FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards the Partitioning and Operation of Web Applications leveraging Platform and Function as a Service

Rui Pedro Moutinho Moreira Alves

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Tiago Boldt Pereira de Sousa

Second Supervisor: André Monteiro de Oliveira Restivo

July 12th, 2021

Towards the Partitioning and Operation of Web Applications leveraging Platform and Function as a Service

Rui Pedro Moutinho Moreira Alves

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. João António Correia Lopes

External Examiner: Prof. João Miguel Fernandes

Supervisor: Prof. Tiago Boldt Pereira de Sousa

July 12th, 2021

Abstract

Cloud Computing democratised access to large scale computation resources, offering on-demand solutions to deploy applications at vast scales and featuring service models with different degrees of abstraction. These models, such as Platform and Function as a Service (PaaS and FaaS), differ in billed costs, user response time, and level of abstraction in the resources managed, each having different strengths and drawbacks.

Models that offer less degree of abstraction, such as PaaS, are known to feature higher computational capabilities and a better cost per unit of computation, while their counterparts with higher degrees of abstraction, such as FaaS, are more widely geographically distributed and inherently scale horizontally. Although it is possible to use the best each has to offer, deployments featuring multiple service models (hybrid deployments) are typically not popular since they pose a number of implementation and management challenges.

Our literature review shows that these challenges exist in terms of system observability, monitoring, traffic routing, reconfiguration, infrastructure distribution decision, and others. Existing work and studies in these hybrid deployments are either entirely manual (high development effort), lacking in terms of flexibility (working only with specific services of particular cloud providers), or only focused on specific software systems.

We believe that, given that PaaS and FaaS provide complementary benefits for web applications, developers can leverage them to optimise their applications. As such, we set ourselves to demonstrate that:

> *PaaS and FaaS provide complementary benefits for web applications. Developers can leverage them to optimise their applications, by (1) partitioning their applications to be deployed to the best service model for each functionality, (2) using a dynamic routing strategy to forward requests to the proper resource, and (3) allow the application to dynamically reconfigure itself by moving code between PaaS and FaaS (and vice-versa) in a way that optimises infrastructure costs, client latency, and response time.*

To do so, we (1) conceive a set of techniques for dynamic routing between PaaS and FaaS, with the aim of minimising the resulting client latency, which are empirically validated through load testing, and (2) demonstrate their validity by engineering a reference implementation of a plugin prototype that facilitates the programmatic partitioning of web applications into PaaS and FaaS and their deployment to the cloud.

The evaluation process of the conceived routing techniques shows that moving the routing service to the FaaS layer results in the optimal configuration due to this model's wider geographical distribution. Additionally, we successfully partition and hybrid-deploy two open-source web application projects using our reference plugin implementation with minimal development effort, thus demonstrating the validity of this solution. Finally, we contribute to the state of the art of software engineering and cloud computing by preparing our conclusions for publication, as well as to the web development community by open sourcing our plugin.

Resumo

A Computação em *Cloud* democratizou o acesso a recursos computacionais de grande escala, proporcionando o acesso a soluções mediante solicitação para distribuir aplicações a larga escala, contando com modelos de serviço com vários níveis de abstração. Estes modelos, tal como *Platform* e *Function as a Service* (*PaaS* e *FaaS*), diferem a nível de custos cobrados, tempo de resposta ao utilizador, e nível de abstração dos recursos geridos, com diferentes vantagens e desvantagens.

Modelos que oferecem menor grau de abstração, tal como *PaaS*, contam com maior capacidade computacional e um menor custo por unidade de computação, enquanto que modelos com maior grau de abstração, tal como *FaaS*, estão mais amplamente geograficamente distribuidos e escalam horizontalmente de forma inerente. Apesar de ser possível utilizar o melhor que cada modelo oferece, *deployments* que incluam vários modelos de serviço (*deployments* híbridos) não são tipicamente populares, visto que apresentam diversos desafios de implementação e gestão.

A nossa revisão de literatura mostra que estes desafios existem em termos de observabilidade, monitorização, roteamento, reconfiguração, decisão de distribuição de infraestrutura, entre outros. Estudos existentes relativamente a *deployments* híbridos mostram soluções ou totalmente manuais (elevado esforço de desenvolvimento), com falhas a nível de flexibilidade (funcionando apenas com serviços ou fornecedores de *cloud* específicos), ou apenas orientadas a sistemas específicos.

Acreditamos que, dado que *PaaS* e *FaaS* proporcionam benefícios complementares para o desenvolvimento de aplicações *web*, os programadores podem geri-los a fim de otimizar as suas aplicações. Desta forma, propomos demonstrar que:

> *PaaS e FaaS proporcionam benefícios complementares para aplicações web. Os programadores podem geri-los a fim de otimizar a sua aplicação (1) através da sua partição para serem deployed para o modelo de serviço mais adequado para cada funcionalidade, (2) utilizando uma estratégia de roteamento dinâmico para encaminhar pedidos para os serviços corretos, e (3) permitindo que a aplicação se reconfigure dinamicamente movendo partes de si entre PaaS e FaaS (e vice-versa) de forma a otimizar custos de infraestrutura, latência do cliente e tempo de resposta.*

Para tal, (1) concebemos um conjunto de técnicas de roteamento dinâmico entre *PaaS* e *FaaS*, com o objetivo de minimizar a latência do cliente, que são empiricamente validadas através de testes de carga (*load testing*), e (2) demonstramos a sua validade através de uma implementação de referência de um *plugin* que permite, programaticamente, a partição de aplicações *web* em *PaaS* e *FaaS*, bem como o seu *deployment* para a *cloud*.

O processo de avaliação das técnicas de roteamento mostra que a alocação do serviço de roteamento à camada *FaaS* resulta na configuração ótima devido à distribuição geográfica mais ampla deste modelo de serviço. Adicionalmente, efetuamos a partição e *deployment* híbrido de projetos *web* publicamente abertos utilizando o nosso *plugin* com um esforço de desenvolvimento mínimo, demonstrando assim a validade desta solução. Finalmente, contribuímos para o estado da arte da engenharia de *software* e de computação em *cloud*, preparando as nossas conclusões para publicação, e publicando abertamente o nosso *plugin* para a comunidade de desenvolvimento *web*.

Acknowledgements

Firstly, I would like to thank my supervisors, Professors Tiago Boldt de Sousa and André Restivo, who provided me with guidance and all the help I needed throughout this dissertation, and whose feedback and ideas contributed to the success of this work. Additionally, I would also like to thank Professor Hugo Sereno Ferreira, who followed my work closely throughout the whole process, and who provided valuable advice and insights [39].

Then, I would like to give the biggest word of appreciation I can give to my mother, Conceição, who always provided me with all I needed and who was always present for me throughout all of my academic journey, and all of my life — Thank you for everything.

To my girlfriend, Filipa, who was always there to cheer me up and remind me that there's more in life than work, and who was always with me since day 1 in this long, however short, journey through university.

Lastly, I would like to thank my friends, for all the moments of fun, laughter, and much more. A special thanks to Miguel for our never-ending arguments about nothing and everything, which always motivated me to work harder.

Rui Alves

"Do. Or do not.
There is not try."

Yoda

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command-line Interface
DNS	Domain Name System
FaaS	Function as a Service
GCP	Google Cloud Platform
IT	Information Technology
IaaS	Infrastructure as a Service
IoT	Internet of Things
JSON	JavaScript Object Notation
NIST	National Institute of Standards and Technology
OS	Operating System
PaaS	Platform as a Service
PoC	Proof of Concept
QoS	Quality of Service
RT	Routing Technique
SLA	Service-level Agreement
SaaS	Software as a Service
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
VM	Virtual Machine
VPS	Virtual Private Server

Chapter 1

Introduction

This chapter introduces the problem under study, describing its motivation and context, as well as 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

Cloud Computing democratised access to large scale computation resources, offering developers an on-demand solution to deploy their applications at vast scales (often across multiple regions), with guaranteed flexibility and Quality of Service (QoS) [105]. This computing model was popularised in 2007 by Google and IBM — although AWS had already introduced their *Elastic Compute Cloud* (EC2) [8] in the previous year [5], and its origins in cluster and grid computing date back to the 1980s [25] — contrasting to more traditional on-premise architectures.

Cloud solutions are available through multiple service models, such as Infrastructure and Platform as a Service (IaaS and PaaS), which is introduced in Chapter 2 (p. 5). They differ based on the level of resources that is managed by the Cloud provider, and each has its own perks and drawbacks. However, there is no *"silver bullet"* when it comes to Cloud service models.

Function as a Service (FaaS), which is commonly interchangeably referred to as Serverless in the literature, is a Cloud Computing execution model that became popular throughout the past

decade due to the architectural shift of enterprise systems towards containerisation and microservices [19]. In FaaS, the Cloud provider executes computational tasks (functions), which are more geographically distributed than in the other models, and automatically manages scalability, which has brought a few advantages over the other models.

These computing models differ in billed costs, user response time, and levels of abstraction in the resources managed. Depending on the specific scenario, one model may be more adequate than another. If this is the case, there should be a way to make use of the best of each one, within the same system.

1.2 Motivation

Cloud Computing has been consistently growing throughout the past decade, mainly due to the progress of computing hardware and subsequent cost lowering, an increasing number of related Information Technology (IT) research trends [101], and the wide range of applications.

Even though the leading Cloud providers — Microsoft Azure, Amazon Web Services (AWS), Google Cloud Platform (GCP), and IBM — currently retain most of the market share in the Cloud Computing services industry [6], the need for a variety of new architectures that can accommodate recent research trends (such as the growth of IoT, fog and mobile Edge Computing, and Serverless Computing) is ever-changing. [101].

With the addition of more layers of abstraction for developers, single-unit tasks (functions) can be executed when requested without the need for the pre-allocated infrastructure to be running at all times, and abstracting every infrastructure management aspect from the developer, except for the function's source code. This Cloud Computing model is known as FaaS, and provides several advantages, such as its detailed level of management granularity — functions.

That being said, every Cloud Computing model features strengths and weaknesses. It would be ideal if web applications could benefit from deployments with multiple service models and using the best of each model, that is, hybrid deployments. However, due to technical challenges and limitations (such as cumbersome infrastructure setup and management), these types of deployments are not a common practise.

Thus, it would be ideal if an engineer could define parts of the application that could move among different models at runtime (automation at compile-time has already been explored in the literature [82]) and, with this in mind, this work intends to take the first steps towards practical hybrid deployments.

1.3 Problem

The different Cloud Computing models follow a *pay only for what you use* philosophy that only utilises the amount of resources necessary to satisfy the levels of demand in a scalable and cost-efficient way. However, they differ in the level of granularity of the billed computational unit: whereas in IaaS it is typically a Virtual Private Server (VPS), in FaaS it is a single function.

Cloud models such as IaaS and PaaS feature higher computational capabilities and a better cost per unit of computation. On the other hand, FaaS solutions inherently scale horizontally [46] and are more widely geographically distributed, which results in the improvements of metrics such as latency and availability. Nevertheless, FaaS can suffer from cold starts, execution time deviations and state management issues [19, 74].

However, to reach a state where a given system (or parts of it) can be moved from IaaS/PaaS to FaaS and vice-versa, engineers must redesign the architecture of their application. This process is hard to assess since (1) it is hard to estimate the required effort to redesign the application, (2) it is hard to ensure that it will result in improvements, (3) it requires significant manual development effort, and (4) it is troublesome to revert, due to the architectural changes applied to the application [71, 59].

Additionally, deploying projects with multiple levels of service models (such as hybrid deployments featuring PaaS and FaaS) is not a trivial task. It poses issues and challenges in terms of observability, monitoring, traffic routing, reconfiguration, infrastructure distribution decision, *et cetera*. Although we believe that numerous advantages arise from these hybrid deployments by benefiting from the strengths of each cloud service model, the aforementioned cumbersome factors may be the origin of these deployments not being a popular choice.

With this research, we set ourselves to explore how an engineer can programmatically define parts of their web application that should run and be deployed to different cloud service model layers (such as FaaS and PaaS), which would result in significantly less development effort. The problem under study is thoroughly formalised in Chapter 4 (p. 33).

1.4 General Goals

Although all the challenges presented in Section 1.3 (p. 2) are relevant to be explored, this work focuses on the partitioning and operation of web applications leveraging FaaS and PaaS, by investigating techniques that facilitate the programmatic partitioning of web applications into these service models and their deployment to the cloud:

Manage traffic routing in hybrid web applications:

Routing traffic among different service model layers is not a trivial task, since it may lead to issues such as communication delays and bottlenecks. Additionally, different routing architectures may lead to different advantages and drawbacks. We intend to conceive and experiment upon a set of distinct routing techniques, with the aim of minimising the resulting routing latency;

Study the requirements to operate and leverage hybrid web applications:

In hybrid scenarios, when there are multiple providers or different types of services being used, an application's deployment is a cumbersome task. We want to understand how we can manage these deploys whilst minimising development effort, and thus abstracting most configuration from developers;

Investigate how to programmatically partition applications at the source code level:
The process of refactoring a system and redesigning its architecture for its deployment to the cloud is a demanding endeavour. We believe that, by identifying partitioning sections using annotations at the source code level, developers would have an expeditious way to tackle this task, in a way that significantly decreases development effort;

Provide a reference implementation plugin that facilitates the above tasks:
Implementing a plugin that addresses these goals would both provide a reference implementation for the studied methods and techniques, and contribute to the web development community with a tool that facilitates hybrid web application management and deployment.

1.5 Document Structure

This document is composed of seven chapters, structured as following:

- Chapter 1 (p. 1), **Introduction**, introduces the problem under study, as well as its motivation, goals and validation process;
- Chapter 2 (p. 5), **Background**, explores the background's key concepts that are needed to fully understand this work;
- Chapter 3 (p. 15), **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. 33), **Problem Statement**, formalises the problem under study, and elaborates upon the scope and main focus of this work;
- Chapter 5 (p. 39), **Techniques for Dynamic Routing between PaaS and FaaS**, details the implementation and validation of the dynamic routing techniques that were conceived in this work;
- Chapter 6 (p. 59), **Reference Implementation**, discusses the various aspects of the implementation, experimentation and validation of the reference plugin prototype;
- Chapter 7 (p. 71), **Conclusions**, summarises the developed work, its main contributions, and discusses the possibilities of future work.

Chapter 2

Background

On the previous chapter we introduce this dissertation. This chapter reviews the key concepts used throughout this work. Firstly, Section 2.1 introduces the main concepts of Cloud Computing, describing its origin and characteristics, as well as its deployment and service models. Then, Section 2.2 discusses the use of source code annotation to add metadata to programs. The concept of reflection of a system is defined in in Section 2.3. Finally, Section 2.4 summarises the key concepts that characterise this work's background. This chapter may be skipped by readers which are familiar with web application development and their operation with cloud computing.

2.1 Cloud Computing

Cloud Computing is an on-demand computing model that is defined by National Institute of Standards and Technology (NIST) [80] as:

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"*

This model was popularised in 2007 by Google and IBM [5] (although AWS had already introduced their EC2 solution [8] in the previous year), democratising access to large scale computation resources by offering developers an on-demand solution to deploy their applications at vast scales. It features a set of main characteristics (described in Subsection 2.1.1), four service models (detailed in Subsection 2.1.2, and four deployment models (discussed in Subsection 2.1.3).

2.1.1 Characteristics

Cloud Computing features a set of characteristics that may be regarded as conceptional, technical, business-related or even user-experience-related [80, 47].

- **On-demand self services** allow clients to automatically use computing resources (*e.g.*, server time and storage space) without the need for direct human contact with the Cloud provider;
- **Broad network access** indicates that computing resources are available over a network, being heterogeneously accessible by different types of client-platforms;
- **Resource pooling** stipulates that the Cloud provider's resources are simultaneously accessible by multiple clients, (re)assigning the physical/virtual resources according to demand. The infrastructure's control and details (*e.g.*, location) are abstracted from the client;
- **Rapid Elasticity** means that resources can be, automatically or not, scaled according to clients' demand by means of releasing or provisioning additional resources;
- Cloud providers typically offer a **Measured Service**, monitoring and controlling resources' usage, thus providing service transparency for both the client and the Cloud provider;
- Cloud Computing is also conceptually **Service-oriented**. Through means of abstraction and virtualisation, clients can access resource service models, classified as IaaS (Infrastructure as a Service), PaaS (Platform as a Service) and SaaS (Software as a Service);
- Services are **Loosely Coupled**. Cloud infrastructure is physically or logically separated, and clients connect loosely with servers/Cloud providers, which results in low levels of control-dependency among Cloud services.

This set of characteristics offer clients flexibility, both technical and business-related, one of the main reasons for its popularity.

2.1.2 Service Models

The Cloud service models separate the client's and the Cloud provider's managing responsibilities at different abstraction levels, in opposition to a traditional on-premise model where the client manages everything (from the application itself to the networking layer). The differences in abstraction levels are illustrated in Figure 2.1 (p. 7).

2.1.2.1 IaaS - Infrastructure as a Service

In this model, the networking, server, storage and virtualisation parts are abstracted for the client, being managed by the Cloud provider [80]. The client is able to expand/reduce the amount of resources of the underlying infrastructure on demand.

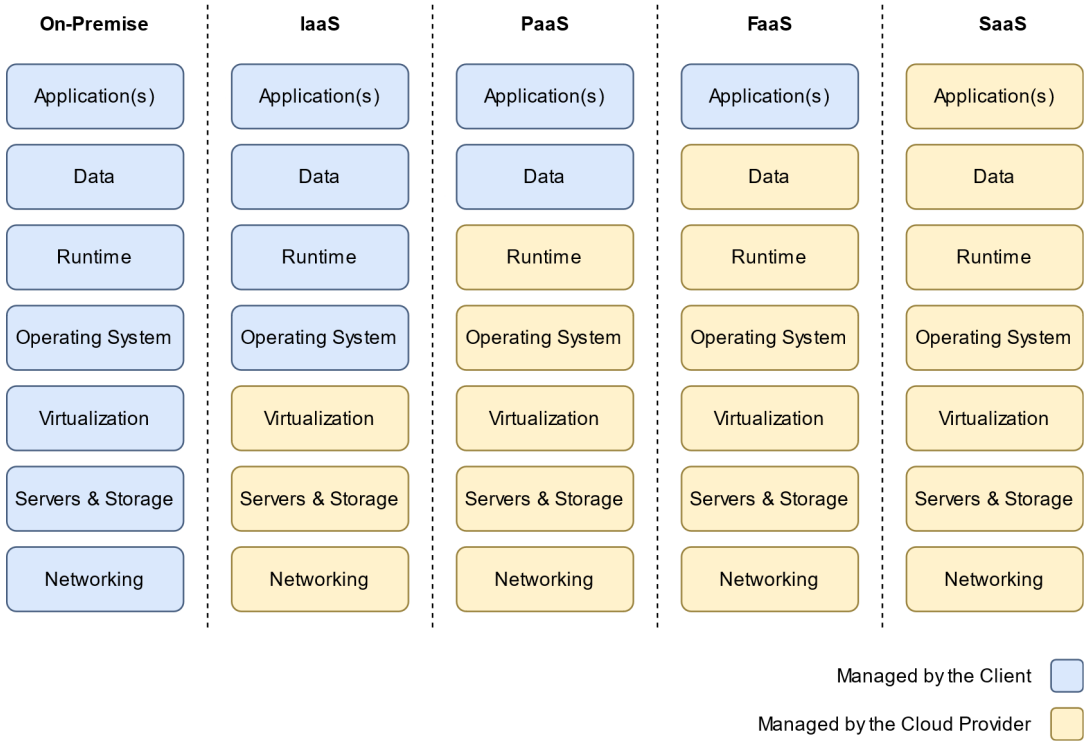

Figure 2.1: Comparing On-Premise, IaaS, PaaS, FaaS, and SaaS service models (adapted from [55]). From left to right, there is an increasing management responsibility by the cloud provider, and thus an increasing level of abstraction for the user.

