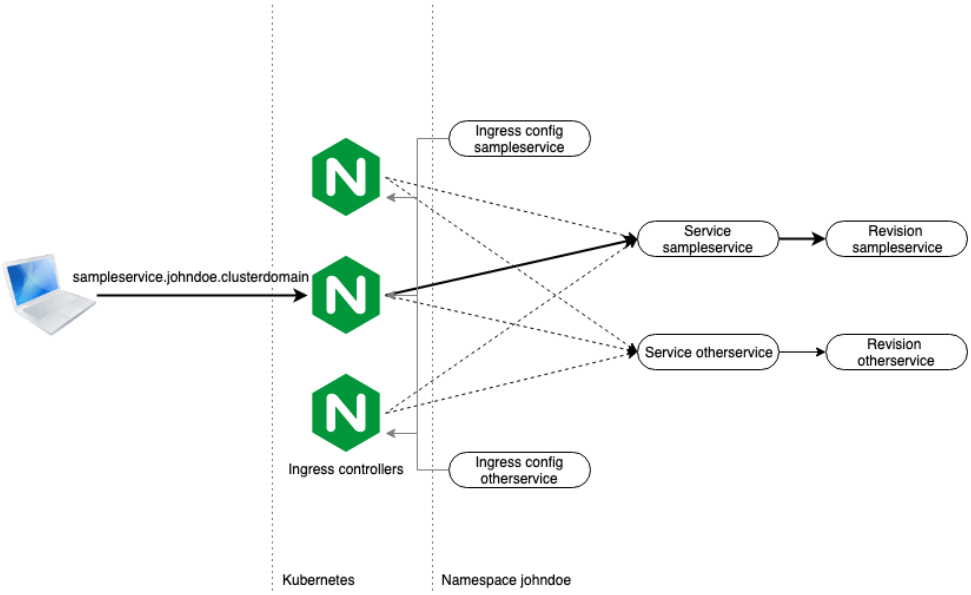

Figure 5.3: How an ingress controller routes traffic to the correct service

5.6 Adding a new service

In this section, we show what's needed to add a service from any codebase to our development cluster. Adding a new service to the cluster is fairly trivial. There are two components needed to make it work: (1) A Dockerfile capable of building a container image with the built service running, and (2) an Application configuration file to configure how the development cluster needs to use this Dockerfile, and what runtime configuration needs to be supplied when the service runs, such as ports and environment variables. We show an example of both of these configuration files in Example 5.4 and Example 5.5. We will discuss some best practices for this configuration in Section 7.5.

```
FROM golang@sha256:8cc1c0f534c0fef088f8fe09edc404f6ff4f729745b85deae5510bfd4c157fb2
  as builder
ENV GO111MODULE=on
RUN addgroup -g 1000 -S user && \
    adduser -u 1000 -S user -G user

WORKDIR /go/src/example.com/newservice
RUN chown user:user /go/src/example.com/newservice

USER user

COPY --chown=user go.mod .
COPY --chown=user go.sum .
RUN go mod download
COPY --chown=user . .
RUN go mod vendor

# Build the binary
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -ldflags="-w -s" -o
  /go/bin/newservice cmd/app/main.go

FROM scratch
# Import from builder.
COPY --from=builder /etc/passwd /etc/passwd

# Use an unprivileged user.
USER user

# Copy our static executable
COPY --from=builder /go/bin/newservice /go/bin/newservice
EXPOSE 5000
ENTRYPOINT ["/go/bin/newservice"]
```

Example 5.4: Example of a Service's Dockerfile

```
apiVersion: adyen.com/v1beta1
kind: Application
metadata:
  name: newservice
spec:
  service: true
  external: false
  buildTemplate:
    fromVolume:
      volumeName: codebase
    buildSteps:
      - kaniko:
          dockerfile: ./Dockerfile
          buildContext: /workdir/src/newservice
          destination: yourregistry/newservice
          registrySecretVolume: kaniko-secret
  revisionTemplate:
    count: 1
    baseImage: yourregistry/newservice
    port: 5000
    env:
      - name: PORT
```

```
23          value: "5000"
24    volumes:
25      - name: kaniko-secret
26        secret:
27          secretName: harbor-registry
28          items:
29            - key: .dockerconfigjson
30              path: .docker/config.json
```

Example 5.5: Example of the configuration of an Application resource

For our Dockerfile, we'll go into an overview of the best techniques we've found in Section 7.5. For the Application specification, only four questions are truly important:

Should this service be able to communicate with other services in the workspace? If so, `service` should be set to true

Should this service be available with users or external applications over HTTP? If so, `external` should be true. Note this will create a hostname of `servicename.namespace.clusterdomain.com`.

What needs to happen to go from source to a built Docker image? In the `buildSteps` you can configure any container to be run, as well as a Kaniko process that will actually build and publish the image. This means it is possible to do pre- or postprocessing via the container steps, or integrate other means of building the container image if Kaniko doesn't work for the build process of this service. Note this also means that the Build resource can be used, even if the end result isn't (directly) a Docker image. This leaves room for further development and experimentation.

What configuration should be passed to a container image when it boots? This can be any runtime configuration such as environment variables, deploying multiple copies of the service or opening up ports for the service to send and receive traffic with.

5.7 Example: The new development workflow

With this system in place, let's compare in what aspects the development workflow has changed. First of all, let's revisit Alice's bugfix in Section 4.1.

To spin up a new environment, Alice first pulls in the latest version of her system to version control. If Alice is doing multiple tasks at once, she can separate out her tasks in different git worktrees[1]. Once she has a worktree set up with the latest version of the codebase, Alice calls the gatekeeper service to request a new workspace. Her client is configured to work on workspace `alice-a9df4b`. Her client then configures the workspace with all her services. Since Alice hasn't made any changes yet to any of her services, the client can launch the services as already built by the continuous integration process. This means that within seconds of her workspace booting, all her services start up and are ready for Alice to debug the issue. This means that while Alice just checked out

[1] `https://git-scm.com/docs/git-worktree`

an entirely new version of the platform compared to what she may have been working on before, because the services were already built, she can immediately start reproducing the bug. This means Alice will reach her Edit-Compile-Test loop faster.

During that loop, every time Alice has made a change she wants to test, she kicks off the synchronization and rebuilding command. This copies over all her changes to her workspace and rebuilds and redeploys the targeted services. If Alice gets stuck, she can share her environment's frontend with a colleague to discuss and reproduce the case, whether it is one desk over or in a different country over a telephone call, the colleague can navigate Alice's workspace on his own, discuss the change even before committing it. This way Alice can get early feedback if she desires. Once her fix is ready, Alice commits her changes. This kicks off the continuous integration, which can benefit from Alice's latest build by reusing layers from the cache. This means it can give feedback faster for Alice and whoever will review Alice's change.

6

Evaluation

In order to evaluate our system, we simulate various forms of developer activity. To evaluate our system, we test it against an open source microservices demo project from Google called HipsterShop[1]. The purpose of this project is to show the versatility of cloud-native applications. Because of this, the various microservices in the project are built in a variety of different programming languages (Go, Java, C#, Javascript and Python) and frameworks. First of all, this helps us illustrate the added flexibility of our development platform: As long as a service can be built in a container, our system can handle the process. In the case of HipsterShop, there is an obvious overkill of different techniques and programming languages for a single system, but it does help showing the flexibility of our platform. To give an idea of the HipsterShop application, Figure 6.1 shows the architecture diagram for this application. The HipsterShop application consists of 10 services that communicate and together create a webshop. The services are depicted in square boxes in the architecture diagram.

To integrate the HipsterShop application in our platform, we add the configuration as described in Appendix A's Example A.1. This configuration, in combination with the pre-existing Dockerfiles for every service, is the only thing needed for HipsterShop to integrate with our platform. As a note, we will not discuss Challenge III in this chapter, but rather in the Discussion in Chapter 7.

[1] `https://github.com/GoogleCloudPlatform/microservices-demo`

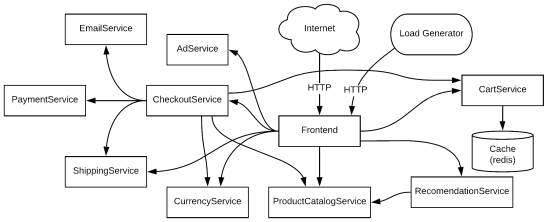

Figure 6.1: The architecture diagram of HipsterShop, the demo project we evaluate our platform with.

