FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Leveraging Serverless Computing for Continuous Integration and Delivery

André Filipe Magalhães Rocha

Mestrado em Engenharia Informática e Computação

Supervisor: Tiago Boldt Pereira de Sousa

External Supervisor: Dmitry Treskunov

July 5, 2022

Leveraging Serverless Computing for Continuous Integration and Delivery

André Filipe Magalhães Rocha

Mestrado em Engenharia Informática e Computação

Approved by . . . :

President: Prof. Filipe Correia

External Examiner: Prof. Davide Taibi

Supervisor: Prof. Tiago Boldt Sousa

July 5, 2022

Abstract

Continuous Integration and Continuous Deployment practices have revolutionised the way we develop and deploy software. These pratices offer a means of guaranteeing a constant feedback loop where source code is used to create and test deployable artefacts, which can, in turn, be deployed in different environments. CI/CD systems allow a series of tasks to be executed in their product. These tasks are usually distributed across a cluster of runners, also called agents in some literature. CI/CD products make use of different kinds of computing services for their pools of runners, such as integrating with Infrastructures as a Service (IaaS) services, like Virtual Machines, or Platforms as a Service (PaaS), like Elastic Clusters.

The choice of infrastructure for runners is based on many factors: latency, cost, scalability and more. Literature shows that CI/CD products' customers face challenges when it comes to provisioning hardware that can fulfil their necessities as runners, especially while also keeping costs low. While IaaS and PaaS tend to have complementary benefits for deploying runners, recently, a new means of computation, called Function as a Service (FaaS) or Serverless, has shown itself to be an alternative. However, there is no effort to apply this to the CI/CD ecosystem, besides some products that offer a fully Serverless alternative to CI/CD.

FaaS provides complementary benefits to IaaS and PaaS for applications, by allowing a scalable deployment with minimal configuration and a different cost model. With that in mind, we aim to demonstrate that developers may benefit from leveraging FaaS for their CI/CD tasks with this research. As such, we set to ourselves to demonstrate that:

> *There are CI/CD tasks that, given their intricacies, can be optimised by executing within the function as a services cloud paradigm. The decision on the paradigm to adopt should be influenced namely by their execution time, computational needs, or geographic distribution requirements. Doing so, enables the team to improve their CI/CD efficiency.*

To test this hypothesis, we will (1) elaborate a state of the art on CI/CD runners, their intricacies, and how their underlying infrastructure influences them, (2) design and provide a reference implementation of a runner that leverages FaaS and guidelines for its adoption, and (3) evaluate how a FaaS runner compares to the state of the art.

We contribute to the state of the art of software engineering, cloud computing, and CI/CD by showing how FaaS can be used for running CI/CD tasks, and its usage can even optimise some tasks. These tasks must be executed in under 15 minutes (or be splittable into 15 minutes) and would benefit more if they are either a high throughput of tasks in a small period of time or very few executions over time. These tasks will benefit from both higher performance and cost-efficiency.

Keywords: Cloud Computing, Function as a Service, Continuous Integration, Continuous Deployment, Computing Services, Leveraging FaaS.

Resumo

As práticas de Integração Contínua e Entrega Contínua (CI/CD) revolucionaram a forma como desenvolvemos *software* e fazemos *deployment* de *software*. Estas práticas oferecem meios para garantir um ciclo de feedback onde o código fonte é usado para criar e testar artefactos entregáveis, que podem, pela sua vez, ser *deployed* em diferentes ambientes. Sistemas de CI/CD permitem executar uma série de tarefas no seu produto. Essas tarefas são normalmente distribuídas por *clusters* de *runners*, que também podem ser chamados de agentes nalguma literatura. Produtos de CI/CD fazem uso de diferentes serviços de computação para os seus aglomerados de *runners*, através de integrações com serviços de *Infrastructure as a Service* (IaaS), como Máquinas Virtuais, ou serviços de *Platform as a Service* (PaaS), como *Clusters* Elásticos.

A escolha da infraestrutura para *runners* é baseada em vários fatores: latência, custo, escalabilidade e mais. A literatura mostra que utilizadores de produtos de CI/CD enfrentam desafios na providência de recursos de *hardware* que consigam cumprir as suas necesidades de *runners*, especialmente enquanto mantendo os custos reduzidos. Enquanto que IaaS e PaaS tendem a ter benefícios complementares para os *deployments* de runners, recentemente uma nova forma de computação, chamada de *Function as a Service* ou *Serverless*, tem-se demonstrado como uma alternativa complementar. No entanto, não há ainda qualquer esforço para aplicar esta forma de computação ao ecossistema de CI/CD, para além de alguns produtos que oferecem uma alternativa de CI/CD completamente *Serverless*.

FaaS providencia benefícios complementares comparativamente a IaaS e PaaS para aplicações, ao permitir *deployments* escaláveis com um mínimo de configuração e com um diferent modelo de custo. Com isto em mente, o nosso objetivo é demonstrar que desenvolvedores podem beneficiar de usufruir de FaaS para as suas tarefas de CI/CD, com esta pesquisa. Sendo assim, nós propomonos a demonstrar que:

> *Há tarefas de CI/CD que, devido às suas complexidades, podem ser optimizadas ao serem executadas dentro do paradigma Function as a Service da Cloud. A decisão do paradigma a adoptar deve ser influenciada pela seu tempo de execução, as necessidades de computação, ou os requisitos de distribuição geográfica. Isso irá desbloquear a equipa para melhorar a eficiência do seu CI/CD, quer seja em termos de custo ou tempo.*

Para testar esta hipótese, iremos (1) elaborar um estado da arte em *runners* de CI/CD, os seus detalhes, e como a infraestrutura por debaixo destes os influencia, (2) desenhar e providenciar uma implementação de referência de um *runner* que use FaaS e orientações gerais para como o adoptar, e (3) avaliar como um *runner* de FaaS se compara ao estado da arte.

Nós contribuímos para o estado da arte de engenharia de software, computação na cloud e CI/CD ao demonstrar que FaaS é capaz de correr tarefas de CI/CD sendo que algumas destas até beneficiam do seu uso. Essas tarefas necessitam de ser executadas em menos de 15 minutos (ou podem ser divididas em tarefas de 15 minutos) e beneficiam mais se forem executadas num ambiente em que haja um elevado *throughput* de tarefas num curto espaço de tempo, ou se houverem

poucas execuções ao longo do tempo. Estas tarefas beneficiarão de uma elevada *perfomance* e eficiência de custo.

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Tiago Boldt de Sousa, who has been able to guide the means of suggestions rather than corrections, which was able to create a much more productive environment to contribute to the success of this work.

Then, I would like to give a big word of appreciation to TeamCity's Team Lead, Dmitry Treskunov, who allowed me to pursue my ideas for this dissertation by creating an open debate space for ideas but also for its mentoring.

I would also like to thank the Cloud Integrations team of TeamCity for embracing this challenge and the constant support on all matters they could help with.

Lastly, I would like to thank my friends for all the constant support and moments of fun throughout this journey. A good company truly enables you to work better and happier.

André Rocha

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CDE	Continuous Delivery
CI/CD	Continuous Integration/Continuous Deployment
FaaS	Function as a Service
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
PoC	Proof Of Concept
SaaS	Software as a Service

Chapter 1

Introduction

This chapter introduces the problem under study, describing its motivation and context and the general goals we aim to achieve. Firstly, Section 1.1 describes the context of this work. Then, Section 1.2 explains the motivation of the proposed work. Section 1.3 elaborates upon the problem under study and its details, followed by an enumeration of the general goals we aim to achieve in Section 1.4. Finally, Section 1.5 describes how this document is structured.

1.1 Context

Continuous Integration/Continuous Deployment (CI/CD) practices have revolutionised how software is developed. By allowing developers to introduce means to ongoing automation and continuous monitoring throughout the lifecycle of software development, from integration and testing stages until it is delivered and deployed into a production environment [63], CI/CD allows for the significant speedup of the process of delivering the changes to the customer (or an environment like the customers') from when the changes are committed [77]. In CI/CD, users run these automation and monitoring processes, commonly called tasks, through certain triggers, such as a new commit to a version control system. Even if one of the first CI tools in the market was CruiseControl [96], only with the appearance of Jenkins [41] did the usage of CI/CD start being popularised.

Usually, CI/CD uses clusters of machines to execute their tasks. These tasks are in a defined order, creating a pipeline. Providing an isolated environment for the tasks to be executed, the clusters, typically called runners or agents, return the feedback of the tasks and their logs so that users can analyse the execution [83]. Any computer with a compatible architecture can have a runner processing CI/CD tasks. For private CI/CD solutions, Cloud Computing services are some of the most common ways of providing such computers. These can come from Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) offers from cloud providers, topics exhibited in Subsection 2.1.2.

While Cloud Computing services started growing in popularity by 2007, with the collaboration between Google, IBM and, several American universities [102], it has grown much further, with

some research indicating 94% of enterprises stating they use cloud services [3]. Virtual machine services like Amazon EC2 [8] gave way to services with more elasticity, such as Amazon ECS [34], better discussed in Section 2.1.

By 2010 [51], a new service model named FaaS (Function-as-a-Service) made its appearance. This model provides an abstraction where developers only develop code in a reactive way, that is ready to respond to triggers (usually by calling a function, but also by subscribing to events of other cloud services), leaving to the cloud provider all matters related to server management [76].

```
def lambda_handler(event, context):
    message = 'Hello {}!'.format(event['name'])
    return {
        'statusCode': 200,
        'message' : message
    }
```

Listing 1: AWS *Lambda Function* [54], their FaaS, code example. The user is only responsible for defining the function itself. It receives the request's details via parameters, and its *return* operation defines the returned response.

This new service model provides a difference in billed costs, response time, and level of abstraction compared to its predecessors, IaaS and PaaS. FaaS is a model that can be advantageous for specific applications in specific scenarios [101]. Thus, considering the trade-offs of FaaS over other paradigms, with this research, we want to evaluate how those can be leveraged to improve CI/CD runners.

1.2 Motivation

CI, CD and Continuous Delivery (CDE) have been a growing trend throughout the past decade, mainly due to the adoption of agile practices within organisations[97]. Even if the usage of CI/CD, as opposed to CI/CDE, is still far from wide adoption [72], it is undeniably growing closer to becoming a standardised practice.

While CI/CD enterprises have been able to create more mature tools over the past decade and better reach the users' necessities, there still are several problems associated with them [72]. Build times are one of their biggest problems, with some tools even reporting 40% of their builds taking over 30 minutes [82]. Some developers even claim that these tools are tailored to web applications, not fulfilling the needs of other markets [97].

Some of these problems can be attributed to the infrastructure used to run the tests. Determining the infrastructure required to run the tasks and obtain the results promptly is a struggle many organisations face [72]. The balance between the cost of the infrastructure and the number of resources to improve response times is stated by Claps as one of biggest technical challenges [72]. While this problem could be partially solved with Orchestration Platforms-based runners, the costs associated with these solutions tend to be relatively high.

With the introduction of FaaS as a possible solution for cloud computing problems, it was clearly understood how all cloud computing service models provide complementary benefits. FaaS has shown itself to scale as much as Orchestration Platforms, though providing a much higher layer of abstraction that simplifies deployments without sacrificing scalability [90].

Even if FaaS has begun to show itself as a complementary alternative for cloud computing problems, its viability for running CI/CD tasks has not been tested. Since every service model provides complementary benefits, it would be ideal for defining a way to use FaaS to run CI/CD tasks and evaluate in which use cases FaaS excels, in opposition to more traditional CI/CD runners.

1.3 Problem

The different CI/CD tools may provide different levels of abstraction on the interaction with the product's deployment, a topic explored in Subsection 2.2.8. Nevertheless, as the project grows in size and complexity, most rely on organisations providing their resources for running CI/CD. It is pretty standard for users to struggle at provisioning these resources [72].

Runners can be provisioned with an organisation's on-premise resources. They can provide a much more secure infrastructure, provided they can secure it, where costs are much more predictable. However, they tend to be much more costly, especially regarding the initial investment and their overall maintenance over time. [80].

Alternatively, cloud providers offer several services that can be used as a means of provisioning runners. Several CI/CD tools end up providing integrations with many of these directly to simplify their usage for their users. While this is a topic discussed in detail in Subsection 3.2, it is crucial to notice how they all end up supporting services in the range of IaaS and PaaS. There is no known support of FaaS as a runner for CI/CD.

FaaS provides some characteristics that introduce problems in running CI/CD tasks. On the one hand, their infrastructure is available to run tasks that take a long time [56]. On the other hand, FaaS are usually stateless services [90], while CI/CD tasks tend to benefit heavily from caching (mainly when they are build tasks).

Additionally, the event-oriented architecture of FaaS is not directly compatible with how most CI/CD tools work. CI/CD tools require a constant connection to their runners since they use their orchestrator to distribute jobs [83]. It is a challenge to integrate the two of them into an existing product.

Furthermore, the literature reveals that FaaS tends to bring complementary benefits compared to IaaS and PaaS [101]. FaaS may only be beneficial in certain conditions, depending on the characteristics of the task itself (e.g. computational intensiveness, the importance of caching) and the social context of the team that triggers the task (size, geographical distribution).

With this research, we set ourselves to explore how a FaaS can be introduced into an existing CI/CD tool as a possible runner while also evaluating its viability when opposed to traditional runners.

1.4 General Goals

Use FaaS for a CI/CD runner:

Existing CI/CD tools cannot leverage FaaS on their pipelines. This task requires understanding how a FaaS could communicate with the CI/CD tool. CI/CD tools tend to implement their runner orchestration, but so do FaaS'. We intend to develop a reference architecture for a FaaS runner and demonstrate it with a Proof Of Concept (PoC) for a given CI/CD platform.

Research the viability of a FaaS runner with different tasks:

All cloud models provide a different subset of problems. Since FaaS has its intricacies, it is crucial to comprehend which tasks could benefit from using FaaS. The characteristics of the task, such as its length, computational intensiveness, and dependence on caching, may influence the quality of FaaS, as opposed to traditional cloud-provided runners. We want to understand which of these scenarios allows FaaS to excel in different parameters, such as task throughput, response time and cost.

Study the viability of a FaaS runner with different kinds of teams:

The social characteristics of the project's team may influence the quality of FaaS. A project with geographically distributed contributions will have tasks distributed over a more extended period than all from a similar timezone. This distribution difference would lead to different resource usage, particularly important when considering how FaaS does not charge any cost when it is not being used. The number of times a task can be triggered may also influence. We want to understand which of these scenarios allows FaaS to excel in different parameters, such as task throughput, response time and cost.

Evaluate how a FaaS runner fairs against other paradigms:

FaaS is known to provide complementary benefits compared to IaaS and PaaS [101]. We want to evaluate which scenarios allow FaaS to be a viable alternative to IaaS and PaaS in the context of CI/CD.

1.5 Document Structure

This document is composed of three chapters, structured as follows:

- Chapter 1 (p. 1), **Introduction**, introduces the problem under study, as well as its motivation, goals and validation process;
- Chapter 2 (p. 6), **Background**, explores the background's key concepts that are needed to understand this work fully;
- Chapter 3 (p. 23), **State of the Art**, describes the literature review process and the current state of the art on the topic of this dissertation;

- Chapter 4 (p. 35), **Problem Statement**, formalises the problem of this dissertation, and explains the scope and main focus of this work;
- Chapter 7 (p. 56), **Conclusions and Work Plan**, summarises the developed work and findings, describing the work plan and future work.;

Chapter 2

Background

In the previous chapter, we contextualise this dissertation and the problem it attempts to solve, followed by a presentation of its goals. This chapter reviews the fundamental concepts used throughout this work. This section can be skipped for readers familiar with Cloud Computing concepts, especially FaaS and Computing Services, and CI/CD concepts, including runners, pipelines and their deployment methods. Initially, Subsection 2.1 will address the main concepts of Cloud Computing, exhibiting its origin, and defining characteristics, followed by their service and deployment models. To conclude, the difference between containerisation and virtualisation in the context of computing is addressed. Then, Subsection 2.2 will explain where CI/CD comes from and what CI and CD means precisely, followed by a distinction between Continuous Delivery and Continuous Deployment. It is also introduced the concept of runners, followed by what are the runtime environments that these make use of; furthermore, the importance of distributed caching in this environment is exposed, as well as what a pipeline is, finalised by a discussion of the different deployment methods of a CI/CD system. To conclude, Section 2.3 introduces the different computing services available in cloud providers, focusing on virtual machines, orchestration platforms and serverless.

2.1 Cloud Computing

Cloud Computing is defined as an on-demand computing model. Coutinho [73] has defined Cloud Computing as:

> *Cloud computing paradigm proposes the integration of different technological models to provide hardware infrastructure, development platforms, and applications as on-demand services based on a pay-as-you-go model. In this well-consolidated paradigm for resource provisioning, customers waive the infrastructure administration. Furthermore, cloud computing providers offer services as third parties, delegating responsibilities and assuming costs strictly proportional to the used amount of resources.*

The idea of cloud computing came from the 1960s when John McCarthy suggested that "computation may someday be organised as a public utility" [81]. During the 1990s came the idea of

grid computing, the concept of making computing power accessible as quickly as the electrical power grid, which also contributed to the first stages of cloud computing. One of the first movers was Salesforce [65], providing enterprise applications via a website. However, an industrywide collaboration between Google and IBM and many American universities truly popularised the term and the model [102].

The model has a set of characteristics common across cloud computing instances (described in Subsection 2.1.1), four defined service models (exhibited in Subsection 2.1.2) and four deployment models (explained in Subsection 2.1.3). Furthermore, Cloud Computing instances may depend on different computational models based on either virtualisation or containerisation (depicted in Subsection 2.1.4).

2.1.1 Defining Characteristics

The National Institute of Standards and Technology has proposed a set of characteristics that define the cloud computing model [91]:

- **On-demand self-service** - without the need for human interaction from the Cloud provider, users can provide any resources they require (e.g. server time and storage space)
- **Broad network access** - resources must be available through standard mechanisms that promote use through heterogeneous platforms
- **Resource pooling** - resources can be available to multiple consumers through a multitenant model and reassigned through consumer demand. Clients are agnostic to the specific location of the resource, even if they are a higher-level abstraction (e.g. country, region or data centre)
- **Rapid Elasticity** - resources may be elastically provisioned and released, automatically or not.
- **Measured Service** - usage metrics should be provided to the client, allowing them to monitor their resources transparently

2.1.2 Service Models

Cloud providers provide services at different abstraction levels that the client's responsibilities and the provider's responsibilities.

2.1.2.1 Infrastructure as a Service

The client can provision processing, storage, networks and other fundamental computing resources. Even if they have no control over the underlying cloud infrastructure, such as virtualisation and machine management, the client has control over operating systems, storage and

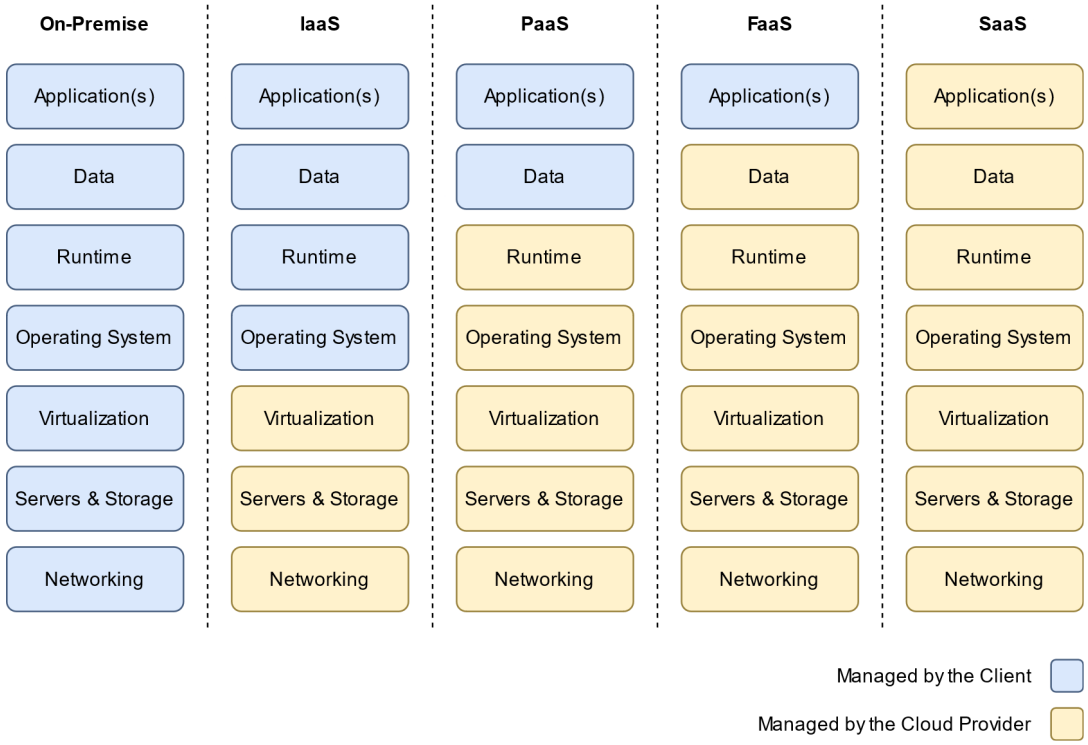

Figure 2.1: Comparing On-Premise, IaaS, PaaS, FaaS, and SaaS service models [66]. From left to right, the cloud provider has increasing management responsibility and thus an increasing level of abstraction for the user.

deployed applications, and possibly limited control over some network components, such as the host's firewalls [98].