It offers the ability to create a Virtual Machine (VM) to the client, who controls everything from the Operating System (OS) level to the application itself. The network tasks such as load balancing, routing and firewalls are managed by the Cloud provider, and IaaS often offers automatic scaling [23]. This model often results in cost reduction both in hardware and human resources.

However, the environment's security (from the application to the OS layer) is managed by the client. Attending to this responsibility is crucial since IaaS is prone to a set of security threats such as (1) attack in virtualisation, (2) attack based on the life-cycle of the VM, and (3) data loss and leakage (directly related to sharing data in public Clouds, which are described in Subsection 2.1.3).

Examples of IaaS solutions offered by market-leading Cloud providers are Google's *Compute Engine* [51] and Amazon's *Elastic Compute Cloud* (EC2) [8].

2.1.2.2 PaaS - Platform as a Service

In this model, besides the layers of abstraction provided by IaaS solutions, the OS and runtime execution are also managed by the Cloud provider. It supports the complete life-cycle of a software application, allowing clients to develop and test services directly on the PaaS Cloud [35].

The client still has over the application itself and its data, as well as the application's environment configuration. Security is supplied by the Cloud provider (featuring recovery and backup

mechanisms), and clients are offered the flexibility of installing tooling alongside their platform(s), according to their requirements [23].

Nonetheless, adopting a PaaS solution results in dependability towards the vendor and the specific software they offer (often known as vendor lock-in). Moreover, migrating from one PaaS Cloud provider to another is a difficult task.

Examples of PaaS solutions offered by market-leading Cloud providers are Google's *App Engine* [48] and Amazon's *Elastic Beanstalk* [9].

2.1.2.3 SaaS - Software as a Service

In this model, the capabilities offered to the client are the usage of the provider's application(s), which are running on their Cloud infrastructure [80].

A SaaS application can be run by multiple clients at the same time (one-to-many basis [23]), and this model removes any responsibility from the client's side. SaaS applications can often be customised via an API.

It is worth mentioning that, given the fact that the application's data is managed by the Cloud provider, there are challenges in terms of client authentication and identity management, as well as in data integrity, privacy, availability and accessibility [23].

Examples of SaaS solutions are *Google Docs* [52] and *Dropbox* [36].

2.1.2.4 FaaS - Function as a Service

This model fits between PaaS and SaaS in terms of which layers are managed by the Cloud provider and the client. Although it is very similar to PaaS in degree of abstraction, in FaaS the platform may not be running at all until the invocation of a function to be executed, whereas in PaaS the application is typically running in at least one server at all times [19]. Clients provide the source code of typically short functions, defining triggers and events [74] for executing these functions.

However, due to the entire infrastructure being managed by the Cloud provider, issues may arise regarding (1) state management, (2) unpredictable execution time deviations, (3) delays when executing functions (also known as cold start), and (4) high dependability towards the specific chosen Cloud provider (also known as vendor lock-in) [19, 74].

Examples of FaaS solutions are Amazon's *AWS Lambda Functions* [10], Google's *Cloud Functions* [49], and Microsoft's *Azure Functions* [15]. Listing 1 (p. 9) features an example of a Python *AWS Lambda Function*. The developer provides the function's code itself to the Cloud provider, and can configure settings such as the runtime programming language version and the name of the function handler, as well as environment variables. On execution, the function may access details about the event that triggered its action, as well as its context. The developer may then monitor the execution details, accessing output and error logs. In *AWS Lambda*, the developer has the option of managing their functions in a user-friendly way using the *AWS Console* web dashboard, or in a programmatic way using the *AWS* Command-line interface (CLI).

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

Listing 1: AWS *Lambda Function* code example. The function itself is the only source code created by the user. It receives request's details via parameters, and its *return* operation defines the returned response.

It is worth mentioning that there are other options to use FaaS. An example is *OpenFaaS* [84], a framework for building serverless functions that makes use of Docker and Kubernetes.

2.1.2.5 Comparing FaaS to IaaS and PaaS

The different Cloud Computing service models follow a *pay only for what you use* philosophy that only utilises the amount of resources necessary to satisfy the levels of demand in a scalable and cost-efficient way. However, they differ when in the level of granularity of the billed computational unit: whereas in IaaS it is typically a Virtual Private Server (VPS), in FaaS it is a single function [19].

IaaS and PaaS options offer higher levels of performance than FaaS. Malla and Christensen [75] developed a comparative study on the performance of the IaaS and FaaS models, in the context of HPC (high-performance computing, which frequently features workloads of multiple tasks that may be run independently). The study focused on the GCP provider, using the IaaS *Google Compute Engine* [51] (GCE) and FaaS *Google Cloud Function* [49] (GCF) services. Their results show that although GCF was is less expensive (14% to 40%) than GCE for similar performance levels, it shows high performance variation levels. Moreover, GCE better higher levels of performance, computing similar workloads about 1.65x faster.

In a study involving the migration of a monolithic web application firstly into microservices and secondly into a Serverless architecture, Villamizar *et al.* [104] explored the infrastructure cost efficiency of deploying the application using the AWS Cloud provider. In the first instance, the web application was deployed into a set of VPS machines, using the IaaS *AWS EC2* [8] services, and cost estimations showed reductions of up to 13.42%. In the second instance, the application was deployed using an AWS-based Serverless architecture (*cf.* Subsection 3.6.1, p. 27), which includes the FaaS *AWS Lambda Functions* [10] services. This deployment version showed an infrastructure cost reduction of about 77%, due to the granularity of the billed computation unit.

Finally, in the context of deployment of mobile applications based on microservices, Albuquerque *et al.* [2] carried out a comparative study on PaaS and FaaS services based on the AWS Cloud provider. The study was aimed at understanding the impacts of performance, scalability, and infrastructure-related costs, comparing deployment versions using PaaS *AWS Elastic Beanstalk* [9]

and FaaS *AWS Lambda Functions* [10] services. Their results showed equivalent levels of performance, although FaaS exhibited irregular results due to cold starts and unexpected execution times. Regarding cost efficiency, the resulting values were inconclusive and highly dependant on the workload types, varying according to the amount of read/write database operations. Lastly, FaaS showed more efficient scalability, presenting a granular and linear allocation of resources according to demand, unlike PaaS where the scaling process occurred in resource jumps.

2.1.2.6 Cloud providers Solutions

The market-leading Cloud providers offer Cloud solutions across all service models. Table 2.1 synthesises the IaaS, PaaS and FaaS services offered by AWS, Azure, GCP and IBM.

Table 2.1: IaaS, PaaS and FaaS solutions offered by market-leading Cloud providers. Each provider features services in all cloud service model layers.

Cloud provider	IaaS	PaaS	FaaS
AWS	Elastic Compute Cloud (EC2) [8]	Elastic Beanstalk [9]	Lambda Functions [10]
Azure	Azure IaaS [16]	App Service [14]	Azure Functions [15]
GCP	Compute Engine [51]	App Engine [48]	Cloud Functions [49]
IBM	Cloud Infrastructure [63]	Cloud Platform [64]	Cloud Functions [62]

2.1.3 Deployment Models

A Cloud deployment model is defined by the location of the deployment infrastructure and by who has control over that infrastructure. These deployment models are categorised as follows [80, 35]:

- **Private Cloud**, the Cloud infrastructure is used by a single organisation and their comprising consumers. The Cloud may exist either in or off premises, and can be managed by either the organisation itself or by a third-party entity;
- **Community Cloud**, the Cloud infrastructure is used by a community of consumer organisations that share the same infrastructure, policies and concerns. Similarly to private Clouds, it can be owned or managed either by a third-party entity or by any of the community's organisations;
- **Public Cloud**, the Cloud infrastructure is open for use by the general public. It may be owned, managed, and operated by any organisation (or group of organisations), and exists on the premises of the Cloud provider;
- **Hybrid Cloud**, the Cloud infrastructure consists of a combination of at least two of the aforementioned distinct types of Clouds. They remain unique entities but are bound together by standardised/proprietary technology that enables data and application portability.

The chosen deployment model consists of a critical decision, given that it impacts the overall infrastructure security, accessibility, as well as the business itself [90].

2.2 Source Code Annotations

Source code annotations, often referred to as attribute-oriented programming or attribute-enabled programming (AEP) in literature, is a programming technique that allows enriching program elements (*e.g.*, variables, methods, classes, and packages) with metadata [99]. These have been used to manage inheritance on Object-oriented Programming (OOP) languages, identify parallelisation opportunities, declare elements as deprecated, among others.

Due to its general-purpose flexibility, this technique became popular in various programming languages (*e.g.*, *Java* and *Python*). It consists of adding custom structured and parsable declarative tags to annotate elements without directly changing the source code's semantics.

Annotations may, however, pose difficulties when being interpreted and maintained [27]. Sulir *et al.* [99] formalised source code annotations by analysing the syntax and semantics aspects of the language, hypothesising a correlation between source code annotations and conventional formal languages. They propose the definition of the *annotation-based language* term to *"describe a given set of annotations that are processed by the same reference implementation with the same goal"*.

The following Subsections demonstrate the use of annotations under different scenarios and for different purposes.

2.2.1 Parallelisation

Dig *et al.* [34] suggested the use of source code annotations to identify sections that should run in parallel when refactoring sequential programs for performance. Although their work focuses on a multitude of refactoring approaches towards parallelism, they propose an integration with the IDE (integrated development environment) to expand annotations and see the parallel code that resulted from the programmer's annotation (*e.g.*, in a loop structure).

This concept is used by the *Intel Advisor* [65] tool to identify places in serial program (in programming languages such as *C*, *C++*, and *Fortran*) sections that should execute in parallel with the need for synchronisation, in a way very similar to the *OpenMP* [28] framework.

2.2.2 Preserving Design Patterns

A design pattern is a formalised solution to a problem that frequently occurs when designing software [45]. To ensure that their application is preserved throughout the development process, Sabo and Poruban [91] proposed a way of preserving design patterns through means of using source code annotations. With the goals of both improving traceability and ensuring the pattern's correct application in a given scenario, they annotate code constructs (*e.g.*, classes and methods) to explicitly identify and represent design patterns, providing feedback to the programmer if the pattern is not followed correctly. Listing 2 (p. 12) provides an example of a concrete implementation of an annotated *Abstract Factory* design pattern.

```
@AbstractFactory(role=Role.CONCRETE_FACTORY, id=3)
public class AfricanFauna extends ContinentalFauna {
    private static int lionCount = 0;

    @Override
    @AbstractFactory(role=Role.CONCRETE_METHOD, id=3)
    public Carnivore createLion() {
        lionCount++;
        return new Lion();
    }
}
```

Listing 2: *Java* annotated *Abstract Factory* concrete implementation (based on [91]). The *CONCRETE_FACTORY* and *CONCRETE_METHOD* annotations are used to indicate the class and method roles in the *Abstract Factory* design pattern implementation.

Meffert [79] also proposed a methodology to annotate design patterns with the intent of documenting their usage. In this work, they focus on the semantics of the pattern, proposing annotations to indicate the pattern's intent, problem, drawbacks and needs.

2.2.3 Web Frameworks and Services

Several popular web frameworks use annotations to declare networking details (*e.g.*, *Spring*, *Flask*, and *Django*). Listing 3 features an example of an *@action* annotation in the *Django* framework to make a method routable, through the use of a Python decorator. The annotations are used to specify how a web service's logic is accessible, and, at compile or execution time, these are analysed to generate the required mechanisms to expose the functionality [77].

```
class ViewSet(viewsets.ModelViewSet):
    @action(detail=True, methods=['post'])
    def set_password(self, request, pk=None):
        # ...
        return Response({
            'status': 'Password set.'
        })
```

Listing 3: *Django ViewSet @action* annotation. The *@action* decorator is used to specify which HTTP methods should be routed the function.

On the frontend side, Heinrich *et al.* [58] proposed a set of source code annotations to improve the development of collaborative web applications. The annotations introduce lightweight mechanisms to control concurrency, working on top of the *Knockout* web framework, and their study

shows a 40% effort reduction when incorporating collaboration capabilities into web applications.

2.2.4 Annotated Code Quality

The quality of the code annotations should, similarly to the source code itself, be taken into account. However, a number of anti-patterns (or *"bad smells"* [31]) often arise in annotation-based frameworks.

Correia *et al*. [31] proposed a set of strategies to detect and mitigate these anti-patterns, as well as an open-source tool to automate this process. They evaluate the annotated source code based on several metrics (*e.g*., number of annotated lines and annotation line length), labelling annotated constructs with detected *"bad smells"* (*e.g*., the *Crowded Party* anti-pattern consists of a class with a large number of annotation lines, which results in low readability and high complexity). Finally, they suggest refactoring paths for each identified problem.

2.2.5 Serverless

Although source code annotations have been used in the context of platforms to build Serverless solutions, their use to leverage Serverless usage has not yet been explored.

Crucial is a system to program concurrent Serverless applications proposed and created by Barcelona-Pons *et al*. [20]. It provides annotation-based abstractions to identify code sections according to their degree of isolation (with the aim of managing state). They proposed an architecture with three layers, with (1) a FaaS computing layer, (2) a distributed shared memory layer and (3) the application itself. The solution was validated and evaluated using *AWS Lambda Functions* [10], supporting fine-grained support for mutable state and synchronisation with less than 3% of source code differing from a conventional solution.

Hendrickson *et al*. proposed *OpenLambda* [60], an open-source platform for building Serverless web services and applications. By comparing solutions based on *AWS Lambda Functions* [10] and *AWS EBS* [7], this work proposes solutions to various Serverless research problems (*e.g*., execution engines, data aggregation, load balancing, debugging, and language support), such as web application modularisation and partitioning. They refer the opportunities of framework-aware tools to automate this process, suggesting the usage of source code annotations to associate endpoints with handler functions. However, their proposed solution does not explore these mechanisms.

2.3 Reflection

Given that we intend to explore web applications' reconfiguration at runtime, it is important to understand the concept of *Reflection* in software engineering. Bobrow *et al*. [22] define reflection as:

> *"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution."*

A system/application with this property can change its behaviour and composition by manipulating data that represents its own state [38]. Bobrow *et al*. distinguish this manipulation between *introspection*, the ability to observe and reason about its own state, and *intercession*, the ability to alter its own execution state, or its interpretation or meaning.

Cazzola [26] classifies this ability as either *structural reflection*, which is the ability of a language to reify its own execution and abstract data types, or *behavioural reflection*, which is the ability of a language to reify its own semantics and data used to execute the program.

2.4 Summary

This chapter describes Cloud Computing, FaaS, and source code annotations, representing key concepts to the understanding of this work.

Section 2.1 (p. 5) defines Cloud Computing, detailing its main characteristics, service models (providing examples of specific Cloud providers' solutions), and deployment models. Cloud's characteristics may be regarded as conceptional, technical, business-related or even user-experience-related, such as its on-demand capabilities, broad network access, and measured services [80, 47]. Cloud solutions are available in different service models (*e.g.*, IaaS, PaaS, FaaS, and SaaS) which differ in the set of computing layers that are managed by the Cloud provider [23, 80] (*cf.* Figure 2.1, p. 7). A Cloud deployment model is defined by the location of the deployment's infrastructure and by which entity has control over that infrastructure, and the Cloud may be regarded as private, community, public, or hybrid [80, 35].

Section 2.2 (p. 11) discusses a few of the uses of source code annotations to add metadata to programs, as well as their common anti-patterns. It is a programming technique that allows enriching program elements (*e.g.*, variables, methods, classes, and packages) with metadata [99]. Due to its general-purpose flexibility, this technique has been applied to domains such as parallelisation [34, 65], design patterns preservation [79, 91], and web frameworks and services [77, 58]. The application of this concept in Cloud and Serverless computing has also been explored with the aim to improve state management [20].

Finally, Section 2.3 (p. 13) introduces the concept of reflection of a system — *"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution."* — distinguishing this property between introspection (the ability to observe and reason about its own state) and intercession (the ability to alter its own execution state, or its interpretation or meaning) [22, 38].

On the following chapter we discuss the current state of the art on the domain of this dissertation.

Chapter 3

State of the Art

On the previous chapter we review the background key concepts used throughout this work. This chapter details the state of the art for refactoring partitioned applications at runtime and FaaS leveraging. Section 3.1 describes the methodology used in the research process. Then, Section 3.2 introduces the techniques used to partition systems. Section 3.3 explores approaches to inspect and refactor applications at runtime, and Section 3.4 discusses topics related to dynamic routing between FaaS and PaaS. Subsequently, Section 3.5 describes the known challenges in hybrid deployments, followed by the description of the methodologies used to leverage the usage of FaaS in Section 3.6. Finally, Section 3.7 summarises the main findings on the aforementioned topics.

3.1 Methodology

To study the current state of the art of this dissertation's domain, an iterative literature review methodology was followed. It consisted of defining a group of search queries, which were adapted (that is, by changing the search engine's specific operators, *e.g.*, NOT, AND, OR) accordingly to the used databases.

The search queries were aligned with the sections defined throughout this document, to support the investigation of the domain's background and state of the art. Although no specific inclusion or exclusion criteria was 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 by means of *snowballing* — analysing the references of each result to identify other works of interest, as well as relevant terms, topics and concepts in the domain.

3.1.1 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, as well as popular reference books. Moreover, it also included technical reports and grey literature, consisting of web articles and Cloud providers' official documentation.

3.1.2 Research Questions

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

LRQ1: How should traffic be dynamically routed between different cloud service models?
Focusing on PaaS and FaaS, it is important to understand the advantages and disadvantages of different routing architectures, in terms of robustness, latency efficiency, and others;

LRQ2: What trends and practises exist within hybrid deployments?
Deployments that utilise different service models in their architecture feature additional challenges, which makes them usually not a popular choice. We aim to understand what these challenges are and how to mitigate them;

LRQ3: How should the use of FaaS and PaaS be leveraged?
It is important to understand both the theoretical advantages and drawbacks of each of these service models, as well as analyse existing works and case studies that include practical evidence of them;

LRQ4: How do applications refactor themselves at runtime?
Understanding how applications can inspect themselves and, based on the gathered operational data, reconfigure their architecture at runtime (*e.g.*, moving parts of themselves among different cloud services) is crucial for the automation-related goals of this work;

LRQ5: What techniques have been used to partition software systems?
We want to understand the different approaches on partitioning systems, with main focus on web applications, including both automatic and manual methodologies.

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 (*e.g.*, include 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. Notwithstanding that, an exemplification of part of the process is depicted in Appendix C (p. 81).

3.2 System Partitioning

In order to migrate part of an application to FaaS, that part of the application must be expressed as a stateless function — otherwise, there must be system state stored externally to the FaaS function or functions [44]. Thus, the process of partitioning an application to a set of microservices by determining its boundaries is a necessary step towards that goal.