6.1 Challenge I: Local build time

In Section 4.2, we found that one of the main bottlenecks for developer's productivity is developers having to wait for their build tool to build a full set of services for them. Large software systems that have many services under active development suffer from this issue particularly, since building all services in such a system is a lot of computing work, and a developer's laptop only has a limited pool of resources. Because of this bottleneck, we defined our first challenge for the development cluster to be able to quickly build a full set of services for a particular software system. To evaluate how well our cluster can handle this task, we pose the first research question:

RQ 1 How can our development cluster help us speed up the build process of a large software system?

As we have discussed in Chapter 1, developers need a relatively large amount of computing power during their build processes, but this requirement does not extend to all other parts of a developer's activities. For example, the actual process of writing code doesn't need much computational power. Our development cluster is designed to be a solution that can provide large amounts of (distributed) computing power, by spreading the work of building all services over different nodes in the cluster, and therefore making use of much larger and stronger infrastructure to build the services. Because developers

We note that our cluster, besides being able to distribute multiple build processes across nodes, also heavily relies on the concept of artefact reuse. We will go into this concept more in Section 7.1, but for this test we are interested in build performance for when a developer does need to rebuild multiple services.

To measure how much faster we can build a set of services, we will use our HipsterShop demo project. We first set a baseline measurement. We take a sample of revisions, and build all containers locally on a (relatively powerful) laptop. We measure the time every service took to build, as well as the total time required to build all services. For these runs, the docker process had access to 10 cores and 16 GB of memory. An overview of average build times is visible in Figure 6.2. These are build statistics of services over the last 70 commits of the HipsterShop project. All builds of a particular revision were started at the same time.

Due to the distributed nature of a cluster, as well as its deployment in a datacenter with high-speed connections to third party sources for external dependencies, we hypothesize our development cluster should be able to build services significantly faster.

To evaluate the performance of our platform's build setup and test our hypotheses, we ran 2794 builds of services of the HipsterShop project, spread over 392 commits. For every build, we recorded the commit hash, start- and endtime, duration, result and digest of the built image. We also pushed the artefact to a registry for re-use in other tests.

Protocol 1 Building services in a range of commits for a system

Inputs. List of commit hashes C
Goal. Build all existing services in all commits with hash $i \in C$.
The protocol:

1. **Setup.** Prepare template T which configures Build resources for all services with placeholders for the commit hash. Every Build resource starts with a container that checks if the service has a *Dockerfile* available for the commit and cancels the build immediately if it does not.
2. **For all** $i \in C$
   (a) Request a new namespace N from gatekeeper service
   (b) Inject commit hash i in template T
   (c) Apply build configuration T_i to namespace N
   (d) Wait until no more builds are in a `Pending` phase, e.g. all builds have a terminal phase of either `Succeeded` or `Failed`.
   (e) Record build statistics (service name, namespace, commit hash, terminal phase, artefact digest, start time, end time and duration) from all Build resources.
   (f) Delete namespace N

Of these runs, 2003 builds were successful in creating an artefact. An aggregrated overview of the different builds of services can be found in Figure 6.3.

We note the low number of successful builds for the cartservice. Upon inspection of log files, this turned out to be an issue between the (C#) compiler's debug output interfering with the cache. Building the same image without the build cache works, but since it's not relevant for this test we have not built the full history again for this service.

Another interesting observation during this test is the high number of cache misses, which required a rebuild of the image. Upon inspection of these cache misses, we observed changes in dependencies that were not version locked. Additionally, some package managers (apt-get in particular) fetched lists of latest versions in its repository on an `apt-get update`. Because this list updates whenever a package updates, this caused a lot of cache misses. These issues could be avoided by building a base image with binaries from repositories, and assigning a version to these dependencies that way. Additionally, the Dockerfile could also be modified to not store the indexed lists of dependencies in the image.

We've executed these builds on a test cluster with 5 nodes, each with 2 CPUs and 4 GB of memory, which is a relatively limited setup, especially compared to our developer's

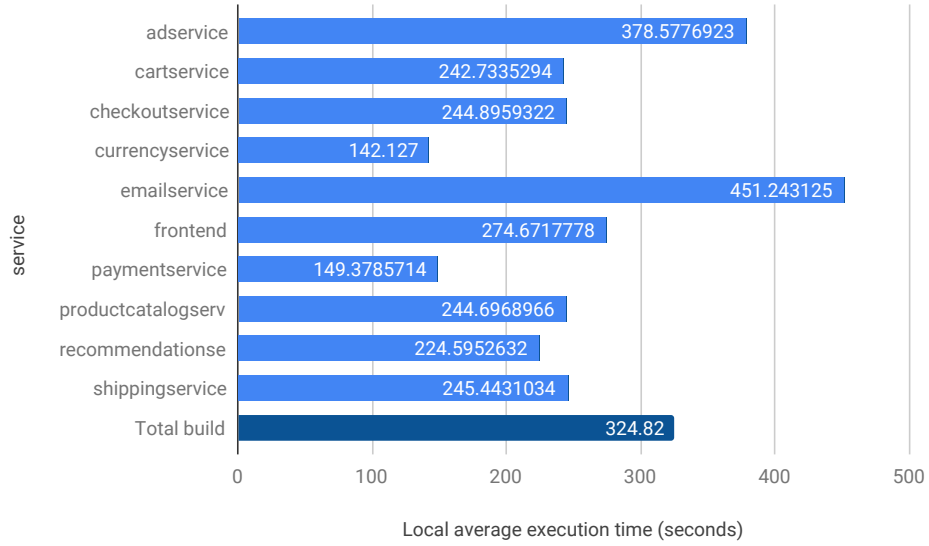

Figure 6.2: The average build times on a developer's laptop

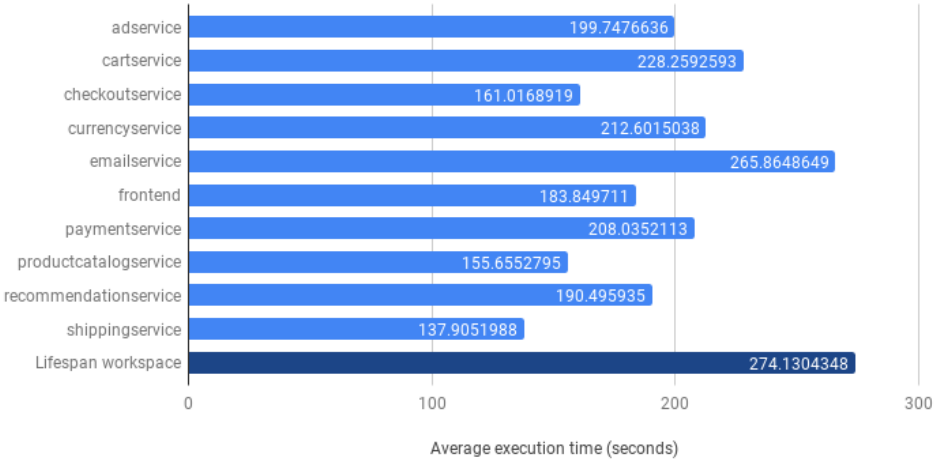

Figure 6.3: The average build times on the developer cluster

laptop. However, it is enough to show the general idea of our system. We note that in a real world scenario with many developers, the nodes can be more taxed, so it is advisable to monitor the load on the cluster and make sure enough computing capacity is available. In the case of deployment on a cloud environment like AWS, the cluster could even be dynamically resized if needed. Regardless, Kubernetes helps us by scheduling builds on nodes that have the most capacity available[2].

Observation 1 The development cluster is able to help developers build faster by providing strong computing hardware on demand, and distributing the build processes of various services over the nodes in the cluster to further speed up the build by not having build processes interfere with each other.

Observation 2 Builds relying on external dependencies without version locking or with indexing capabilities often rebuilt their services without application logic changing.

6.2 Challenge II: Running a local environment

In Chapter 1, we noted that for Adyen's codebase, it was infeasible to run a full system of services due to the size of the system. Since the HipsterShop application is a demo application, it is obviously way smaller. However, we can simulate launching a larger system by duplicating services under different names and launching those together.

RQ 2 How quickly can the development cluster start a large software system?

To get a decently sized simulated environment, we've taken 100 tagged revisions of the different services in the HipsterShop project, and deployed them all using a (fake) unique service name. This way, we simulate a larger software system.

We note that starting 100 services at the same time on a laptop is an intense task that takes quite a bit of memory. In fact, during our run, only 59 services were able to boot before the laptop ran out of available memory. The average boot time of the services that were able to start was 4 minutes and 22 seconds. A histogram of the start delay of the booted services is available in Figure 6.4.