The service provider offers a virtual server powered by one or more central operating units. These virtual servers are usually available through virtualisation technologies, better discussed in Subsection 2.1.4, which can be treated as machine-like servers from the user's perspective. [92]. IaaS tends to provide an infrastructure that can scale on-demand, and sometimes even automatically, while also providing network capabilities as a service, from load balancing to hardware for routers and firewalls. [99]

In the IaaS model, as Bokhari [99] mentions, the client is responsible for the environment's security. There are several security threats that the model is prone to, such as (1) attack in virtualisation, (2) attack based on the life-cycle of the VM, and (3) data loss and leakage (related to how data may be shared in Public Clouds, discussed in Subsection 2.2.8).

Some examples of Cloud providers' IaaS solutions are Amazon Elastic Compute Cloud (EC2) [8] and Google Compute Engine [28], both explored in Subsection 2.3.1.

2.1.2.2 Platform as a Service

The client can use the PaaS model to deploy onto the cloud infrastructure applications and their configurations without managing the development environment[99]. In comparison to IaaS, clients

abdicate the control over the OS and runtime execution to the cloud provider while gaining the ability to build and test their applications directly in the Cloud. [92]

In PaaS, clients lose control over the OS and runtime execution, a core layer whose security is now over the Cloud provider's control. Any security issue found will impact all instances. Furthermore, there tends to be a vendor lock-in, where clients will depend on some PaaS features to operate thoroughly. With this, migration to another Cloud provider, whether out of their own volition or the deprecation of the PaaS, becomes a difficult task [99].

Examples of PaaS applications offered by cloud providers are Google App Engine [10] and AWS Elastic Beanstalk [17].

2.1.2.3 Software as a Service

SaaS is the model through which Cloud providers provide their own applications that run on their cloud infrastructure. Those applications must be accessible through various thin client interfaces. The client forfeits any control over the capabilities of the applications, besides some limited user-specific configuration settings [98].

SaaS usually focus on the end-user visual interface, but it can also be an API [92]. Usually, an application instance is handed over to many clients, being that the provider is responsible for controlling and limiting how they are used. [99].

Examples of SaaS are Google Docs and Facebook.

2.1.2.4 Function as a Service

FaaS is a model that started becoming more popular almost a decade after the surge of cloud providers [68]. It provides clients with high-level software abstractions, such as functions or events, which are transparently deployed, giving the client the idea that there is no server management involved, the reason why the term 'Serverless' is usually used in conjunction. [76].

In FaaS, the cloud provider is responsible for managing the data centre server and the runtime environment, usually using containerisation (exhibited in Subsection 2.1.4). FaaS is said to be between PaaS and SaaS when it comes to service models since the client does not have any management or maintenance complications while still maintaining control over the application deployed [94].

While better explored in Subsection 2.3.3, the entire infrastructure being under the control of the Cloud provider also has its issues, from (1) vendor lock-in, (2) unpredictable execution time deviations, (3) delays when executing functions (also known as a cold start), and (4) state management [68].

The term Serverless is largely used to refer to *FaaS* services, but the term can be expanded much more than that. Eismann [78] defined Serverless computing as any computing platform that hides server usage from developers. In fact, it is possible to classify as a "serverless application" any application that combines *FaaS* with *SaaS* from any cloud provider (such as a queue, a database or an authentication service).

Examples of FaaS are AWS Lambda [54] and Google Cloud Functions [26].

2.1.3 Deployment Models

A Deployment model is characterised by the location of the infrastructure and who has control over it. NIST has proposed the following categories [91]:

- **Private Cloud** - the infrastructure, owned by the organisation or a third party, exists solely for the exclusive use of the organisation and their consumers.
- **Community Cloud** - the infrastructure is owned by one or more community organisations who share common goals (e.g. security requirements, compliance considerations) or a third party. The infrastructure is provisioned for the exclusive use of the community of organisations and their consumers.
- **Public Cloud** - the infrastructure is owned by a third party, which provisions it for open use by the general public.
- **Hybrid Cloud** - the infrastructure comprises two or more of the deployments models above, even if remaining distinct entities. They are bound by technology that enables data and application portability

2.1.4 Comparing Virtualisation to Containerisation

Cloud providers' services imply the usage of some technology that provides an isolation and multi-tenancy layer, meaning that computing resources are split between clients through techniques that allow for this multi-tenancy model. Two technologies dominate this market: the hypervisor and the container [70]. These are associated with the concepts of Virtualisation and Containerisation, respectively.

Virtualisation was the original foundation platform of cloud computing [102]. It represents a set of technologies that abstract compute resources (CPU, storage, network) from applications that allow multi-tenancy models since all tenants are provided with the same scalable yet shared platform. Virtual machines possess an isomorphism between the virtual guest and the real host, managed by the hypervisor. This isomorphism allows users to interact with specific resources via a view that will behave the same way as the real machine.

Containers are similar to virtualization[70]. While virtualization allows for running an Operating System on top of the host's OS, through a hypervisor, containers share as much as possible with the host OS (kernel, binaries and libraries), which reduces their size substantially. Furthermore, since the kernel is shared, this increases performance as the mapping latency is removed. Docker is a standard tool associated with containers. It provides a systematic way to build portable containers that isolate process trees, networks, user IDs, and file systems while still building container images compliant with an open standard, the Open Container Initiative [49].

While most cloud providers use both technologies, since they need to provide solutions encompassing the range of needs of their users, there is usually a prevalent one, especially when it comes to building their highest level services. AWS is known to rely more on hypervisors (initially relying on the XEN hypervisor [39], then moving to their own, AWS Nitro [19]), as well as Azure Cloud (using their hypervisor, Hyper-V [38]). Google, however, is famously known to rely on Container for most of their services, making use of control groups (cgroups) [87].

2.2 CI/CD systems

Continuous Integration and Continuous Deployment (CI/CD) stand for a set of practices commonly used in software development, which are close to becoming an integral part of the development process. Dullmann [77] defined CI/CD as:

> *While CI stands for the creation and testing of deployable artefacts from source code (e.g., compilation, code quality checks, unit tests), CD also comprises the eventual deployment to a production system after all previous steps have been successful (e.g., integration tests, test deployments). CI/CD aims to significantly speed up the process from the commit of code changes to a source code repository to the deployable artefact, respectively the deployment of that changes to the production system.*

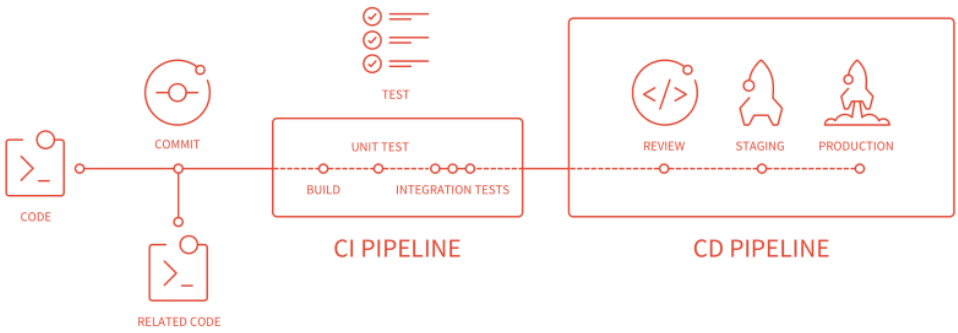

Figure 2.2: CI/CD Pipeline in GitLab CI [35]. Initially, the code is committed into a repository, which triggers the CI/CD pipeline. The CI pipeline will ensure that the application is built and automates all the tests that can be run in that environment. The CD pipeline will ensure the code is deployed to the production environment.

Even if the term CI was first coined by Grady Booch [71] (even if at the time, the term was not associated with the creation of artefacts on a daily basis), its popularity only started to rise with Kent Beck's book on Extreme Programming (XP) [69]. One of the first tools to be developed for CI was named CruiseControl [96], though the first product to popularise the usage of CI/CD was Jenkins [41], formerly known as Hudson.

An initial definition of Continuous Integration (Subsection 2.2.1) and Continuous Deployment (Subsection 2.2.2) will be introduced. Even though the term CD has different meanings in literature (discussed in Subsection 2.2.3), all CI/CD systems allow for tasks to be triggered through various ways (most commonly being VCS changes), which will generate events for runners to execute the tasks (described in Subsection 2.2.5), which must be able to support different software requirements (discussed in Subsection 2.2.6), while guaranteeing the task is executed in promptly, with caching (detailed in Subsection 2.2.7), in order to be executed in different steps of a pipeline (defined in Subsection 2.2.4). CI/CD systems themselves might be deployed through different methods (depicted in Subsection 2.2.8).

2.2.1 Continuous Integration

The idea of Continuous Integration is heavily related to allowing multiple developers to work together, ensuring their features are integrated. Integration involves merging the code and building the application, and carrying several tests within an ephemeral environment [31].

While Continuous Integration does not need to be directly associated to XP Programming and agile practices [69], it is common to see these hand-in-hand since those practices transmit the idea of frequent integration, which prompts the need to automate this integration while also performing checks prior to the integration [93].

Red Hat defines a flow that is common for various applications [31]:

- **Pushing code to the repository:** Usually, there will be a code repository and some workflow on contributing new code. While it depends on the workflow, it is typical for committing code to kick off the CI tasks, often starting with static analysis.
- **Static Analysis:** This step aims to ensure the code does not have any bugs and conforms to standard formatting and styling.
- **Pre-deployment testing:** At this point, any test that can be run without deploying a server should be. Tests can include unit tests, functional or even integration tests. This step ensures the change does not break functionalities and works well with the rest of the code.
- **Packaging and deployment to the test/staging environment:** While it depends on the project, some projects are then built and packaged before being sent to an environment that mimics production, usually called staging.
- **Post-deployment testing:** Now that the application is deployed in the staging environment, tests can be run to ensure the new changes are compatible with other libraries and the deployment environment. These tests can range from functional integration tests to performance tests. Should this step end successfully, all CI tasks are typically concluded.

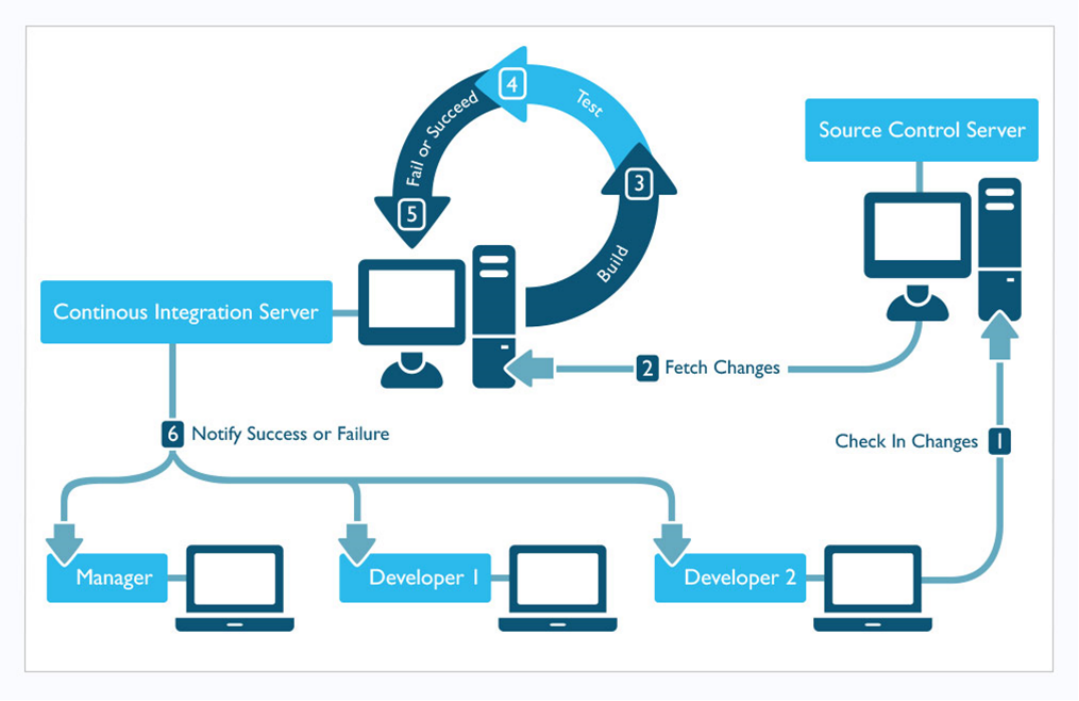

Figure 2.3: Example of a Continuous Integration process [11]. While this simplified image abstracts several technical concepts (including runners), it is helpful to understand the process itself.

2.2.2 Continuous Deployment

Continuous deployment refers to deployment automation to release a developer's changes from the repository to production. CD relies heavily on CI since there is no manual deployment gate before production, making it essential that tests automation is well designed [64].

The CD process points to deploying software to customers as soon as developed. This can lead to several benefits, ranging from a reduced risk for each release and the prevention of the development of wasted software [72].

2.2.3 Continuous Delivery vs Continuous Deployment

There is some confusion of terminology when it comes to CD: continuous delivery or continuous deployment. Throughout this thesis, the reader should interpret CD as continuous deployment and CDE as continuous delivery.

The difference between CDE and CD can be tenuous: albeit CDE refers to the process of creating an application that is potentially capable of being deployed, CD indicates that the application is automatically deployed to a production environment on every update [97]. Shahin's survey [97] exhibits that, while a large number of the surveyed organisations has successfully adopted CDE practices, only 36.1% of participants adopted CD practices.

2.2.4 Pipelines

Usually, CI/CD practices tend to be developed in a pipeline of work: a series of tasks that should be executed in a defined order.

As Jackson [86] shows, CI/CD practices could themselves be described through a pipeline, which that also helps understand the difference between Continuous Delivery and Continuous Deployment (Subsection 2.2.3).

Figure 2.4: CI/CD Pipeline proposed by Jackson [86].

Jackson [86] proposed a pipeline definition that would fit the DevOps ecosystem:

- **Code** - code is usually stored using a VCS since it allows developers to collaborate and track changes. VCS is one of the most common triggers for CI/CD tasks
- **Build** - the project's source code must be compiled in order to produce one or more deployable artefacts
- **Test** - now that the artefact has been created, a series of tests should be done on the artefact before its deployment. This kind of testing ranges from unit tests to end-to-end testing.
- **Release** - after having received the new artefact, it must be distributed in a way that ensures that there is little disruption in the availability of the service
- **Deploy** - the artefact is now considered stable by the standard of all tests previously executed and can be released to a cloud provider or an on-premise server to automate its deployment.

From all of the steps mentioned, only the release step usually does not fit into a task executed in a CI/CD system since there is no interaction between them and the service's existing infrastructure.

2.2.5 Runners

Generally, in a modern CI/CD system, these tasks are then managed by an orchestrator, responsible for distributing this task across other independent forms of computation, usually called runners or agents.

A runner provides an environment for the task to be executed. This environment requires the support of the runtime (see Subsection 2.2.6 for details) used by the project, but it usually also uses some caching (be it a centralised cache for all runners or only its local cache) capabilities to ensure the task is executed as fast as possible. Runners return the feedback of the task and the logs about its execution so that users can understand how the execution went [83].

Depending on the deployment method (Subsection 2.2.8), runners may be available to the user, though complex projects require instantiating their infrastructure. This can be done manually or through integrating with cloud providers, better discussed in Subsection 3.2.

2.2.6 Software Requirements

CI/CD tasks require a series of software dependencies in them. Those dependencies need to be available in the runner, should they need to be installed during the tasks or already made available, with the help of caching (discussed in Subsection 2.2.7) or even virtualisation (exhibited in Subsection 2.1.4).

In CI/CD, one of the most common requirements is runtime environments. Whenever a language needs direct support, a language runtime environment is prepared and installed in the machine's environment for every given operating system supported by the runner. In the case, for instance, of Java, this would mean installing the JRE (Java Runtime Environment) [74].

With the appearance of Virtualisation and Containerisation (*cf.* Subsection 2.1.4, p. 10), dependencies support could be more straightforward. By merely supporting virtual machine or container runtimes, through both of these technologies, users may only provide an image for the virtual machine or container where the code is to be executed, which delegates the support of the runtime to another layer, abstract to the CI/CD system. This is, for instance, some of the offers of GitLab for runners [33].

2.2.7 Caching

CI/CD tasks usually require a series of dependencies that are immutable from that project's perspective, usually treated as artefacts themselves. Moreover, a previous compilation of the same project tends to be reused by compilers to understand which files need to be compiled. Both of these cases are particularly fond of caching mechanisms.

Runners can cache these values locally. Users can typically indicate which folders are persisted between runs, where the dependency mentioned above artefacts and compiled will appear. Some services also offer distributed caching for the CI/CD ecosystem, allowing tasks whose runners' environments frequently vary to benefit from a dramatic speed-up [22]. This occurs when the runners executing the task are usually not the same. Distributed caching is usually centralised for all runners. Gitlab's distributed caching [22] is based on Amazon S3 [27].

2.2.8 Deployment Methods

CI/CD systems are offered in different ways that are usually related to their deployment methods. They are related to the cloud computing deployment models (described in Subsection 2.1.3).

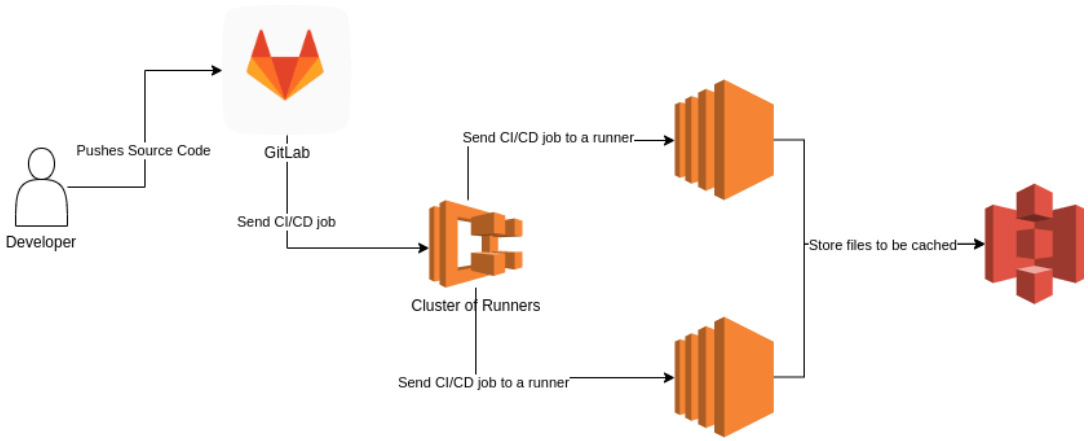

Figure 2.5: Distributed Runner Caching Example in Gitlab CI [33] (adapted from [37]). After each task execution, the runner has the responsibility of storing all of the files (or folders) that the user indicated to be cached into the S3 bucket

2.2.8.1 Private Cloud

CI/CD platforms are products offered as artefacts so that the user can deploy by its means all of the infrastructure. This is the example of, for instance, Jenkins [41], that provides its monolith for the central server to be installed and another range of installers for distributing the runners

Fisher [80] argues that on-premise allows better control of the costs of the whole infrastructure, even if they are much higher in the beginning. Cloud providers tend to charge for every single transaction and storage unit, costs that not only are dependent on any decision made by the cloud provider but also can increase with scale, even if that does not directly into more resources being used (e.g. the transaction costs associated to communication outside the cloud's network). Moreover, on-premise tends to be less vulnerable to data leakage and external security threats, provided the internal security is mature. In the case of CI/CD products, this can be even more the case since these products do not need to be exposed outside a company's internal network.

However, Fisher [80] also points out some problems for on-premise products. The initial investment is much higher than with a cloud provider. It will also be required to have dedicated support teams running these products.

2.2.8.2 Public Cloud

CI/CD services are usually offered as a SaaS, being that the user only needs to interact with it through its interfaces, having no control of the infrastructure being used for it. This is the case of Google Cloud Build, better described in Subsection 3.3.1.