3.2.1 Automatic Techniques

The literature features approaches that combine model-driven, static analysis, and dynamic analysis techniques to address this task in an automatic/semi-automatic manner [18].

It is worth mentioning that this process does not necessarily aim to grant a system (or its parts) the capacity of being migrated to FaaS, but to partition a monolithic system to achieve the benefits of better scalability, shorter release cycles, and better maintainability [78].

3.2.1.1 Model-driven Techniques

This technique uses design elements as input (such as business data-flow diagrams, function/non-functional requirements and domain entities), and consists of iteratively (1) analysing the domain's

architecture, (2) extracting the candidate microservices, (3) verifying that data-related dependencies are consistent with the candidate partitioning, and (4) reorganising and refactoring the codebase [88].

It is often combined with the reusing of knowledge of migration patterns [18]. Although not systematic and dependant on human intervention, it is a popular *ad-hoc* way of achieving a microservices architecture.

3.2.1.2 Static Analysis Techniques

This technique uses source code itself as input [88]. Through its analysis, the coupling levels and dependencies (*e.g.*, imports and method invocations) among components (*e.g.*, packages, modules or classes) are evaluated, aiming to discover partitioning sections [78]. Understanding dependencies among components often depends on programming languages and frameworks, as well as on the environment itself.

The system and its components can be modelled as a graph, where nodes represent components and edges represent dependencies. The edges are weighted according to the dependency level between two node components [78, 57].

3.2.1.3 Dynamic Analysis Techniques

This technique analyses a system's behaviour and functionalities at runtime [88]. By doing so, operational data is gathered, which provides insight on how dependencies among components are exercised while executing [78]. Together with results from static analysis, this information may be used to formulate new dependency values among components.

Groups of tightly coupled components (having a high degree of dependency among each other) may be grouped into *clusters* by using a clustering algorithm. Given that *clusters* are loosely coupled among each other, they are good candidates to become isolated services [78].

3.2.2 Manual Techniques

Although the automatic techniques that were introduced in the previous subsection have proven to achieve decent results (it is worth mentioning that even certain automatic approaches may rely on manual steps [29]), several works have used *ad hoc* manual techniques to achieve microservices-like architectures, due to their focus being on the resulting system and not on the decomposition process itself.

With the aim of understanding and evaluating how a Serverless architecture would scale and speedup a *FinTech* document processing system, Goli *et al.* [46] partitioned a monolithic manually based on their business knowledge of the architecture, packaging low-coupled parts of the system as Docker containers.

In a work that discusses decomposition into microservices and its influence on a system's elasticity, Kecskemeti *et al.* [70] suggest the annotation of software with dependency information as a way to identify partitioning sections. The usage of source code annotations to associate

independent section with handler functions is also suggested by Hendrickson *et al.* [60] in the context of web application modularisation and partitioning.

3.2.3 Summary

The process of partitioning a system (typically a monolith) to a set of microservices by determining its boundaries has been explored in multiple research works [18, 57, 78, 88].

Automatic approaches typically involve using a combination of techniques:

- **Model-driven** techniques [18, 88] are generally iterative, and use design elements (*e.g.*, business data-flow diagrams, function/non-functional requirements, and domain entities), often reusing knowledge of known migration patterns;
- **Static Analysis** techniques [57, 78, 88] examine the system's source code to understand dependencies (*e.g.*, imports and method invocations) among components (*e.g.*, packages and classes);
- **Dynamic Analysis** techniques [78, 88] observe the system's behaviour at runtime, gathering operational data on how dependencies among components are exercised while executing.

Manual approaches have also been used or referred to in several research works [46, 60, 70], consisting of *ad hoc* techniques that are often non-systematic. They are typically used not by their strategic value *per se*, but as a mean towards an end when the partitioning process is not the main focus. This subject is discussed in Section 4.2 (p. 34).

3.3 Runtime Application Refactoring

For an application to change its execution behaviour (intercession) at runtime, it must be able to inspect itself (introspection) (*cf.* Section 2.3, p. 13).

3.3.1 Monitoring and Self-Inspection

Monitoring is the process of a system inspecting itself at runtime to gather operational data that results from its activity. Monitoring strategies and tools should not directly affect the system being monitored, and thus should not be intrusive or introduce overheads resulting from their action [56].

3.3.1.1 Techniques

Several runtime monitoring and self-inspection approaches have been proposed and developed, differing in purpose, methodology and application domain. Rabiser *et al.* [89] presented a comparison among 32 different approaches, evaluating them based on the application context (*e.g.*, domain and architectural style), usage (*e.g.*, needed skills and user guidance level), content (*e.g.*, usage language and data/event manipulation) and validation process. Their results revealed key

differences in aspects such as temporal behaviour (supporting occurrence, logging and ordering of runtime events).

The system dynamic analysis technique that was introduced in Subsection 3.2.1.3 as a step for partitioning monolithic systems may also be applied to runtime self-inspection. It allows gathering operational data that indicates how the system behaves at runtime [78]. This technique has been applied in several domains, such as behaviour monitoring for security assessment of web applications [61] (to assist security software testing techniques such as fault injection and SQL injection), and system scalability monitoring (to analyse web application API requests processing details, such as memory usage and response time).

3.3.1.2 Web Applications

These techniques have also been applied to leverage Cloud resources. Vasar *et al*. [102] proposed a framework for monitoring and testing scalability and QoS of Cloud-based web applications. This framework focuses on both self-inspection and Cloud resources management and allocation policies (by adapting the number of running *AWS EC2* [8] virtual server instances). It analyses metrics such as average response time to requests, average memory and CPU usage, uptime, infrastructure costs and percentage of successfully handled requests. These metrics are measured using the sysstat [100] utility and monitoring package, making the analysed information available via a Java web service TCP endpoint. The introduced overhead for this approach is minimal, with statistics loads averaging at 330 bytes. The *MediaWiki* web application was used to validate this solution, using several workload patterns to simulate different scenarios. Although the dynamic analysis techniques allowed to identify application bottlenecks and insights on future traffic arrival rates, this technique lacks automation and requires significant manual effort, especially if multiple machines in the target Cloud are being analysed.

Web application frameworks often feature profiling tools that can be used to gather runtime operational data. *Silk* [69] is a runtime profiling and self-inspection tool for the Django [42] framework. It intercepts and stores data from API requests and database queries, providing insight such as request rates, request processing time, and the number of database accesses. The tool allows accessing the gathered operational data in both a web user interface and programmatically. This tool has been used for dynamic analysis purposes, by Matias *et al*. [78], in a case study aiming to determine microservice boundaries.

3.3.2 Dynamic Refactoring

Fowler [43] defines refactoring as follows:

> *"Refactoring is a controlled technique for improving the design of an existing codebase. Its essence is applying a series of small behaviour-preserving transformations, each of which "too small to be worth doing". However, the cumulative effect of each of these transformations is quite significant."*

This concept has been redefined over the years by different authors, being first introduced by Opdyke [83] in 1992. The goals of the internal structure improvements may reside in improving the system's extensibility, reusability, and efficiency [73]. Refactoring may occur at various levels, such as a specific function, a class, or the system's architecture itself. Additional challenges arise when a system is refactored at runtime while maintaining its availability (that is, without halting the system's execution).

In a work exploring strategies of refactoring applications' architectures for the Cloud, Zimmermann [107] emphasises the difference between *"architectural refactoring of the system"* and *"code refactoring of programs"*, referring that the former (which is the one that is related to this work) is not as commonly practised as the latter.

Although it is possible for a system to adapt itself using programming language features (*e.g.*, conditional expressions, environment-based parameterisations, and exceptions), this approach introduces design and maintainability issues, low flexibility, and does not scale. Thus, this responsibility should be delegated to an application-independent middleware that focuses solely on refactoring and adapting the underlying application [41].

A set of approaches that rely on describing a system's architectures as models have been proposed for this purpose. The techniques feature a middleware layer to apply refactoring operations on the system's components at runtime.

Lan *et al.* [73] introduced an approach to refactor several parts of a system at runtime. Their solution relies on a MOF-based (meta-object facility, which is a meta-model architecture) definition of bad/good patterns in the application using a meta-model. Then, a middleware layer automatically detects, removes, and updates components based on the predefined patterns. They distinguish different types of meta-models, such as hosts, services, components, methods, and their invocations.

On a different approach, Irment *et al.* [66] proposed a methodology for runtime adaptation of components in SOAs (service-oriented architectures).

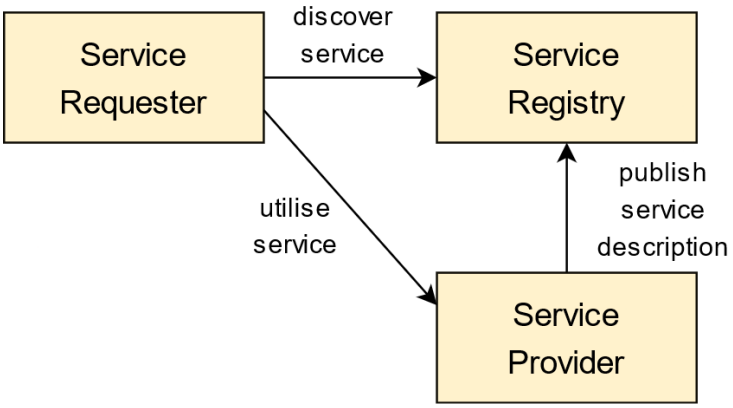

Figure 3.1: SOA service publishing and discovery (adapted from [66]). Providers publish service descriptions on a Registry. Then, requesters search the registry, choosing a service provider based on its description.

These architectures feature a service provider that publishes a service description (service interface, functional and non-function properties) to a registry. Then, requesters search the registry, choosing a service provider based on its description — this process is visible in Figure 3.1 (p. 21).

To ensure that availability is not impacted when a service is updated, all instances of the outdated service remain active until all requests prior to the update are fulfilled, as shown in the UML sequence diagram in Figure 3.2.

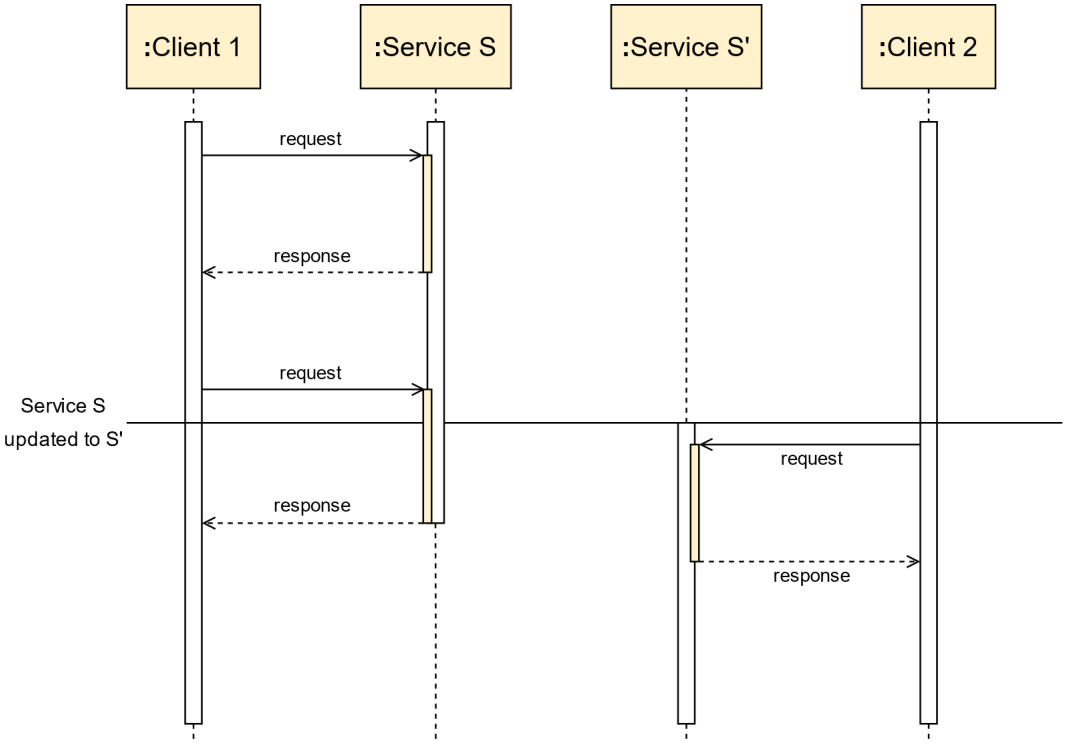

Figure 3.2: SOA service updating UML sequence diagram (adapted from [66]). Instances of the outdated service remain active until all requests prior to the update are fulfilled, to ensure availability is not negatively impacted.

3.3.3 Summary

Monitoring is the process of a system inspecting itself at runtime to gather operational data that results from its activity. These strategies should not be intrusive to the system itself [56]. Monitoring and self-inspection techniques differ in aspects such as application context, usage language, data/event manipulation, and temporal behaviour.

Dynamic analysis techniques, which may also be used for system partitioning purposes (*cf.* Subsection 3.2.1.3, p. 18), can be used to gather operational data that indicates how the system behaves at runtime [78]. In the context of web applications, such operational data consists of requests' response time, memory usage, database accesses, and API request rates [102]. This can be done through the usage of runtime profiling and self-inspection tools such as *Silk* [69] (a tool for the Django web framework).

This information can then be used to refactor a system's architecture (improving the system's structure without impacting its external behaviour) [43, 83]. Although the change in behaviour can result from primitive techniques (*e.g.*, conditional expressions in the source code), these are often difficult to maintain, inflexible and do not scale [41].

The runtime refactoring/adapting process can be delegated to an application-independent middleware layer. This layer relies on a representation of the system it is going to adapt [41] (*e.g.*, meta-object and meta-model representations [73]). These techniques should not impact the system's availability. Refactoring techniques have also been applied to service-oriented architectures, updating components at runtime without impacting the system's availability [66].

3.4 Dynamic Routing Between FaaS and PaaS

The subject of using resources and services from multiple clouds has been explored in the literature for over a decade. It is commonly named as Multi-Cloud (or Multiple-Clouds), although a number of publications refer to this topic as Cloud Federation, Inter-Cloud, or Cloud-of-Clouds [86]. Even though the primary concerns around this subject are usually about the challenges of integrating and managing multiple cloud providers in the same context, the used resources are often distributed among different service levels (*i.e.*, IaaS, PaaS, FaaS, ...). In a work that explores multi-cloud architectures migration, Jamshidi *et al.* [68] present a pattern-based approach profile and migrate cloud resources. Although their proposed solution can result in *"parts of the application ... deployed on PaaS, IaaS or both"* (with some detail being provided on the orchestration of these resources), they do not discuss traffic routing or communication details among the cloud resources.

In Multi-Cloud, the same abstractions within each cloud are used to deploy various parts of an app, typically to achieve redundancy. However, in this work we explore the possibility of given a specific microservice, making it possible for it use the best abstractions (possibility within multiple clouds) to be optimised — that is, in a way, facilitating the separation of a microservice in two, aiming to obtain the resulting advantages of this separation (in both its FaaS and PaaS parts) in a transparent manner.

In the IoT (Internet of Things) domain, the orchestration of local resources with cloud computational resources has also been explored. In a work examining the Dynamic allocation of serverless functions (FaaS) in IoT environments, Pinto *et al.* [87] propose the creation of a fog layer that leverages resources between the local IoT devices network (the edge layer) and the cloud functions (the cloud layer).

By introducing a proxy between the function execution requester and the aforementioned layers (*cf.* Figure 3.3, p. 24), they are able to forward execution to the cloud or to the local network, based on the evaluation of the available runtime environments (estimating time taken to execute the function in these environments). This type of architecture also represents a relevant approach to be used for dynamic routing in the cloud.

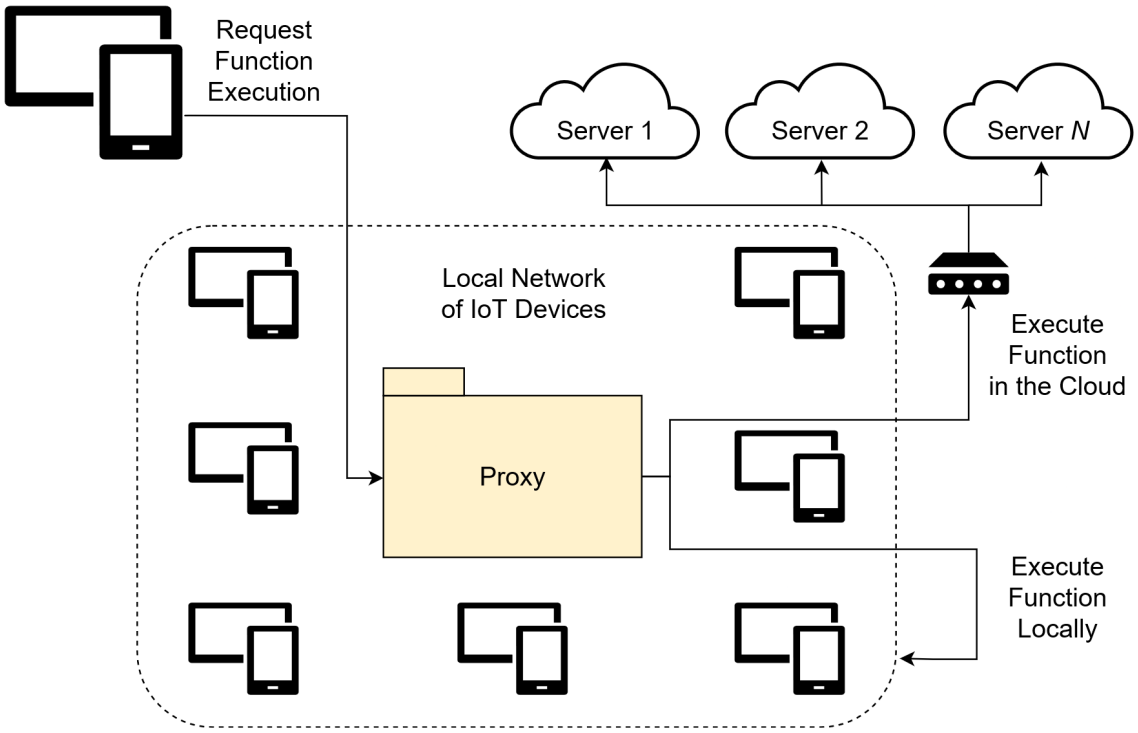

Figure 3.3: IoT and Cloud Proxying Architecture (Adapted from [87]). The proxy component forwards requested function's execution to either the local network of IoT devices or to the Cloud.

Regarding the usage of multiple service models within the same context, Jain *et al.* [67] proposed a system called *SplitServe*, a system that manages the execution of *Apache Spark* workloads between IaaS virtual machines and FaaS cloud functions, using AWS resources. Although this work does not detail how the workloads traffic is routed within the system, their solution features a stateful layer which manages the load in all the available virtual machines (IaaS layer, *EC2* elastic compute cloud instances), creating cloud functions (FaaS layer, *Lambda* function instances) and moving any new or on-going workloads to them when desirable (either in terms of cost-management of computational performance).

Although certain studies (such as the aforementioned one) propose techniques that automatically manage infrastructure at different service models layers and move workloads among them, there have also been manual-driven attempts to tackle this task. For example, in a comparative study on FaaS and PaaS, Albuquerque *et al.* [3] manually partitioned and deployed a web application using AWS cloud resources. Their system features an AWS *API Gateway* that, based on setup configurations, routes traffic to microservices that are either deployed in AWS *Elastic Beanstalk* (PaaS) or running in AWS *Lambda Functions* (FaaS). However, this study focuses mostly on evaluating metrics such as performance, error rates and cost-efficiency, and does not explore the impact of different routing techniques on the resulting latency.