On the cluster, there are a few advantages that help the development cluster deliver a better result. First of all, the total available memory in the cluster can be way higher than what ever could be in a developer's laptop. Second, many nodes sharing the burden of launching the services make for little work to be done per node, so this peak of computing power required is spread over different physical machines and can be dealt with fairly easily. During our test run, our cluster was configured for with our more powerful nodes (5 nodes, with 8 CPUs and 32 GB of memory each). Remarkably, the cluster had no issues with booting all 100 services in a very short time window. Similar to the experiment on the developer's laptop, we have a histogram of the start delay available in Figure 6.5. Note that the values on the x-axis here are significantly smaller, with timings in the order of seconds, not minutes. In fact, **all 100 services in the cluster booted within a minute, at an average start delay of 16 seconds.**

This means that instead of the developer having to wait for (in the case of the slowest service) 13 minutes, all services are running within a minute. Also, the developer is actually

[2] `https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes/`

able to run the full software system, instead of having to resort to managing a runnable subset of services for their work. This shows the power of our development cluster, that by distributing the workload over multiple nodes, we can run way more complex workloads, and reduce our bottleneck of booting that system to a fraction of the time required for a laptop.

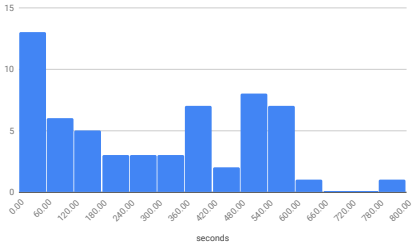

Figure 6.4: Histogram of start delay of services on a developer's laptop

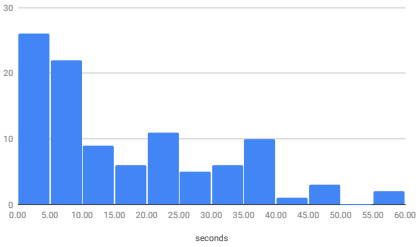

Figure 6.5: Histogram of start delay of services on the development cluster

Observation 3 Starting up even a huge system with many services will only take time in the order of seconds on the development cluster.

Observation 4 Running a system with many services is no problem for the development cluster, where it is impossible for a laptop to run the same collection of services.

6.3 Challenge IV: Resource exhaustion test

Before we evaluate how well our development cluster can grow with a software system in the next section, we are interested in a worst-case scenario. We would like to see what happens when our development cluster is put under too big of a load by developers trying to run their workloads on the cluster, and explore ways on how we can prevent these scenarios from occurring. We therefore pose the research questions:

RQ 3 What happens when the load on our development cluster becomes too big?

RQ 4 What can we do to prevent cluster failure?

To simulate an extreme load scenario, we use a single revision of the HipsterShop project and use pre-built images. This gives us a controlled load to test the cluster with. To find an answer to **RQ III**, we deploy multiple workspaces running a full HipsterShop system on the development cluster, and monitor its resources.

We have performed this experiment in two cluster configurations. The first configuration consists of 5 nodes, each with 2 CPUs and 4 GB of memory. The second configuration also consists of 5 nodes, but these each have 8 CPUs and 32 GB of memory.

Because we were interested in failure scenarios instead of degraded performance in this test, we hypothesized node memory would be the limiting factor. Out-of-memory errors will block or even crash processes that try to allocate more memory when a node's memory is full with data from other processes.

With both clusters, we deployed increasingly large amount of services and monitored the health of the processes and cluster nodes. The smaller cluster encountered its first node failure after 400 services were deployed to the 5 nodes. As we hypothesized, memory was the limiting factor. Once the cluster received signals from the failing node that its processes were dying due to out-of-memory errors, the Kubernetes API server disabled further scheduling on that node to try and reschedule the failing workloads elsewhere in the cluster, and let the failing node recover. However, with the amount of workloads deployed to the cluster, we observed a cascading failure.

For the larger cluster, we applied the same protocol. However, this time, after 530 deployed services the API server stopped scheduling workloads on any of the nodes. Upon inspection, it became clear we hit the `--max-pods` limit Kubernetes puts in place to prevent too many processes running on a particular node at once. This is one of Kubernetes' safety limits, and in this case we could clearly observe existing workloads kept running healthily, while new workloads were queued up, waiting for more capacity to become available.

Besides the `--max-pods` safety limit, Kubernetes has more ways of preventing failures such as the one we observed. Workloads can be given maximum budgets and/or reservations for resources. These help the Kubernetes scheduler make an informed decision on whether or not to schedule a new workload on a cluster under high load. Upon repeating the experiment on the small cluster, this time with these resource budgets in place and set to 128 MB per payload, we see the scheduler stop scheduling nodes after XXX deployments, to prevent overcommitting resources.

Observation 5 A cluster that runs out of resources without safety measures in place will cause node failures and in some cases cause cascading failures, grinding workloads to a halt in the cluster.

Observation 6 With the proper safety measures (such as maximum pod counts and resource limits) in place, the cluster will remain healthy by delaying the scheduling of new workloads until capacity becomes available. This allows for delayed, but healthy workloads on the cluster, until cluster administrators can either provide more capacity or reduce total workload in other ways.

Observation 7 Regularly auditing the limits set on these safety measures can help provide more security in preventing cluster failure.

6.4 Challenge IV: Cluster scalability for resource-intensive workloads

Since the development cluster should scale with the growth of the number of services for every developer and with the number of developers, we need to verify how our system scales with bigger workloads. A big factor here is how the cluster responds to a larger number of concurrent build processes.

RQ 5 How well can our development cluster scale with a growing software system?

To test this, we took our cluster with powerful nodes (8 core CPU and 32 GB of memory each), and simulated large peaks in build workloads. Note we simulate a worst-case scenario here, where a large number of developers all build all services in their system at the same time. Therefore, the computational load per workspace is way larger than we expect from actual developers in a real world scenario. With this test, we want to see how the cluster behaves under different configurations, and see if we can find a predictable relation between the load on a system and execution performance, given different sizes of the cluster. Since we expect to be able to schedule the same number of workloads on every node, we hypothesize that the development cluster scales it's capacity for resource-intensive workloads linearly with the number of nodes (with identical specifications) available in the cluster.

To test our hypothesis, we take our cluster, and "cordon off" all but a specified number of nodes. A cordoned node is excluded from scheduling workloads, and therefore will never accept new workloads. This way, we test different numbers of nodes in the cluster against different sizes of simulated workloads.

Protocol 2 Cluster scalability for growth in number of developers

Inputs. Commit hash of project h, number of active nodes in the cluster available for scheduling s, template T which contains Build templates for all services in the HipsterShop project.

Goal. Compare how the average build time grows for different values of s to see how well our cluster scales for larger number of developers.

The protocol:

1. **For** $s \in$ **{1,2,4}**
   (a) **Setup.** Disable scheduling on all but s nodes using `kubectl cordon`.
   (b) **For** i **times with** $i \in$ **{1,5,10,15,20,25,30,35}:**
      i. request a new namespace N_i from gatekeeper service
      ii. Inject image hash j in template T
      iii. Apply build configuration T_j to namespace N_i
   (c) **Record** After all builds are completed, record the average build duration for every pair of s and i.
2. **Compare** Plot average build durations and compare the trend lines for different values of s

---

The experiment ran a total of 2080 builds of services in the HipsterShop project under the various scenarios. We've plotted the average build time per scenario in Table 6.1 and plotted it in Figure 6.6. The "N/A" entries are situations where the amount of builds exceeds the `--max-pods` safety barrier set by Kubernetes.

In the graph, we can clearly see an (as good as) linear relation between the number of builds scheduled on the cluster, and the average execution time in every configuration.

# total builds	Avg. build time (s): 1 node	Avg. build time (s): 2 nodes	Avg. build time (s): 4 nodes
10	140	139.50	134.80
50	567.34	261.94	174.56
100	995.23	497.89	257.76
150	N/A	696.80	362.79
200	N/A	1002.93	518.46
250	N/A	N/A	657.33
300	N/A	N/A	812.78
350	N/A	N/A	904.25

Table 6.1: Average build time of various sizes of builds, on various sizes of clusters

However, what is more interesting is the relation between the different setups. We note that when the nodes available for scheduling double in capacity, it also takes almost twice as many builds worth of workload before the average build time hits the same mark. Our hypothesis therefore looks fairly accurate, but some of the biggest peak tests in our cluster of 4 nodes slowed down slightly sooner than expected. After investigating this with a couple more runs, we theorize this is due to the bandwidth of the container registry in our setup, as the node hosting this registry peaked its network traffic at a little over 70MB/s during the run. We also see significantly more retries in the logs when builds are try to push their container images to the registry. Of course, due to the setup of our experiment, a lot of builds pull and push their dependencies and artefacts around the same time causing peak loads, but making sure the infrastructure such as the container registry is set up to be highly available is critical to get the most out of a scaling cluster.