Fisher [80] describes how using cloud products has a high cost-benefit, especially in the first five years of adoption. Removing the necessity to control machines directly results in a significant reduction of staff dedicated to supporting the product (assuming the CI/CD service offers customer support).

Aryotejo [67] also discusses how cloud providers' solutions can scale much better should the loads increase. This is particularly important when it comes to the management of runners.

2.2.8.3 Hybrid Cloud

In the context of CI/CD systems, Hybrid deployments refer to a model where the vendor provides the whole CI/CD infrastructure through their Cloud service, but the user can provide its runners and control their environment.

Aryotejo [67] exhibits that Hybrid deployment methods end up providing a security level that may be lower than on-premise but also higher than Public Cloud. This is widely observed in the context of CI/CD since the tasks themselves would be run in a much more contained environment, which can, for instance, block attempts to access specific web resources not recognised as a necessity for the task.

2.2.9 Tools Comparison

Throughout this chapter, many different CI/CD concepts were discussed. Polkhovskiy [93] prepared an in-depth comparison of how these concepts fair in CI/CD tools, exhibited in Figure 2.6 (p. 19).

Polkhovskiy defined his search criteria in the following way:

- The number of functionalities it provides is considered by him to be one of the most of significant factors for choosing a CI/CD tool. Features can include:
  - The possible ways to define a trigger to a task and how the methods it uses to interact with the repository used. CI/CD tools must be able to recognise changes files and their versioning. This is defined by the *GiHub* and *Other repositories fields*.
  - The feedback of the task execution is a key element. This not only encompasses how a user is warned, but how it can interact with execution logs. This is defined by the *Feedback* field.
  - The user interface is not considered to be essential, though good designed interface can dampen the learning curve. This is defined by the *Usability* field.
  - It is important for CI/CD tools to provide authentication and authorisation features, in order to control the access to results and configuration modification. This is defined by the *Security* field.
  - The tool's ability to allow the community to extend its functionality is also crucial, in order to ensure the lack of support for a given tool does not render the tool useless. This is defined by the *Extensibility* field.
- The *Compatibility* field defines how well a tool is integrated with the other elements of the development process, for instance, the programming language or the build tool.
- The *Reliability* field defines the amount of time a given tool has been in the market and its reputation, by looking into the the size of the community of users and developers.

- The *Longevity* field is defined using the same markers as the *Reliability* one, though understanding what are the future trends for those values, while *Reliability* focuses on the past and present.
- In order to understand the cost of the CI/CD tool, the *Commercial, Totally Free, Trial, Free for open-source* fields will give different inputs on such matters.
- The *Support* field defines if a tool offers dedicated support to solving a company's specific issues.
- The *Easy entry-period* field defines the complexity that it is to setup the tool until the first tasks can be ran.
- The *Flexibility* field details how customisable are the settings of the tool.
- The *Community* field entails how active are the users and the developers on online forums.

Taking into account all the eighteen criteria used, Polkhovskiy [93] provides the biggest amount of ticks to Jenkins, though this does not necessarily mean that Jenkins is the best tool for every possible scenario.

2.3 Computing Services

Any CI/CD system offers the ability to run a series of scripts, used to express the user's needs for the given problem. This requires the usage of any form of computation. Therefore, different CI/CD services have, over time, made use of different computing services for this, even if those are not transparent for the user.

2.3.1 Virtual Machines

These services were the first option that CI/CD services made use of. Virtual machines provide isolated computational devices by making use of virtualization techniques.

Cloud providers offer this service with a wide variety of options for their virtual machines. From different Operating Systems to different hardware requirements, the variety of characteristics offered is always increasing. For instance, AWS in their EC2 [8] wide selection of instances is able to cover different types of CPU architecture (x86 or ARM), different types of storage (HDD, SSD or a mix of both), the need for a dedicated GPU or not and more.

Their payment system is usually based on how much time they are spent running, usually counted in seconds. Besides the ability to shut them down and run them again on-demand, some cloud providers are already able to hibernate their machines as well, which allows for the content in RAM to be stored until it is back running, without any additional cost.

Major Cloud providers are also able to offer upgrades or downgrades on the machine's hardware (usually called vertical scalability), without the necessity to lose their storage (though, at the

Criteria	Jenkins	Bamboo	CircleCI	Codeship	TeamCity	Travis
GitHub	✓	✓	✓	✓	✓	✓
Other repositories	✓	✓	✓	✓	✓	×
Multi-Language	✓	✓	✓	✓	✓	✓
Feedback	✓	✓	✓	✓	✓	✓
Usability	×	✓	✓	✓	✓	✓
Security	✓	✓	✓	✓	✓	✓
Extensibility	✓✓✓	✓✓	✓	✓	✓	✓
Compatibility	✓✓✓	✓	✓	✓	✓	✓
Reliability	✓✓✓	✓	✓	✓	✓	✓
Longevity	✓✓✓	✓	✓	✓	✓	✓
Commercial	×	✓	✓	✓	✓	✓
Totally free	✓	×	✓*	✓*	✓*	×
Trial	×	✓*	✓*	×	×	×
Free for open-source	✓	✓	×	✓*	✓*	✓*
Support	×	✓	✓	×	×	×
Easy entry period	×	×	✓	✓	×	×
Flexibility	✓	×	×	×	×	×
Community	✓✓✓	✓	×	✓	✓	✓

Figure 2.6: Comparing different CI/CD Tools [93]. Polkhovskiy uses the number of ticks in some comparisons in order to be able to distinguish the ones considered to be far better than their competitors. A tick with a star denotes limited functionality.

time of writing, no provider is able to do so without shutting down the machine)[25]. Horizontal scalability is also offered, meaning that if the workload is too much for the current cluster of virtual machines (based off on metrics such as CPU utilisation or response time), further machines can be added to the cluster.

2.3.2 Orchestration Platforms

While Virtual Machines have already been able to provide some forms of horizontal elasticity, by allowing the resizing of the number of instances according to workloads, these technologies have evolved over the years, particularly with the emergence of Kubernetes. Kubernetes [45] is a cluster manager of containers.

Kubernetes, and other cluster managers such as Amazon ECS [6], abstract the orchestration of containers by provisioning them according to the needs of the current workload. Clusters are able to have their own IP address and load-balance loads across instances. Clusters are also able to automate deployments, by progressively rolling out any changes in order for them to be monitored. Managers also provide self-healing of nodes, should they not respond.

Orchestration platforms themselves tend to not have a cost on major cloud providers, besides the cost of all the machines that are instantiated, usually using a time-based cost model.

2.3.3 Serverless

Cluster management systems were able to provide flexible yet elastic solutions. However, their setup and maintenance tend to require a reasonable amount of resources, which make them less viable for small teams. With this in mind, the concept of Serverless was created, with the first service that ever existed being called Facebook Parse Cloud Code, though the term has been popularized by Amazon's AWS Lambda [68]. Baldini defines Serverless as a broad term for all applications that don't control the resources used by their applications. Baldini also defines Serverless computing services as FaaS, which is the definition that will be used for throughout this thesis [68]. It is considered that *Serverless* refers to serverless computing services, and, therefore, FaaS.

Serverless computing services offer a platform where the user is only responsible for the code it is being run. All the container orchestration is completely handled by the cloud provider, providing the necessary elasticity while also guaranteeing that the service is always able to provide a response [85]. Albeit these services were first advertised for their capabilities when it came to easing deployments and cluster management, they have already been proved to be able to scale as much as previous solutions. This is mainly due to the fact that it is able to spin up as many nodes as the workload requires very efficiently (though there are very high concurrency soft limits).

With the intent of testing how serverless can perform against orchestration platforms, Völker [101] designed a case study application divided into several microservices that he would use to test the performance of AWS Lambda against Amazon ECS. He was able to find several of his services that were outperformed by Lambda against ECS, especially when it came to big workloads. Whenever the throughput had to increase dramatically in a short time, Lambda was able to surpass ECS; however, services that relied heavily on local caching proved themselves to be dramatically more costly, monetarily speaking.

This can be explained by the clear difference in cost models when it comes to serverless services, as opposed to any of the previous computing services discussed. These services tend to be charged by the time they are spent running, usually measured in milliseconds, as well as the number of requests processed. In order to avoid incredibly costly invocations, services tend to provide hard limits on the length of the invocations. Serverless services allow the configuration of how much memory is allocated to the runtime of the function, being that the CPU power is proportional to its memory. Due to the nature of its containers, the number of programming languages supported tends to be more limited than other services, though they are able to support OCI [49] container images as a runtime.

One of the biggest problems related to serverless is their cold starts. Whenever a function does not receive any sort of workload for a given period of time, the number of containers allocated to the problem becomes 0. Whenever the function receives a new request, it will have to wait for the spin-up of a new container [90]. This cold start can be further aggravated by the runtime of

the function. For instance, JVM languages will have a significantly higher cold start, due to the JVM's timely warm-up.

2.4 Summary

This chapter described the critical concepts of Cloud Computing, with a particular focus on FaaS and the concepts of CI/CD, mainly focusing on runners.

Section 2.1 (p. 6) defines Cloud Computing by detailing its defining characteristics, service and deployments models, and comparing virtualisation and containerisation. Most cloud computing's characteristics are a mixture of conceptional, technical and business characteristics, such as on-demand self-service, resource pooling and rapid elasticity [91, 99]. Service models (e.g. IaaS, PaaS, FaaS and SaaS) are a way of characterising different cloud solutions based on their level of abstraction over computing layers [99, 98, 91] (*cf.* Figure 2.1, p. 8). Cloud deployment models are defined by who controls the infrastructure, classified as private, community, public or hybrid [91]. Containerisation and virtualisation are two technologies that allow isolation and multi-tenancy over computing resources [70, 102].

Section 2.2 (p. 11) defines CI/CD systems and their most important concepts, such as continuous integration, continuous deployment, continuous delivery, pipelines, runners, software requirements, caching and deployment methods. On the one hand, Continuous Integrations involves merging code to integrate new features, proceeded by building the application and carrying out several tests to ensure the quality of the work [31, 63] (*cf.* Figure 2.3, p. 13). On the other hand, Continuous Deployment ensures that the process that leads software to production is automated [64, 72]. It is common to confuse Continuous Delivery with Continuous deployment since Continuous Delivery stands for the process of creating an application that is capable of being deployed. In contrast, Continuous Deployment ensures the application is automatically deployed to production on every update [97]. Pipelines are commonly associated with CI/CD since CI/CD practices are usually organised in a series of tasks that should be executed in order, leading to a pipeline of work [86] (*cf.* Figure 2.4, p. 14). Runners are the form of computation that CI/CD systems use to execute the tasks. Their primary responsibility is executing the tasks and providing logging and feedback [83]. CI/CD tasks commonly have software requirements that need to be available to the runner [33]. Some of these requirements are immutable or have very few changes between tasks executions, making them suitable for caching practices. Caching can be done locally or distributedly [22] (*cf.* Figure 2.5, p. 16). CI/CD systems, as a cloud solution themselves, can be deployed through the different cloud deployments methods.

Section 2.3 (p. 18) defines the different cloud computing services available for CI/CD. Virtual machines were the first option existing that provided computation devices isolated by virtualization. Cloud providers' clients can choose from different hardware requirements to different operating systems. Their cost model is based on the amount of time (measured in seconds) spent running [25]. Orchestration Platforms abstract the management of containers or virtual machines by provisioning them according to the workload's needs, which is already load balanced across

instances. Orchestration Platforms can also automate deployments since they progressively roll out changes and self-heal non-responding nodes. These services tend not to have an additional cost but the underlying machines and the time they spend running [6]. Serverless computing offers a platform where the user is only responsible for the code that is being run, leaving all the orchestration to the cloud provider, while still guaranteeing the necessary elasticity for it to be able to always provide a response [85]. A case study was even able to to demonstrate that there are scenarios where Serverless has been able to outperform Orchestration Platforms for big workloads [101]. Serverless' cost model depends on the execution and the number of requests received.

In the following chapter, we discuss the state of the art of how CI/CD systems support runners, what kind of serverless CI/CD solutions there are and some open challenges in the CI/CD ecosystem.

Chapter 3

State of the Art

The previous chapter reviews the critical background concepts used throughout this work—this chapter details state of the art for refactoring partitioned applications at runtime and FaaS leveraging. Section 3.1 describes the methodology used in the research process. Section 3.2 provides an overview of the different types of runners offered by the different CI/CD products in the market, followed by an introduction in Section 3.3 of serverless CI/CD products and their differences to traditional CI/CD products. Finally, in Section 3.4, a presentation of open problems and challenges in the CI/CD ecosystem is presented. Finally, Section 3.5 summarises the main findings on the aforementioned topics.

3.1 Methodology

The study of the current state of the art for this dissertation's domain was done using an iterative literature review methodology [89]. A defining group of search queries was defined, which were adapted (that is, by changing the engine's operators) accordingly to the user databases. These search queries can be explored in Appendix B (p. 68)

The search queries were initially defined by the sections name's themselves and further developed upon exploring its results in order to investigate the domain's background and state of the art. Although no specific inclusion or exclusion criteria were directly applied, the selection of relevant articles followed an exploratory analysis that was influenced by (a) number of citations, prioritising the most cited publications, (b) publish date, prioritising the most recent publications, and (c) domain similarity, evaluated by analysing the article's title, abstract and conclusions.

The literature results obtained from the search queries were complemented through *snowballing* — analysing the references of each result to identify other works of interest and relevant terms, topics, and concepts in the domain.

3.1.1 Scientific Databases

The search process included the usage of IEEE Xplore, ACM Digital Library, Scopus, Science Direct, and Research Gate.

The researched literature consists of conference papers, journal and survey articles, and popular reference books. Moreover, it also included technical reports and grey literature, consisting of web articles and Cloud providers' official documentation.

3.1.2 Literature Review Questions

To guide the literature review process, the following literature research questions (LRQs) were taken into account:

LRQ1: How is CI/CD differently supported by difference cloud computing models?
CI/CD tools already provide some integrations with many cloud solutions. We wish to understand which integrations currently exist, how to make use of them and their limitations;

LRQ2: How is FaaS leveraged for CI/CD?
It is essential to comprehend if FaaS technologies are already used in any way for CI/CD. We aim to understand if these technologies are leveraged in any way, and,if they are, what technologies are made use of for this;

LRQ3: What are some of the struggles the industry still faces with CI/CD practices and tools?
Understanding how organisations use CI/CD tools and what struggles are faced key to understanding which benefits can FaaS bring to complement CI/CD products.

3.1.3 Process

A bottom-up approach was used to find the studied literature. It consisted of iteratively executing the following steps:

1. Given a survey research question, build a set of queries that express the information need;
2. Submit the queries, refining them based on the obtained results (like including synonym expressions and keywords);
3. Identify the retrieved documents as relevant or non-relevant based on their similarity to the domain by analysing their title, abstract, and conclusions;
4. Analyse the relevant documents in detail, filtering false positives. Among the relevant, inspect their references (snowballing) to identify missing relevant terms, topics and concepts;
5. Accommodate the newly identified terms, topics and concepts in the search queries, repeating the process.

Within the results obtained in each search query, the most recent and cited (as well as the combination of both) were considered for thorough analysis. However, the various submitted queries were not captured, which results in the replication of this literature review process not being trivial

3.2 CI/CD Systems Runners

Even if CI/CD systems have some homogeneous characteristics to them, the means provided to run their tasks are certainly not of them. These are usually called runners, or agents in some contexts, and options can be significantly different from one service to another, especially when it comes to the differences between on-premise and cloud services.

3.2.1 TeamCity

Teamcity allows runners, called agents in their literature, to be manually installed on machines by the user, as long as they have a java runtime or a container runtime. Runners can also be automatically installed into a machine through SSH [40].

However, users who prefer to use cloud providers for this are already provided with integration with these technologies through plugins. EC2 Fleets [32] can be used in order to automate the process of creating new instances, according to loads. In a similar fashion, Azure Virtual Machines can be used with its plugin, as well as Google Compute Engines (their virtual machines).

Kubernetes clusters can also be made use of, allowing the requests to be load-balanced instead. Thanks to this, any on-premise cluster or service offered by a cloud provider (such as EKS, GKE and AKS [6]) can be used for running builds.

3.2.2 Jenkins

Jenkins' runners, called agents in their literature, require a constant connection to the host server. The software for the agent to connect to the instance can be installed in a machine that has a Java runtime or that has a container runtime.

Despite this default means for adding runners, EC2 is supported through the means of a plugin [1]. This makes use of Jenkins' own load-balancing, while also providing its own horizontal scalability, should the build cluster not befitting for the current load. A similar plugin is provided for Azure VMs [4], however, this seems to be deprecated of detriment of creating a cluster through Azure's CLI and integrating it with Jenkins.

When it comes to Elastic Cluster technologies, while a plugin for using Kubernetes is provided [2], there are plugins in order to use GKE [61] and AKS [20] directly, as well as ECS [9]. However, at the time of writing, both the ECS and the AKS plugins have been orphaned, but maintainers are being looked for.

Serverless technologies can be used, through a plugin for Google Cloud Build [42], which makes Jenkins completely agnostic to the state of the runners and their orchestration. When using this plugin, users must define their tasks' steps using Google Cloud Build's language, explored in Subsection 3.3.1.

3.2.3 GitLab CI

GitLab CI's offers are slightly different from previous CI/CD services due to their lack of extensibility. Rather than that, guides are provided in order to make use of different technologies to provide your runners.

Several different executors for runners are maintained [33]:

- **Shell Executor** - a shell script that connects the machine directly to the GitLab CI's instance, making use of the machine's own environment for running tasks. This is the simplest executor of them all.
- **SSH Executor** - similar to the shell executor, which allows commands to be executed in the remote machine over SSH
- **Virtual Machine Executor** - making use of VirtualBox or Parallels for virtualization, it is possible to run builds using virtual machines inside the runner, providing a much more isolated approach.
- **Docker Executor** - making use of Docker's container runtime, Docker Engine [30], builds are all ran inside an isolated container using the image specified in the configuration file.
- **Docker Machine Executor** - docker machine was a tool that allowed Docker hosts to be created in a number of places, such as cloud providers. While docker machine has been deprecated, GitLab maintains its own fork due to the popularity of this sort of runners. Instructions are given on how to use this executor in order to use ECS instances to make runs [12]. An EC2 is responsible for receiving the jobs and distributing them to an ECS Cluster. AWS Fargate [53] is used to manage the cluster and load-balance the requests it receives. A similar logic can be applied in order to use EC2 Spot Instances [13] instead of AWS Fargate.
- **Kubernetes Executor** - a Kubernetes cluster can be used as an executor, which allows a new Pod, a group of 1 or more containers, to be created for each task. Each Pod will contain at least 3 containers: a build container, a helper container and one container per each service defined in the configuration.
- **Custom Executor** - Whenever these options are not enough, an API exists in order for your executor to be defined, such as using LXC runtime [46] instead of Docker Engine

3.2.4 Azure DevOps

Azure DevOps, being part of Microsoft's Azure Cloud platform, offers great interactions with its own technologies. Runners, called agents in their literature, may be managed by Azure itself, who spins up AVMs as needed for the workload of jobs existing. Furthermore, a runner may be installed manually in a machine, either directly or using the Docker Engine runtime [30]; although there is no means to automatically manage and orchestrate them. [21]

	Virtual Machines	Orchestration Platforms	Serverless
TeamCity	✓	✓	✗
Jenkins	✓	✓	✓*
Gitlab CI	✓	✓	✓*
Azure DevOps	✓	✗	✗
CircleCI	✓	✓	✗

Table 3.1: Runner Support in Various CI/CD tools. It is possible to see that the annotated tools who claim of support serverless technologies are both accompanied by asterisks, related to what is discussed in Subsection 3.2.6

3.2.5 CircleCI

CircleCI provides its own in-house orchestration using a shared pool of runners, called agents in their literature, which may take jobs from several different users, even if the environment is completely isolated [23]. As all services discussed, a runner may be manually installed, directly into the machine or using Docker Engine runtime [30] to contain it.

Instructions are also provided on how to use Kubernetes, therefore adding support for any Cloud Provider who servers Kubernetes clusters.

3.2.6 Conclusions

After an in-depth analysis of several tools, it is possible to claim that all analysed CI/CD tools possess some support for using Virtual Machines services for runners. Even if AzureDevOps is the only tool to fail to support Orchestration Platforms, we can determine this is to ensure vendor lock-in into their cloud platform.

To conclude, it is possible to see already some ways of interacting with Serverless technologies, albeit with some limitations:

- Jenkins allows the usage of Google Cloud Build; however, this is already a CI/CD tool of itself, which can be used independently of Jenkins. In this case, Jenkins is merely a tool for managing tasks reports and the CI/CD triggers.
- While none of GitLab's documentation, nor any literature, discuss how could serverless technologies be leveraged for runners, their executors logic has been observed to be used to serve FaaS by the community [55]

These findings are summarised in Table 3.1.