Thus, since different studies aim to investigate the usage of multiple cloud service models to optimise distinct metrics (*e.g.*, performance and cost-efficiency), it is of interest to explore tech-

niques to dynamically route traffic among cloud resources deployed at different service models, namely PaaS and FaaS.

3.5 Hybrid Deployments

Deploying projects with multiple levels of service models (such as hybrid deployments featuring PaaS and FaaS) is not a trivial task. It poses issues and challenges in terms of observability, monitoring, traffic routing, reconfiguration, infrastructure distribution decision, *et cetera*. Although we believe that numerous advantages arise from these hybrid deployments (*e.g.*, lower client latency), the aforementioned cumbersome factors may be the origin of these deployments not being a popular choice.

In terms of observability (having knowledge of the system's internal state at any point in time [85]) and monitoring (that actions involved in observing a system), deploying a service at multiple levels (possibly in different providers) makes it more difficult to perform typically trivial tasks such as accessing integrated execution logs, errors and performance information, due to this data being more scattered among different cloud services.

It is also difficult to assess and decide which parts of a system should be deployed at each layer, since many variables (*e.g.*, user load, performance, cost management, latency) must be taken into account (*cf.* Section 3.6). Moreover, the optimal deployment architecture often changes over time (as the aforementioned variables fluctuate), which results in the need of system reconfiguration. This is also a challenging task since parts of the system may not be ready to be reconfigured (*cf.* Section 3.3, p. 19), and may need human intervention and development effort. Additionally, routing traffic among these cloud service layers may result in communication delays, which may ultimately result in bottlenecks that affect the whole system.

3.6 Leveraging FaaS

FaaS is commonly interchangeably referred to as Serverless in the literature. This Cloud Computing execution model became popular throughout the past decade due to its unit of computation granularity.

The motivation for using FaaS lies in the user convenience when compared to traditional Cloud solutions: It abstracts the complex management of infrastructure from the user, with automatic scaling elasticity and its granular function unit of computing [44]. However, FaaS is not in itself a *"silver bullet"*, resulting in both advantages and drawbacks (*cf.* Subsection 2.1.2.4, p. 8). Thus it is imperative to leverage its usage.

When it comes to pricing, costs are computed based on each function's execution time and memory usage. The fact that only active compute time is charged is highly beneficial for applications that rely on massive parallelism [92]. Barga [21] identifies other domains that benefit from this computing model, such as IT (Information Technology) automation, Chatbots, IoT (Internet of Things), Big Data, and web applications (which is the focus of this work).

Aiming to study the economic, business and architectural impacts of migrating web applications into FaaS, Adzic and Chatley [1] studied the migration of two applications using *AWS Lambda Functions* [10]. By migrating (1) parts of an *Heroku* platform (PaaS), and (2) a monolith-like system that made use of *AWS EC2* (IaaS), the first company achieved a cost reduction of 66% in the following year, while the second achieved a monthly cost reduction of 95%, and increasing the number of production releases in more than 10x with an engineering team of the same size, due to the business agility provided by the service-oriented architecture.

When it comes to scalability, it is completely elastic, automatic and managed by FaaS provider — the number of function instances executing for a given task is scaled horizontally based on demand. Goli *et al.* [46] migrated a *FinTech* document processing system from a monolithic architecture to Serverless. The system was partitioned manually, migrated to a Cloud-like architecture, and deployed using *GCP Cloud Run* [50]. By comparing the processing of 400,000 documents using both architectures, they achieved a speedup of 93x due to the service's horizontal scaling in the new architecture. Even though their estimation showed that costs would double with the migration, the assumption that Serverless functions would continuously be running makes the monetary estimation inaccurate.

FaaS solutions also benefit from their geographical distribution, which results in improvements in client latency [103], since providers feature a wide distribution of computing machines in multiple cities across different regions. In a work proposing an architecture for deploying FaaS platforms in hybrid clouds (featuring both public and private clouds) composed by multiple providers, Vasconcelos *et al.* [103] focus on infrastructure global distribution to allow the execution of requests as geographically close to the client as possible (aiming to reduce latency). Their proposed architecture features a *FaaS Cluster*, which contains all the computing nodes, and a *FaaS Proxy*, which offers the same interface as a regular FaaS gateway (*cf.* Subsection 3.6.1, p. 27). However, instead of processing requests directly, this component redirects traffic to different services according to factors such as the location of the client's request and available servers.

Nonetheless, due to FaaS being a somewhat recent Cloud Computing model, it still features a set of business and technical issues that make it an unattractive solution for particular use-cases.

Cloud providers that offer FaaS solutions only provide Service-level agreements (SLAs) for availability. This lack of assurance may result in these solutions not being considered adequate for data and performance-sensitive applications in specific domains (*e.g.*, Government, Banking, Military, and Healthcare) [44]. Due to this lack of SLAs, even FaaS solutions from the market-leading Cloud providers often result in unexpected startup latency and irregular function execution times [92].

The function startup latency, often referred to as cold starts, is an inherent problem of virtualisation techniques [76]. Since functions are executed in containers, it experiences a latency period when the container is initialised prior to the execution. However, there are strategies to mitigate this issue from both the provider and client sides:

- From the provider side, to enhance performance, containers are not shut down immediately. This allows for subsequent executions of a function to use the warm execution environment

of an existing container [76];

- From the client side, the strategy of periodically "pinging" the function (even if there is no task that needs to be executed) pre-warms the function instances, preventing them from shutting down and, thus, reducing cold starts [74]. However, this technique results in additional costs and contradicts the *"scale to zero"* FaaS principle (no resources are allocated or billed when there is no demand [19]).

Given that networking, virtualisation and runtime are managed by the Cloud provider, FaaS architectures are often highly dependent on the specific provider's ecosystem without having an expeditious way to migrate to a different one (a problem known as vendor lock-in). Although the adoption of standards (*e.g.*, messaging and configuration standards) to mitigate this issue and support the easy movement of business logic among different providers has been debated [44], there is no evidence that such standards have been developed and adopted.

Due to the stateless nature of FaaS, issues with state management also arise. Hellerstein *et al.* [59] discussed Serverless issues, pinpointing common failures in Serverless architectures based on several case studies. They describe issues related to both orchestration (*e.g.*, state management) and function execution itself (*e.g.*, lifetime limitations, and I/O and storage bottlenecks), concluding that although current FaaS solutions are suitable to handle workloads of independent tasks, the challenges related to function orchestration and composition result in these architectures being inadequate for numerous use cases.

Although most Cloud providers offer FaaS solutions to their clients, as referenced in Subsection 2.1.2.4, the aforementioned lack of standards and SLAs results in substantial differences in the existing solutions. Kritikos and Skrzypek [71] reviewed a set of popular Serverless provisioning and abstraction frameworks (*e.g.*, *Fission*, *Kubeless*, *Spart*, and *Fn*). In this work, they focus on frameworks that rely on *Kubernetes*, *Docker Swarm*, or specific Cloud provider solutions (*e.g.*, *AWS Lambda Functions* [10] and *Azure Functions* [15]), evaluating their design, development, deployment, testing, execution, monitoring, and security aspects, concluding that there is no solution that delivers optimal results in all the aforementioned characteristics.

3.6.1 Serverless Architectures

The FaaS/Serverless Cloud Computing execution model features significant architectural changes when compared to traditional monolithic systems. Serverless architectures alleviate developers' need to explicitly manage servers, appointing that responsibility to Cloud providers [32, 72].

At its core, a FaaS/Serverless platform consists of an event processing system. It manages a group of functions, receiving requests and events (typically sent over HTTP) sent by a User Interface (UI) or another service. Events are dispatched to different service/function instances via an *API Gateway*, that acts as a reverse proxy to the Serverless functions. After the event is received and processed, a set of execution logs are produced, and the response is made available to the requesting entity [19].

However, since most of the architecture's components are managed by the Cloud provider, these architectures often lead to vendor lock-in issues (technical and business dependability towards the provider). Figure 3.4 features an example of a fully AWS-based Serverless architecture for a web application [13].

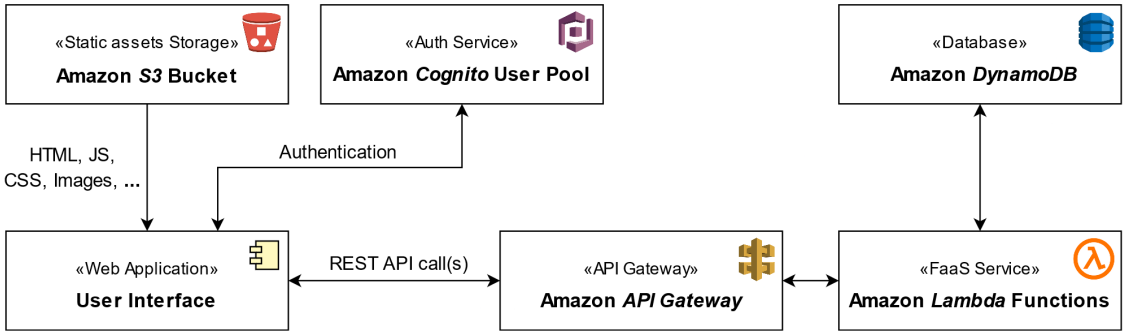

Figure 3.4: AWS-based Serverless architecture for a web application (based on [72, 13]). *S3 Bucket* manages static assets storage, *Cognito* manages user authentication, *API Gateway* manages routing and load balancing, *Lambda* manages the cloud functions execution, and *DynamoDB* is the storage database.

3.6.2 Programmatic FaaS Function Management

FaaS solutions offered by Cloud providers typically feature user interfaces (*e.g.*, web page dashboards) that allow managing function configurations and observe usage statistics. However, to deploy and remove a system's function without the need for human intervention at runtime, programmatic interfaces are needed.

These interfaces are commonly available as language-specific libraries or packages, and as Command-line Interfaces (CLIs). A few examples are the *AWS CLI* [11], used to manage *AWS Lambda functions*, and the *GCP gcloud* [53], used to manage *GCP Cloud functions*.

3.6.3 Programming Languages and Technologies supported in FaaS

The number of technologies and programming languages to develop systems such as web applications is quite vast. Thus, even the market-leading Cloud providers (such as AWS, GCP, Azure and IBM) do not offer FaaS support for every available language, as visible in Table 3.1 (p. 29).

Among these providers, IBM offers support to the largest number of languages (eight). Moreover, they claim to support any given language if provided the Docker Container for the function (through means of virtualisation) [62].

Even though popular languages such as Python, Java and NodeJS are supported by all these Cloud providers, they may highly differ in both performance and costs. Bortolini and Obelheiro [24] investigated performance and cost variations in the AWS, GCP and IBM FaaS platforms. They evaluated these metrics based on (1) the allocated memory for the cloud function, (2) the programming language used in the cloud function, and (3) the FaaS Cloud provider, comparing over

Table 3.1: Programming Languages and Technologies available for FaaS solutions offered by market-leading Cloud providers. IBM claims to support the most languages and technologies, while GCP supports the least.

Language/Technology	**AWS Lambda Functions** [12]	**GCP Cloud Functions** [54]	**Azure Functions** [17]	**IBM Cloud Functions** [62]
Python	✓	✓	✓	✓
Java	✓	✓	✓	✓
Javascript/NodeJS	✓	✓	✓	✓
Ruby	✓	✓	-	✓
C#	✓	-	✓	-
F#	-	-	✓	-
Go	✓	✓	-	✓
.NET	-	✓	-	✓
TypeScript	-	-	✓	-
PowerShell	✓	-	✓	-
Swift	-	-	-	✓
PHP	-	-	-	✓

45 testing configurations. The provider comparison showed differences of about 67x in costs and 8.5x in performance, and the language comparison showed differences of 67.2x in costs and 16.8x in performance. The observed results reflect the function inconsistency that results from the lack of SLAs discussed in the beginning of the present section.

3.6.4 Summary

FaaS is not in itself a *"silver bullet"*. On the one hand, there are a set of advantages and key features that popularised the usage of FaaS [1, 21, 44, 74, 92]:

- **On-demand execution of resources**, which provides a "pay-as-you-go" pricing model to the client instead of paying for a continuously running platform;
- **Horizontal scaling** of resources according to demand, given that every function is an isolated computational unit;
- **Deployment simplicity**, since the client needs only to supply the function's source code;
- Providers may **serve a large number of clients** without the need for large infrastructure resources;
- **Supports iterative development**, allowing to get applications up and running in an agile manner (which results in a product's faster time to market).

On the other hand, the degree of abstraction of FaaS comes with a set of restrictions, which may negatively impact some architectures [19, 44, 59, 74, 76, 92]:

- Most FaaS solutions **do not allow retaining state** between consecutive invocations;
- There are often **unpredictable execution time deviations**, since the infrastructure is managed by the Cloud provider;
- There may be a **delay when executing a function**, given that the platform may not be running (also known as cold starts);

- **Vendor lock-in**, there is high dependability towards the specific chosen Cloud provider.

Several strategies may be used to mitigate the aforementioned problems. Cold starts may be alleviated by periodically "pinging" the function (ignoring the produced workload) to reduce latency in other requests [19]. However, this does results in an increase in costs and is far from ideal [74]. Regarding state management, functions should be stateless, and the application's state should be externalised and persisted in data stores [59].

When compared to its IaaS and PaaS peer models, FaaS shows better and more granular scalability, which results in more efficient resources allocation that often reduces costs. However, its peers show higher levels of performance, being able to compute similar workloads faster.

Using FaaS has a meaningful impact in a system's architecture, alleviating the need for developers to manage infrastructure. However, since most of the architecture's components are managed by the Cloud provider, these solutions may lead to vendor lock-in issues [19, 74] that, together with the lack of SLAs [44], may make FaaS an unfeasible approach. Moreover, different providers show differences of about 67x regarding costs and about 8.5x regarding performance, with similar differences regarding distinct programming languages.

Finally, to leverage FaaS, it is essential to move and remove parts of an application from this model. To do so, Cloud providers offer a set of options for developers (*e.g.*, CLI tools and libraries) to perform this task in a programmatic way.

3.7 Summary

Section 3.1 (p. 15) introduces the methodology used to perform the literature review on the topic of this work. First, the reasoning behind the developed search queries and the criteria to select relevant literature is described. Then, the databases used are enumerated, as well as the types of reviewed literature documents. The iterative process used to answer the research questions is then outlined and discussed.

Section 3.2 (p. 17), which focuses on **LRQ5**, elaborates upon techniques to partition systems, exploring both automatic and manual techniques, as well as providing practical examples where these approaches have been used in previous literature works. Automatic approaches typically involve combining a set of techniques — in this section we explore (1) model-driven techniques, which make use of system design elements and known migration patterns, (2) static analysis techniques, which examine source code to understand dependencies among components, and (3) dynamic analysis techniques, which observe the system to understand how dependencies are exercised at runtime. Manual approaches are typically *ad hoc* and non-systematic and tend to be used as a mean towards an end when the partitioning process is not the main focus.

Section 3.3 (p. 19), which focuses on **LRQ4**, describes methodologies for system monitoring and self-inspection, exploring different techniques and their usage in the context of web applications, as well as approaches for dynamic refactoring. System monitoring aims to gather operational data of a system's execution at runtime, without negatively affecting its performance and

behaviour. Dynamic analysis techniques allow gathering operational data of web applications such as requests' response time, memory usage, data accesses, and API request rates. This information can then be used to refactor a system's architecture, without impacting its external behaviour. Although primitive techniques can be used for this purpose (*e.g.*, conditional expressions in the source code), they are difficult to maintain, inflexible and do not scale. Thus, techniques based on application-independent middleware layers that adapt the system at runtime without impacting its availability have been explored.

Section 3.4 (p. 23), which focuses on **LRQ1**, discusses topics related to dynamic routing between FaaS and PaaS. Multi-Clouds explore the usage of resources distributed among difference service levels (*i.e.*, IaaS, PaaS, FaaS). Although typically the same abstractions within each cloud are used to usually achieve redundancy, it is possible to apply this concept to the deployment of a specific microservice, making it possible for it to use the best available abstractions (possibly within multiple clouds) to be optimised. Routing a function's execution among different abstraction levels has also been explored in the IoT domain, in which functions can either be executed locally in a network of IoT devices or be routed for cloud execution. There have also been case studies of projects deployed using different cloud service models, both manually and with some degree of automation.

Section 3.5 (p. 25), which focuses on **LRQ2**, describes the main challenges in hybrid deployment scenarios (that is, featuring multiple service models, such as PaaS and FaaS), which results in these not being a very popular approach. These deployments pose challenges in terms of observability and monitoring — which makes it more difficult to perform typically trivial tasks such as accessing integrated execution logs, errors and performance information — as well as in terms of system reconfiguration and infrastructure distribution decision.

Finally, Section 3.6 (p. 25), which focuses on **LRQ3**, discusses the importance of leveraging the usage of FaaS. The different advantages and drawbacks of traditional Cloud-based web application and FaaS approaches are debated, supported by results of practical case studies and comparative research. FaaS was popularised by its inherent horizontal scaling capabilities, deployment simplicity, and on-demand execution of resources. However, it is not a *"silver bullet"* and has issues related to state management, cold-starts, and unpredictable execution time deviations. A comparison between the IaaS/PaaS and FaaS models showed that the former can achieve higher levels of performance, while the latter features more granular scalability of resources that often results in cost reduction. The dependability towards the Cloud provider (vendor lock-in) together with the lack of SLAs in this model make it an unattractive approach for certain scenarios, with providers showing differences of about 67x regarding costs and about 8.5x regarding performance, and similar differences regarding distinct programming languages. FaaS has a meaningful impact on a system's architecture and provides a number of programmatic ways for developers to deploy functions.

Thus, based on the gaps identified in this literature review, on the following chapter we thoroughly describe and formalise the problem under study.

Chapter 4

Problem Statement

On the previous chapter we discuss the current state of the art on 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. Secondly, Section 4.2 identifies the scope and focus of this work. Then, Section 4.3 defines the set of *desiderata* we intend to address in this work. Section 4.4 presents the main hypothesis we aim at validating, followed by the main research questions that guide this work in Section 4.5. The methodology that is used to validate and evaluate the obtained results is outlined in Section 4.6. Finally, Section 4.7 provides an overview of the topics addressed in this chapter.

4.1 Current Issues and Open Problems

In Chapter 3 (p. 15) we present the current state of the art on a variety of topics related to leveraging FaaS, hybrid deployments features PaaS and FaaS, as well as techniques to partition and dynamically refactor applications. However, there are still a number of open challenges and issues on these topics, given that most research works operate on very specific assumptions, scenarios, and restrictions.

Firstly, deployments featuring multiple cloud service models (hybrid deployments) showed not to be a popular deployment decision. Although each service model features its own strengths and, ideally, the best of each of them can be used to optimise the deployment context of a given

application, they pose challenges in terms of observability, monitoring, traffic routing, reconfiguration, distribution decision, among others. Additionally, existing hybrid deployment scenarios either require mostly manual configuration, which leads to elevated development effort, or can only operate with particular services of specific cloud providers. Thus, there is a lack of ways to automate hybrid deployments in a way that is agnostic to the cloud provider.