Observation 8 The capacity of the cluster scales almost linear with the number of nodes in the cluster.

Observation 9 The loss compared to exact linear scaling can be mostly attributed to network-based bottlenecks. This can be mitigated by properly caching dependencies in separate layers and by running a High-Availability registry.

Observation 10 We observed significant issues in the cluster in experiments that depleted the nodes' available memory, with nodes crashing, and their workloads getting rescheduled on other nodes, increasing their load as well.

Observation 11 In experiments that depleted the nodes from their available computing power, but preserved enough memory, processes simply slowed down and were affected less critically compared to an out-of-memory failure. In general, **sensible budgets and limits for workloads prevent these issues all together by waiting to schedule superfluous workloads during peak loads.**

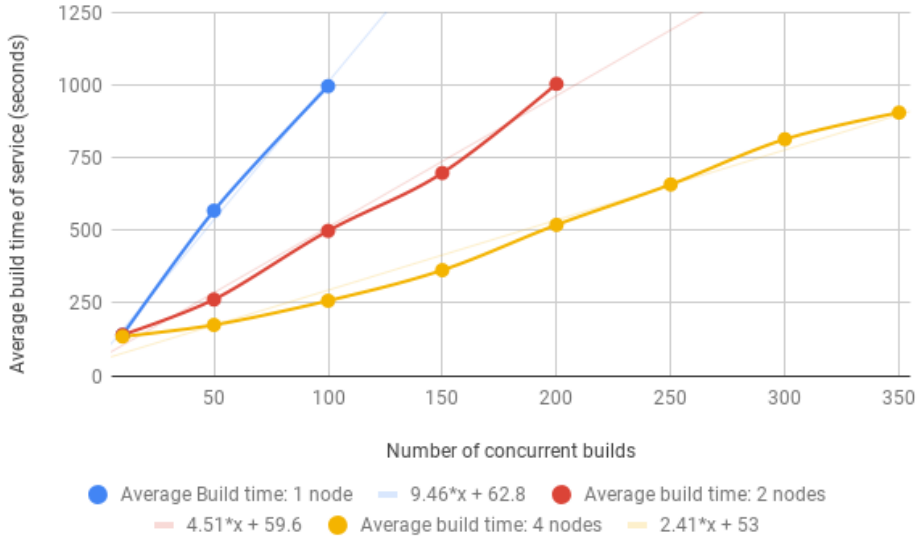

Figure 6.6: An overview of average build time under different loads for different size clusters

6.5 Challenge V: Deploying pre-commit work

To verify deploying pre-commit work is also working as expected, we simulate developer's work by replaying changes from a past commit on top of its previous parent. This effectively simulates the end-result of a developer's work on a task. In reality, the developer would possibly build intermediate changes a few times, but since it follows the same process every time, we simplify this to a single change.

RQ 6 Can a developer identify a bug, and build and test their fix on the development cluster more effectively?

Protocol 3 Deploying workspace and making changes

Inputs. Commit hash of project h, number of active nodes in the cluster available for scheduling s, template T which contains Build templates for all services in the HipsterShop project.
Goal. Simulate how a developer would start up his system and make iterative changes.
The protocol:

1. Request new workspace N from gatekeeper
2. Deploy system based off of last upstream commit in the git tree
3. Synchronize changes made to codebase to workspace N
4. Trigger build of changed services
5. Confirm new version is deployed

We've run this test, simulating the work done in commit `f2769955`[3] of the HipsterShop project, which changes a CVV input field in the frontend service's checkout page. We first create a new workspace and launch all services from `f2f382f6b`, the parent commit from our change. This simulates the developer having done a git pull and built his local system. Note that since we We then replicate the change from our test commit, synchronize over the source code and rebuild and redeploy the frontend service. We then verify the frontend service has been updated by navigating to the page (see Figure 6.7 and Figure 6.8). We show an overview of timings in Table 6.2

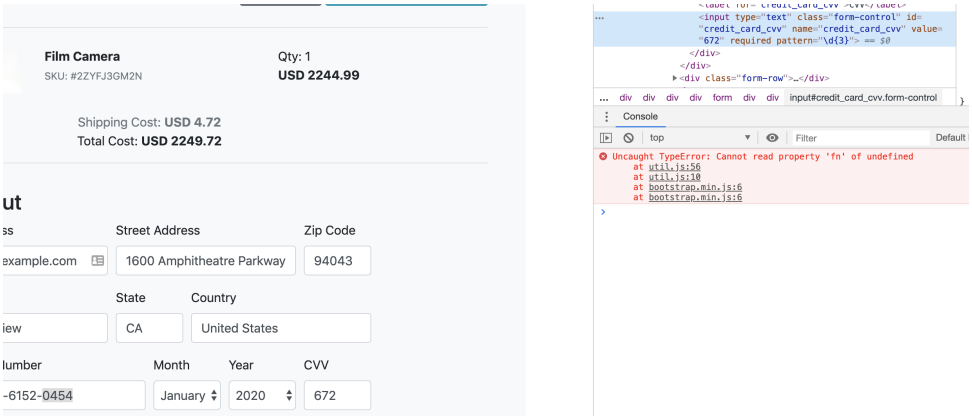

Figure 6.7: A screenshot of the frontend service's cart page before the change in our example commit

[3] `https://github.com/GoogleCloudPlatform/microservices-demo/commit/f276995585251b7b88554ff563b41e857a12d2dd`

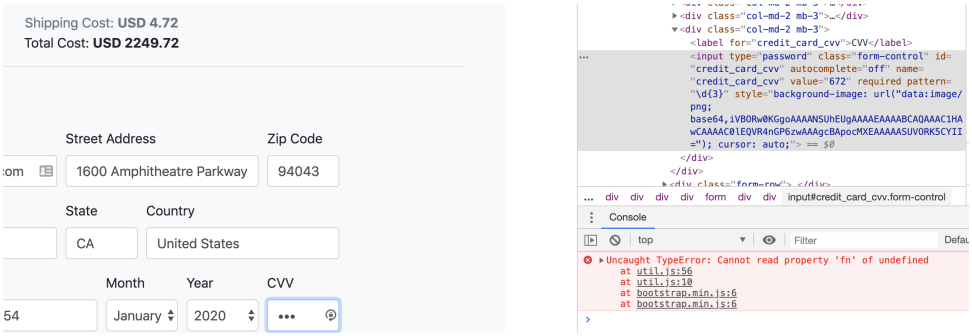

Figure 6.8: A screenshot of the frontend service's cart page after the change in our example commit. Note the change in CVV field, both in the inspector and visible on the page

Action	Duration devcluster (s)	Duration local (s)	Speedup
Creating new workspace	3	N/A	N/A
Starting local system	22	334	15.18 x
Synchronizing code change	2	N/A	N/A
Rebuild of changed service	118	90	0.76x
Total	145	424	2.92x

Table 6.2: Overview of timings during the experiment

We see that especially setting up our local environment is extremely fast, because we can simply pull in existing images and boot them. Synchronizing the code change is almost negligible for small changesets. However, rebuilding a single service is not quite as fast, especially compared to the incremental compilation of a local build. This is largely due to our system not allowing for incremental compilation, as we are building from a clean source every time. This means that whilst the cluster an re-use the dependencies for this service, it must run the full build process for the source files, instead of just the changed files. This step in the process can be sped up by combining our platform with remote caching features from the underlying build tools. The choice for this clean build approach is made by design, in order to not lock in our setup to a particular programming language or build tool. If desired, one could even create additional build steps as described in Section 5.4 to fetch and reuse bytecode from previous runs, however, we have not included such an approach in our current system.

Even without any compiler-level caching or re-use of compiled sources, in this case the total waiting time of this workflow was way less compared to local development. This means the developer gets (1) a faster startup, allowing them to begin their work way faster, and (2) a fully up to date system to test his change on, such that any interaction with recently committed code from other developers can also be observed.

Observation 12 Setting up a working system from a known revision is extremely fast compared to classic local development

Observation 13 Rebuilding a service from scratch takes the most time (on the developer

cluster) in this workflow, but can be optimized by proper Dockerfile design and use of build tool remote caching mechanics.

6.6 Challenge VI: Network traffic evaluation

A major concern raised at Adyen was the expected network load of continuously shipping a codebase to a remote cluster. To verify our development cluster has covered this concern, we ask:

RQ 7 How high is the network load between a developer and the development cluster during active development?

To answer this question, we highlight the most important sources of network traffic between the developer and the cluster. Note that we are looking for a network load low enough such that an office of developers can comfortably connect to the development cluster to do their work.