3.3 Serverless CI/CD Solutions

Cloud providers have also started defining their own platforms for CI/CD that are announced as Serverless; meaning that developers have no real interaction with the servers used to run the platform. It is an example of this Google Cloud Build and AWS CodeBuild. These platforms

differ from other aforementioned ones such as CircleCI mentioned in Section 3.2.5 since the user has no way of controlling the infrastructure used to run these builds. While they can be called Serverless from the perspective of Runners, they may be interpreted by many simply as SaaS that users interact with.

3.3.1 Google Cloud Build

Released in 2018, Google Cloud Build allows for building, testing and deploying software without ever having to think of the infrastructure these tasks are run on. It uses a container runtime to run every build in order to guarantee the platform's environment does not affect the reproducibility of the build itself. Since every build is stateless, it heavily relies on caching of source code, dependencies or assets in order to increase the build speed [24].

Container orchestration is completely controlled by the server itself, being that by default a user will make use of a shared pool of machines across all users. It is possible to allocate a private pool of machines for a team, should it be found necessary, consequently, with an additional cost.

Google Cloud Build allows the choice of the characteristics of the VMs that will be used to run the tasks. The offers displayed are the same ones offered by Google Compute Engine [29]. The cost model of this service is dependent on the number of minutes spent running tasks, as well as the cost of the caching of artifacts, adding as well any network costs. Secrets, used for security purposes, will also add to the costs [100].

In practice, using Google Cloud Build is quite similar to using any other SaaS. In order to further explore this, let us examine a simple boilerplate project for NodeJS [48] that serves only a *Hello World* in the root route (*cf.* Listing 2, p. 29). A unit test accompanies that server example, only ensuring that the route returns successfully (*cf.* Listing 3, p. 30).

As with any NodeJS project, it also requires the presence of a package file that specifies the app's dependencies (*cf.* Listing 4, p. 31). To finalize, it is only required to create a YAML file dedicated to explaining the steps of this application's CI/CD pipeline to Google Cloud Builds. This file must have a specific name, *cloudbuild.yaml* (*cf.* Listing 5, p. 32). Each of the fields provided in the list of steps is directly related to the container images used:

- *name* - represents the name of the Docker image used to run the step
- *entrypoint* - Docker images can provide an entrypoint, a command that is prepended to every execution. This ensures the created image is executable
- *args* - the list of arguments that will be passed to the *entrypoint*

In order to now trigger CI/CD tasks on a push to the *main* branch of the project, it is required to create a repository trigger through the Google Cloud Platform (*cf.* Figure 3.1, p. 33).

3.3.2 AWS CodeBuild

Released in 2016, AWS CodeBuild allows for building and testing software with the promise of continuous scaling [14]. It makes use of a container runtime to ensure reproducible builds.

```
const express = require('express');

const app = express();

app.get('/', (req, res) => {
        res.send('Hello World!');
    });

const port = 3000;
const server = app.listen(port, () => {
        console.log('listening on port %s.\n', server.address().port);
    });

module.exports = app;
```

Listing 2: NodeJS Code Example. Should a user run this code, they will be able to access *http:localhost:3000* to receive the *Hello World* response.

CodeBuild offers some instance types that resemble the VMs used in other services, though there's no evidence to point out as to how they're related. All the build artifacts created during a build are stored in S3 Buckets [27].

AWS CodeBuild is part of a family of products related to CI/CD. In order to ensure a similar solution as Google Cloud Build, mentioned in Section 3.3.1, in AWS it is required to make sure of two more products:

- AWS CodePipeline - a continuous delivery service to automate deployments [16]
- AWS CodeDeploy - a fully managed deployment service that automates deployments to any AWS compute service, as well as on-premise servers [15]

It is possible to integrate with others services that are AWS not specific as well, though integration may not be as simple and less costly.

CodeBuild's cost model is based on the number of minutes used to run the tasks. Furthermore, additional charges may apply, related to the usage of S3 for storage of artifacts and storage of credentials or any data transfer outside the AWS network [100].

3.4 Current Challenges in the CI/CD Space

Even if the usage of CI/CD is growing closer and closer to becoming a standard, there are still challenges that hinder its adoption by some sectors. Furthermore, there are still challenges that several teams tend to have whenever their product scales.

```
const app = require('./index');
const should = require('chai').should();
const request = require('supertest');

describe('test.js', () => {
  describe('GET /', () => {
    it('responds with 200', (done) => {
      request(app)
        .get('/')
        .expect(200)
        .end((e, res) => {
                should.not.exist(e);
                done();
        });
    });
  });
});
```

Listing 3: NodeJS Code Unit Test Example. We can ensure the *Hello World* route will return successfully with this unit test.

3.4.1 Hardware Requirements and Long Builds

Long builds are a big problem when it comes to the usage of CI/CD products. A dataset of builds taken from TravisCI [62] has over 40% of its builds taking over 30 minutes to run. A study by Ghaleb [82] on this dataset has revealed some essential factors that contribute to this:

- **Lack of Caching** - the usage of means to cache assets that are not often changed impacts dramatically build times. It was observed that the introduction of caching into an already existing project reduced the build duration by a median of 11 minutes. Ghaleb states that the most common reasons associated with to not using caching capabilities is related to its misconfiguration or lack of knowledge that it is possible to configure it. Since CI/CD tasks will run normally without it, caching is one of the most common oversights.

- **Non-deterministic tests** - TravisCI allows developers to configure build steps to rerun failing commands multiple times. This configuration is shown to have a practical impact on maintaining a more stable build status, indicating how brittle some of the developers' pipelines are. This instability was usually associated with using certain concurrency features, such as sleep, in their tests in the case of a Finnish software company [84].

- **Time of the day** - it was observed how builds were more likely to have a longer duration if triggered on weekdays during the daytime. This is associated with higher workloads and their effect on server usage, suggesting that TravisCI servers have a concurrency limit. A study on the experience of several different software engineers in Atlassian in their adoption of CI/CD has shown that one of their main challenges when adopting it has been the difficulty in measuring the infrastructure required for their problem [72].

```
{
  "name": "nodeapp",
  "version": "1.0.0",
  "description": "my node app",
  "main": "index.js",
  "scripts": {
    "start": "node .",
    "test": "mocha test --exit",
    "build": "mocha test --exit"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "chai": "4.3.6",
    "mocha": "9.2.0",
    "supertest": "6.2.2"
  }
}
```

Listing 4: NodeJS Code Package File. With this file, we specify the project's dependencies (for server itself and the unit tests) as well as the commands one can run.

3.4.2 Domain Constraints

Many challenges associated with the applicability of CI/CD are associated with the differences between some software applications' environments and web applications. In a survey conducted by Shahin [97], many of the software developers interviewed faced this problem when deploying their software less frequently than web-based applications such as embedded systems or financial systems.

Furthermore, in a round of interviews with a car manufacturing company, it was found that CI/CD tools in the market failed to be helpful for their use case. Due to the heterogeneity of tools being used by teams, no CI/CD tool could integrate with everything they used correctly. It was also mentioned how the number of simulations (around 200) required during the testing process was tough to run in CI since their hardware requirements would be too costly [88]. Due to this, this manufacturing car company ended up creating their in-house CI/CD solution, whose main goal was to provide integrations with all of the tools they developed in-house or used, such as engineering management tools or systems architectures tools.

3.4.3 Lack of Research on Runner Specific Challenges

Even while using the methodology explained in Section 3.1 (p. 23), very little research was found on CI/CD challenges related to runners. The query elaborated for this search was the following:

```
# [START cloudbuild_npm_node]
steps:
# Install dependencies
- name: node
  entrypoint: npm
  args: ['install']
# Run tests
- name: node
  entrypoint: npm
  args: ['test']
# Run custom commands
- name: node
  entrypoint: npm
  args: ['run', 'build']
# [END cloudbuild_npm_node]
```

Listing 5: Google Cloud Build Steps File. This YAML file list indicates three distinct steps. Each step is noted by the initial hyphen and is composed of three arguments: *name*, *entrypoint* and *args*

> (ci/cd OR "continuous integration" OR "continuous deployment") AND (challenges OR constraints) AND (runners OR agents OR executors)

The amount of research done for Runner-specific challenges is incredibly scarce. From all the work search engines used, only Debroy's [75] work displayed some runners' challenges, and it only displays an intuitive example. Their work focused on Varidesk's work on changing their monolith application to a microservices-based approach and the implications of their CI/CD process.

Since their product only possessed a single pipeline to run, all of their workloads went through two Virtual Machines, a single runner installed on them, using Visual Studio Team Services (VSTS). With microservices, this lack of scalability deteriorated their task times dramatically. Two different approaches were tested, where the Virtual Machines were used without the VSTS service, which introduced its latency, and where the Virtual Machines had Kubernetes clusters installed in them, in order to run builds containerised and so that multiple runners could be spawned in the same machine.

The usage of clustering software dramatically improved the response time of all tasks, in the order of several orders of magnitude (*cf.* Figure 3.2, p. 34).

3.5 Summary

Section 3.1 (p. 23) introduces the methodology used to perform the literature review on the topic of this work. First, the logic under the developed search queries and the criteria to select relevant

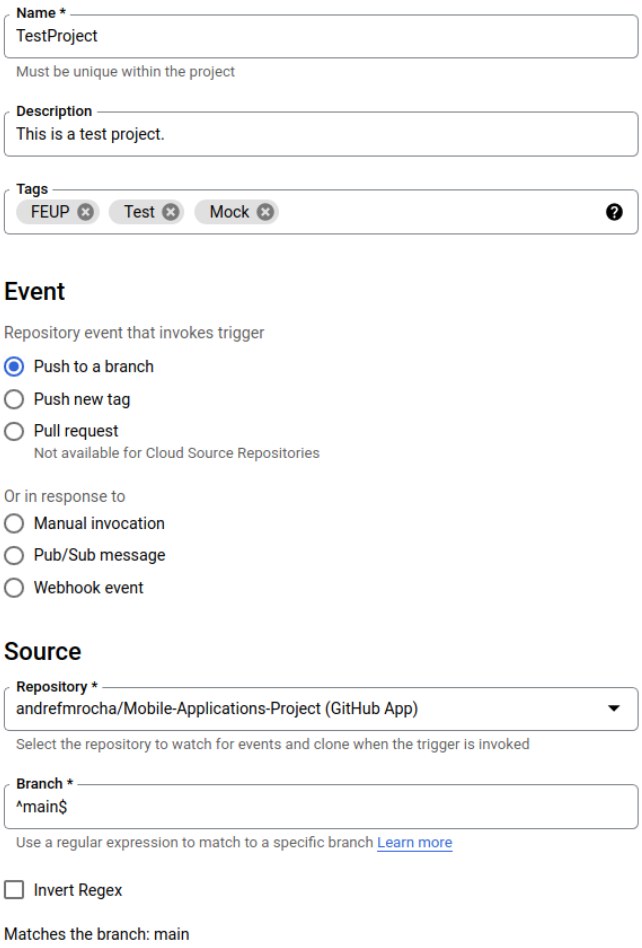

Figure 3.1: Setup of Google Cloud Build for a repository. Should a Google Cloud account already be set up, from a financial perspective, Google Cloud Build only requires the setup of the integration with the repository server, in this case, Github.

literature is described. Then, the databases used are enumerated and the types of reviewed literature documents. The iterative process used to answer the research questions is then outlined and discussed.

Section 3.2 (p. 25), focused on LRQ1, demonstrated how the market's different CI/CD tools are integrating with cloud computing services. Most CI/CD tools already integrate cloud computing services, mainly virtual machines and orchestration platforms. The level of integration varies widely since some tools only require the user to fill a form with some information of importance, while others require this integration to be done manually. Nevertheless, the vast majority does not offer any integration with Serverless technologies, only integrating with serverless CI/CD tools.

Section 3.3 (p. 27), which focuses on LRQ2, details the existing Serverless CI/CD tools. These tools differ from the ones described in Section 3.2 (p. 27) since the user cannot control the infrastructure used to run these builds. These tools focus on providing the most simple experience in order to be able to build, test and deploy software without having any interaction with the infras-

	Agents		
	Hosted	Basic	Containerized
Average build/release queue time	1 h, 18 min	23.1 min	4.4 s

Figure 3.2: Comparison of the response times for CI/CD tasks of the different options of runners [75]. *Hosted* represents the Virtual Machines using VSTS, *Basic* the non-containerised agents and *Containerized* the Kubernetes cluster.

tructure the tasks run on. These rely heavily on container technologies to ensure build's security and reproducibility. They also rely heavily on caching of the source code, dependencies, and assets to increase build speed. These services' cost model depends on the number of minutes spent running tasks, the cost of the cached artefacts, the network costs, and the cost of secrets storage.

Section 3.4 (p. 29), focused on LRQ3, describes CI/CD users' different challenges with their tools. Long builds are one of the problems that most users face, which can be attributed to (1) the lack of caching of assets that would dramatically reduce build times, (2) some non-deterministic tests, which forces them to be rerun multiple, (3) and the time of the day the task is triggered, as bigger workloads have an impact on server usage. Some developers in environments that are not web applications also consider that CI/CD tools may be unfit for their needs due to their requirements of, for instance, deploying their product less frequently. Other developers also mentioned how the lack of integration with tools being used by teams made it hard to find a CI/CD tool that fitted their needs. To conclude, there is very little research on challenges specific to runners, with only one article even mentioning these sorts of issues, which shows the lack of scientific research into CI/CD runners.

Chapter 4

Problem Statement

In the previous chapter, we discuss the current state of the art in the domain of this dissertation. This chapter thoroughly describes and formalises the problem under study. Firstly, Section 4.1 describes the current issues and open problems in the domain of this dissertation. Then, Section 4.2 defines the set of *desiderata* we intend to address in this work. Section 4.3 presents the central hypothesis we aim at validating, followed by the leading research questions that guide this work in Section 4.4. The methodology used to validate and evaluate the obtained results is outlined in Section 4.5. Finally, Section 4.6 provides an overview of the topics addressed in this chapter.

4.1 Current Issues and Open Problems

In Chapter 3 (p. 23) displays the current state of the art on the different computing services offered by Cloud Providers, how these are applied for CI/CD runners, taking a particular concern on how the serverless CI/CD solutions work, and a look into known issues and challenges shown by CI/CD users. Nevertheless, there is a lack of formal research on CI/CD runners, which affects the research on the usage of cloud services for CI/CD. When it comes to FaaS, even grey literature does not have many examples.

On the one hand, while the industry displays many examples of how CI/CD systems can be integrated with cloud providers for different runners (*cf.* Subsection 3.2, p. 25), there is very little research on how these runners fair. Should an organisation deploy their infrastructure in a specific cloud provider, even choosing the integration that benefits them the most is not backed up in researched data. Thus, it is recognised that organisations lack the means to determine their best offer without doing their own empirical research.

On the other hand, there is a lack of research on how CI/CD runners' integrations with cloud providers are achieved. Though CI/CD tools provide these integrations, there is very little research on how they are architectured or could be replicated in other CI/CD tools. Grey literature provides some insights into how specific integrations have been achieved, though their goal is to document their path to the achievement. Therefore, besides its implementation, providing an architecture for a FaaS runner may allow further runners to be developed in other platforms or for other services.

4.2 Desiderata

Based on the literature's current lack of research on FaaS for CI/CD runners, this dissertation aims to investigate how can FaaS be leveraged for CI/CD runners, thus improving the ecosystem by bringing a new option to developers. Moreover, this work explores the viability of FaaS compared to the current existing runners.

Thus, the scope of this work will include the development of a reference architecture for FaaS runners, followed by the implementation of this architecture in a CI/CD system.

We can formalize this research's scope as having the following desiderata:

D1: **Develop a reference architecture for how to use FaaS for runners,** so that developers can understand how a FaaS could be integrated into any CI/CD tool;

D2: **Implement a FaaS runner for a specific CI/CD tool** to validate the architecture from **D1**, proving it is possible to use it to implement a runner for FaaS.

D3: **Confirm the viability of FaaS for runners** to understand how the outcome of **D2** could be used. It should be analysed how it fairs against different kinds of tasks and demand levels of tasks compared to the current existing cloud-based solutions.

4.3 Main Hypothesis

We propose to research the viability of adding FaaS as a means for running tasks in a CI/CD software system. Our research is guided by the following hypothesis:

> *There are CI/CD tasks that, given their intricacies, can be optimised by executing within the function as a services cloud paradigm. The decision on the paradigm to adopt should be influenced namely by their execution time, computational needs, or geographic distribution requirements. Doing so, enables the team to improve their CI/CD efficiency.*

In order to better understand our hypothesis, a deconstruction of it is now displayed:

There are CI/CD tasks that, given their intricacies, can be optimised by executing within the function as a services cloud paradigm
We believe that there are specific recurring tasks in Software teams that are better fit to serve Serverless runners in their CI/CD system rather than the traditional alternatives.

The decision on the paradigm to adopt should be influenced namely by their execution time, computational needs, or geographic distribution requirements
While no two contexts for a CI/CD task are the same, there are characteristics that can provide insight for engineers to decide when to use each cloud model to optimize their runners..

Doing so, will enable the team to improve their CI/CD efficiency in cost and/or time
The concept of outperformance can be related to several different metrics. As in many engineering problems, it is common for an alternative to outperform specific metrics while underperforming in others. Since these trade-offs tend to have implications specific to the context inserted, our goal is to analyze specific metrics without reaching a conclusion on which one is most appropriate.

4.4 Research Questions

To validate the presented hypothesis and to achieve the proposed goals, the following four research questions (RQs) were identified to guide this work:

RQ1: Is it possible to use FaaS services to run CI/CD tasks?
CI/CD systems usually require a constant connection to the runner, which is not possible using FaaS. CI/CD systems also tend to orchestrate CI/CD tasks themselves. It is crucial to understand whether it can integrate FaaS with existing CI/CD systems.

RQ2: What metrics should evaluate the viability of a CI/CD runner?
In order to evaluate the viability of a CI/CD runner and compare it to others, it is first needed to decide which metrics can assess the quality of a runner.

RQ3: What CI/CD tasks can FaaS runners be a viable alternative to other orchestration platforms?
FaaS provides complementary benefits to other cloud service models, which means that it can potentially be a valuable tool for CI/CD runners, though certainly not in all cases. Thus, it is essential to conceive and experiment with different kinds of CI/CD tasks and different demand levels against other orchestration platforms. These experiments will allow comprehending where FaaS is a viable alternative, in terms of the types of tasks but also the task throughput limitations. The different tasks are influenced by the project's characteristics, such as the programming language or the project's scope, but also the team's characteristics, such as the number of active developers or their working hours.

4.5 Validation and Evaluation

Zelkowitz and Wallace[103] defined four different general categories for experimental models (quoted from their publication):

Scientific Method: Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis.

Engineering Method: Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement.

Empirical Method: A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis.

Analytical Method: A formal theory is developed, and results derived from that theory can be compared with empirical observations.

For this dissertation, the validation and evaluation process is separated into two phases, which use **Engineering** and **Empirical** methods of validation, respectively. The **Engineering** method will allow the development of a proposed solution for this problem while the **Empirical** will validate the viability of the solution.

A reference architecture for a FaaS runner will be developed in the first phase. This architecture will be demonstrated with a PoC into an existing CI/CD platform. The plugin will be experimented with by running several kinds of CI/CD tasks in different projects through the CI/CD platform. The results can be validated by comparing the outcome of the run in the CI/CD platform to another runner that does not rely on FaaS. This phase follows an Engineering Method approach towards validation, as the implementation is developed to test the solution engineered for the hypothesis.

Afterwards, the reference implementation will be tested against different kinds of CI/CD tasks in the second phase. Several open-source projects will be chosen for these tests to ensure they are applied against real-world use case scenarios. These tests will also compare the FaaS runner against other cloud-based solutions. To do so, a cluster of virtual machines and orchestration platforms will be set up with characteristics similar to the FaaS runners, and the same tests will be run on them. These scenarios will be evaluated through their response time (how much do they take to complete the task), their task throughput (the number of tasks it can complete in a specific period of time) and their cost (associated with the cloud provider). This phase follows an Empirical Method since data on the response time, task throughput and cost of these scenarios is gathered to validate the hypothesis.

4.6 Summary

Section 4.1 (p. 35) describes the current issues and open problems in the dissertation domain. We believe there is a lack of formal research on CI/CD runners, which affects the research of the usage of cloud services for CI/CD. While the industry can detail how to use cloud computing services with their CI/CD systems, there is very little research comparing the different services. There is also very little research on how these integrations are architectured or could be replicated in other CI/CD tools.