Additionally, to reach a state where a system can be deployed to the cloud, engineers must redesign its architecture. This process is hard to assess since (1) it is hard to estimate the required effort to redesign the application, (2) it is hard to ensure that this step will be successful, (3) it requires substantial manual effort, and (4) it is troublesome to revert, due to the architectural changes applied to the application. There is a lack of programmatic methodologies to declare these separation points at the source code level, which would mitigate most of the aforementioned points.

Within the previously described hybrid deployments, there is also a lack of research on specific dynamic techniques for routing traffic between service model layers, namely PaaS and FaaS. The optimisation of these techniques towards goals such as minimising client latency when accessing the cloud resources, or saving infrastructure-related costs, could be a step towards the popularisation of cloud hybrid deployments.

Finally, we believe there are several gaps regarding the ability of cloud-deployed applications to maximise their operational efficiency through runtime inspection and self-reconfiguration. Data gathered on self-inspection at runtime should be used to decide if parts of the system should be migrated to different cloud layers (*e.g.*, from PaaS to FaaS, and vice-versa) with the aim of optimising metrics such as latency or infrastructure-related costs.

4.2 Scope

Based on the literature shortcomings and current issues identified in the previous subsection, this dissertation aims to investigate strategies to facilitate the programmatic partitioning of web applications into PaaS and FaaS and their deployment to the cloud, thus improving the software development process for engineers. Additionally, this work explores techniques for dynamic routing between PaaS and FaaS in hybrid deployment scenarios.

Thus, the scope of this work includes the implementation of a plugin prototype that utilises and facilitates the aforementioned tasks.

Even though that self-inspection and runtime infrastructure reconfiguration is considered worthy of being investigated, it is not be the main focus of this work, and any advances towards that goal are secondary.

4.3 Desiderata

As discussed in the previous section, this work includes the implementation of a plugin that facilitates the partition and cloud deployment of a web application. With its implementation, we

propose to achieve the following *desiderata* (Ds):

D1: **Provide an interface to annotate where resources are deployed and executed,** so that developers can programmatically declare in which cloud service model (PaaS or FaaS) a web application's resource should run;

D2: **Deploy the annotated web application resources to the cloud,** so that the annotated resources from **D1** can quickly be deployed without effort, in a way that is agnostic to the used cloud providers;

D3: **Manage the deployed web application cloud resources,** so that it is possible to eliminate, re-deploy, and change the location of deployed resources;

D4: **Efficiently route traffic between PaaS and FaaS,** in a way that ensures that user requests are routed to the correct resource location (PaaS or FaaS) as fast as possible (with minimal latency);

D5: **Automatic reconfiguration of annotated resources at runtime,** that is, moving annotated resources from PaaS to FaaS, and vice-versa, at runtime (based on efficiency metrics such as user load, memory usage, ...) to minimise user latency and costs.

The main priority consists of fulfilling *desiderata* **D1**, **D2**, **D3**, and **D4**. Progress towards **D5** is secondary.

4.4 Main Hypothesis

We can summarise our *desiderata* in the following hypothesis:

> *PaaS and FaaS provide complementary benefits for web applications. Developers can leverage them to optimise their applications, by (1) partitioning their applications to be deployed to the best service model for each functionality, (2) using a dynamic routing strategy to forward requests to the proper resource, and (3) allow the application to dynamically reconfigure itself by moving code between PaaS and FaaS (and vice-versa) in a way that optimises infrastructure costs, client latency, and response time.*

Providing web developers with interfaces to *partition their application* can significantly reduce development effort. We believe that programmatically declaring in which service model layer a functionality should be deployed and executed, in a way that is agnostic to the cloud provider, avoids manual infrastructure management and its cumbersome configuration, allowing developers to focus on choosing *the best service model for each functionality*.

For this to be possible, it is imperative to use a *dynamic routing strategy* that guarantees that requests are routed to the correct resource. We believe this process should be crafted in a way that

optimises its robustness and routing efficiency when forwarding traffic between Paas and FaaS, by minimising latency when *forwarding requests to the proper resources*.

Finally, it should be possible to optimise the execution context of a deployed application, *in a way that optimises infrastructure costs, client latency, response time*, and others, by allowing it to inspect itself at runtime. The operational data gathered on self-inspection should allow it *to dynamically reconfigure itself by moving code between PaaS and FaaS (and vice-versa)*, thus making use of the best each service model has to offer based on the application's runtime necessities.

It is worth pointing out that the hypothesised premises 1 and 2 are a requirement and step towards premise 3. Thus, this dissertation primarily focuses on these first two premises.

4.5 Research Questions

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

RQ1: Can we use source code annotations to identify parts of the application that should be executed/deployed in PaaS and FaaS?
Refactoring a system and redesigning its architecture for its deployment to the cloud is a demanding endeavour. Identifying partitioning sections using annotations at the source code level would provide an expeditious way to tackle this task, in a way that significantly decreases development effort;

RQ2: Can we deploy a programmatically partitioned web application to the cloud in a way that is entirely provider-agnostic to the developer?
A cumbersome part of deploying a web application to the cloud, especially if there are multiple providers or different types of services being used, is coping with provider-specific configurations and details. Can we abstract this effort from developers?

RQ3: How can we optimise the routing of traffic between PaaS and FaaS? What techniques can be used for this purpose?
Routing traffic between PaaS and FaaS is not a trivial task and can introduce communication delays and bottlenecks. Thus, it is important to conceive and experiment with different routing techniques in order to minimise client latency and routing robustness;

RQ4: How can a system self-inspect, refactor and migrate parts of itself at runtime?
Gathering operational data and adapting an application's codebase and structure at runtime poses a number of challenges. Thus, it is important to identify ways of addressing these tasks, investigating the best approaches and strategies to execute them;

RQ5: What techniques have been used to leverage FaaS and with what results?
The different cloud models have distinct advantages and drawbacks. Moving parts of a web application to/from FaaS should optimise a set of metrics. What policies have been used for this purpose? How have they impacted these metrics?

4.6 Validation and Evaluation

In a work on *"Experimental Models for Validating Technology"* [106], Zelkowitz and Wallace defined the following four 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.

The validation and evaluation process of this dissertation are separated into two phases, which use both the **Engineering** and **Empirical** methods of validation.

In the first phase, a set of techniques for dynamic routing between PaaS and FaaS are implemented and evaluated, using load testing techniques to simulate typical real-world use case scenarios. To do so, a cluster of machines is set up to create simulated users that execute web requests to the routers implemented in each technique. These machines should be geographically distributed around the globe to simulate usage from multiple regions. Each technique is evaluated in terms of their latency performance (how long they take to fulfil users' requests) and their robustness (that is, their ability to fulfil requests without failing). This phase follows an **Empirical Method** approach of validation — data on the latency performance and robustness of the proposed routing techniques is gathered to validate the hypothesis.

Then, in the second phase, the routing technique that shows the best results is applied in the implementation of the plugin PoC itself — this way, the plugin itself serves as a reference implementation of the proposed optimal routing technique. The plugin is experimented by using it to programmatically partition and deploy two open-source web application projects. Then, the validity of the resulting deployments are verified by accessing all the deployed resources URLs and confirming they are routed executed in the correct service model, PaaS or FaaS. This phase follows an **Engineering Method** approach of validation — the reference implementation plugin is developed to test the solution we engineered for the hypothesis.

The complete experimentation process, the obtained results, and instructions on how to replicate all its steps on both phases are available in Section 5.2 (p. 48) and Section 6.2 (p. 66).

4.7 Summary

The current issues and open problems in the domain of this dissertation are described in Section 4.1 (p. 33). We believe there are several gaps in the literature regarding the use of hybrid deployments, especially related to their automation and ease of utilisation. There are challenges associated with its use, as well as in the routing between service model layers within these hybrid deployments.

In Section 4.2 (p. 34) we define the scope and focus of our work. It consists of investigating strategies to facilitate the programmatic partitioning of web applications into PaaS and FaaS and their deployment to the cloud, and features the implementation of a plugin prototype to facilitate these tasks.

Section 4.3 (p. 34) details the *desiderata* we propose to achieve with the implementation of the aforementioned plugin prototype. This *desiderata* is then summarised into the main hypothesis (*cf.* Section 4.4, p. 35) we propose to validate, which lies in the belief that PaaS and FaaS complementary benefits can be achieved through their use in hybrid deployments, that can be facilitated by allowing developers to (1) partition their applications, (2) use dynamic routing strategies to forward requests to the proper resources, and, ultimately, (3) by the automation of the reconfiguration process of these deployments.

Section 4.5 (p. 36) lists the main research questions that guide this work. These lie in the investigation of methodologies to partition web applications programmatically, their deployment to different service models (PaaS and FaaS) based on the identified partitioning points, and the optimisation of traffic routing between these service models after deployment. Additionally, we intend to investigate ways of automating migration between these service models, based on self-inspection, with the aim of optimising a set of performance metrics.

Finally, Section 4.6 (p. 37) discusses the validation process of this dissertation. It is separated into two phases, which consist of (1) evaluating a set of conceived and implemented techniques for dynamic routing between PaaS and FaaS using load testing techniques, and (2) experimenting the implemented plugin (using the technique that showed the best results in the previous phase) in public open-source web application projects, demonstrating its validity by verifying the success of the resulting deployments.

On the following chapter we discuss the implementation and validation of the techniques for dynamic routing between PaaS and FaaS that were conceived in this work.

Chapter 5

Techniques for Dynamic Routing between PaaS and FaaS

On the previous chapter we describe and formalise the problem under study. This chapter discusses the implementation and validation of the techniques for dynamic routing between PaaS and FaaS that were conceived in this work. Firstly, Section 5.1 explores the considerations and implementation process of each technique. Section 5.2 describes the evaluation and validation process of these techniques, as well as their results and replication instructions. Finally, Section 5.3 summarises these topics.

In this work, we provide a reference implementation of a plugin for a web framework (*cf.* Chapter 6, p. 59). However, as a step towards this goal, it is imperative to study distinct techniques for dynamic routing between different cloud service models, namely PaaS and FaaS, with the aim of minimising the client latency that results from routing traffic between these models. With the evaluation and validation of the conceived and implemented routing techniques, the best performing technique is selected to be included in the reference plugin implementation. This chapter's work provides progress towards research question **RQ3** (*cf.* Section 4.5, p. 36).

5.1 Implementation

As observed in Section 3.4 (p. 23), routing traffic between PaaS and FaaS is not a trivial task and poses a set of already identified challenges. With the aim of addressing them, we define a set of techniques that are able to route traffic between PaaS and FaaS.

These techniques aim to achieve *desiderata* **D4** (*cf.* Subsection 4.3, p. 34), thus intending to minimise client latency when accessing hybrid deployed web applications in a robust manner.

Additionally, they were conceived with the aim of exploring routing in different cloud service layers, using both existing services (such as web proxies, *e.g.*, Nginx or Apache) and existing protocols (such as DNS). With that in mind, this worked focused on the following four routing techniques (RTs):

RT1 — Proxy Server: A proxy web server (such as Nginx or Apache) instance routes incoming requests to either PaaS or FaaS based on a configuration file;

RT2 — Cloud Function Router: A cloud function routes incoming requests to either PaaS or FaaS based on the requested resource;

RT3 — Cloud Function Router + Web Server: A cloud function either routes incoming requests to a PaaS instance or executes and answers to the request locally, based on the requested resource;

RT4 — DNS Routing: Web requests are resolved into PaaS or FaaS instances based on DNS configurations, as opposed to having a device responsible for routing traffic as in the aforementioned techniques.

It is worth mentioning that although these techniques are designed for routing for PaaS and FaaS, they can be applied to other service models.

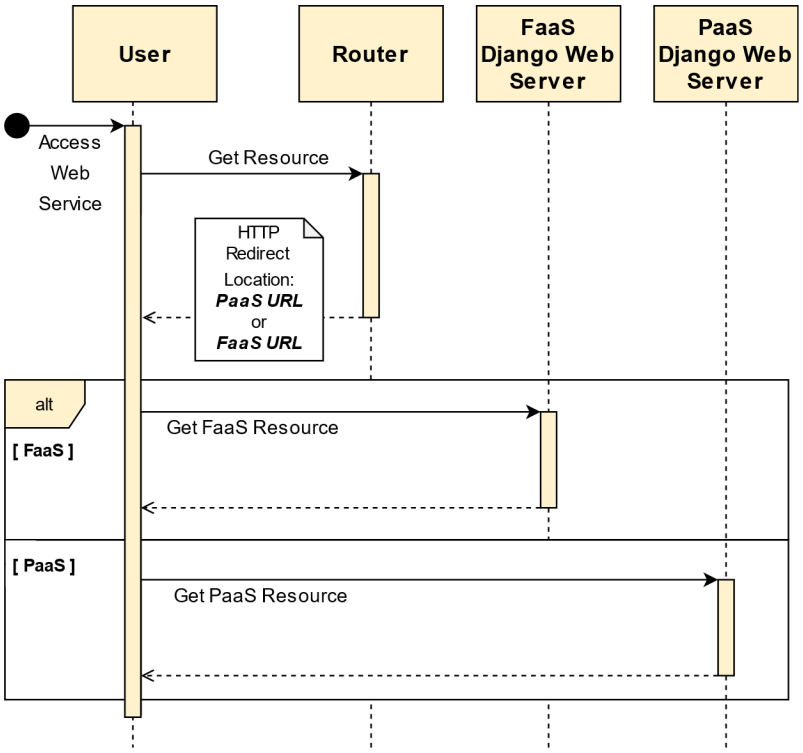

Figure 5.1: *RT1*, *RT2*, and *RT3* redirect UML sequence diagram. The router evaluates received requests, redirecting the user to the appropriate FaaS or PaaS resource location.

Techniques *RT1*, *RT2*, and *RT3* follow a similar procedure to route traffic (*RT4* operates in a different way, which is detailed further ahead in the present section). Firstly, they rely on the existence of a set of rules that dictate which resources (identified by URLs) should be redirected to PaaS or FaaS. After a web request is received on the router service, the requested resource is evaluated, and a redirect response is returned together with the location of the remote resource that should be accessed (either in PaaS or FaaS). Then, the user who created the original request repeats it to the correct destination it was redirected to. This process is illustrated by Figure 5.1 (p. 40). Additionally, this scenario closely resembles the *"Service Discovery"* design pattern described by Sousa *et al*. [33, 93, 94, 95, 96, 97, 98] which abstracts the details of the service network (in this case, the PaaS and FaaS services) by relying on an external mechanism (the router, in either *RT1*, *RT2*, or *RT3*) to route traffic.

As for the redirects themselves, they can be of the temporary or permanent type. They differ on their permanency, that is, on how long the location of the resource that the request was redirected to is cached. Although these are not fixed, temporary redirected are usually cached for very short periods of time (or not cached at all), while their permanent counterparts are usually cached for future requests. They offer both advantages and drawbacks [40].

On the one hand, permanent redirects (*"301 Moved Permanently"* and *"308 Permanent Redirect"*) aim to reduce request latency. Since requests subsequent to the first one are accessed directly without visiting the original resource, this period of time is saved in future requests. However, they pose challenges when the redirect resource location changes, a problem known as *"Cache Invalidation"*, which is cumbersome in scenarios where a system needs to reconfigure itself.

On the other hand, temporary redirects (*"302 Found"* and *"307 Temporary Redirect"*) avoid these reconfiguration issues. Yet, typically the original resource in the redirecting process must be visited in all requests, which introduces communication delays.

With this in mind, in this work we opted to study scenarios using temporary redirects. Even though system reconfigurability is not focused on this work, this type of redirects is more suited to accommodate those requirements in further work on this topic.

5.1.1 RT1 - Proxy Server

This routing technique relies on a device hosting a proxy web server (such as Nginx or Apache) receiving incoming requests and forwarding them to either PaaS or FaaS. The UML deployment diagram on Figure 5.2 (p. 42) illustrates this technique's architecture.

The proxy server configurations used to route incoming traffic to the different available cloud resources can be dynamically generated by inspection of an application's source code, or based on other metrics such as usage rates and cost prediction (although the latter was not explored in this study).

It is worth mentioning that, since user requests need to be routed to the proxy server instance before being routed to cloud functions, which are generally closer to the user, the router in this architecture may be a system bottleneck (*e.g.*, latency, system failure).

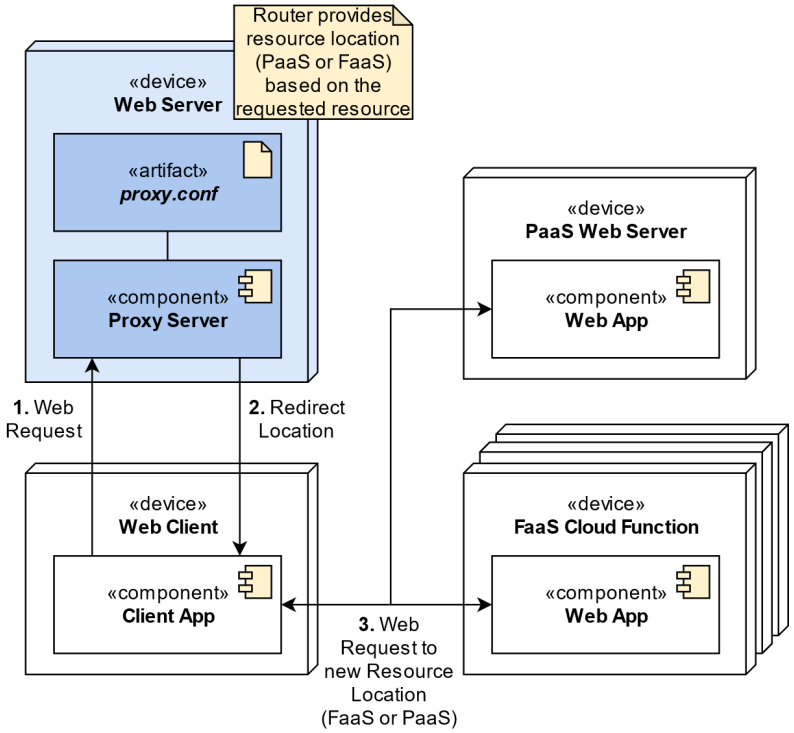

Figure 5.2: *RT1* — Proxy Server UML deployment diagram. The proxy web server evaluates and redirects incoming requests to the appropriate PaaS or FaaS resource location.

Furthermore, updating the routing configurations may imply the need to restart the proxy instance, which may result in system downtime.

```
http {
    # ...

    server {
        # ...

        location <%=ENV['PAAS_REGEX']%> {
            rewrite ^/(.*) <%=ENV['PAAS_URL']%>/$1 redirect;
        }

        location <%=ENV['FAAS_REGEX']%> {
            rewrite ^/(.*) <%=ENV['FAAS_URL']%>/$1 redirect;
        }
    }
}
```

Listing 4: *RT1* — Nginx configuration file snapshot. Requests are evaluated and redirected to either PaaS or FaaS using regular expression matching on the requested URL.

To implement this routing technique, an Nginx proxy server was used, which was deployed using the free tier Heroku Nginx platform. By evaluating the requested resource's URL using regular expressions, the service redirects incoming requests to either PaaS or FaaS, as illustrated in the configuration snapshot in the configuration present in Listing 4 (p. 42).