Source synchronization As we've discussed in Section 5.3, our source code synchronization process has been designed with this network load in mind. We observe that any work done on the source code is only modifying, adding or deleting small parts of the codebase, compared to the previous version. Since this previous version will always be a state of the system that is checked in to source control, we can minimize the data we need to send to the development cluster ourselves. In other words: A developer does not have to synchronize their entire codebase to their workspace on the cluster, but instead can send only a set of updates, compared to a previous version. The size of these changes is **approximately the size of a commit**, since we're only sending the changes and some metadata to negotiate the starting point of the change. This network load is in the order of a few kilobytes for every synchronization, occasionally growing to a megabyte or two for something like images or larger datasets.

Interacting with the software system Another additional factor of network load is the interaction a user has with their running version of the system. This will generally consist of interacting with web services or otherwise transferring payloads (for Adyen this could be test transactions, capture requests etc). General usage during active development will roughly be equivalent to browsing a web page.

Log streaming One of the possible ways of reading logs for deployed services is to stream them directly to the user. Since this is a stream of text, the load of network traffic will generally be pretty manageable. Should the rate of logging be high enough to become a problem, a potential solution could be to run a log aggregation tool in the development cluster that will pre-process and filter the logs that need to be sent to a developer. In the worst-case scenario, this may be considered equivalent to streaming a YouTube video. Note also that not every developer will need or want this rate of logging, and many will not stream the logs continuously but rather consult them on demand.

Cluster management interaction The last way a developer will have traffic going to and from the cluster is the traffic needed to control the cluster. Generally, this will

not be more than applying configuration files, which is equal if not smaller than the source synchronization payload we discussed earlier.

Besides these efforts to minimize network traffic between the developer and the cluster, we also note that some traffic, such as fetching dependencies, is no longer needed for the developer's laptop. All in all, we can conclude network traffic between the developer and the development cluster is minimal.

6.6.1 Datacenter traffic

To lighten the load of our connection between the user and the cluster, we have made a few key decisions. Because of that, we also discuss the network traffic in the datacenter.

Cluster management and application chatter First of all, because our cluster will consist of multiple nodes, there is a fair bit of communication between them, both by the cluster management services, and by application traffic.

Artefact transfer Because services may be built on different nodes they are run, we distribute all images via our container registry. This means that every build will result in an upload, and most deployments will require a download of the image. Depending on the application, the dependencies bundled in the container image, and the availability of cached image layers on the destination the size of this transfer may differ significantly. It is therefore advisable to (1) keep images as small as possible and (2) re-use base layers, for example by using a shared base image. See our recommendations in Section 7.5 in this aspect. We also noticed in Section 6.4 that under high load, the registry can become a bottleneck for the throughput of the system.

Version control traffic Because of our source synchronization optimizations between the developer and the cluster, we are forced to make fresh clones of our projects to the cluster more often. This might somewhat impact the load on the version control servers. We've opted for this optimization because (1) it doesn't matter much for the cluster itself, as we need to get the source code on the cluster regardless, and (2) having a larger data transfer between two servers in a datacenter has significantly less impact due to the high bandwidth between servers in a datacenter.

An additional concern for network traffic is the geographic distribution of nodes in the cluster. If nodes are distributed over multiple datacenters in different parts of the world, this may impact the data transfer speeds. To mitigate this, cluster administrators could use `nodeAffinity`[4] annotations to indicate a manual preference of nodes to deploy a set of services on.

Observation 14 Network traffic between the developer and the cluster has been kept to a minimum in our environment, particularly through minimizing the data transfer needed to submit a code change to the development cluster.

[4] `https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity`

Observation 15 The general network requirements of a development cluster (both between nodes and between edge services such as container registries, version control and dependency sources) are not to be underestimated, particularly at scale. However, by managing a strong physical network, making sure services are highly available and making clever use of cached resources this load can be managed effectively.

7

Discussion

In the previous chapter, we have seen a number of experiments and evaluations of the development cluster, with respect to the challenges we've defined in Chapter 1. In this chapter, we discuss our findings with respect to Challenge III, as we have not ran a formal experiment for this challenge. We'll also go in various learnings and recommendations for anyone developing cloud-native applications.

7.1 Challenge III: Decreasing build avoidance

Build avoidance is the concept of avoiding (re)building parts of a system to not get blocked by a long build process. A known best practice in software development in teams is to make small, iterative changes to a system and share these as early as possible with others, either via code review and/or by letting other developers run the changed code in their own system, so they can give early feedback. This last concept is known as "dogfooding". However, if developers go as long as possible without rebuilding services they do not actually work on, they do not get exposed to these recent changes, and will therefore not give feedback early enough. Because this challenge is a psychological issue as well as a technical one, we were not able to run an experiment to see the effects of our development cluster on build avoidance yet.

Instead, we will discuss what our development cluster can do to mitigate the problem of build avoidance. The main way this cluster can help developers in this aspect is by the ability to re-use build artefacts (in the form of container images) directly. As soon as either a developer or a continuous integration pipeline has built the image of the changed service, any developer can simply pull and run this version of the service on their workspace in the development cluster. They can skip the build step that would be needed for their local setup.

Using these container images, the developer can simply pull and rebase their codebase from their version control system, and request the development cluster to update the services in their workspace. If the developer hasn't made changes to a particular service himself, the container image containing the other developer's changes can simply be pulled and started immediately. If the developer has made changes themselves as well, the next time they rebuild their own service will incorporate those changes as well.

Therefore, we expect our development cluster to be a major help in decreasing build avoidance and exposing recent changes to other developers sooner.

Observation 16 Because newer revisions of unchanged services are pre-built, updating a developer's setup to the latest revision can generally happen in a matter of seconds to a few minutes, depending on the changes in the developers workspace.

Observation 17 Services that the developer is actively working on will still need a re-compilation, but this will happen when the developer builds the next version of his service with changes.

7.2 Contribution to Kaniko

During the work done with Kaniko in this thesis, it became clear that Kaniko was missing a feature central to how our pipeline works. Specifically, Kaniko allows to specify a destination to where an image can be pushed by tag, but there is no way to conveniently and consistently get the digest (cryptographic hashes that represent a container image) of the image built by the Kaniko container. This is unfortunate, as tags can be reassigned to different container image versions, potentially causing issues or even security risks. In fact, this means an actor with push rights to the container registry could inject a payload in any new image, and move the tag in the registry to this compromised image. Because of this threat model, our development platform works with digests to ensure we're always running the code we are expecting.

To get the ability to work with digests through the entire pipeline, we needed a new feature added to Kaniko. By writing the digest of the built container image to a specific file (`/dev/termination-log`), Kubernetes can pick up the image digest in its container termination message. That allows us to extract the digest on completion of the build, and make sure we're running the correct image. To make this possible, a contribution to Kaniko[1] was made. This contribution adds a flag, that if set writes the final image's digest to a file specified in the flag's argument. We use this feature in our Build controller to update the Build resource's status with the digest, so we can use it during runtime. This is further illustrated in Section 5.5.

7.3 Application architecture

As we've said in Section 1.2.1, Adyen's codebase is a Service-Based Architecture. The HipsterShop project also has a number of different services, separated even further by not sharing any underlying modules, and even having services implemented in different programming languages. Of course, service based architectures, while very popular, are not the only Application architectures. For some software systems, it may be very beneficial to keep a monolithic structure. While our development cluster is geared more towards the service-based model, which would benefit more from the ability to distribute building, other application architectures would still be able to use the development cluster to benefit from the elastic infrastructure, the ability to share environments and the ability to switch between tasks while keeping workspaces running. Having ready-to-go artefacts will work

[1] `https://github.com/GoogleContainerTools/kaniko/pull/655`

with clean working directories, but after even a single change the full system will need to be rebuilt in that scenario. We theorize that a system closely following the Twelve-factor app model[44] or other cloud-native methodologies may have the best results working on this development platform as they already have (a large part of) the practices upon which we built our platform.

7.4 Development Cluster, Build tools and compiler-level caching

As we have seen in Section 2.4, a strategy often taken by big organizations is to have a caching system on the level of the build tool. At first glance, it may seem our system must compete against build tools such as Bazel or Buck. However, what is important to note is the difference in scope and the interaction between these scopes. Bazel and Buck work on a module level, where it is providing scalability by building and caching artefacts of code and dependencies. Our platform concerns itself about services as a whole, and defers the exact build process of each service to the build tool inside the container. This allows build tools to use their own (remote) caching solutions, as they will have the best knowledge of how to manage the exact build. Our solution aims to have a development cluster that allows the developer to build and deploy their services and have their own environment to run their version of a software system on.