Section 4.2 (p. 36) details the focus and scope of our work and exhibits the desiderata proposed to implement the prototype for the integration mentioned above. The scope consists of architecting and implementing an integration of FaaS for running CI/CD tasks in a given tool and evaluating how that solution compares to the other existing integrations with cloud computing services. The

desiderata is then summarised into the main hypothesis (cf. Section 4.4, p. 38) that we wish to validate, that focuses on proposing that CI/CD tasks may benefit from using FaaS, given their intricacies. Such decisions should be determined by the task's execution time, computational needs or geographic distribution requirements, enabling the team to improve their CI/CD efficiency from either or both a cost and time perspective.

Section 4.4 (p. 37) lists the main research questions that guide this work. These first focus on determining if it is possible to use FaaS to run CI/CD tasks. Furthermore, we intend to investigate the viability of using FaaS runners, when comparing to other cloud-based computing services, against different kinds of tasks and different demand levels of tasks, in order to detail the possible scenarios where FaaS may provide complementary benefits to the other computing services in CI/CD. This comparison will be based on several performance metrics, such as response time and metrics.

To conclude, Section 4.5 (p. 37) introduces the validation process of this dissertation. It starts with (1) the development of a reference architecture for a FaaS runner and also its implementation into an existing CI/CD tool as an extension, and (2) it ends with validation of the extension by comparing the outcome of such a runner against other cloud computing-based runners against different open-source projects of different dimensionalities.

Chapter 5

Reference Implementation

In order to provide the ability to use FaaS services for running CI/CD tasks, it is necessary to implement such a solution within an existing CI/CD system. Moreover, a decision must be made on which existing FaaS service should be used for this implementation.
TeamCity has been the CI/CD system chosen for this implementation due to its high extensibility [1], allowing plugins to be externally developed with access to all the features existing within TeamCity, which meant that the development was not dependent on any changes to internal APIs, and could be considered fully open-sourced.
In terms of FaaS services within the market, AWS Lambda has been chosen as the service to be used due to being one of the most mature solutions existing on the market, having been released in 2014, and due to the AWS' market share (at the time of writing, it has been the leader since its appearance, standing with a 33% of market share [7]). Moreover, AWS Lambda stands as the leader FaaS service used by users, with a market share of 96% [58], at the time of writing. This popularity makes it stand off regarding the amount of online support for issues and its limitations.

5.1 TeamCity

TeamCity is a general-purpose CI/CD product developed by JetBrains that offers a flexible solution to all sorts of development workflows and development practices [59]. TeamCity can offer several integrations that are available through plugins so that the base product is as extensible as possible without providing too many base features that are not useful for the user.

Developing plugins for TeamCity is done through harnessing the power of Spring [57] and its depending injection features. With this, TeamCity can use XML descriptor files that specify the location of the classes Spring should be able to instantiate. These classes implement some specific APIs that represent a TeamCity functionality. This can range from new means of running tasks to new settings tabs for projects. This extensibility allows most of its features through these plugins, which are developed by the TeamCity team and third parties.

[1] It is also important to mention how JetBrains is the host of this research and TeamCity is also one of their products.

5.2 Architecture Research and Development

The complex architecture of TeamCity allowed for several implementations of how FaaS could serve tasks. Two of those possibilities have stood out: Server-side Management and Detached Tasks. While Detached Tasks has been chosen as the approach, it is crucial to overlook both options and understand their trade-offs. All solutions must follow a few requirements:

R1: It must be able to make use of AWS Lambdas Any solution must make use of AWS Lambdas to execute their CI/CD tasks

R2: It must be able to create an AWS Lambda function To invoke a Lambda function, this must be registered within AWS, with all of its characteristics, from the container image to use to the memory and storage size allocated to the containers that are going to be invoked.

R3: It must be able to adapt to the Lambda function's environment All of the characteristics of a Lambda function are re-configurable and must be updated if its settings are changed.

R4: It must be able to access the source code that will be used to run the task All CI/CD tasks are associated with a revision of a source code repository, which must be communicated to the lambda function environment in some way.

5.2.1 Server-side Management

Server-side management is the most straightforward possible architecture to envision for such a solution (*cf.* Section C.1, p. 70). In this option, the CI/CD server is responsible for all the management regarding the AWS Lambda function. Lambda functions' environment would be created (delivering on the requirements **R1**, **R2**, and **R3**), and the source code would be packaged within the server (delivering on the requirement **R4**). The diagrams detailing this architecture can be seen in Appendix C.1 (p. 70).

This option has the following advantages:

- It allows the execution of tasks in a fully serverless environment - in this approach, the CI/CD server might not require any infrastructure maintenance besides the server since all of its tasks would be executed in a Lambda function.
- It removes most of the overhead of tasks in a queue - like in most CI/CD servers, TeamCity places all tasks in a queue until a runner is available to run them. With this option, the time spent on tasks in a queue would be minimal since Lambda functions can virtually scale infinitely

However, there are some drawbacks to such a solution that hinders its possibilities:

- Handling a pipeline of tasks would be complex - pipelines, or build chains as they are called in TeamCity, are the concept of connected tasks. A simple yet typical example of this would be a task to initially compile a project, followed by a division into several tasks that can be

parallelised, such as running unit tests and integration tests. Since Lambda functions are isolated containers, it would be necessary to introduce all of the logic to store the execution state between tasks. In TeamCity, usual runners are already able to do this by default.

- Runners are a means of caching that Lambda functions cannot be - Runners can keep state regarding the source code repository, such as compilation results or the repository source code itself that the server-side cannot do. The lack of caching can create additional overheads since the source code of multiple repositories cannot be kept on the server-side without high storage costs

- Internal changes to TeamCity - due to how TeamCity currently works, all CI/CD tasks must eventually end up connected to an agent. In order to implement this logic, some changes to how TeamCity manages tasks would have to be implemented. This would also mean that this solution would not be backwards compatible.

5.2.2 Detached Tasks

The Detached Task architecture (*cf.* Section C.2, p. 72) uses the concept within TeamCity of Agentless Build Steps [5]. In TeamCity, a certain CI/CD task is able to detach itself from the runner, or agent as they're called in TeamCity, and start executing some work that reports back to the server through a REST API. With this logic, an agent can first package the source-code for the Lambda function to use (delivering on the requirement **R4**) and can call the server to execute the Lambda function (delivering on the requirements **R1**, **R2**, and **R3**), and the Lambda function invocation can report the logs of the task and its result through this API. In this, the Lambda function is invoked through the server to reduce the execution time spent on a runner even further. The diagrams detailing this architecture can be seen in Appendix C.2 (p. 72).

A Detached task brings several interesting characteristics to the user:

- Agents can still cache tasks - since the agent makes the packaging of the environment, it can cache any task that has been executed beforehand, including compilation results. This can dramatically help the quality of the solution.

- Independent solution - this solution does not require any additional change to TeamCity since Agentless Build Steps are already a concept within TeamCity. The API also allows for logging to be quickly reported back.

Albeit its qualities, it is essential to address its most significant disadvantage: it does not entirely remove the usage of other types of runners. While these runners are used for a minimal time, since runners in TeamCity end up caching all repositories they interact with, their existence is nonetheless a decisive factor since it does not allow for a fully serverless experience for running CI/CD tasks.

Notwithstanding, this solution has been the one chosen for this implementation.

5.3 Implementation Details

5.3.1 Development Details

This work has been done with the help of the Cloud Integrations team of TeamCity. As the team responsible for all the integrations with cloud platforms in TeamCity, their expertise actively played a role in the architectural decisions and the implementation.

The work is hosted in a public GitHub repository under the JetBrains' organisation [18]. The development's planning and organisation have been done with JetBrains' issue tracker, YouTrack, and is also publicly available [60]. YouTrack also displays tasks that are being considered for future iterations of the plugin.

5.3.2 Implementation Workflow

With the selected architecture (*cf.* Appendix C.2, p. 72), it was essential to address some implementation details regarding how TeamCity and Lambda should integrate. One of the biggest questions is how to make the source code reach the Lambda invocation. While the Lambda function could be responsible for downloading the repository into its storage, this would mean that it would not be possible to use the results of any previous tasks that could have been executed in a usual runner. Besides, it would imply that much information to understand the repository type (Git, Subversion, Perforce, or others) and the credentials to access them would need to be delivered to the Lambda function. Since these portions of TeamCity are not open-sourced, it would also mean duplicating logic.

With all of that in mind, the fact that any Lambda CI/CD task will first go through an agent could be used. With this, the source code, and the result of any previous task, can be packaged and stored in any service that is responsible for storage. Since this work is being done within the AWS environment, using Amazon S3 [27] as a means of storage was fitting for the problem.

5.3.3 The Introduction of Caching

Caching task executions can be relevant to improve the execution startup time. This can include dependencies or even the build system, since, for instance, Gradle downloads its own instance and caches it outside the source code repository, based on the specified version in the project settings. Since in FaaS services it isn't guaranteed that the storage will be maintained between executions, this caching can be quite relevant. The current solution still possesses its own limitations at this point in time, since only each task has its own cache, which means that some data might be different between different revisions of the same source code (in the case that a new branch adds a new dependency, for instance). Storage of this cache is also done using Amazon S3.

5.3.4 AWS Lambda Limitations

Due to how AWS Lambda is a containerised environment that the clients have very little control over, it is important to discuss its limitations and how they may affect the tasks they are used to running:

- Invocations are limited to 15 minutes - any Lambda function must execute within 15 minutes. This is a limit that is commonly overpast by CI/CD tasks. Due to the size of this problem, the concept of parallel testing has been explored in Subsubsection 5.3.5.
- Both storage size and memory size are limited to 10GB - while this number can be considered relatively high, it is not uncommon to consider a project that would require, for instance, more than 10GB of storage size. For instance, monorepos [47] are a typical example of this.
- Complex environments depend on container images in private Amazon ECR repositories - AWS provides its base container to execute Lambda functions. It is a container based on Amazon Linux 2, and, given that the Lambda functions in this context are developed within the JVM, it has Java 11 installed on it. However, more complex environments that, for instance, need to use any other programming environment would require publishing under their private container repository, created under Amazon ECR, AWS's image repository. This increases the overhead of setup for languages, though it can be reduced by images for all significant runtimes being ready to be published through scripts that can be provided to clients.
- Lambda Functions are limited to Linux-based runtimes - At the time of writing, Lambda functions cannot run any runtime that is not based on Linux, making tasks that require any other operating system impossible to run in such an environment.
- The number of file descriptors is quite low - While this is not a limitation explicit within the Lambda functions' documentation, usage has allowed to determine that the number of files that can be open at the same time, in Linux called file descriptors, is much lower than usual in an operating system. This can be limitating for tasks that require a high I/O interaction. A particular example of this is explored in Subsection 6.3.4.

5.3.5 Parallel Testing

As mentioned in Subsection 5.3.4, Lambda functions are limited to 15 minutes for each invocation. While this is undoubtedly limiting, it also forces the implementation to understand how to handle such a problem and how this could be used to improve the solution compared to regular runners. Therefore, that is where Parallel Testing [50] is introduced. In TeamCity, there is the concept of running tests in parallel. Since TeamCity can capture, from specific build systems, how much time a task took to run and how much time each test took, it is also able to split the tests into a number

of batches that are specified by the user, to reach an optimized division of the tests through each task. These tasks can then be run in parallel in order to decrease the time a complex testing task takes.

With this in mind, it was considered how Lambda functions could be a great candidate for executing such tasks. Due to its virtually infinite scalability, splitting a test task into multiple tasks becomes cheap and reasonably performant. Due to the complexity of the Parallel Testing feature in TeamCity and how it was only finished after the development of the Lambda functions had already begun, no integration between these two features has been developed; however, an invocation can be split with a particular keyword, detailed in Subsubsection 5.3.6. Given the Proof of Concept nature of the current implementation, the goal was only to create an environment where it was possible to compare this solution to the Parallel Testing feature existing in TeamCity for agents.

5.3.6 Task Settings

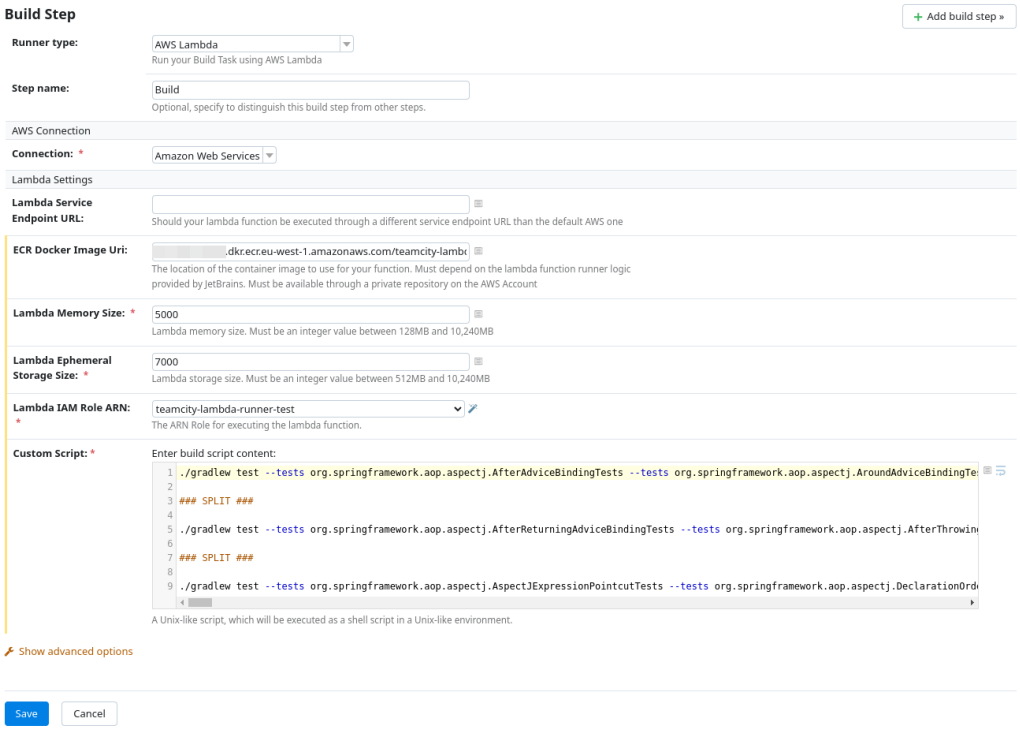

Figure 5.1: AWS Lambda Build Step settings. Each time a user wants to make create a new task that makes use of AWS Lambda as a means to run their task, this is the form they must fill. All form inputs with asterisks are required to be filled, while the others are optional and have default values. A portion of the ECR Docker Image URI has been redacted for privacy reasons.

In order to make use of Lambda Functions, a new type of task had to be created so that TeamCity users could make use of it. This task can be installed through an external plugin, available at TeamCity's plugin marketplace [18]. While choosing to add a new task through the UI, in TeamCity called build step, an option for "AWS Lambda" should appear. TeamCity also allows

a Domain-Specific Language based on Kotlin to be used to fill the details of CI/CD for a project, but that is not currently supported.

The settings that are required to be filled, shown in Figure 5.1 (p. 45), are the following:

- AWS Connection - the means to connect to the AWS account. This connection must be able to provide a series of permissions related to AWS Lambda, Amazon S3, and the permissions service (IAM).
- Lambda Service Endpoint URL - this particular input is specific towards users who run their own private AWS infrastructure and not one of the public ones. It is used to understand where to connect to the Lambda functions. By default, it is not used, and the region presented in the AWS Connection is used.
- ECR Docker Image URI - as mentioned in Subsection 5.3.4, Lambda functions may use a runtime a docker image that has been published to a private repository in Amazon ECR. This field is used to provide such an image. Should it not be filled, the standard Java 11 runtime will be used, which runs under Amazon Linux 2.
- Lambda Memory Size - the amount of memory allocated to the Lambda function. As mentioned in the documentation, the CPU power of a Lambda function will be directly proportionate to the memory size selected.
- Lambda Ephemeral Storage Size - the storage size allocated to the Lambda function. It is called ephemeral because whatever is stored in the container is not guaranteed to be present in another invocation.
- Lambda IAM Role ARN - the IAM role associated with the Lambda function. IAM roles are used to provide which permissions an AWS resource has. A role with the least amount of permissions required is created by clicking the magic wand. The smallest amount of permissions is only to publish logs to Amazon's monitoring service, CloudWatch.
- Custom script - the script that will be executed during the task. In this script, it is also seen in Figure 5.1 (p. 45) the special keyword discussed in Subsubsection 5.3.5 to split the invocation.

5.4 Summary

Section 5.2 (p. 41) discusses the two major possible architectures that have been proposed for the solution. The most straightforward architecture is Server-side management since its principle is the substitution of the usual process for agent management by invoking a Lambda function as the task is ready to be invoked, which could be genuinely efficient and truly serverless. However, this proved to complicate the reproduction of several standard features in CI/CD, such as the pipeline of tasks and the caching of results.

Section 5.2 (p. 41) also described the architecture that has been chosen for this project, the Detached task. In this approach, an agent first starts the CI/CD task, packages the server's environment to call a Lambda function, and detaches itself from the task, being left available to pick up other tasks. This flexible solution can keep most of the original characteristics of a CI/CD agent, though it does rely on the usage of other sorts of agents simultaneously.

Section 5.3 (p. 43) first describes the workflow of an execution of the Lambda function for CI/CD, which packages the environment within the agent and publishes it under an S3 bucket, followed by an invocation to the lambda function from the Server-Side. During the Lambda function's execution, it reports to the Server all of the logs through a REST API. Moreover, it discusses the Lambda environment's limitations that restrict its usage, such as limiting a 15-minute invocation or the low number of file descriptors. The 15-minute is addressed by using parallel testing features, where the CI/CD server can split test invocation into different batches of tasks for them to parallelise.

Section 5.3 (p. 43) finishes with the details required to create a Lambda function CI/CD task. The details range from AWS-specific requirements, like its memory and storage size, to the script that will be executed during the CI/CD task.

Chapter 6

Evaluation and Validation

6.1 Methodology

To evaluate the viability of FaaS for executing CI/CD tasks, it is necessary to analyse their behaviour, both in terms of execution time, throughput and price, compared to current used and tested techniques for leveraging CI/CD tasks.

To that goal, a series of experiments were devised using Amazon ECS, an elastic cluster like the ones discussed in Subsection 2.3.2, which is used in TeamCity to serve runners. It is essential to notice that TeamCity uses its node orchestration to manage the necessary runners. Its logic is based on the fact that each runner can only run a single task at any point in time and that a runner idle by more than a time specified by the user is taken down.

As mentioned in Subsubsection 5.3.5, the objective is to see the behaviour of FaaS when in comparison with the Parallel Testing feature of TeamCity. In order to do so, all projects were created with a step that compiles the project beforehand and then runs the tests in parallel in a series of usual runners. After the task was run a series of times until the test division stabilised (*cf.* Subsection 5.3.5, p. 44), the division of the tests would stabilise, and it could be obtained since it was published as a hidden output of each run. These files could be used to create the custom script used to run in a FaaS task. The FaaS task was also connected to the same initial task that ensured the compilation was done beforehand. With the project and its tasks ready to be executed, the experiment could be executed by running each of these test tasks a number of times.

In order to ensure the experiments are as realistic as possible, the CI/CD process of several open-source projects has been recreated for this project. The projects selected are the following:

- Gradle IntelliJ Plugin - the plugin in order to use Gradle projects, a build system for the JVM, within IntelliJ IDEA, a multi-language IDE by JetBrains.
- Xodus - a transactional schema-less embedded database developed by JetBrains.
- Spring Framework - an application framework for inversion of control container for the JVM platform, which is commonly used for building web applications.

In order to evaluate each task, it is crucial to capture and take into account each of the following metrics:

- Price - indicates the overall cost of the AWS resources used during the execution of the CI/CD tasks
- Throughput - indicates the number of finished tasks over the period that they have executed
- Execution Times (Average, Minimum, Q1, Median, Q3, and Maximum) - to understand the overall performance of the computational resources

Due to the lack of formal research on this subject, these metrics have been chosen based on internal experience from the TeamCity team regarding validating and assessing CI/CD runners. Execution time is fundamental since it is the metric that measures how much time a user will have to wait for feedback on their CI/CD pipeline in optimal conditions (where runners are readily available to execute their). However, throughput can be an even more significant influence on understanding how many users can receive feedback on their pipeline in a period of time. Nevertheless, as all enterprises handle monetary budgets, providing the price of each option can help understand if the performance differences are worth the monetary differences.

The execution time metric is captured through a plugin for TeamCity internally developed for the scope of this thesis, that captures tasks execution details. These detais are scraped through Prometheus [52], a metrics system and a time-series database. Those execution details were then exported from Prometheus in CSV format. In those execution details, it could be found the execution time and a task timestamp, which can be used to infer the other metrics.