The Heroku Nginx platform allows the specification of a ruby template file as the proxy configuration file. In the aforementioned figure, a set of environment variables are used in the configuration, which facilitated experimentation since it was possible to make configuration modifications without changing the source code file itself. The environment variables were set on Heroku's service dashboard directly.

5.1.2 RT2 - Cloud Function Router

This routing technique relies on an HTTP-triggered cloud function device receiving incoming requests and forwarding them to either PaaS or FaaS. The UML deployment diagram on Figure 5.3 illustrates this technique's architecture.

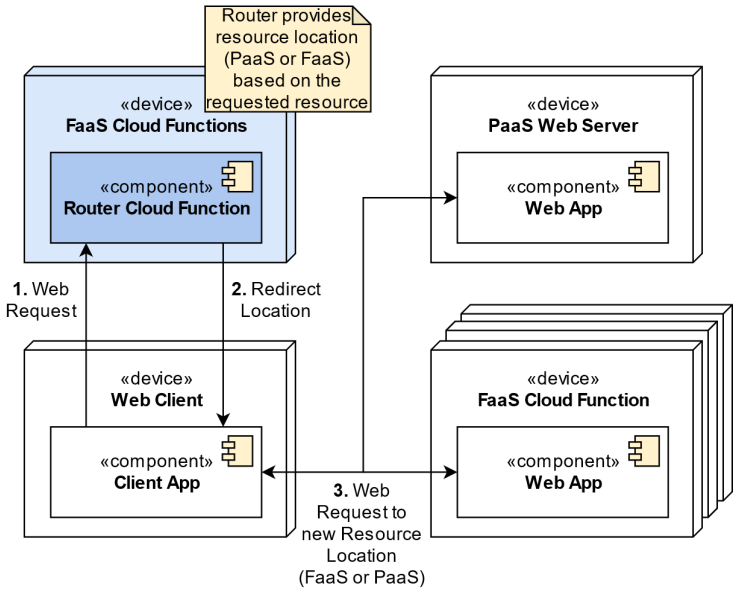

Figure 5.3: *RT2* — Cloud Function Router UML deployment diagram. A cloud function evaluates and redirects incoming requests to the appropriate PaaS or FaaS resource location.

Similarly to *RT1*, the configurations used to route incoming traffic to the different available cloud resources can be dynamically generated.

However, in contrast to *RT1*, these configurations may be dynamically updated over time without the need to restart the system (without downtime). Additionally, on the one hand, the client latency that may occur in *RT1* from redirecting traffic from the PaaS to FaaS is eliminated since the routing process is, in this architecture, present in the FaaS layer itself.

On the other hand, a new latency from redirecting traffic from FaaS to PaaS is introduced — however, given that FaaS services are more geographically distributed and generally closer to any user location, this latency is expected to be significantly smaller than the former.

To implement this routing technique, the Azure cloud function service was used. Similarly to *RT1*, the router function evaluates the requested resource's URL using regular expressions, and then redirects received requests to either PaaS or FaaS. The routing cloud javascript function is illustrated in Listing 5.

```
module.exports.route = async function (context, req) {
    const segment = req.params.segments || "";

    if (process.env.PAAS_REGEX.test(segment)) {
        context.res
            .status(302)
            .set("location", `${process.env.PAAS_URL}/${segment}`)
            .send();
    } else if (process.env.FAAS_REGEX.test(segment)) {
        context.res
            .status(302)
            .set("location", `${process.env.FAAS_URL}/${segment}`)
            .send();
    } else {
        context.res.status(400).send();
    }
};
```

Listing 5: *RT2* — Azure router cloud function. Requests are evaluated and redirected to either PaaS or FaaS using regular expression matching on the requested URL.

The cloud function is triggered by incoming HTTP requests, matching any routes, and verifies whether the request should be redirected to PaaS or FaaS. In the case no match is found, an error response is returned. The deployment of this function was facilitated by the usage of the Serverless framework, which allowed creating the function with minimal additional configuration. Additionally, the environment variables used for the URL matching were defined via a configuration file.

5.1.3 RT3 - Cloud Function Router + Web Server

This routing technique is very similar to *RT2* in the sense that requests are always received in FaaS. However, besides the routing function, the web application's functions are also deployed in the FaaS layer. In the case that requests can't be fulfilled locally in the FaaS layer, these are redirected to the PaaS layer by the routing function. The UML deployment diagram on Figure 5.4 (p. 45) illustrates this technique's architecture.

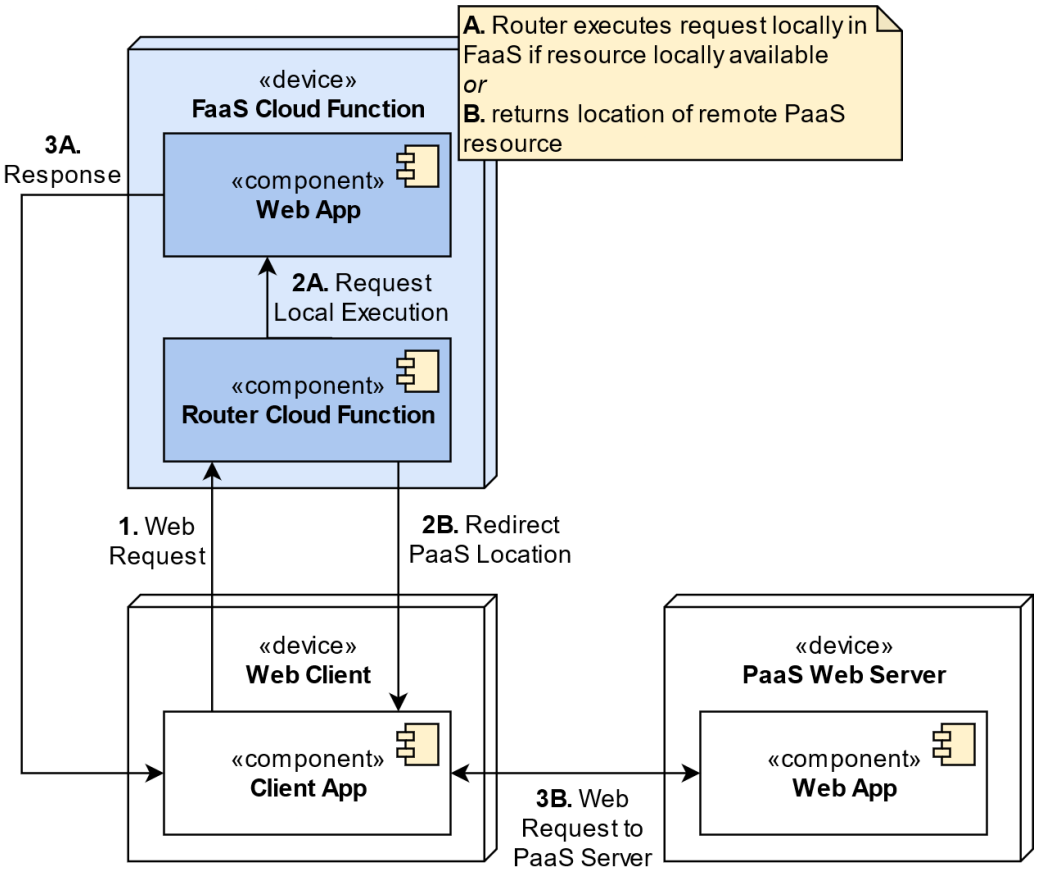

Figure 5.4: *RT3* — Cloud Function Router + Web Server UML deployment diagram. A cloud function evaluates incoming requests and either (a) fulfills the request locally if the requested resource is available, or (b) otherwise, redirects the request to the PaaS resource location.

Despite offering almost the same advantages and drawbacks as *RT2*, this technique altogether removes the latency of routing traffic to remote cloud functions FaaS instances by having these resources available locally. For that reason, although this can result in larger amounts of storage and memory being needed to host the router function, this technique aims to be an improvement on *RT2* in terms of latency optimisation.

Similarly to the previous one, the Azure cloud function service was used to implement this routing technique. The router function evaluates the requested resource's URL using regular expressions, and if the requested resource is locally available, then it executes the request and returns the answer. If not, the received request is redirected to PaaS. The routing cloud python function is illustrated in Listing 6 (p. 46).

Like in *RT2*, the cloud function's execution is triggered by incoming HTTP requests and the deployment was facilitated by the usage of the Serverless framework. To allow the execution of web server functions locally, we used the *AzureFunctionsWsgi* [30] package. It facilitates the execution of web application requests inside python azure functions by providing a middleware for WSGI (Web Server Gateway Interface) applications.

```
def main(req: func.HttpRequest, context: func.Context) -> func.HttpResponse:
    match = re.match(urlAndSegmentsRegex, req.url, re.M|re.I)
    segments = match.groups()[-1]

    if re.search(os.environ.get('PAAS_REGEX'), segments, re.M|re.I):
        return func.HttpResponse(
            status_code = 302,
            headers = {
                "location": f"{os.environ.get('PAAS_URL')}/{segments}"
            }
        )
    elif re.search(os.environ.get('FAAS_REGEX'), segments, re.M|re.I):
        return AzureFunctionsWsgi(application).main(req, context)
    else:
        return func.HttpResponse(status_code = 400)
```

Listing 6: *RT3* — Azure router cloud function. Requests are evaluated using regular expression matching on the requested URL and either the request is (a) locally fulfilled if the requested resource is available, or (b) otherwise, redirected to the PaaS resource location.

5.1.4 RT4 - DNS Routing

Instead of featuring a device that is responsible for routing traffic as in the three aforementioned techniques, this technique relies on domain name resolution to route traffic to different service models. The UML deployment diagram on Figure 5.5 illustrates this technique's architecture.

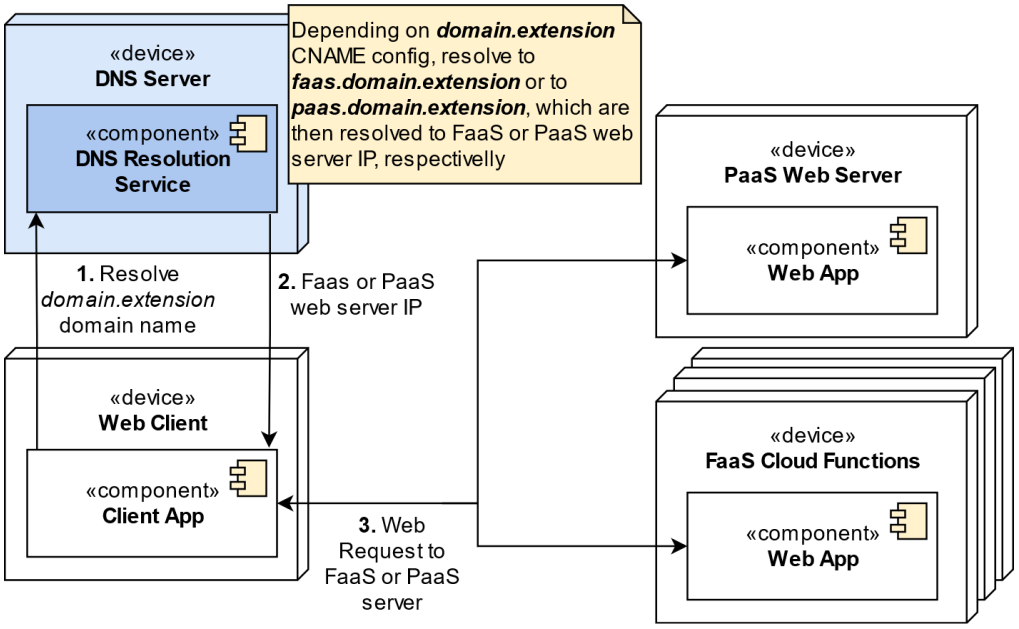

Figure 5.5: *RT4* — DNS Routing technique UML deployment diagram. The requests' domain named is resolved to either PaaS or FaaS based on the DNS configuration.

To implement this technique, the *tese.ruialves.me* subdomain was used. This technique relies on the following DNS configuration:

1. A CNAME record to resolve *faas.tese.ruialves.me* to the FaaS cloud functions;
2. A CNAME record to resolve *paas.tese.ruialves.me* to the PaaS web server;
3. A CNAME record, with a TTL value of 300 seconds, to resolve *tese.ruialves.me* to either of the previous names

To route traffic, the web requests are made to *tese.ruialves.me* and requests are resolved to either FaaS or PaaS by updating the aforementioned record 3. It is worth pointing out that this record should have a low TTL value, so that it is possible to reconfigure it in an expeditious way, whilst avoiding the risk of DNS caches becoming outdated — the selected value of 300 seconds was enforced due to the available service's limitations.

The implementation steps of this technique included configurations at a PaaS Heroku web server, a FaaS Azure function app, and Cloudflare domain management.

Firstly, regarding PaaS configurations, the Heroku web dashboard was accessed to add a custom domain. Adding the custom domain $*.paas.tese.ruialves.me$ resulted in the creation of a Heroku DNS target, which can be used to create a CNAME record for DNS resolution.

Secondly, regarding FaaS configurations, the Azure web dashboard was accessed to configure the FaaS custom domain. Creating the custom domain *faas.tese.ruialves.me* resulted in the creation of an Azure DNS target, which can be used to create a CNAME record for DNS resolution, and in a custom domain verification ID string, which must be used to create a TXT record for domain ownership confirmation.

Then, the necessary DNS configurations were created using Cloudflare domain management dashboard. These DNS records are depicted in Table 5.1. The *paas.tese* entry resolves to the DNS target previously created on Heroku. The *faas.tese* entry resolves to the DNS target previously created on Azure, and the *asuid.faas.tese* entry contains the domain verification ID created on Azure for ownership confirmation purposes. Finally, the *tese* record is updated to resolve to either the PaaS or FaaS names, thus routing incoming traffic by DNS resolution.

Table 5.1: *RT4* — Cloudflare DNS records. A CNAME record resolves *paas.tese* to the PaaS server, a CNAME and a TXT record resolve *faas.tese* to the FaaS cloud functions, and a final CNAME record resolves *tese* to either *paas.tese* or *faas.tese*.

Type	Name	Content
CNAME	paas.tese	hidden-baboon-7ltuxj23r6rx3vo8z4l85k4x.herokudns.com
CNAME	faas.tese	sls-weur-dev-serverless-framework-app.azurewebsites.net
TXT	asuid.faas.tese	82BA3A98D17291BD832288F19EF0C711C6541B8FCC0A53...
CNAME	tese	paas.tese **or** faas.tese

However, this technique lacks the ability to control where traffic is routed to based on, for example, the requested resource URL — the only way of indicating to where the domain should

be resolved to is by changing a DNS address (*A* or *AAAA*, if pointing to a specific IPv4 or IPv6, respectively) or canonical name (*CNAME*, if point to another domain) records. Moreover, these record changes take time to propagate based on the record's TTL (time to live) and, even if the TTL value is minimal, some DNS servers fail to acknowledge TTL values below a given threshold.

5.2 Evaluation and Validation

5.2.1 Methodology

To evaluate the traffic routing latency that results from each of the routing techniques that were presented in the previous chapter it is important to analyse their behaviour, both in terms of performance and robustness, when accessed by different loads of users throughout a period of time.

In order to do so, we use load testing techniques to generate web traffic in a set of different experiments — a number of simulated users are generated, and each of them periodically executes requests to the router.

Additionally, given that the latency performance is a key factor in evaluating performance, the simulated users should be widely geographically distributed across the globe. With that in mind, a cluster of machines (their infrastructure details are specified in Subsection 5.2.3) was used to generate traffic, and distributed in various regions of the world — Europe, Asia, North America, South Africa, and Middle-East.

This cluster features a total of six computation nodes, with one of them (located in Europe) serving as the master node, which orchestrates the load testing procedure among the worker nodes (including itself).

For evaluating the performance and robustness of each routing technique, it is important to capture and take into account the following metrics throughout each load test:

Total Number of Requests: indicates the total amount of traffic that was received in the router;

Total Number of Failed Requests: indicates the total number of times the routing failed (regardless of the failure reason);

Request Failure Rate: derived from the aforementioned metrics, which indicates the robustness of the routing technique;

Average, Minimum, Q1, Median, Q3, and Maximum Response Times: to understand the latency performance of the routing technique;

Request Failure Reason: only captured on failed requests, for qualitative evaluation purposes.

To facilitate the load testing process, the Locust[1] open-source load testing framework was used, given that it features a small learning curve, capabilities for setting up tests using a cluster

[1] Locust — An open-source load testing tool, https://locust.io/

of machines, vast community support (due to its open-source nature), and a simple programmatic interface using python, as visible in the locust test configuration present in Listing 7.

```
class User(HttpUser):
    wait_time = between(0.5, 2)

    @task(1)
    def paas_request(self):
        self.client.get(url=os.environ.get('PAAS_URL'), timeout=10.0)

    @task(1)
    def faas_request(self):
        self.client.get(url=os.environ.get('FAAS_URL'), timeout=10.0)
```

Listing 7: Locust test configuration. Each generated user periodically executes a task with equal probability (each has a priority weight of 1). These tasks consist of performing a request to either the PaaS or FaaS resource.

Regarding the tests, in each test the cluster nodes periodically generate web requests to the routing device, depending on the technique being evaluated. Each request features a timeout period, in which if no response is received, the request is deemed as failed (timeout failure). The user load is evenly distributed among all the cluster machines and, since the number of users may be high in various tests, a startup period is taken to generate the simulated user workers — data gathered during this startup period is not considered, and results start being gathered and the test phase begins after all the simulated users are created. Moreover, the user requests are approximately evenly distributed between FaaS and PaaS, with each model receiving about half the requests. The experiments were done under the following configurable parameters:

Duration: each test run took a period of 200 seconds;

Number of Users: constant in each specific test. Ranged from 500 to 2000 (the reasoning for these values is provided in Subsection 5.2.4), with an increment of 500 users per test;

User Request Periodicity: each user generated a new request every 0.5 to 2 seconds, following a uniform distribution probability between these two values;

User Timeout Period: each user timed out (resulting in a timeout failure) after a 10 seconds period without receiving a response.

We chose values that aim to represent gradual traffic growth throughout a given period of time for these parameters. However, their optimisation towards a specific scenario is out-of-scope of this work, and thus was not explored.

These tests were repeated 5 times for each routing technique, and the average results for each metric were taken into account. Furthermore, the scripts used to automate these tests and the full

results data (as well as instructions on how to replicate this process) are available in this work's replication package [4].

5.2.2 Test Web Application

A simple web application was developed with the purpose of being used in the routing techniques test sets. The application features a REST API and serves a simple website, and uses an SQL database to persist data.

The web application closely resembles the sample web app from the Django *Writting your first Django app* tutorial[2], and its source code, as well as its setup instructions, are available on this work's replication package [4].

5.2.3 Cloud Infrastructure Details

The load testing cluster was deployed using 6 Azure VMs, and its deployment diagram is illustrated in Figure 5.6.