7.5 Designing container build recipes

To start using the development cluster a developer needs a build recipe (in our case a Dockerfile) and an Application resource configuration. While the Application resource is pretty straightforward, and only needs a bit of application-specific configuration, the Dockerfile is a bit more complicated to set up effectively. We offer a few pieces of advice, which are reflected in the example Dockerfile in Section 5.6:

Start with a base image By maintaining a base image for every programming language/runtime setup, upgrading runtime dependencies will be easier. For example, we have built two (see the next point) base images, one containing the Java JDK, Gradle, and other build time depenencies, and one containing the JDK and TomCat for use during runtime. For both of these, we can use a simple FROM statement in the Dockerfile of an application to load in a particular version of a known compatible set of dependencies. This way, every application has the power to choose its runtime environment while separating the difficulty of building that environment from the build phase of an application. For the development cluster, this also means the application's container build can potentially run without the need for administrative privileges in the container.

Use multi-stage Dockerfiles to separate run- and build dependencies In the Dockerfile for the service, we recommend the usage of multi-stage builds. This essentially gives you the ability to build something in a container, then start over and copy over files from a previous stage. This means that we can have a build- and an execute image. The build image has all build dependencies, such as the build tool, compilers

and other tools needed during the build. Then, we can copy in the built artefact to a fresh container that only has the runtime dependencies. The reason this is beneficial for the development cluster, is that the final image does not contain the source code or build dependencies, and therefore is way smaller. This helps the development cluster, as a node in the cluster that is tasked with running the built image can download a smaller payload, and may even have the execution environment's base layer (see the next point) cached.

Use container layers to minimize repeated work Layers are the building blocks of containers. They are comparable to diff files in Git, and are applied on top of each other to create the full filesystem for an image. The interesting mechanic here is that layers can be cached if their inputs do not change. By cleverly using layers to our advantage, we can reduce the work that needs to be done during building. A good example of this is the installation of dependencies. If only the files needed to fetch dependencies are loaded in a layer and then dependencies are fetched, the resulting layer can be re-used as long as dependencies are not changing. The build then can avoid the dependency fetching the dependencies in subsequent builds.

Only use version-locked dependencies One thing we noticed with the HipsterShop services is that they do not version-lock some of their dependencies. This causes a high percentage of cache misses and requires many layers to rebuild. Make sure that any external resource you pull in to the image is version-locked.

7.6 Debuggers, test runners and other IDE tools

A concern voiced by developers at Adyen is whether they would be able to use the tools they use today, for example the IntelliJ debugger, as well as specific tools that generate test traffic. As long as these tools send traffic over a network port, we can use `kubectl port-forward` to redirect the tool's request to the correct service.

We have seen in Chapter 5 how we use ingresses to expose our webservices. Other services that do not have this ingress configured cannot be externally reached. However, if a developer would for example want to attach a remote debugger, there is a second way of connecting services. By using the `kubectl port-forward` command, we can proxy any network-based traffic directly into a running pod. This means that to an external user the service is still not directly reachable, but the developer that owns the workspace has the ability to send and receive data and instructions. This means that for example an external debugger can be attached via this proxy to a service.

However, some tooling will have more issues. Especially tools that deeply integrate in an application, such as PMD, which reads Java bytecode to detect potential issues, may need more work to work effectively. While there are solutions to these issues, we put it outside of the scope of our prototype.

8

Conclusion and future work

The goal of this thesis is to see what is needed to create a scalable development platform for large-scale software systems. We explored this challenge by charting the size of the problem for a system such as that of Adyen. Based on these findings, we presented an idea for a scalable development platform, which is based on the concepts of orchestration and cloud-native architecture. We designed and implemented this platform based on previous engineering work done by the Kubernetes team, building upon their platform to create custom workflows that help developers work more effectively, by offloading build work to powerful machines and deploying these build artefacts in a developer's individual workspace. We then explored the scalability of our solution by simulating developer activity on the cluster and charting the benefits of our platform. In the process, we added an additional contribution in the form of a open source contribution to Kaniko that helps us and other integrators of Kaniko to get more information from the Kaniko tool.

During our evaluation, we've identified a few key conclusions:

1. Due to the our basis on containerization technology, re-using images that have been built by Continuous Integration or other developers can massively speed up the time needed for a developer to get their own running system.

2. Due to the distribution of compute-intensive work over multiple machines, build processes, in particular those that have to build multiple services, can cause a significant speed-up when run on our development cluster as compared to a developer's local machine.

3. Scalability of our development cluster is as simple as growing the number of nodes in the cluster. This means that our development cluster can handle both growth in size of a software system, as well as growth in the number of users.

We also note a few concerns and areas of improvement. First of all, dealing with high-velocity changes such as those during active development can be straining for supporting resources such as the container registry, which is a key service in our build-and-run workflow. Critical to the success of a development cluster such as this is the availability of

services such as these, and with all developers of a company building on such infrastructure, it becomes a single point of failure that can stop all developers from building and running their code, effectively blocking them from working. There are ways to prepare for these issues, such as segmenting the cluster into multiple clusters, running edge infrastructure such as registries with high-availability configurations and having close and active monitoring on the health of the cluster and surrounding services.

Running a local development environment in a distributed setting may also be challenging to developers, particularly in the beginning. By providing the correct tools for developers to be able to log, debug and trace what is happening in their system, we can mitigate this issue, but our current development cluster proof-of-concept does not include in-depth solutions for this yet. Basic logs and the ability to enter a container remotely are available, but more tool integration will be needed to make development even smoother.

8.1 Future work in Software Engineering research

Many concepts used in our development cluster are based off of (relatively) recent developments in cloud-native architectures. While a lot of attention from industry as well as research has focused on production-related topics such as automatic scaling, alerting and resilience to failure, there has been very little research with respect to the developer experience using these new techniques. We found a lot of cloud-native methodologies generate a lot of data, that is available at larger scales. We expect that gathering insights of developer activity will be significantly easier with the use of cloud-based tools compared to trying to gain access to individual developer machines. As the integration of software in our society continues to grow, we expect the challenge of providing a scalable way for developers to do their job will shift significantly. Our contribution can be a first step in exploring this direction by bringing the power of these cloud-based systems to developers, but many more steps, such as tools for testing, analysis, debugging and visualization still need to be taken to help software engineers stay productive in software systems at massive scale. The scientific community, in collaboration with industry, can aid massively in the development of these tools and practices.

8.2 Future engineering work for the development cluster

While this thesis describes a first version of our platform with a scalable build-and-run workflow, many more extensions to this platform can be made. We highlight a few potential extensions that can be implemented as follow-up steps to this work.

Integrate more tooling By creating and integrating more diagnostic tools, such as traffic simulators, circuit breakers that test network failure or even advanced service discovery, the development platform can aid the developer in many new ways. Having a development environment that can take away barriers and even build new ways to improve efficiency is of great value to any large-scale software system, and our platform is a solid start for this effort.

Improve traceability One of the major challenges for developers in a distributed system is traceability. Keeping track of these highly asynchronous systems can be difficult

for a developer, as messages are passed between services are not always observable by the developer.

There are multiple ways our platform could aid in solving this problem in the future. Application logs can be extracted by attaching logging "sidecars"[42] and sent to a central service in the workspace.

Another potential solution could be to not just rely on log messages, but monitor and visualize all network traffic in the user's workspace. This can be done in a similar matter, by injecting sidecars in the service pods that monitor the service's network traffic.

Both of these solutions would help developers understand the flow through the distributed system faster, catching unexpected interactions before they become a problem.

Workspace collaboration and hand-off By taking advantage of the fact that a developer's workspace has moved off of his physical machine, new ways of collaboration and even hand off are possible. In our platform, multiple users can already use a workspace's deployed services, giving users the ability to let other developers join in on interacting with their system for debugging specific scenarios, or ask for expertise from another team. In the future, the platform could expand upon this to fully transfer a workspace (including any potential changed code) to a different developer.

Workspace hand-off for Continuous Integration Building from the previous point, a Continuous Integration service running on this platform could give unique debugging abilities. A complaint frequently made about complex bugs in CI environments is that they are difficult to recreate. If a CI run would run on our platform, the exact environment could be handed over to a developer to investigate an issue with his/her build. This has a few key benefits:

1. Developers can access the exact environment the CI was running in, so state-dependent bugs are easy to inspect, not having to try and reproduce the situation first.
2. The Continuous Integration service uses the exact same platform as developers, ruling out many "it works on my machine" situations.
3. Developers that get back the results of a failing build can have the CI environment transferred back to them as a developer workspace immediately. This means the developer, who may have moved on to a new task while the CI build was running, immediately gets his exact environment for the original task in a secondary workspace. This means his barrier to switch context back to fix his bug is lowered significantly, resulting in higher developer productivity and satisfaction.