To ensure the FaaS' environment is as truthful to the ECS environment as possible, it was ensured that the ECS nodes had the same memory as the FaaS, while ECS nodes had the double CPU units allocated as those of the memory. It is important to remember that Lambda functions do not allow the CPU units allocated to it to be defined, though it is mentioned they will be directly proportionate to the allocated memory size. The ECS cluster used was one instance of the type m6id.12xlarge, which possesses 48 vCPUs and 192GiB of memory.

The solution has not been compared against other cloud-based solutions because all cluster implementations existing for TeamCity use the same orchestration logic, which would lead to similar results.

6.2 The Projects Evaluated

In order to better put into scope the results that have been recorded, an analysis on the projects that were used is necessary.

6.2.1 Gradle IntelliJ Plugin

The Gradle IntelliJ plugin allows users of the IntelliJ IDE to work better with projects that use Gradle as its build system. This plugin can extract information from Gradle by communicating

with a local server that Gradle instantiates to handle builds, but it is also able to provide syntax highlighting and suggestions for writing the files that represent the projects' definitions. This means this is a project with heavy interaction with the filesystem and local servers and its logs, which translates into heavy usage of I/O.

The project is hosted on GitHub [43] and uses Gradle itself. Besides having unit tests for its classes, it also has a series of integration tests which interact with some template projects and try to extract the information the Gradle plugin displays to users, such as the project's dependencies and tasks that can execute. For each project, the plugin's integration tests require opening several files and maintaining an open connection with the Gradle server that outputs information regarding the compilation and the tasks it executes.

The environment created to execute the plugin's unit and integration tests required 4GiB of memory allocated to them.

6.2.2 Xodus

Xodus is a transactional schema-less embedded database. This means that the database runs inside the application, making it self-contained without needing any deployment or administration. Xodus is written in Java and is ready to run on any platform that can execute the Java Virtual Machine. Xodus is a log-structured database, meaning all its information is stored sequentially inside log files. Any data stored in a log file is never modified, as all changes are merely appended to the log. This makes it believe that Xodus also relies heavily on I/O interactions, though without relying on too many open files.

Xodus is a Gradle project hosted on GitHub [44] whose tests also interact with the filesystem, being that it is a database, though it does not open enough files so the number of file descriptors is a problem, as stated in Subsection 5.3.4. This is since its tests rely primarily on using old logs to recreate a database state and then acting upon this state. This leads to very few open file descriptors per test since they reuse the same logs repeatedly.

For this particular project, 3GiB of memory have been allocated per node.

6.2.3 Spring Framework

The Spring Framework is a comprehensive framework aiming to offer more control over the Java Virtual Machine for enterprise applications. Spring offers a wide variety of libraries that can solve several common problems, such as HTTP Servers, WebSockets, Concurrency, Serialization, Database Interaction, Messaging Systems, Dependency Injection and even Common Java annotations.

The Spring Framework is another Gradle project on GitHub [57] whose tests have a wide variety. While some are responsible for maintaining the different language abilities it brings (such as Dependency Injection), there are also modules related to connections to databases, handling different kinds of Web connections (HTTP, WebSockets), and more. For such a complex project, allocating 5GiB of memory per node was necessary.

6.3 Evaluation Scenarios

6.3.1 Xodus Scenario

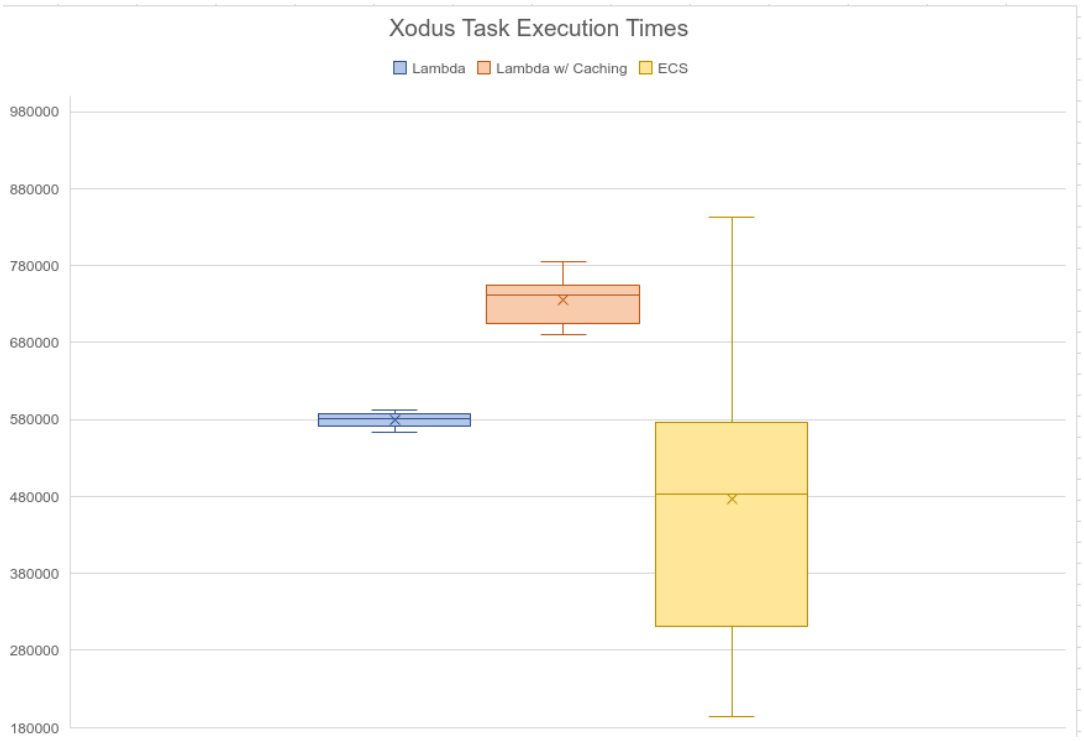

Figure 6.1: Xodus Execution Times.

In Figure 6.1 it is shown how the ECS cluster can provide overall lower response times, though the variance in response times is much higher than FaaS. While in this scenario, the response times seem to favour the ECS cluster, there is a clear distinction when it comes to throughput (*cf.* Figure 6.2, p. 52). As FaaS can scale virtually infinitely, they can handle all of the different executions in parallel, while the ECS cluster cannot scale more than its instance allows. This leads to a more than ten times higher throughput and a lower cost (*cf.* Figure 6.2, p. 52).

6.3.2 Xodus Scenario - Caching

In this scenario, the concept of caching is activated for FaaS. With such caching, FaaS can cache in S3 all of the dependencies required to execute the project and the binaries to execute Gradle. As seen in Figure 6.1, fetching the cache from S3 is slower than fetching them directly from the source.

6.3.3 Spring Framework Scenario

Figure 6.3 (p. 53) shows how FaaS can execute the tests of Spring in a slightly less amount of time while still maintaining its high throughput (*cf.* Figure 6.4, p. 54) (even higher in this case) and

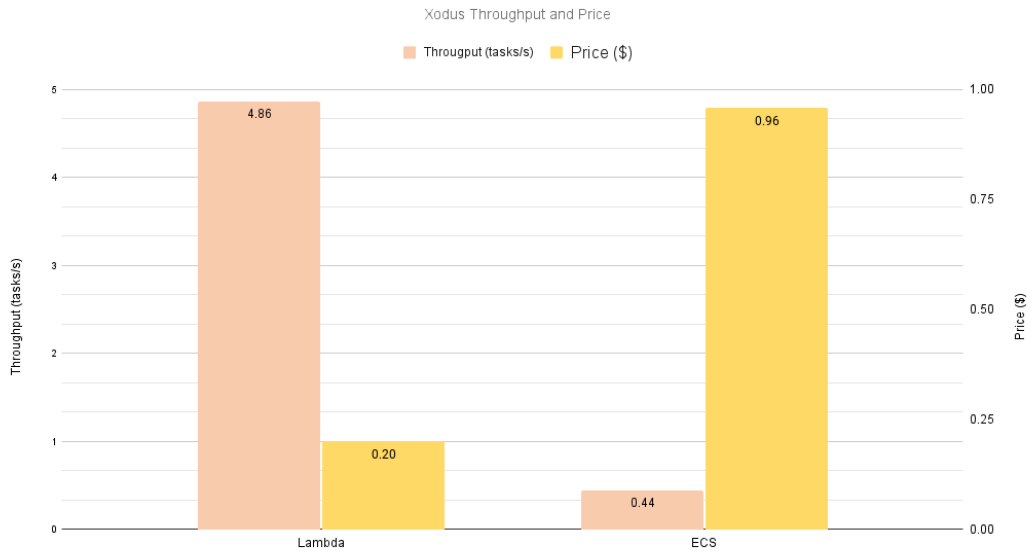

Figure 6.2: Xodus Throughput and Prices for the runs.

low cost (*cf.* Figure 6.4, p. 54). Since the difference in execution times is, on average, just slightly lower, the emphasis is still on the throughput.

6.3.4 Gradle IntelliJ Plugin Scenario

The Gradle IntelliJ plugin tests presented a challenge to execute within the environment of FaaS. Most of its integration tests require instantiating example projects within the filesystem and using it to determine if the plugin can correctly interpret the characteristics of such a project. Therefore, that meant much I/O activity happened during these tests, which led to founding the hidden limitation discussed in Subsection 5.3.4.

Since these tests interacted several times with files, a relatively high number of files were open at any given moment in these tests. In UNIX, those opened files are called file descriptors, and there is a limit set to how many can be opened at a given moment. In FaaS, that number is relatively low and not enough to handle a separation into five batches of tests.

This affirmation is backed by information found in community blogs, where it is commonly discussed how Lambda functions cannot handle many file descriptors open and that there is no way to circumvent this limitation other than reducing the number of open files.

As not even five batches had been enough to run the project's tests successfully, it was determined that Lambda functions were not suitable to run this sort of project.

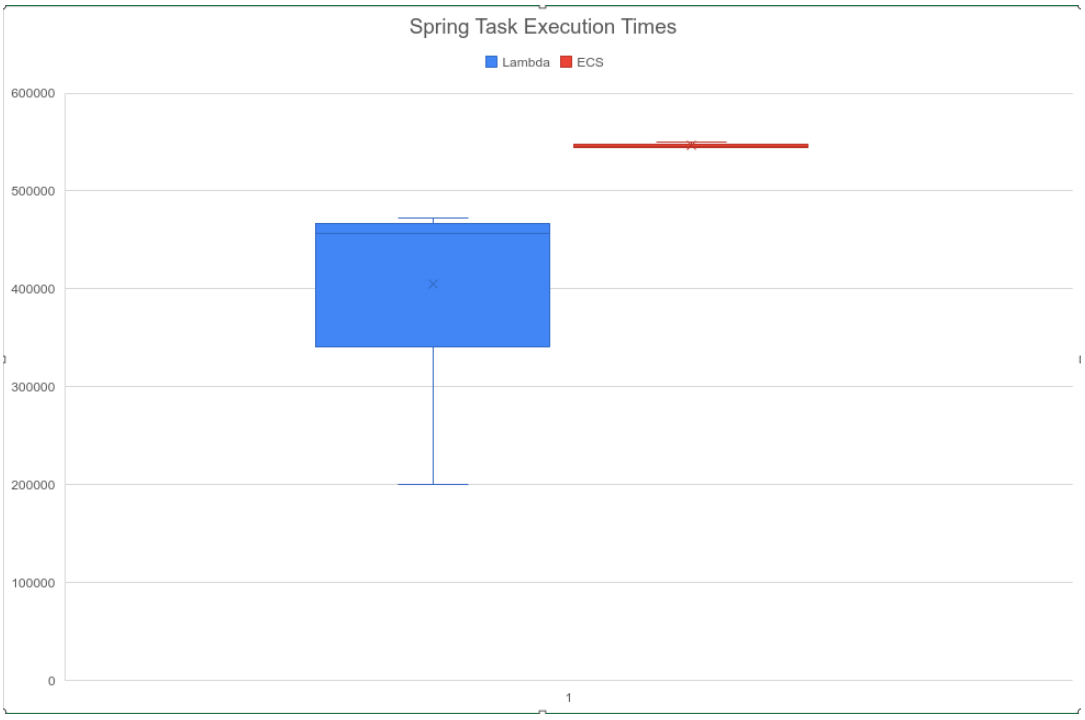

Figure 6.3: Spring Execution Times.

6.4 Threats to Validty

Feldt divides into two categories: internal and external threats [79]. The current solution has some threats to its validity due to some problem that can't be generalised by the results of this work. These are called a external threats:

- All projects were using the JVM with Gradle - due to the current implementation of the Parallel Testing feature's limitations, it is impossible to test any project that is not on the scope of the JVM. Other programming languages and build systems provide other challenges to handle and may even react different to the means of caching used. For instance, the NodeJS language is known for compiling some its dependencies when installing them, which would mean that the FaaS cache could prevent such a compilation from happening;
- More unknown limitations of AWS Lambda - the projects used to test the solution have been able to provide information about some hidden limitations, like the limited number of file descriptors, in the Lambda function's environment. However, there are no guarantees that there are not more that have not been discovered yet.

Moreover, the solution and how the results were treated internally also poses a threat. These are called internal threats:

- The number of projects evaluated does not allow to cover all scenarios - the current projects that have been used for testing do not cover all cases of possible scenarios for CI/CD. The

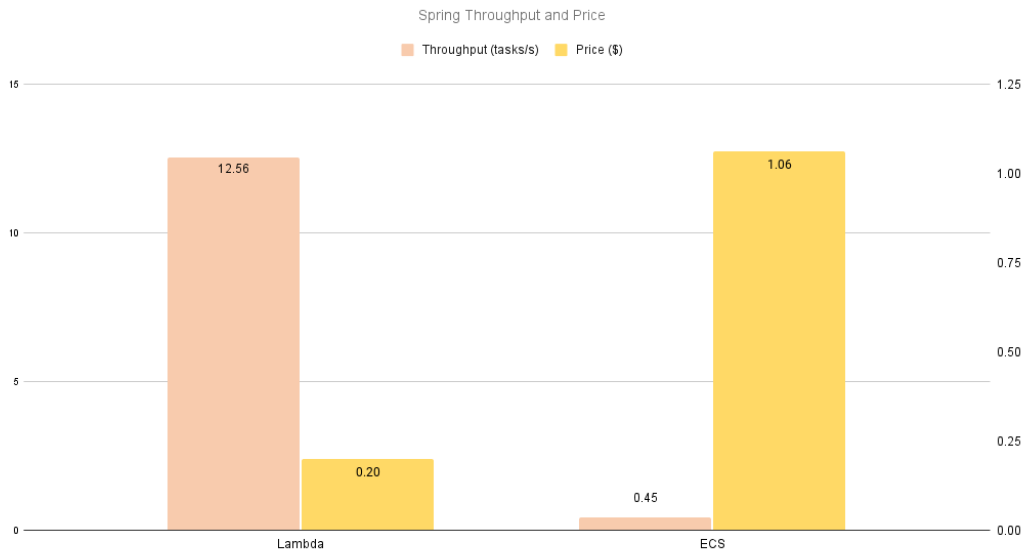

Figure 6.4: Spring Throughput and Prices for the runs.

number is not higher to due time restrictions. A use case worthy of testing that could not be covered is testing that requires instantiating a complex engine, such as functional testing with a browser or game engine. This may lead to unknown problems or potentials that are unknown to this analysis.

- No tests for long-term for running tasks over more extended periods - in the context of TeamCity, most CI/CD tasks are run a few times over a day. While the results of these experiments allow us to infer what would happen in those situations, it would nevertheless be relevant to have results sustaining such claims;

6.5 Replication Package

A replication package was built to allow replication of the various experiments or the validation steps presented throughout this chapter. This package [95] includes:

- TeamCity Project Configurations' Backup - backup of all the CI/CD configurations of all the projects used for these experiments. This backup will create three different projects: *Gradle Intellij Plugin*, *Spring Framework*, and *Xodus*. Each of these projects will have the following task configurations: *Build* (representing the compilation task), *Test* (representing the split tests run in the ECS cluster), and *TestLamba* (representing the split tests run in FaaS). It is important to mention that the replication of these experiments does require a license for TeamCity and to apply a new connection to AWS;
- Tasks Performance tests - all of the results obtained during the experiments led to this chapter's conclusions.

6.6 Summary

Throughout this chapter, the different experiments done to evaluate and validate the reference implementation are detailed, and the conclusions regarding each scenario are discussed. It is essential to focus on some of the limitations of this solution, as exhibited by the project discussed in Subsection 6.3.4 since the lack of control of the container's environment can pose unaware threats that are well documented. These experiments have led us to understand how CI/CD tasks with a high number of concurrent I/O interactions would not fit Lambda functions, and ECS Clusters pose a much safer option.

Nevertheless, the two projects that have been able to execute the tasks with FaaS experimented on (discussed in Subsection 6.3.1 and Subsection 6.3.3) have provided enough data to conclude some scenarios where FaaS can stand out over orchestration platforms such as ECS from a performance perspective. From this perspective, it is crucial to define that performance is achieved by providing a higher number of successful tasks over some time (therefore, throughput), other than tasks finishing in less time (therefore, execution time).

In the tested scenarios, FaaS has proved to be much more performant whenever there is a high throughput of tasks in a short time, usually defined as bursts of tasks. This performance is attributed to its capability to scale virtually infinitely. Lambda functions can do so while still maintaining a lower cost than ECS clusters.

However, if such a high throughput, or even a low one, is spread through a more continuous amount of time, ECS proved to be more performant, albeit more costly. In these scenarios, its lower execution time proves to be more critical since there is a reduced necessity for concurrent tasks.

Nevertheless, whenever there is a low throughput of tasks with occasional triggers, FaaS becomes more performant and cheaper since ECS cluster caches are lost once an agent is disconnected, and these caches are the highest factor in the lower execution times of ECS cluster-based agents.

Chapter 7

Conclusions and Work Plan

7.1 Conclusions

This work proposes an investigation into how FaaS could be leveraged for running CI/CD tasks. We aim to architecture and develop a reference implementation on how FaaS could be used for CI/CD runners and its adoption guidelines and to evaluate the viability of FaaS runners compared to traditional runners and in different scenarios. The critical background concepts for understanding the domain are introduced, such as the characteristics of Cloud Computing, the characteristics of CI/CD and the details of the different computing services used for CI/CD runners.

To understand the current state of the art on the domain of the proposed work, a literature review was done on the different runners offered by CI/CD tools, on the concept of Serverless CI/CD tools, and the challenges customers face with CI/CD tools. The results showed that the support of Virtual Machines and Elastic Clusters is already implemented in most CI/CD tools in the market, but most of them lack any support for Serverless technologies. Moreover, Serverless CI/CD tools can provide a serverless and scalable CI/CD experience while fully abdicating control over the machines where these jobs run. To conclude, several publications discuss the problems with long builds that users still face while also explaining the struggle some users face regarding their infrastructure; nevertheless, very little research on runner specific issues can be found.

In this work, we proposed to fill this literature shortcoming by investigating how FaaS could be leveraged for running CI/CD tasks as an extension to an existing CI/CD tool. The developed solution will be validated by comparing this new extension to the other options for serving cloud-based runners, using cost estimation and response time metrics. The testing process will consist of running CI/CD tasks on several projects of different dimensionalities with different types of tasks and different demand levels.

7.2 Desiderata Revisited

It is crucial to return to the Desiderata (*cf.* Section 4.2, p. 36) and understand how this thesis was able to answer them:

D1: Develop a reference architecture for how to use FaaS for runners: Section 5.2 (p. 41) provides details of two possible architectures that can use FaaS for runners. One of them substitutes the usage of traditional agents as a whole, making this a genuinely serverless solution and giving up a runner's ability to be a cache. The other solution, the one implemented, makes usage of agents at a first step and then detaches the task from the agent and executes it in the FaaS, which keeps the useful caching abilities of traditional runners while allowing the usage by this sort of runner in pipelines.

D2: Implement a FaaS runner for a specific CI/CD tool: Section 5.3 (p. 43) describes the solution developed, using AWS Lambda as the FaaS server and TeamCity as the CI/CD tool. Amazon S3, an object storage service, was used to share the repository and execution caches between the agents and different containers of the FaaS.

D3: Confirm the viability of FaaS for runners: Chapter 6 (p. 48) discusses the viability of FaaS runners, by testing the solution against 3 different projects. While one of them allowed us to learn some hidden limitations of AWS Lambda as a service, (*cf.* Subsection 5.3.4, p. 44), the other two provided information on how FaaS runners can compete with orchestration platforms.

7.3 Research Questions Revisited

It is important to return to the Research Questions (*cf.* Section 4.4, p. 37) to understand how this thesis work was able to answer them:

RQ1: Is it possible to use FaaSs to run CI/CD tasks?