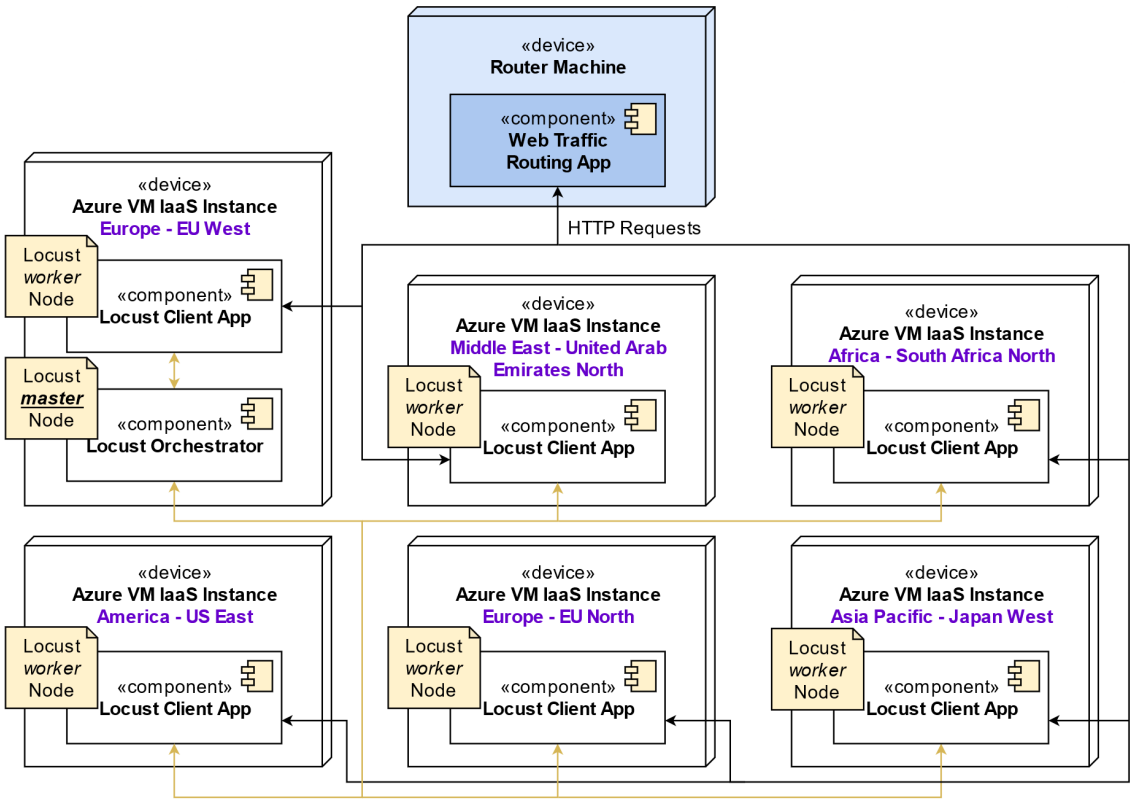

Figure 5.6: Load Testing Cluster— VMs architecture. Yellow associations represent load testing orchestration communication, while black associations represent HTTP requests.

[2]Django Project — Getting started documentation guide, https://docs.djangoproject.com/en/3.2/intro/

The clusters featured 6 virtual machine (VM) instances, located in Europe (one in Europe West and another in Europe North), America East (United States), Asia Pacific (Japan West), Africa (North of South-Africa) and Middle-east (United Arab Emirates). The VMs consisted of Azure *B1s* instances, featuring 1 vCPI, 1 GiB of RAM, 2 SSD datadisks, and a maximum caps of 320 IOPS. The VM located in the Europe West region was used as the orchestrator machine.

As for the VMs networking, each of the machines was set up to allow inbound HTTP/HTTPS traffic, to allow communication in the Locust cluster. However, since each of the VMs belonged to a different deployment region, it was not possible to include them in the same network. For that reason, a gateway subnetwork was added to the address space of each of the machines in the cluster, together with a VPN gateway to communicate with machines in other regions. Then, a VNet-to-VNet (Virtual Network) connection was created for each orchestrator-cluster VM region pair, thus allowing communication among all the cluster machines. The resulting network setup in illustrated in Figure 5.7.

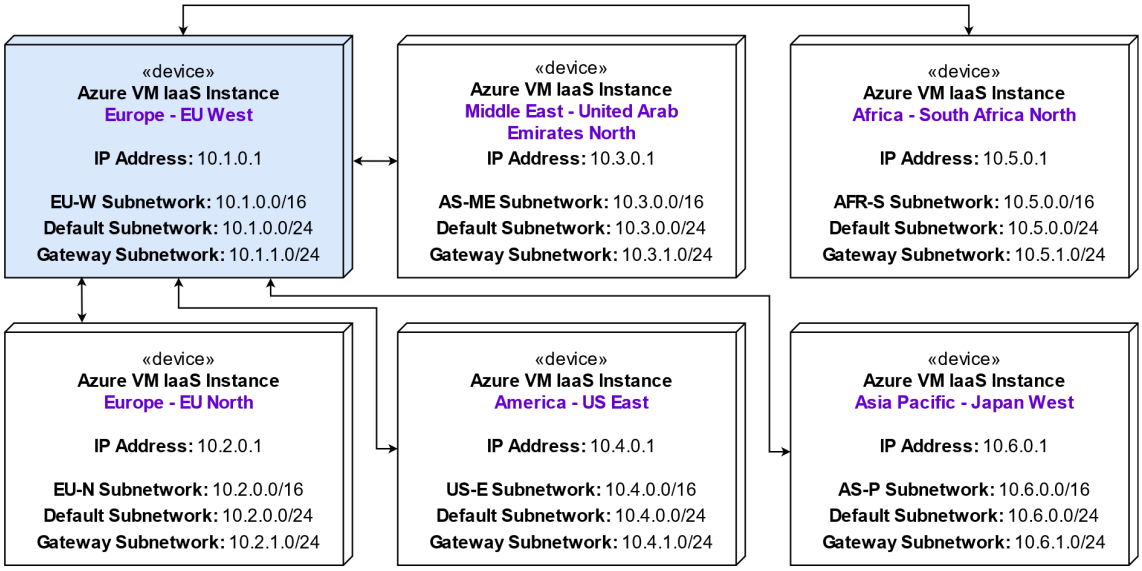

Figure 5.7: Load Testing Cluster — Networking Setup. Each of the associations represents a VNet-to-VNet connection.

As for the routers, each routing technique was deployed as follows:

RT1: Heroku PaaS Nginx instance, featuring 1 vCPU, 512 MiB of RAM, and an SSD datadisk;

RT2: Azure Cloud Function App services;

RT3: Azure Cloud Function App services (both the router function and the web application);

RT4: CloudFlare was used for domain management.

Additionally, the test web application was deployed in both PaaS and FaaS. As for PaaS, it was deployed in a Heroku instance (featuring 1 vCPU, 512 MiB of RAM, and an SSD datadisk) in

the Europe region. The database used by the application was also deployed in Heroku (featuring 1 shard and 1GiB of storage capacity), in the Europe region. As for FaaS, the web app was deployed using Azure python cloud functions.

However, since a free-tier Heroku instance with only 512MiB of RAM is being used to deploy the test web application in PaaS, the machine is likely to crash after a certain load threshold is met, given that the free-tier does not feature resources scaling. Thus, given this infrastructure limitation, it is important to discover the user threshold that the web server PaaS machine can handle — this threshold can then be used as a baseline for future load tests.

5.2.4 PaaS User Threshold Discovery

A set of load tests were made to find the PaaS machine threshold that was introduced in the previous subsection. These tests were also made using the Locust framework, and the same Azure VM cluster was used to perform the tests in a distributed manner, thus simulating traffic from multiple regions of the globe.

The tests consisted of progressively increasing the number of incoming user requests until a given average response time threshold was met. Each test run took 150 seconds, and the number of simulated users ranged from 100 to 2400, with an increment of 100 users per test. Each of these users generated a new request every 0.5 to 2 seconds, following a uniform distribution probability between these two values, and timed out (resulting in a timeout failure) after a 10 seconds period without receiving a response. These tests were executed a total of 5 times, and the average value of the various captured parameters was taken into account for analysis purposes.

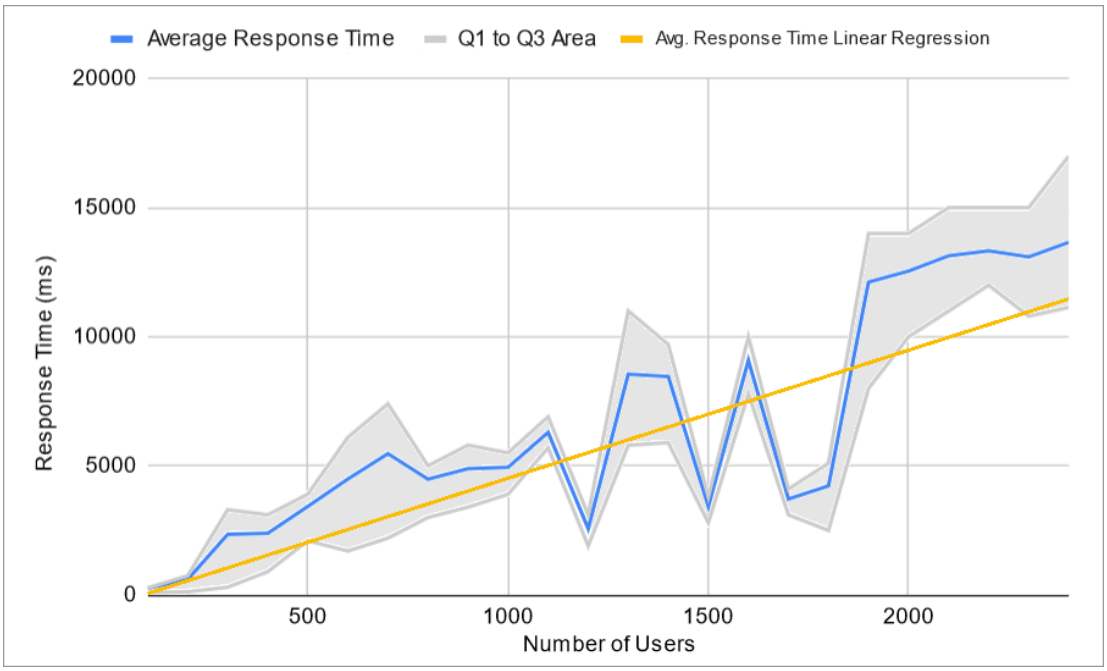

Figure 5.8: Response time Average, lower quartile Q1 to upper quartile Q3 area, and expected growth linear regression based on number of users.

Considering an average response time maximum threshold of 10 seconds, the limit of the PaaS machine under study was reached at the value of 1900 users, as illustrated in Figure 5.8 (p. 52), which indicates a maximum capacity (on average) threshold of 1800 users.

Thus, for the routing tests purposes, this value can be used as a baseline, and a number of users values ranging from comfortable loads (*e.g.*, 25% of this threshold, that is, about 500 users) to above-capacity loads (*e.g.*, 110% of this threshold, that is, about 2000 users) would be interesting to explore.

Although the response time tends to fluctuate in spikes as the number of users increases (as visible in the figure), it shows an overall tendency to grow (as expected). Even though a total of 5 test sets were executed, the response time to number of users ratio is expected to stabilise as the number of executed tests increases — the average response time linear regression in the figure illustrates the overall expected growth rate.

The scripts used to automate these tests and the full results data (as well as instructions on how to replicate this process) are available in this works replication package [4]. Additionally, the collected raw results are available in Appendix A (p. 75).

5.2.5 Results

The response time metrics were used to evaluate the techniques' quality and the failure rate was used to evaluate the techniques' robustness. Additionally, in a qualitative perspective, the failure cause (*e.g.*, timeout and server error) was also considered.

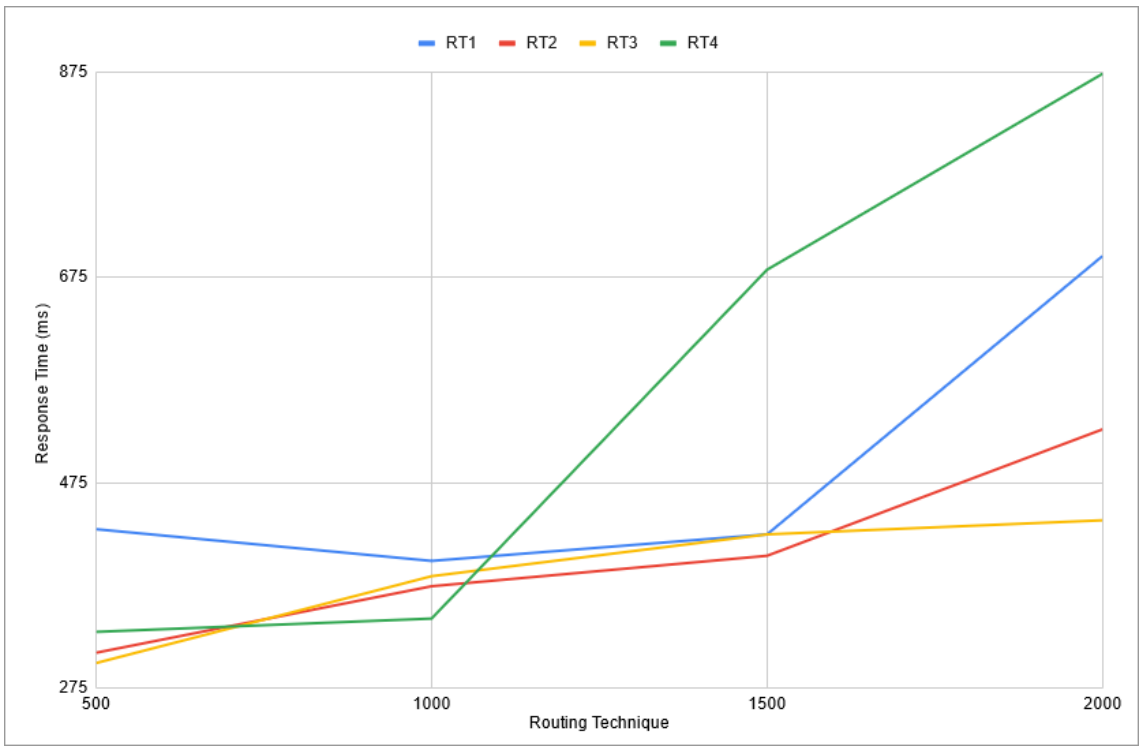

Figure 5.9: Average response time based on the number of users for each routing method.

The results obtained for the average response time based on the number of users for each routing method is visible in Figure 5.9 (p. 53). Although differences are not significant for a small number of users values, the average response times grow at a lower rate (with the increasing of the number of users) with the techniques that perform the routing task at the FaaS level, that is, techniques *RT2* and *RT3*.

Figure 5.10 depicts a more detailed view of the results obtained for user loads greater than 1500 users. *RT1* and *RT4* consistently showed larger response times than their FaaS-level counterparts (with *RT4* showing the worse results), and *RT3* showed even lower response times, which was expected since this technique removes the latency of routing traffic to remote cloud functions that exists in *RT2* by having the web application resources available locally. It's worth mentioning that the *maxima* obtained for each scenario was not included in the plot, given that all results were similar (around the 10 seconds timeout threshold). The complete results for these tests are available in Appendix B (p. 77).

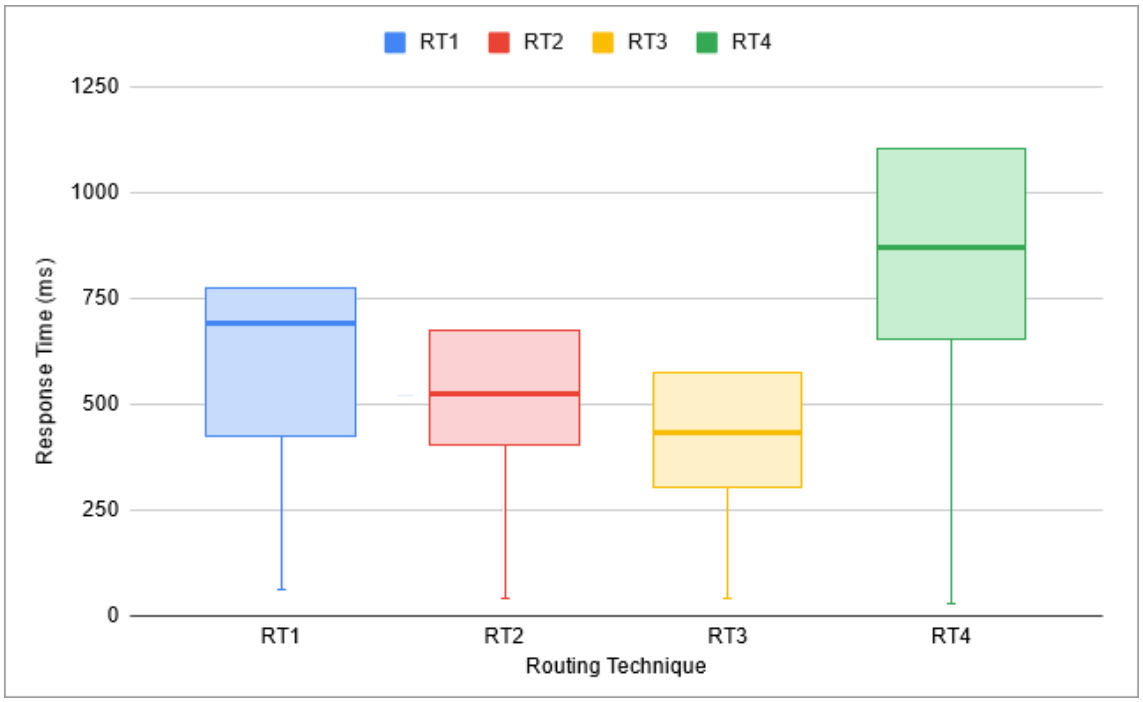

Figure 5.10: Response time for each routing method box plot on user loads greater than 1500 users. Each plot box features, from bottom to top, the minima, lower quartile Q1, average, and upper quartile Q3.

When it comes to failure rates, techniques *RT1*, *RT2*, and *RT3* showed values close to zero, with the reason of these vestigial failures being request timeout. However, technique *RT4* showed a failure rate close to 21%. After closer examination, the vast majority of these failures (nearly 100%) seemed to be caused by server error on the DNS resolution process (as observed in locust report logs). These occurred throughout the whole testing process, both in the beginning and end. However, the root cause for these failures could not be identified.

Furthermore, it is of interest to inspect the average response time that was spent on requests redirected to FaaS and PaaS in each routing technique, keeping in mind that the number of requests was divided between them approximately evenly (*cf.* Figure 5.11).