There are endless more scenarios to think of where a distributed development environment can aid the developer in their work. Today, the platform's initial implementation helps developers by offloading computationally intensive build work to a powerful environment,

and by providing a distributed runtime environment for active development, capable of more closely providing a production-like experience during development. By extending the platform in the future, application developers can become even more productive. They will be able to use more powerful tools to help them produce higher quality software. We hope our development cluster can be the first step in this process, allowing researchers to gain more insights and helping developers stay productive.

A

Evaluation configurations

Below is the full configuration of all services in the HipsterShop project, needed to integrate with our system. While it may look like a lot, the setup for every service is very similar. Good to know is that the Frontend service is the only one with `external: true` configured, so only the frontend service will get an endpoint for http traffic.

```
apiVersion: adyen.com/v1beta1
kind: Application
metadata:
  name: frontend
spec:
  service: true
  external: true
  buildTemplate:
    source:
      git:
        repository: git@github.com:GoogleCloudPlatform/
          microservices-demo.git
        commit: COMMITID
    buildSteps:
      - kaniko:
          dockerfile: ./Dockerfile
          buildContext: /workdir/src/frontend
          destination: registry2.kubernetes.gijsweterings.nl/
            library/frontend
          registrySecretVolume: kaniko-secret
  revisionTemplate:
    count: 1
    baseImage: registry2.kubernetes.gijsweterings.nl/library/
      frontend
    port: 8080
    env:
```

A

```
        - name: PORT
          value: "8080"
        - name: PRODUCT_CATALOG_SERVICE_ADDR
          value: "productcatalogservice:3550"
        - name: CURRENCY_SERVICE_ADDR
          value: "currencyservice:7000"
        - name: CART_SERVICE_ADDR
          value: "cartservice:7070"
        - name: RECOMMENDATION_SERVICE_ADDR
          value: "recommendationservice:8080"
        - name: SHIPPING_SERVICE_ADDR
          value: "shippingservice:50051"
        - name: CHECKOUT_SERVICE_ADDR
          value: "checkoutservice:5050"
        - name: AD_SERVICE_ADDR
          value: "adservice:9555"
        - name: ENABLE_PROFILER
          value: "0"
    volumes:
      - name: kaniko-secret
        secret:
          secretName: harbor-registry
          items:
            - key: .dockerconfigjson
              path: .docker/config.json
---
apiVersion: adyen.com/v1beta1
kind: Application
metadata:
  name: adservice
spec:
  service: true
  external: false
  buildTemplate:
    source:
      git:
        repository: git@github.com:GoogleCloudPlatform/
          microservices-demo.git
        commit: COMMITID
    buildSteps:
      - kaniko:
          dockerfile: ./Dockerfile
          buildContext: /workdir/src/adservice
          destination: registry2.kubernetes.gijsweterings.nl/
            library/adservice
```

```
67            registrySecretVolume: kaniko-secret
68    revisionTemplate:
69      count: 1
70      baseImage: registry2.kubernetes.gijsweterings.nl/library/
          adservice
71      port: 9555
72      env:
73        - name: PORT
74          value: "9555"
75        - name: ENABLE_PROFILER
76          value: "0"
77    volumes:
78      - name: kaniko-secret
79        secret:
80          secretName: harbor-registry
81          items:
82            - key: .dockerconfigjson
83              path: .docker/config.json
84  ---
85  apiVersion: adyen.com/v1beta1
86  kind: Application
87  metadata:
88    name: cartservice
89  spec:
90    service: true
91    external: false
92    buildTemplate:
93      source:
94        git:
95          repository: git@github.com:GoogleCloudPlatform/
              microservices-demo.git
96          commit: COMMITID
97      buildSteps:
98        - kaniko:
99            dockerfile: ./Dockerfile
100           noCache: true
101           buildContext: /workdir/src/cartservice
102           destination: registry2.kubernetes.gijsweterings.nl/
                library/cartservice
103           registrySecretVolume: kaniko-secret
104   revisionTemplate:
105     count: 1
106     baseImage: registry2.kubernetes.gijsweterings.nl/library/
          cartservice
107     port: 7070
```

A

```
    env:
      - name: PORT
        value: "7070"
      - name: REDIS_ADDR
        value: "redis-cart:6379"
      - name: LISTEN_ADDR
        value: "0.0.0.0"
      - name: ENABLE_PROFILER
        value: "0"
  volumes:
    - name: kaniko-secret
      secret:
        secretName: harbor-registry
        items:
          - key: .dockerconfigjson
            path: .docker/config.json
---
apiVersion: adyen.com/v1beta1
kind: Application
metadata:
  name: checkoutservice
spec:
  service: true
  external: false
  buildTemplate:
    source:
      git:
        repository: git@github.com:GoogleCloudPlatform/
          microservices-demo.git
        commit: COMMITID
    buildSteps:
      - kaniko:
          dockerfile: ./Dockerfile
          buildContext: /workdir/src/checkoutservice
          destination: registry2.kubernetes.gijsweterings.nl/
            library/checkoutservice
          registrySecretVolume: kaniko-secret
  revisionTemplate:
    count: 1
    baseImage: registry2.kubernetes.gijsweterings.nl/library/
      checkoutservice
    port: 5050
    env:
      - name: PORT
        value: "5050"
```

```
150         - name: PRODUCT_CATALOG_SERVICE_ADDR
151           value: "productcatalogservice:3550"
152         - name: SHIPPING_SERVICE_ADDR
153           value: "shippingservice:50051"
154         - name: PAYMENT_SERVICE_ADDR
155           value: "paymentservice:50051"
156         - name: EMAIL_SERVICE_ADDR
157           value: "emailservice:5000"
158         - name: CURRENCY_SERVICE_ADDR
159           value: "currencyservice:7000"
160         - name: CART_SERVICE_ADDR
161           value: "cartservice:7070"
162         - name: ENABLE_PROFILER
163           value: "0"
164     volumes:
165       - name: kaniko-secret
166         secret:
167           secretName: harbor-registry
168           items:
169             - key: .dockerconfigjson
170               path: .docker/config.json
171 ---
172 apiVersion: adyen.com/v1beta1
173 kind: Application
174 metadata:
175   name: currencyservice
176 spec:
177   service: true
178   external: false
179   buildTemplate:
180     source:
181       git:
182         repository: git@github.com:GoogleCloudPlatform/
              microservices-demo.git
183         commit: COMMITID
184     buildSteps:
185       - kaniko:
186           dockerfile: ./Dockerfile
187           buildContext: /workdir/src/currencyservice
188           destination: registry2.kubernetes.gijsweterings.nl/
              library/currencyservice
189           registrySecretVolume: kaniko-secret
190   revisionTemplate:
191     count: 1
```

A

```
    baseImage: registry2.kubernetes.gijsweterings.nl/library/
      currencyservice
    port: 7000
    env:
      - name: PORT
        value: "7000"
      - name: ENABLE_PROFILER
        value: "0"
  volumes:
    - name: kaniko-secret
      secret:
        secretName: harbor-registry
        items:
          - key: .dockerconfigjson
            path: .docker/config.json
---
apiVersion: adyen.com/v1beta1
kind: Application
metadata:
  name: emailservice
spec:
  service: true
  external: false
  buildTemplate:
    source:
      git:
        repository: git@github.com:GoogleCloudPlatform/
          microservices-demo.git
        commit: COMMITID
    buildSteps:
      - kaniko:
          dockerfile: ./Dockerfile
          buildContext: /workdir/src/emailservice
          destination: registry2.kubernetes.gijsweterings.nl/
            library/emailservice
          registrySecretVolume: kaniko-secret
  revisionTemplate:
    count: 1
    baseImage: registry2.kubernetes.gijsweterings.nl/library/
      emailservice
    port: 5000
    env:
      - name: PORT
        value: "5000"
      - name: ENABLE_PROFILER
```

```
233           value: "0"
234     volumes:
235       - name: kaniko-secret
236         secret:
237           secretName: harbor-registry
238           items:
239             - key: .dockerconfigjson
240               path: .docker/config.json
241 ---
242 apiVersion: adyen.com/v1beta1
243 kind: Application
244 metadata:
245   name: paymentservice
246 spec:
247   service: true
248   external: false
249   buildTemplate:
250     source:
251       git:
252         repository: git@github.com:GoogleCloudPlatform/
              microservices-demo.git
253         commit: COMMITID
254     buildSteps:
255       - kaniko:
256           dockerfile: ./Dockerfile
257           buildContext: /workdir/src/paymentservice
258           destination: registry2.kubernetes.gijsweterings.nl/
              library/paymentservice
259           registrySecretVolume: kaniko-secret
260   revisionTemplate:
261     count: 1
262     baseImage: registry2.kubernetes.gijsweterings.nl/library/
        paymentservice
263     port: 50051
264     env:
265       - name: PORT
266         value: "50051"
267       - name: ENABLE_PROFILER
268         value: "0"
269   volumes:
270     - name: kaniko-secret
271       secret:
272         secretName: harbor-registry
273         items:
274           - key: .dockerconfigjson
```