We demonstrated it with the implementation of the plugin described in Chapter 5 (p. 40), since it brings an analysis of a possible implementation of how to leverage FaaS to run CI/CD tasks, the Detached Build approach which makes use of traditional runners to provide a better user experience. Moreover, this chapter still brings information of alternative approach to the one implemented that could bring its benefits, should it be implemented.

RQ2: What metrics should evaluate the viability of a CI/CD runner?

As stated in Section 6.1 (p. 48), price, task throughput, and execution time have been the chosen metrics in order to evaluate the viability of a new type of CI/CD runner. Due to the lack of research in this area, this choice has been made based on the experience of the TeamCity team at evaluating different types of CI/CD runners. Execution time is the metric responsible for understanding how much time an isolated task would take to give feedbackt to a user, while throughput is the metric that allows to understand how these tasks would provide feedback for its users in scale. To conclude, price is the metric that puts into perspective how useful a certain improvement in performance is.

RQ3: What CI/CD tasks can FaaS runners be a viable alternative to other orchestration platforms?

Chapter 6 (p. 48) is able to show scenarios with characteristics where CI/CD benefited from the usage of FaaS runners. These range from tasks with a high throughput that come in bursts of tasks, or tasks that have very little executions over time. It is also revealed that, besides the limitations of cloud provided FaaS, there are also hidden limitations that will hinder the quality of FaaS runners. In the case of AWS Lambda, the number of open file descriptor for an execution is remarkably low.

This work can validate the proposed main hypothesis (*cf.* Section 4.3, p. 36) by understanding that CI/CD tasks can be optimised using FaaS. These tasks must be able to be executed in under 15 minutes (or be splittable into 15 minutes tasks) and would benefit more if they are either a high throughput of tasks in a small period of time or very little executions over time. These tasks will benefit from both higher performance and cost-efficiency.

7.4 Contributions

The work developed in this dissertation resulted in the following contributions to the software engineering state of the art:

- Pseudo-systematic Literature Review onto CI/CD Systems Runners, CI/CD solutions that are considered Serverless and the Current Challenges in the CI/CD space - an investigation was made to the state of the art of the existing solutions to procure Runners in CI/CD systems (*cf.* Section 3.2, p. 25), which allowed to understand how most systems allow for the usage of virtual machines and orchestration platforms, but none allowed the usage of FaaS. When looking into the existing CI/CD solutions that are Serverless by nature (*cf.* Section 3.3, p. 27), it was possible to understand how they offer the most simple experience to run a CI/CD task while heavily relying on container technologies for security purposes and reproducibility, and caching mechanisms, to improve execution times. The existing challenges in CI/CD (*cf.* Section 3.4, p. 29) most users face are attributed to the lack of caching in their tasks, non-deterministic tests, and the workloads of their CI/CD; however, another subset is yet unable to make use of CI/CD systems, as they do not provide appropriate features for their industries;

- Investigation on the possible different architectures for adding FaaS to run CI/CD task - we conceived two valid interpretations for how FaaS could be leveraged by CI/CD systems, accompanied by their pros and cons. One of them allows for an authentic serverless experience using FaaS to run tasks. At the same time, the other still uses agents first to execute any initial steps but to also cache the repository and previously executed steps before detaching itself from the agent and executing the FaaS afterwards. The latter has been the chosen implementation for its caching capabilities and how it can be included in a pipeline of tasks.

- Reference Implementation - one of the architectures mentioned above, the detached task one, has been implemented and tested against different scenarios to understand its benefits

and limitations. This implementation is currently published in TeamCity's Plugin Marketplace [18]. This implementation has outperformed orchestration platforms in cases where there are bursts of tasks, as it can scale virtually infinitely. Furthermore, in scenarios where the throughput of tasks is low with occasional triggers, FaaS proves to be more performant and cheaper as the caches of orchestration platforms are lost after it is disconnected.

7.5 Future Work

As mentioned in Subsection 5.3.5, this plugin still does not automatically integrate with the existing Parallel testing feature due to how recent it is. As this plugin grows in maturity, it would be ideal if users could seamlessly use FaaS to run such tasks without directly using a FaaS task. It is possible to pursue an option where using FaaS is nothing more than selecting an option to run a CI/CD task in FaaS without any more configuration. This option would allow users to use TeamCity's different integrations for build systems.

As this plugin currently only integrates with a single FaaS, AWS Lambda, it would be interesting to pursue integrations with other cloud providers, such as Google or Azure, since this would increase the range of users that can use this different type of CI/CD task.

Moreover, some open-source FaaS solutions, such as OpenFaaS [36], provide an alternative with much more control over the environment where the container has run, as it also runs over a Kubernetes cluster directly. This level of control could have different benefits from the current solution, particularly when it comes to avoiding some of its limitations.

This work was welcomed by the JetBrains team supervising it. As a result, the author was invited to continue working on it on the future, while integrating the company in a full time position.

References

[1] Jenkins EC2 Plugin. Available at https://github.com/jenkinsci/ec2-plugin. Accessed in November 2021.

[2] Jenkins plugin to run dynamic agents in a Kubernetes/Docker environment. Available at https://github.com/jenkinsci/kubernetes-plugin. Accessed in December 2021.

[3] 25 amazing cloud adoption statistics [2022] – zippia. Available at https://www.zippia.com/advice/cloud-adoption-statistics/. Accessed in June 2022.

[4] A Jenkins plugin to deploy VHD and managed disk to Azure VM Scale Set. Available at https://github.com/jenkinsci/azure-vmss-plugin. Accessed in November 2021.

[5] Agentless build step | teamcity on-premises. Available at https://www.jetbrains.com/help/teamcity/agentless-build-step.html. Accessed in June 2022.

[6] AKS vs EKS vs GKE: Managed Kubernetes services compared - A Cloud Guru. Available at https://acloudguru.com/blog/engineering/aks-vs-eks-vs-gke-managed-kubernetes-services-compared. Accessed in November 2021.

[7] Amazon aws accounts for 33 Available at https://finbold.com/amazon-aws-statistics/. Accessed in June 2022.

[8] Amazon EC2. Available at https://aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc. Accessed in November 2021.

[9] Amazon EC2 Container Service Plugin for Jenkins. Available at https://github.com/jenkinsci/amazon-ecs-plugin. Accessed in November 2021.

[10] App engine application platform | google cloud. Available at https://cloud.google.com/appenginecom/. Accessed in January 2022.

[11] Automated software testing in continuous integration (ci) and continuous delivery (cd) - continuous improvement. Available at https://pepgotesting.com/continuous-integration/. Accessed in February 2022.

[12] Autoscaling GitLab CI on AWS Fargate - GitLab. Available at https://docs.gitlab.com/runner/configuration/runner_autoscale_aws_fargate/. Accessed in December 2021.

[13] Autoscaling GitLab Runner on AWS EC2 - GitLab. Available at https://docs.gitlab.com/runner/configuration/runner_autoscale_aws/. Accessed in November 2021.

[14] AWS CodeBuild – Fully Managed Build Service. Available at https://aws.amazon.com/codebuild/. Accessed in November 2021.

[15] AWS CodeDeploy - Automated Software Deployment. Available at https://aws.amazon.com/codedeploy/. Accessed in December 2021.

[16] AWS CodePipeline - Continuous Integration and Continuous Delivery. Available at https://aws.amazon.com/codepipeline/. Accessed in December 2021.

[17] Aws elastic beanstalk – deploy web applications. Available at https://aws.amazon.com/elasticbeanstalk/. Accessed in January 2022.

[18] Aws lambda - teamcity plugin | marketplace. Available at https://plugins.jetbrains.com/plugin/19023-aws-lambda. Accessed in June 2022.

[19] Aws nitro system. Available at https://aws.amazon.com/ec2/nitro/. Accessed in January 2022.

[20] Azure Container Services Plugin for Jenkins enables deploying containerized Java apps to Docker on Azure. Available at https://github.com/jenkinsci/azure-acs-plugin. Accessed in November 2021.

[21] Azure Pipelines Agents - Azure Pipelines - Microsoft Docs. Available at https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/agents?view=azure-devops&tabs=browser. Accessed in November 2021.

[22] Caching in gitlab ci/cd | gitlab. Available at https://docs.gitlab.com/ee/ci/caching/. Accessed in January 2022.

[23] CircleCI Runner Overview - CircleCI. Available at https://circleci.com/docs/2.0/runner-overview/. Accessed in November 2021.

[24] Cloud Build Serverless CI/CD Platform - Google Cloud. Available at https://cloud.google.com/build. Accessed in November 2021.

[25] Cloud comparison: AWS EC2 vs Azure Virtual Machines vs Google Compute Engine - A Cloud Guru. Available at https://acloudguru.com/blog/engineering/cloud-comparison-aws-ec2-vs-azure-virtual-machines-vs-google-compute-engin
Accessed in November 2021.

[26] Cloud functions | google cloud. Available at https://cloud.google.com/functions/. Accessed in January 2022.

[27] Cloud Object Storage – Amazon S3 – Amazon Web Services. Available at https://aws.amazon.com/s3/. Accessed in December 2021.

[28] Compute Engine: Virtual Machines (VMs) - Google Cloud. Available at https://cloud.google.com/compute. Accessed in December 2021.

[29] Compute Engine: Virtual Machines (VMs) - Google Cloud. Available at https://cloud.google.com/compute. Accessed in November 2021.

[30] Container Runtime with Docker Engine - Docker. Available at https://www.docker.com/products/container-runtime. Accessed in December 2021.

[31] Continuous integration: A "typical" process | red hat developer. Available at https://developers.redhat.com/blog/2017/09/06/continuous-integration-a-typical-process. Accessed in February 2022.

[32] EC2 Fleet - Amazon Elastic Compute Cloud. Available at https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-fleet.html. Accessed in December 2021.

[33] Executors - GitLab. Available at https://docs.gitlab.com/runner/executors/. Accessed in November 2021.

[34] Fully managed container solution – amazon elastic container service (amazon ecs) - amazon web services. Available at https://aws.amazon.com/ecs/. Accessed in January 2022.

[35] Gitlab ci/cd for cross-platform unreal engine 4 projects. Available at https://blog.zuru.tech/coding/2020/09/29/gitlab-ci-cd-for-cross-platform-unreal-engine-4-projects. Accessed in February 2022.

[36] Home | openfaas - serverless functions made simple. Available at https://www.openfaas.com/. Accessed in June 2022.

[37] How we built an auto scaling cicd solution for 57 cents/month | by lim yi sheng | helicap engineering | medium. Available at https://medium.com/helicap-engineering/how-we-built-an-auto-scaling-cicd-solution-for-57-cents-month-f6c3adeb171a Accessed in February 2022.

[38] Hypervisor security on the azure fleet - azure security | microsoft docs. Available at https://docs.microsoft.com/en-us/azure/security/fundamentals/hypervisor. Accessed in January 2022.

[39] The hypervisor (x86 and arm) - xen project. Available at https://xenproject.org/developers/teams/xen-hypervisor/. Accessed in January 2022.

[40] Install and Start TeamCity Agents - TeamCity On-Premises. Available at https://www.jetbrains.com/help/teamcity/install-and-start-teamcity-agents.html. Accessed in November 2021.

[41] Jenkins. Available at https://www.jenkins.io/. Accessed in January 2022.

[42] Jenkins Google Cloud Build Plugin. Available at https://github.com/jenkinsci/google-cloudbuild-plugin. Accessed in November 2021.

[43] Jetbrains/gradle-intellij-plugin: Gradle plugin for building plugins for intellij-based ides. Available at https://github.com/JetBrains/gradle-intellij-plugin. Accessed in June 2022.

[44] Jetbrains/xodus: Transactional schema-less embedded database used by jetbrains youtrack and jetbrains hub. Available at https://github.com/JetBrains/xodus. Accessed in June 2022.

[45] Kubernetes. Available at https://kubernetes.io/. Accessed in December 2021.

[46] Linux Containers. Available at https://linuxcontainers.org/. Accessed in December 2021.

[47] Monorepo explained. Available at https://monorepo.tools/. Accessed in June 2022.

[48] Node.js. Available at https://nodejs.org/en/. Accessed in February 2022.

[49] Open Container Initiative - Open Container Initiative. Available at https://opencontainers.org/. Accessed in December 2021.

[50] Parallel tests | teamcity cloud. Available at https://www.jetbrains.com/help/teamcity/cloud/parallel-tests.html. Accessed in June 2022.

[51] Picloud launches serverless computing platform to the public | techcrunch. Available at https://techcrunch.com/2010/07/19/picloud-launches-serverless-computing-platform-to-the-public/?guccounter=1&guce_referrer=aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnLw&guce_referrer_sig=AQAAAA2tj6pkxFEcdVHO4ul0njukTZUTJVbLeSIM4oA1mtg9rQMe80Qml Accessed in February 2022.

[52] Prometheus - monitoring system & time series database. Available at https://prometheus.io/. Accessed in June 2022.

[53] Serverless Compute Engine–AWS Fargate–Amazon Web Services. Available at https://aws.amazon.com/fargate/. Accessed in November 2021.

[54] Serverless computing - aws lambda - amazon web services. Available at https://aws.amazon.com/lambda/. Accessed in January 2022.

[55] Serverless gitlab runner builds on lambda | by marius | sharenowtech | medium. Available at https://medium.com/sharenowtech/serverless-gitlab-runner-builds-on-lambda-ded4b24b3c4f. Accessed in February 2022.

[56] Serverless showdown: AWS Lambda vs Azure Functions vs Google Cloud Functions - A Cloud Guru. Available at https://acloudguru.com/blog/engineering/serverless-showdown-aws-lambda-vs-azure-functions-vs-google-cloud-functions-h-the-tl-dr. Accessed in November 2021.

[57] spring-projects/spring-framework: Spring framework. Available at https://github.com/spring-projects/spring-framework/. Accessed in June 2022.

[58] State of serverless today – bmc software | blogs. Available at https://www.bmc.com/blogs/state-of-serverless/. Accessed in June 2022.

[59] Teamcity: the hassle-free ci and cd server by jetbrains. Available at https://www.jetbrains.com/teamcity/. Accessed in June 2022.

[60] Teamcity (tw) - jetbrains youtrack aws lambda plugin issues. Available at https://youtrack.jetbrains.com/issues/TW?q=for:andre.rocha%20tag:%20%7BAWS%20Lambda%7D%20. Accessed in June 2022.

[61] The Google Kubernetes Engine (GKE) Plugin allows you to deploy build artifacts to Kubernetes clusters running in GKE with Jenkins. Available at https://github.com/jenkinsci/google-kubernetes-engine-plugin. Accessed in November 2021.

[62] Travis CI - Test and Deploy Your Code with Confidence. Available at https://travis-ci.org/. Accessed in December 2021.

[63] What is ci/cd? Available at https://www.redhat.com/en/topics/devops/what-is-ci-cd. Accessed in February 2022.

[64] What is deployment automation? Available at https://www.redhat.com/en/topics/automation/what-is-deployment-automation. Accessed in February 2022.

[65] Salesforce. Available at https://www.salesforce.com/. Accessed in January 2022, 2022.

[66] Rui Pedro Moutinho Moreira Alves and Tiago Boldt Pereira de Sousa. Towards the partitioning and operation of web applications leveraging platform and function as a service.

[67] Guruh Aryotejo, Daniel Y. Kristiyanto, and Mufadhol. Hybrid cloud: bridging of private and public cloud computing. *Journal of Physics: Conference Series*, 1025:012091, 5 2018.

[68] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, Philippe Suter, I Baldini, · P Castro, · K Chang, · P Cheng, · S Fink, · N Mitchell, · V Muthusamy, · R Rabbah, · A Slominski, P Castro, K Chang, P Cheng, S Fink, N Mitchell, V Muthusamy, R Rabbah, A Slominski, V Ishakian, and P Suter. Serverless Computing: Current Trends and Open Problems. *Research Advances in Cloud Computing*, pages 1–20, 12 2017.

[69] Kent. Beck and Cynthia. Andres. *Extreme programming explained : embrace change*. Addison-Wesley, 2005.

[70] David Bernstein. Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing*, 1:81–84, 9 2014.

[71] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Continuous Integration*. 2007.

[72] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 57:21–31, 1 2015.

[73] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. Elasticity in cloud computing: a survey. *Annales des Telecommunications/Annals of Telecommunications*, 70:289–309, 8 2015.

[74] Don Davis, Michael Barr, Toby Bennett, Stephen Edwards, Jonathan Harris, Ian Miller, and Chris Schanck. A java development and runtime environment for reconfigurable computing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1388:43–48, 1998.

[75] Vidroha Debroy and Senecca Miller. Overcoming challenges with continuous integration and deployment pipelines: An experience report from a small company. *IEEE Software*, 37:21–29, 5 2020.

[76] Chavit Denninnart and Mohsen Amini Salehi. Efficiency in the serverless cloud computing paradigm: A survey study. 10 2021.

[77] Thomas F. Dullmann, Oliver Kabierschke, and Andre Van Hoorn. Stalkcd: A model-driven framework for interoperability and analysis of ci/cd pipelines. *Proceedings - 2021 47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021*, pages 214–223, 9 2021.

[78] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. Serverless Applications: Why, When, and How? *IEEE Software*, 38:32–39, 1 2021.

[79] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research - an initial survey. In *SEKE*, 2010.

[80] Cameron Fisher. Cloud versus on-premise computing. *American Journal of Industrial and Business Management*, 08:1991–2006, 9 2018.

[81] Simson. Garfinkel and Harold. Abelson. Architects of the information society : 35 years of the laboratory for computer science at mit. page 72, 1999.

[82] Taher Ahmed Ghaleb, Daniel Alencar da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24:2102–2139, 8 2019.

[83] Marcel Grobmann and Christos Ioannidis. Continuous integration of applications for onos. *Proceedings of the 2019 IEEE Conference on Network Softwarization: Unleashing the Power of Network Softwarization, NetSoft 2019*, pages 213–217, 6 2019.

[84] Axel Wikström Helsinki and Axel Wikström. Benefits and challenges of Continuous Integration and Delivery : A Case Study. 2019.

[85] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda, 2016.

[86] Les Jackson. The ci/cd pipeline. *The Complete ASP.NET Core 3 API Tutorial*, pages 305–347, 2020.

[87] Shashank Mohan Jain. Cgroups. *Linux Containers and Virtualization*, pages 45–80, 2020.

[88] Eric Knauss, Patrizio Pelliccione, Rogardt Heldal, Magnus Ågren, Sofia Hellman, and Daniel Maniette. Continuous Integration beyond the Team: A Tooling Perspective on Challenges in the Automotive Industry. *International Symposium on Empirical Software Engineering and Measurement*, 08-09-September-2016, 9 2016.

[89] Mathieu Lavallée, Pierre N. Robillard, and Reza Mirsalari. Performing systematic literature reviews with novices: An iterative approach. *IEEE Transactions on Education*, 57:175–181, 2014.

[90] Garrett McGrath and Paul R. Brenner. Serverless Computing: Design, Implementation, and Performance. *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, pages 405–410, 7 2017.

[91] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[92] Chnar Mustafa Mohammed and Subhi R.M Zeebaree. Sufficient comparison among cloud computing services: Iaas, paas, and saas: A review. *International Journal of Science and Business*, 5:17–30, 2021.

[93] Denis Polkhovskiy. Comparison between Continuous Integration tools. 6 2016.

[94] R. Arokia Paul Rajan. Serverless architecture - a revolution in cloud computing. *2018 10th International Conference on Advanced Computing, ICoAC 2018*, pages 88–93, 12 2018.

[95] André Rocha and Tiago Boldt Sousa. Leveraging serverless computing for continuous integration and delivery. 6 2022.

[96] R. Owen Rogers. *CruiseControl.NET: Continuous Integration for .NET*, volume 2675. Springer, Berlin, Heidelberg, 2003.

[97] Mojtaba Shahin, Muhammad Ali Babar, Mansooreh Zahedi, and Liming Zhu. Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges. *International Symposium on Empirical Software Engineering and Measurement*, 2017-November:111–120, 12 2017.

[98] Eric Simon. Evaluation of cloud computing services based on nist 800-145 national institute of standards and technology (nist). 2017.

[99] Mohammad UbaidullahBokhari. Cloud computing service models: A comparative study. *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 16–18, 2016.

[100] Joni Virtanen. Comparing Different CI/CD Pipelines. 2021.

[101] Christopher Völker. Suitability of serverless computing approaches. 2018.

[102] Yuping Xing and Yongzhao Zhan. Virtualization and Cloud Computing. *Lecture Notes in Electrical Engineering*, 143 LNEE:305–312, 2012.

[103] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *Computer*, 31:23–31, 5 1998.