A

```
            path: .docker/config.json
---
apiVersion: adyen.com/v1beta1
kind: Application
metadata:
  name: productcatalogservice
spec:
  service: true
  external: false
  buildTemplate:
    source:
      git:
        repository: git@github.com:GoogleCloudPlatform/
          microservices-demo.git
        commit: COMMITID
    buildSteps:
      - kaniko:
          dockerfile: ./Dockerfile
          buildContext: /workdir/src/productcatalogservice
          destination: registry2.kubernetes.gijsweterings.nl/
            library/productcatalogservice
          registrySecretVolume: kaniko-secret
  revisionTemplate:
    count: 1
    baseImage: registry2.kubernetes.gijsweterings.nl/library/
      productcatalogservice
    port: 3550
    env:
      - name: PORT
        value: "3550"
      - name: ENABLE_PROFILER
        value: "0"
  volumes:
    - name: kaniko-secret
      secret:
        secretName: harbor-registry
        items:
          - key: .dockerconfigjson
            path: .docker/config.json
---
apiVersion: adyen.com/v1beta1
kind: Application
metadata:
  name: recommendationservice
spec:
```

```
317   service: true
318   external: false
319   buildTemplate:
320     source:
321       git:
322         repository: git@github.com:GoogleCloudPlatform/
              microservices-demo.git
323         commit: COMMITID
324     buildSteps:
325       - kaniko:
326           dockerfile: ./Dockerfile
327           buildContext: /workdir/src/recommendationservice
328           destination: registry2.kubernetes.gijsweterings.nl/
              library/recommendationservice
329           registrySecretVolume: kaniko-secret
330   revisionTemplate:
331     count: 1
332     baseImage: registry2.kubernetes.gijsweterings.nl/library/
        recommendationservice
333     port: 8080
334     env:
335       - name: PORT
336         value: "8080"
337       - name: PRODUCT_CATALOG_SERVICE_ADDR
338         value: "productcatalogservice:3550"
339       - name: ENABLE_PROFILER
340         value: "0"
341   volumes:
342     - name: kaniko-secret
343       secret:
344         secretName: harbor-registry
345         items:
346           - key: .dockerconfigjson
347             path: .docker/config.json
348 ---
349 apiVersion: adyen.com/v1beta1
350 kind: Application
351 metadata:
352   name: redis-cart
353 spec:
354   service: true
355   external: false
356   revisionTemplate:
357     count: 1
```

A

```
    baseImage: registry2.kubernetes.gijsweterings.nl/library/redis
    pinnedDigest: sha256:72c09617b38189123a9b360c8a1998a59be3af52759269d4c740397bd54a31f2
    port: 6379
    env:
      - name: ENABLE_PROFILER
        value: "0"
  volumes:
    - name: kaniko-secret
      secret:
        secretName: harbor-registry
        items:
          - key: .dockerconfigjson
            path: .docker/config.json
---
apiVersion: adyen.com/v1beta1
kind: Application
metadata:
  name: shippingservice
spec:
  service: true
  external: false
  buildTemplate:
    source:
      git:
        repository: git@github.com:GoogleCloudPlatform/microservices-demo.git
        commit: COMMITID
    buildSteps:
      - kaniko:
          dockerfile: ./Dockerfile
          buildContext: /workdir/src/shippingservice
          destination: registry2.kubernetes.gijsweterings.nl/library/shippingservice
          registrySecretVolume: kaniko-secret
  revisionTemplate:
    count: 1
    baseImage: registry2.kubernetes.gijsweterings.nl/library/shippingservice
    port: 50051
    env:
      - name: PORT
        value: "50051"
```

```
397         - name: ENABLE_PROFILER
398           value: "0"
399     volumes:
400       - name: kaniko-secret
401         secret:
402           secretName: harbor-registry
403           items:
404             - key: .dockerconfigjson
405               path: .docker/config.json
```

Example A.1: Template of a full build configuration for the HipsterShop project

Bibliography

References

[1] Robert J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[2] W Royce. The software lifecycle model (waterfall model). In *Proc. WESTCON*, volume 314, 1970.

[3] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[4] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[5] Moritz Beller. Toward an empirical theory of feedback-driven development. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 503–505, New York, NY, USA, 2018. ACM.

[6] Jos Winter. Increasing operational awareness using monitoring-aware ides, 2018.

[7] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122:14, 2006.

[8] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[9] Sallyann Bryant, Benedict Du Boulay, and Pablo Romero. Xp and pair programming practices. *PPIG Newsletter*, pages 17–20, 2006.

[10] Mitch Denny. The inner loop. `https://mitchdenny.com/the-inner-loop/`, 2018.

[11] Dan Lorenc. Build containers faster with cloud build with kaniko | google cloud blog, Feb 2019.

[12] André N Meyer, Thomas Fritz, Gail C Murphy, and Thomas Zimmermann. Software developers' perceptions of productivity. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 19–29. ACM, 2014.

[13] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[14] Fabian Fagerholm and Jürgen Münch. Developer experience: Concept and definition. In *Proceedings of the International Conference on Software and System Process*, ICSSP '12, pages 73–77, Piscataway, NJ, USA, 2012. IEEE Press.

[15] Margaret-Anne Storey and Alexey Zagalsky. Disrupting developer productivity one bot at a time. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 928–931, New York, NY, USA, 2016. ACM.

[16] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[17] Gloria Mark, Daniela Gudith, and Ulrich Klocke. The cost of interrupted work: More speed and stress. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 107–110, New York, NY, USA, 2008. ACM.

[18] Jürgen Cito, Philipp Leitner, Harald C. Gall, Aryan Dadashi, Anne Keller, and Andreas Roth. Runtime metric meets developer: Building better cloud applications using feedback. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 14–27, New York, NY, USA, 2015. ACM.

[19] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, March 2005.

[20] M. Beller, G. Georgios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[21] E Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 564–569. IEEE, 2003.

[22] A. Causevic, D. Sundmark, and S. Punnekkat. Factors limiting industrial adoption of test driven development: A systematic review. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 337–346, March 2011.

[23] Manish Virmani. Understanding devops & bridging the gap from continuous integration to continuous delivery. In *Innovative Computing Technology (INTECH), 2015 Fifth International Conference on*, pages 78–82. IEEE, 2015.

[24] Francesco Colavita. Devops movement of enterprise agile breakdown silos, create collaboration, increase quality, and application speed. In *Proceedings of 4th International Conference in Software Engineering for Defence Applications*, pages 203–213. Springer, 2016.

[25] N Forsgren, J Humble, and G Kim. State of devops report. *Puppet+ DORA, Portland, OR Google Scholar*, 2018.

[26] Fergus Henderson. Software engineering at google. *arXiv preprint arXiv:1702.01715*, 2017.

[27] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. 2016.

[28] Andrew Scott, Johannes Bader, and Satish Chandra. Getafix: Learning to fix bugs automatically. *arXiv preprint arXiv:1902.06111*, 2019.

[29] Aravind Narayanan. Tupperware: containerized deployment at facebook, 2014.

[30] Ron Miller and Ron Miller. How aws came to be, Jul 2016.

[31] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[32] Todd Hoff. How is software developed at amazon?, Mar 2019.

[33] A Singh. An introduction to virtualization. `http://www.kernelthread.com/publications/virtualization/`, 2004.

[34] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers. In *Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing*, pages 826–831. IEEE Computer Society, 2010.

[35] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393. IEEE, 2015.

[36] Joel Kirch. Virtual machine security guidelines, 2007.

[37] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure devops. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 202–211. IEEE, 2016.

[38] Josh Stella. An introduction to immutable infrastructure, 2015.

[39] Chad Fowler. Trash your servers and burn your code: Immutable infrastructure and disposable components, 2013.

[40] Thomas Erl. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall, 2005.

[41] Michael Hüttermann. Infrastructure as code. In *DevOps for Developers*, pages 135–156. Springer, 2012.

[42] Brendan Burns and David Oppenheimer. Design patterns for container-based distributed systems. In *The 8th Usenix Workshop on Hot Topics in Cloud Computing (HotCloud '16)*, 2016.

[43] Kubernetes documentation. https://kubernetes.io/docs/concepts/. Accessed: 2019-04-14.

[44] Adam Wiggins. The twelve-factor app. *The Twelve-Factor App*, 2011.