Appendix A

Work Plan Gantt Diagram

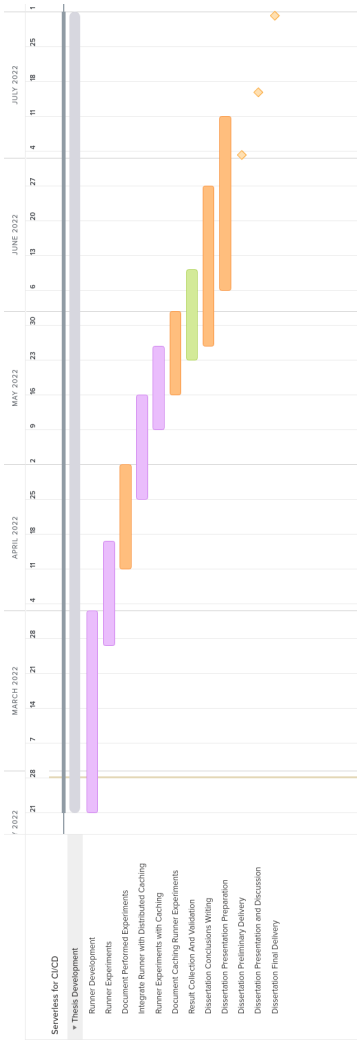

Figure A.1: Work Plan Gantt Diagram.

Appendix B

Literature Review Queries

In this appendix we try to expose the search queries that the *snowballing* technique allowed to create, and exactly which references have been obtained by it. **LRQ1** has been excluded from the table due to it being an exhibition of the current offers by the industry, therefore not directly relying on scientific databases.

Iteration	**Query**	**Results**
1	Since there isn't any formal literature on this topic, it relies on the products' documentation	[40, 6, 1, 4, 2, 61, 20, 9, 42, 33, 12, 53, 13, 46, 21, 23, 30, 6, 1, 2, 61, 20, 9, 42, 33, 12, 53, 13, 46, 21, 23, 30]

Table B.1: Group of search queries that was used to define the references used to answer the Literature Question 1.

Iteration	**Query**	**Results**	**Captured Keywords**
1	(ci/cd OR "continuous integration" OR "continuous deployment") AND FaaS	[24, 100, 14, 16]	serverless, fully managed
2	(ci/cd OR "continuous integration" OR "continuous deployment") AND ("serverless" OR FaaS or "fully managed")	[24, 29, 100, 14, 15, 16, 27]	N/A

Table B.2: Group of search queries that was used to define the references used to answer the Literature Question 2. The last row does not have captured keywords, as it signifies the last iteration.

Iteration	**Query**	**Results**	**Captured Keywords**
1	(ci/cd OR "continuous integration" OR "continuous deployment") AND challenges AND (runners OR agents)	[82, 72, 97, 88]	constraints, executors
2	(ci/cd OR "continuous integration" OR "continuous deployment") AND (challenges OR constraints) AND (runners OR agents OR executors)	[82, 62, 84, 72, 97, 88, 75]	N/A

Table B.3: Group of search queries that was used to define the references used to answer the Literature Question 3. The last row does not have captured keywords, as it signifies the last iteration.

Appendix C

Architecture Diagrams

In this appendix, we display the different diagrams that help comprehend the two proposed architectures in Section 5.2 (p. 41).

Both architectures possess the same components. The only difference between them is how the services are split:

- *VCSBuildListener* - this was an already existing component within TeamCity responsible for listening to the triggers related to Version Control Systems (VCS), such as a commit to git. It determines which tasks are to be executed upon such a trigger.
- *LambdaBuildProcess* - this is an introduced component whose responsibility is to interpret the AWS Lambda CI/CD task and prepare the environment for its execution. At first, it will prepare the repository and the script to be executed. This repository is stored within Amazon S3. The *Lambda* component is then invoked with the information regarding the current task and how to access the repository through Amazon S3.
- *Lambda* - this component was introduced to control the lambda function's execution. Upon the call from *LambdaBuildProcess*, it will obtain the known information on the project and the task to be executed. This information will be used to understand if the Lambda function has already been created and is updated. If it has not been, it will be created, updated, and waited until it is ready to be called. Once that has finished, it will invoke the AWS Lambda with the details it has received from *LambdaBuildProcess*.
- *AmazonS3* - An AWS service that is able to store any sort of objects from any source [27], including *AwsLambda* and the *Server*.
- *AwsLambda* - A FaaS provided by AWS [54]. It is used to execute the CI/CD task, while reporting the execution logs to the *Server* through the *Lambda* component. At the end of its execution, it stores a cache of the execution into [27].

C.1 Server-side Management

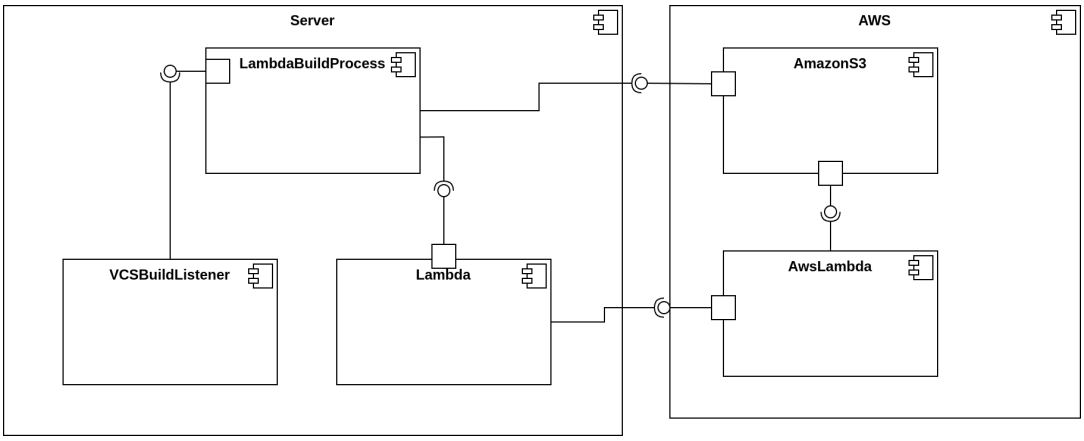

Figure C.1: Component diagram for the Server-side management option for the Architecture. Whenever a task is pushed that is supposed to make use of AWS Lambda, the server would be in charge of preparing an environment for the lambda function to invoked and would also invoke it.

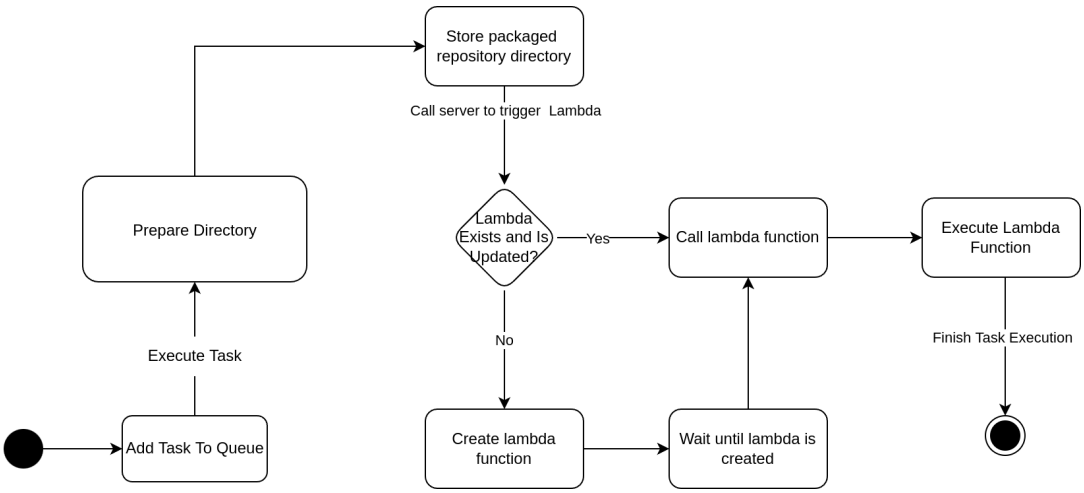

Figure C.2: State diagram for the Server-side management option for the Architecture. Whenever a task is pushed that is supposed to make use of AWS Lambda, the server would be in charge of preparing the directory of the repository and then preparing an environment for the lambda function to invoked and would also invoke it.

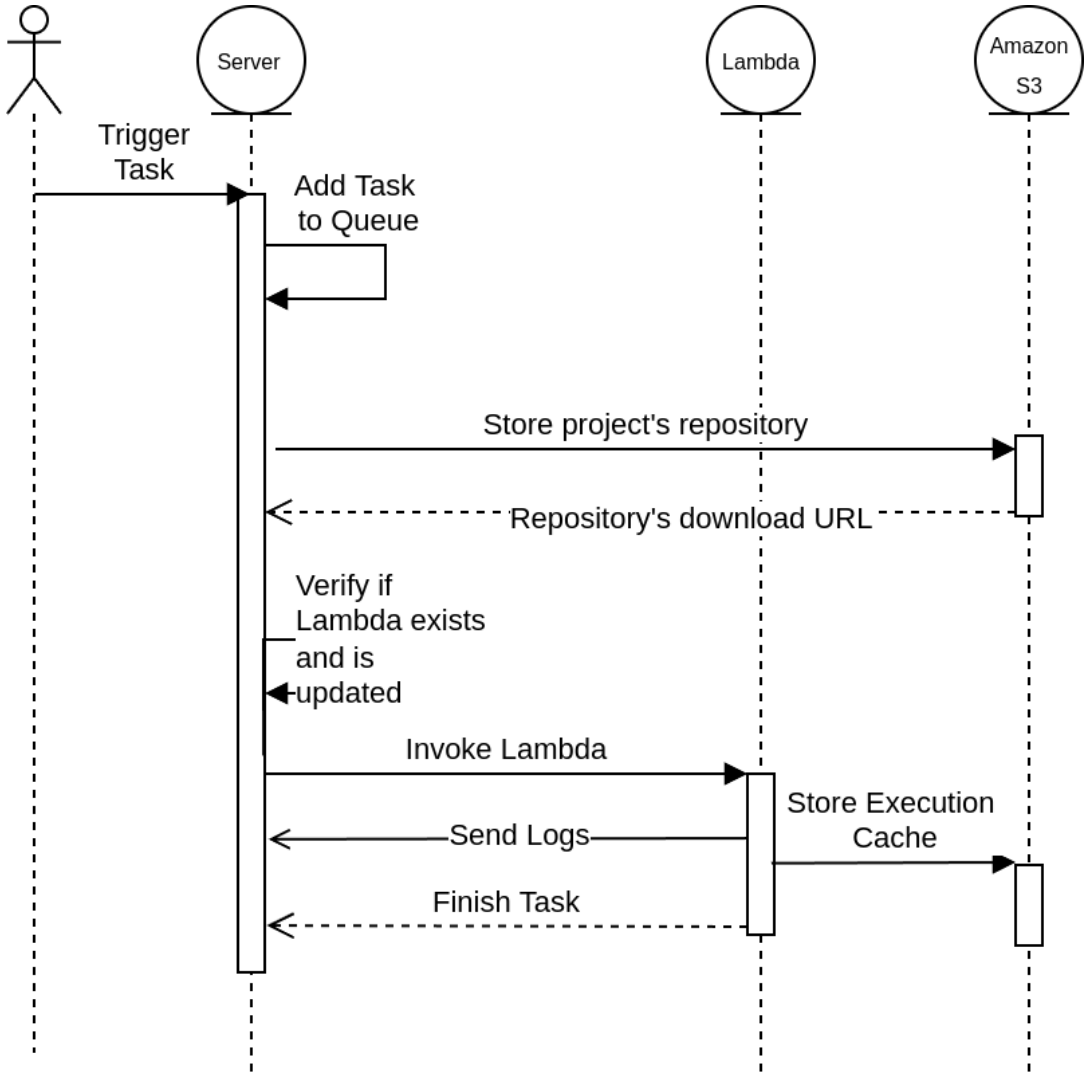

Figure C.3: Sequence diagram for the Server-side management option for the Architecture. Whenever a task is pushed that is supposed to make use of AWS Lambda, the server would be in charge of preparing the directory of the repository and store it within Amazon S3. The server would then prepare an environment for the lambda function to invoked and would also invoke it. The Lambda function would execute the task, reporting its logs back to the server, and would store the execution cache at the end of the execution.

C.2 Detached Task

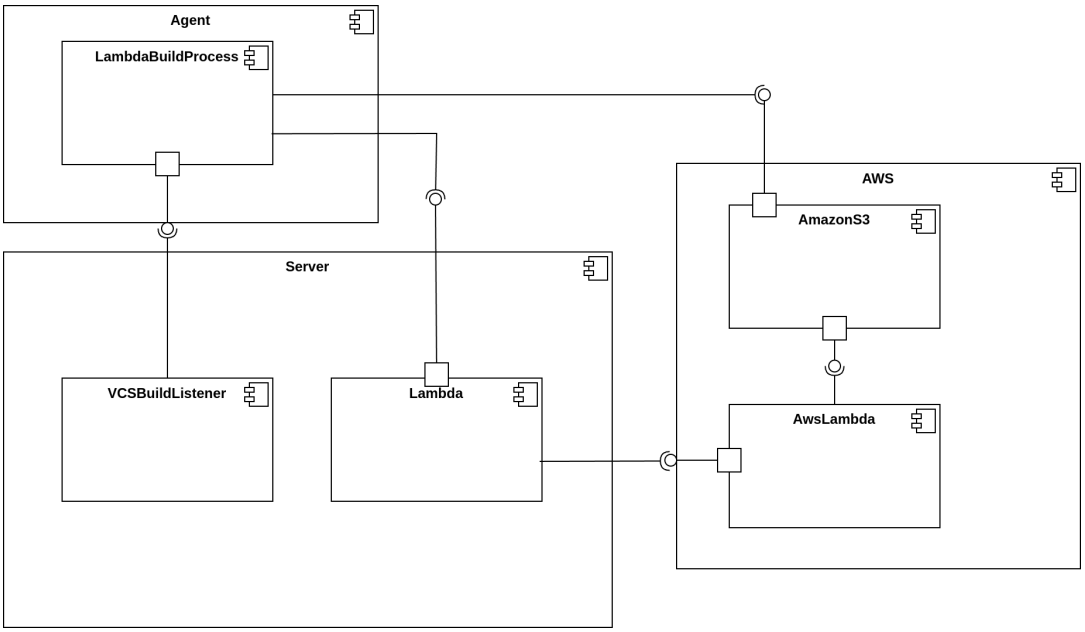

Figure C.4: Component diagram for the Detached task option for the Architecture. Whenever a task is pushed that is supposed to make use of AWS Lambda, the server would still trigger a usual runner which would package the environment of the current task for the server to execute the Lambda function.

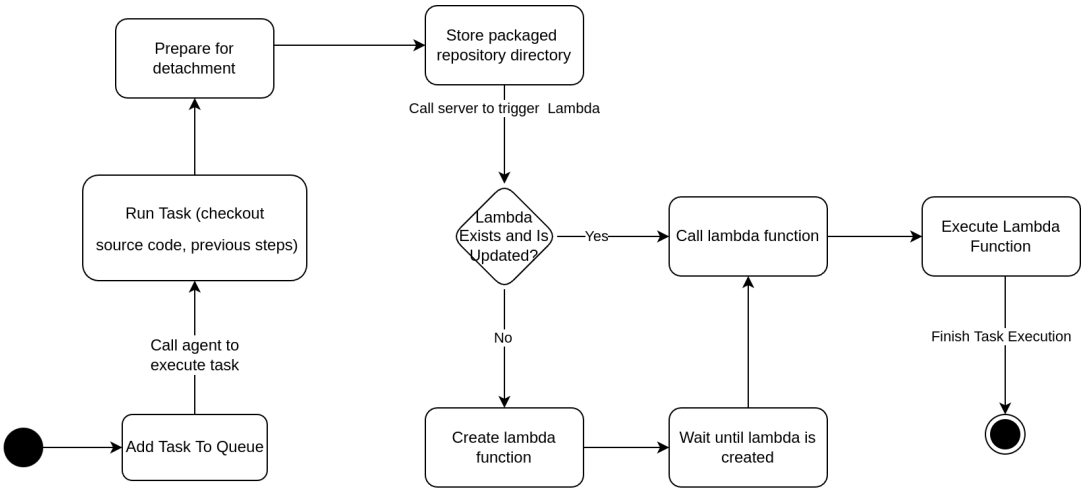

Figure C.5: State diagram for the Detached task option for the Architecture. Whenever a task is pushed that is supposed to make use of AWS Lambda, the server would still trigger a usual runner which first execute any step before the AWS Lambda one. The agent would then package the environment and store in Amazon S3 of the current task for the server to execute the Lambda function.

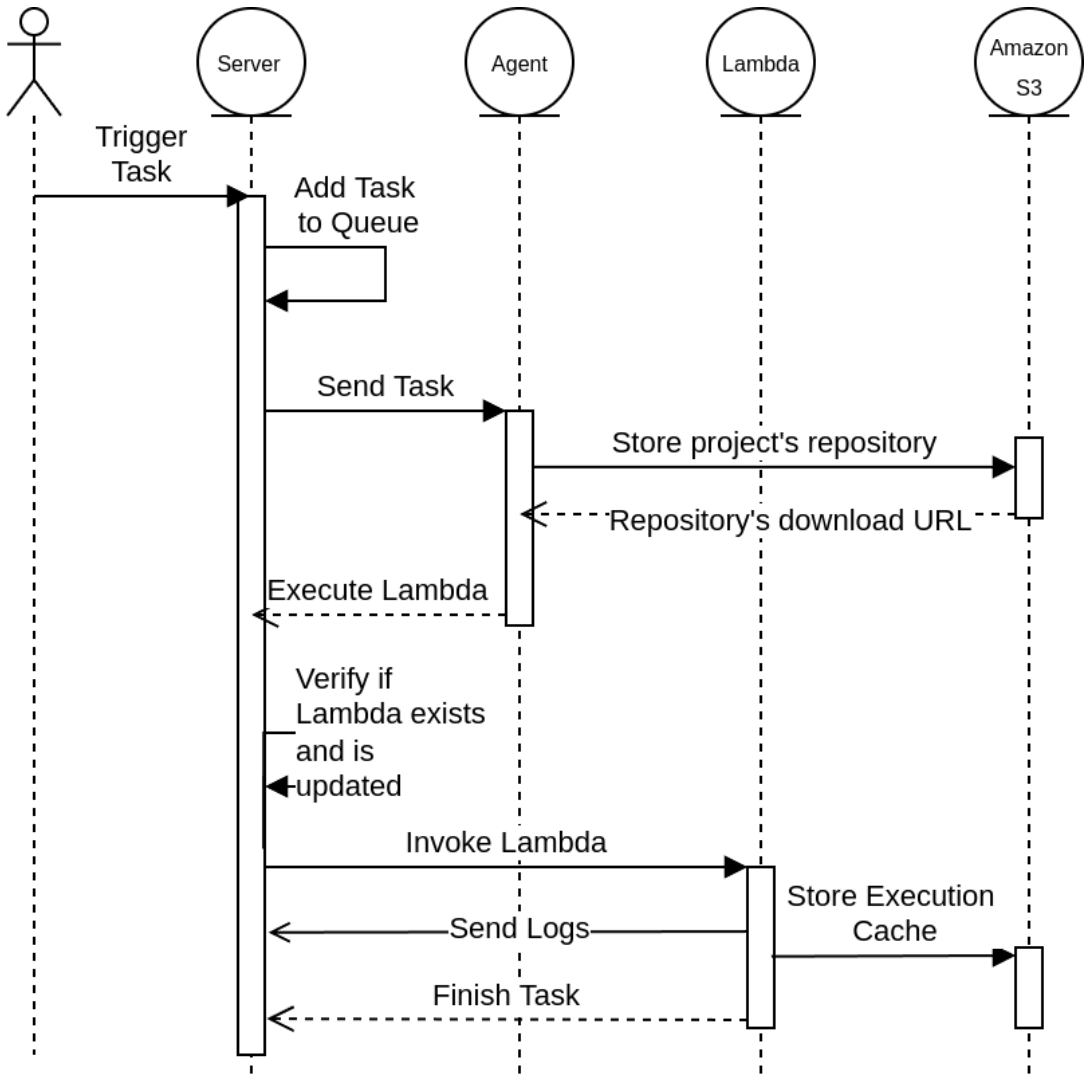

Figure C.6: Sequence diagram for the Detached task option for the Architecture. Whenever a task is pushed that is supposed to make use of AWS Lambda, the server would send the task details to an agent, which would be responsible for sending the packaged project's repository so Amazon S3. The agent would then message the server to execute the Lambda function. The Lambda function would execute the task, reporting its logs back to the server, and would store the execution at the end of the execution.