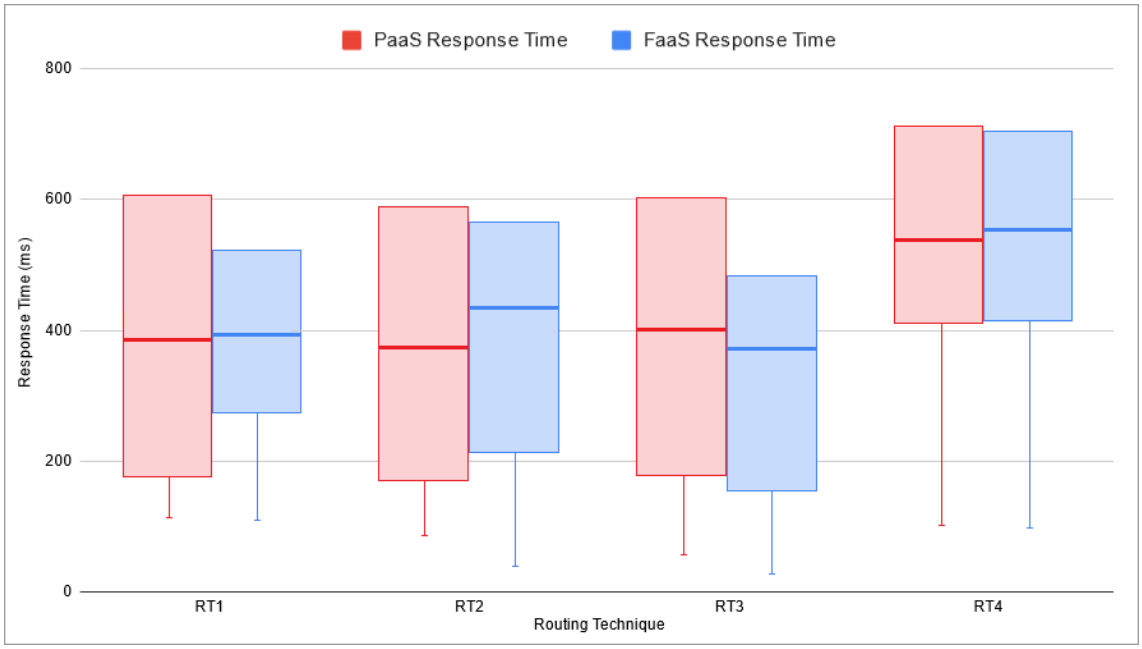

Figure 5.11: PaaS vs FaaS response time Box Plot for each routing method. Each plot box features, from bottom to top, the minima, lower quartile Q1, average, and upper quartile Q3.

The *RT1* technique showed lower response times in PaaS than in FaaS, which was expected since both the proxy server and PaaS test web server were deployed on Heroku while sharing the same region. *RT4* showed similar values for each service model, since this technique does not feature a deployed router device *per se*. When it comes to *RT2*, it showed similar PaaS response times to *RT1*, but much lower FaaS response times (given that both the router function and the *FaaS* web application were deployed in the FaaS layer). Finally, *RT3* showed PaaS response times similar to *RT2*, but lower *FaaS* response times, due to the fact that the router function itself could access the web application locally, as previously explained.

It is noteworthy to point out that the *maxima* obtained for each scenario was not included in Figure 5.11 given that all results were similar (around the 10 seconds timeout threshold). Moreover, FaaS showed more consistent response times, as can be observed by the smaller Q1 to Q3 box plot area than its PaaS counterpart. The FaaS layer techniques, *cf.* *RT2* and *RT3*, were also able to achieve the fastest response times (lower *minima*). The complete results for these tests, separated between PaaS and FaaS, are available in Appendix B (p. 77).

Although it would be interesting to compare other metrics such as routing deployment costs of each technique, this was not possible since the available Azure student subscription did not feature access to detailed costs billed for each resource. Notwithstanding that *RT2* and *RT3* showed similar results, the difference between these techniques response times is expected to increase with the

growth of the number of users. That being said, the *RT3* technique showed the overall best results in terms of response time, as well as a failure rate close to zero.

Finally, the results obtained in this experimentation phase demonstrate the second premise of our hypothesis. We show that moving the routing service to the FaaS layer results in the optimal configuration, due to this service model's wider geographical distribution, minimising client latency when accessing these cloud resources and, thus, optimising the web application in itself.

5.2.6 Threats to Validity

The infrastructure-related constraints and limitations that were present in this validation process pose a set of external (if there is threat that the results can't be generalised the scope of this work) and internal (if there is threat that the results are caused by internal treatment anomalies) threats to validity [37]:

The PaaS machines' performance may influence the results (external threat)

The available cloud infrastructure resources consisted of an Azure student subscription with 100$ worth of credits, Heroku free PaaS instances, and Cloudflare free tier domain management. Not only did these limitations pose difficulties throughout the development process, the fact that only low budget infrastructure was used in the experimentation and validation phases may influence the results, since machines with higher performance levels could present different results — especially when it comes to the PaaS machines used that, unlike the FaaS functions, did not scale, neither horizontally nor vertically, due to tier limitations;

The lack of distinct testing scenarios may influence the results (external threat)

Not enough credit was available to introduce different types of user load distributions (such as user load spikes, that is, high-stress conditions), which is a common pattern in real scenarios, and which would provide valuable insight on the quality of each of the considered routing techniques;

The limitation on captured metrics may influence the decision process (external threat)

The available infrastructure also limited the collection of metrics such as memory consumption and the detailed costs billed for each resource used. The absence of these metrics does not allow us to include criteria such as cost-efficiency in the evaluation of the techniques, which would allow more informed and possibly different conclusions;

The DNS resolution process error on *RT4* was not identified (internal threat)

Given that the root cause of the server errors that resulted from the DNS resolution process on *RT4* was not identified, there could be configuration errors that are negatively influencing this routing technique's behaviour and results.

5.2.7 Replication Package

To allow the full replication of the various experiments and validation steps presented in this chapter, as well as to detail the instructions on how to replicate them, an experiments replication package [4] was built, featuring the following content:

Routing Techniques: Source code and a comprehensive guide of their deployment;

Heroku PaaS machine threshold discovery tests: Results, tests automation scripts, and a comprehensive guide on how to use them;

Routing Latency tests: Results, tests automation scripts (for both single-machine and cluster modes), and a comprehensive guide on how to use them.

5.3 Summary

In this chapter, a set of four techniques were conceived and implemented, along with an analysis of their main strengths and pitfalls.

RT1 uses a proxy web server to route traffic, deployed in PaaS, operating with minimal configuration although prone to be a system bottleneck in terms of either failure or latency introduction.

RT2 aims to improve upon *RT1* by removing the latency of routing traffic from PaaS to FaaS, moving the routing task to a cloud function. However, a new latency from redirecting traffic from FaaS to PaaS is introduced — although this is expected to be significantly smaller than the former, given that FaaS services are more geographically distributed and generally closer to any user location.

RT3 emerges as an improvement to *RT2*, aiming to remove the latency of routing traffic to remote cloud functions FaaS instances by having these resources available locally. If the requested resource is not available locally, the request is still redirected to PaaS, similarly to the previous technique.

RT4 relies on domain name resolution to route traffic to different service models, instead of featuring a device that is responsible for routing traffic as in the three aforementioned techniques. However, when compared to the other techniques, it can't provide the same level of detail when specifying which resources should be redirected to each layer.

Then, the performance of the four implemented techniques for dynamic routing between PaaS and FaaS was studied. Using load testing techniques, a cluster of six machines geographically spread throughout the globe was used to test each of the routing techniques. The latency performance and robustness of each technique's router was evaluated when routing traffic to a test web application that was deployed in both Heroku PaaS and Azure FaaS.

The results showed that *RT3* achieved the best performance in terms of minimising user response time, while showing a robust behaviour (a failure rate close to null), thus demonstrating the second premise of our hypothesis. The results obtained may also be consulted in Appendix A

(p. 75) and Appendix B (p. 77), and a comprehensive guide on how to replicate the deployment of each of the presented routing techniques can be found in this work's replication package [4].

On the following chapter we discuss the implementation and validation of the reference plugin prototype.

Chapter 6

Reference Implementation

On the previous chapter we discuss the implementation and validation of the techniques for dynamic routing between PaaS and FaaS that were conceived in this work. This chapter discusses the various aspects of the reference plugin prototype. Firstly, Section 6.1 provides the implementation details of the plugin, as well as the key architectural decisions in its creation. Then, Section 6.2 discusses the plugin's experimentation on open-source web projects, together with the validation of the resulting deployments. Finally, Section 6.3 summarises these topics.

The previous chapter shows that routing technique *RT3* (*cf.* Chapter 5, p. 39) achieved the best results when it comes to minimising client latency, while maintaining the system's robustness. Thus, in this chapter, we provide a reference implementation for this technique, while working towards research questions **RQ1** and **RQ2** (*cf.* Section 4.5, p. 36).

6.1 Implementation

The plugin intends to facilitate the programmatic partitioning of web applications into PaaS and FaaS and their deployment to the cloud, for a given web development framework. The developed work was directed to the Django python web framework, given that it consists of one of the most popular development frameworks.

Moreover, the plugin software was implemented and packaged as a python Django package[1]. Although it could have been implemented as an external software component, a python package makes it easier to programmatically access a Django project's configuration, since the plugin and the Django project share the same language (python). Besides that, a python package simplifies the plugin's installation and configuration.

[1] Django Cloud Deployer package, available on Pypi: https://pypi.org/project/django-cloud-deployer/

Although the plugin intends to facilitate the deployment of a web application into FaaS and PaaS in a way that is agnostic to the cloud provider being used, the available cloud infrastructure limited the number of implemented supported providers. Given that the Heroku platform free tier and Azure student subscription were available, these providers were adopted as PaaS and FaaS (respectively) implementation targets throughout this work.

Additionally, although the number of available providers was limited due to infrastructure availability restrictions, we chose to use the Serverless framework to facilitate configuration and deployment, given that it abstracts a large part of provider-specific configuration steps.

6.1.1 Deployment Infrastructure

The resulting cloud infrastructure of a deployment using the implemented plugin — which closely resembles the architecture of *RT3* (*cf.* Figure 5.4, p. 45) — is illustrated in the UML deployment diagram present in Figure 6.1.

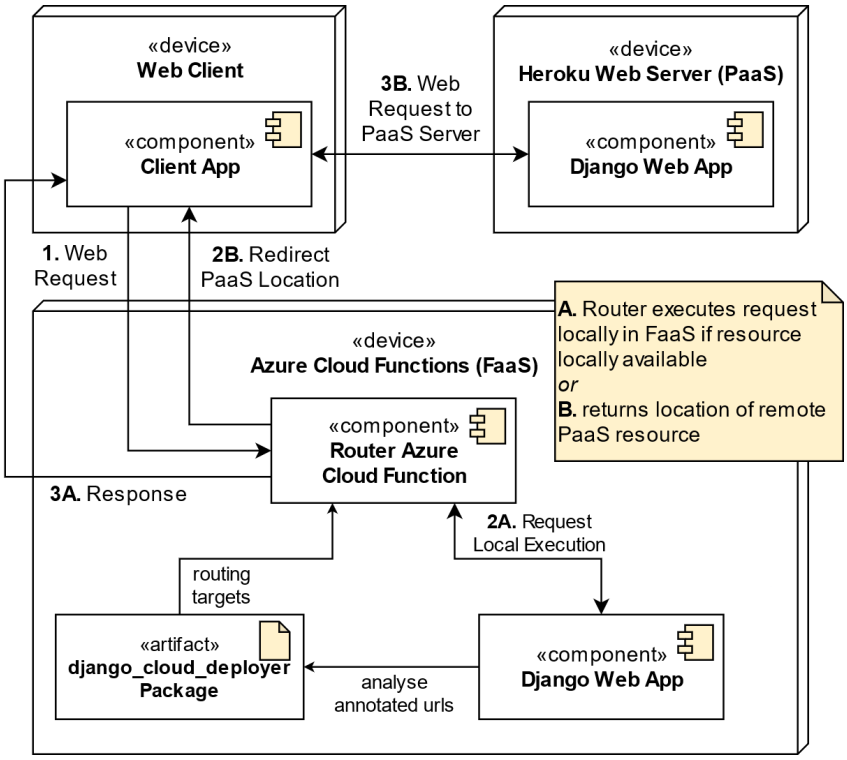

Figure 6.1: Django Cloud Deployer — Infrastructure UML Deployment Diagram. Routing rules are generated by the plugin based on the annotated source code URLs, and router cloud function redirects traffic to either PaaS or FaaS.

6.1.2 Architecture

The plugin provides two programmatic interfaces to facilitate the cloud deployment and routing: (a) a routing annotations API, which operates at the source code level, and that allows a developer

to indicate through the use of two functions — *runInPaaS* and *runInFaaS* — which resources, identified by their respective URLs, should be deployed and routed to PaaS or FaaS (respectively), and (b) a CLI interface that features a set of commands to manage, deploy and delete deployments and infrastructure.

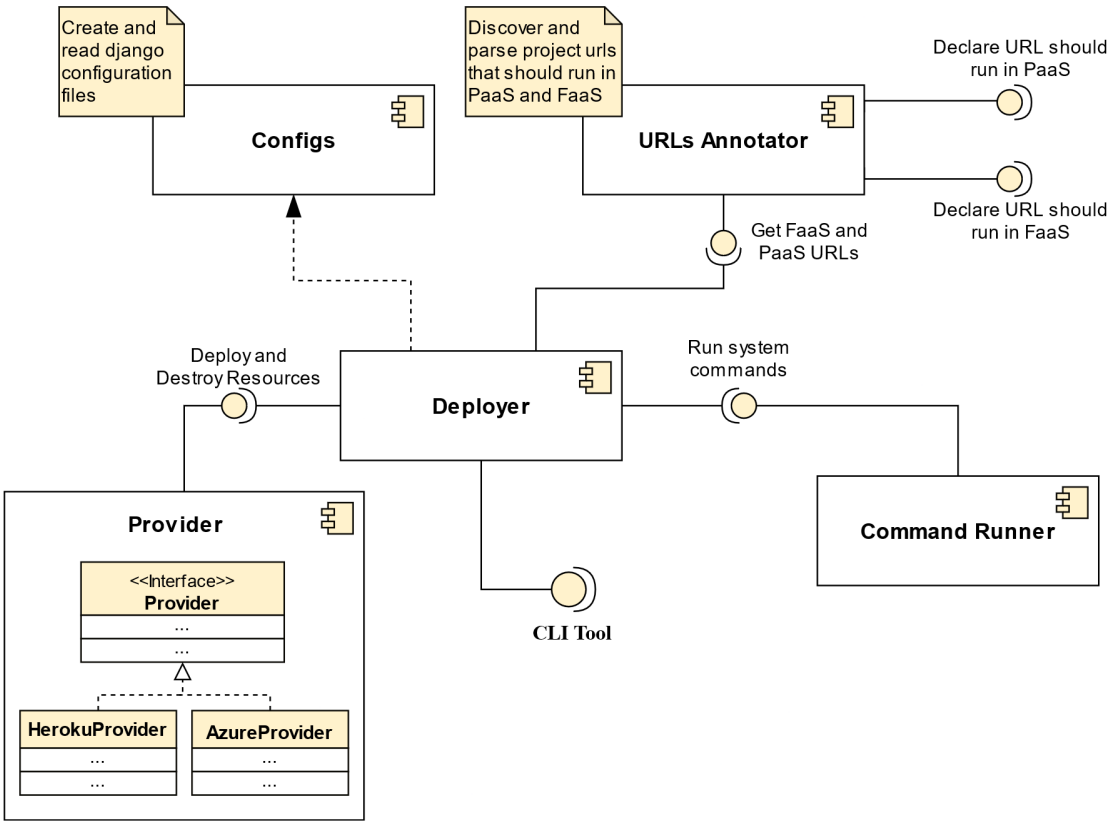

Figure 6.2: Django Cloud Deployer — Architecture UML Component Diagram. The system features five key components and exposes a command-line interface and a programmatic URL PaaS/FaaS annotation interface.

The components architecture that was conceived and implemented for the plugin is depicted in Figure 6.2. Besides the aforementioned interfaces, the plugin features five main modules:

Deployer: Exposes the command-line interface, validating requested commands execution and received input by the user. Additionally, manages the execution of the existing commands, managing exceptions, errors and user feedback;

URLs Annotator: Exposes the programmatic interface to annotate which resources, identified by their URLs, should run in either PaaS or FaaS. Crawls through and explores which URLs are available in the Django project;

Configs: Manages Django settings and configurations, validating modules and settings such as environment variables. Moreover, manages the creation and loading of configuration files generated by the plugin;

Provider: Set of Provider classes, which handle direct communication with the cloud providers (*e.g.*, resources deployment and destruction). Features a *Provider* interface class that specifies the functionality set that each provider (*e.g.*, *HerokuProvider* and *AzureProvider*) should implement;

Command Runner: Executes system command and invokes external tools (*e.g.*, cloud providers' CLIs).

As for the **Provider** module, the *Provider* interface class was conceived with the intent of both normalising and specifying the set of functionalities that each cloud provider, either PaaS or FaaS, should implement in order for it to be possible for the plugin to execute the existing commands. With this object-oriented architecture, new providers can be added to the plugin with minimal development effort, thus making it simple for the plugin to be extended. To support the cloud providers used in this work, the *AzureProvider* (which facilitates managing the Azure provider) and *HerokuProvider* (which facilitates managing the Heroku provider) classes were created, implementing the aforementioned *Provider* interface. Figure 6.3 illustrates how these classes associate with one another, as well as the methods that provider implementations should offer.

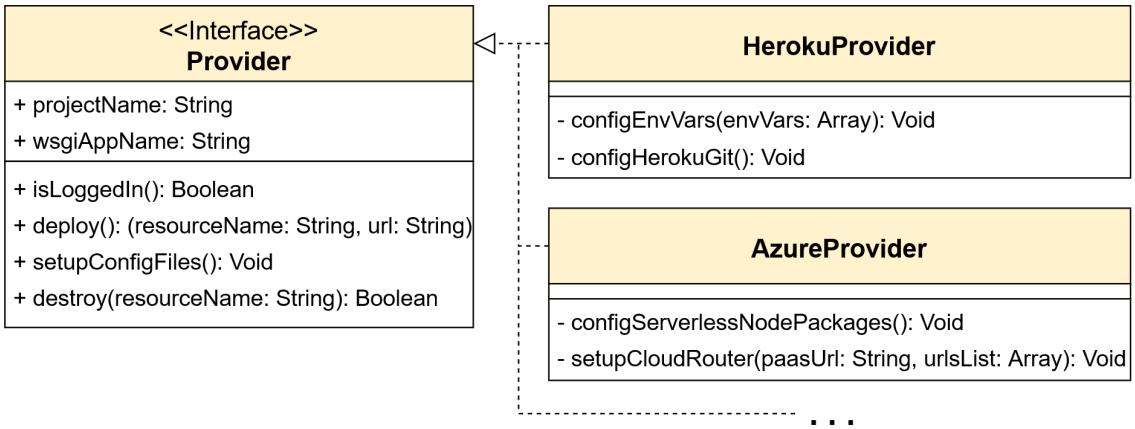

Figure 6.3: Django Cloud Deployer — Provider UML Class Diagram. The *Provider* interface specifies the required cloud provider functionalities, and the *HerokuProvider* and *AzureProvider* classes implement this specification.

6.1.3 Implemented Commands

The plugin features three commands: (a) the *deploy* command, (b) the *destroy* command, and (c) the *check_deploy* command.

The *deploy* command is used to, after annotating the Django project resource URLs with their target service model layer (PaaS or FaaS), deploy the project. This command takes two arguments, which are (a) the PaaS provider and (b) the FaaS provider, that must be available on the supported providers list. Moreover, this command expects the user to be logged in to both

the chosen providers (using their command-line interfaces) — if not logged in, the user will be requested to log in beforehand.

After verifying if the Django project is correctly configured (all setting required settings are defined), the plugin setups the Django project. This allows programmatically accessing Django's URL resolver, and to verify which resources were configured to be deployed in PaaS or FaaS. Then, both providers configuration files are created, based on template configuration files with placeholder values (the Mustache template rendering system was used for this purpose). Finally, the Django project is deployed to both PaaS and FaaS, and the deployment details are persisted in a configuration file (*cf.* Subsection 6.1.4, p. 64). This process is thoroughly illustrated in Figure 6.4.

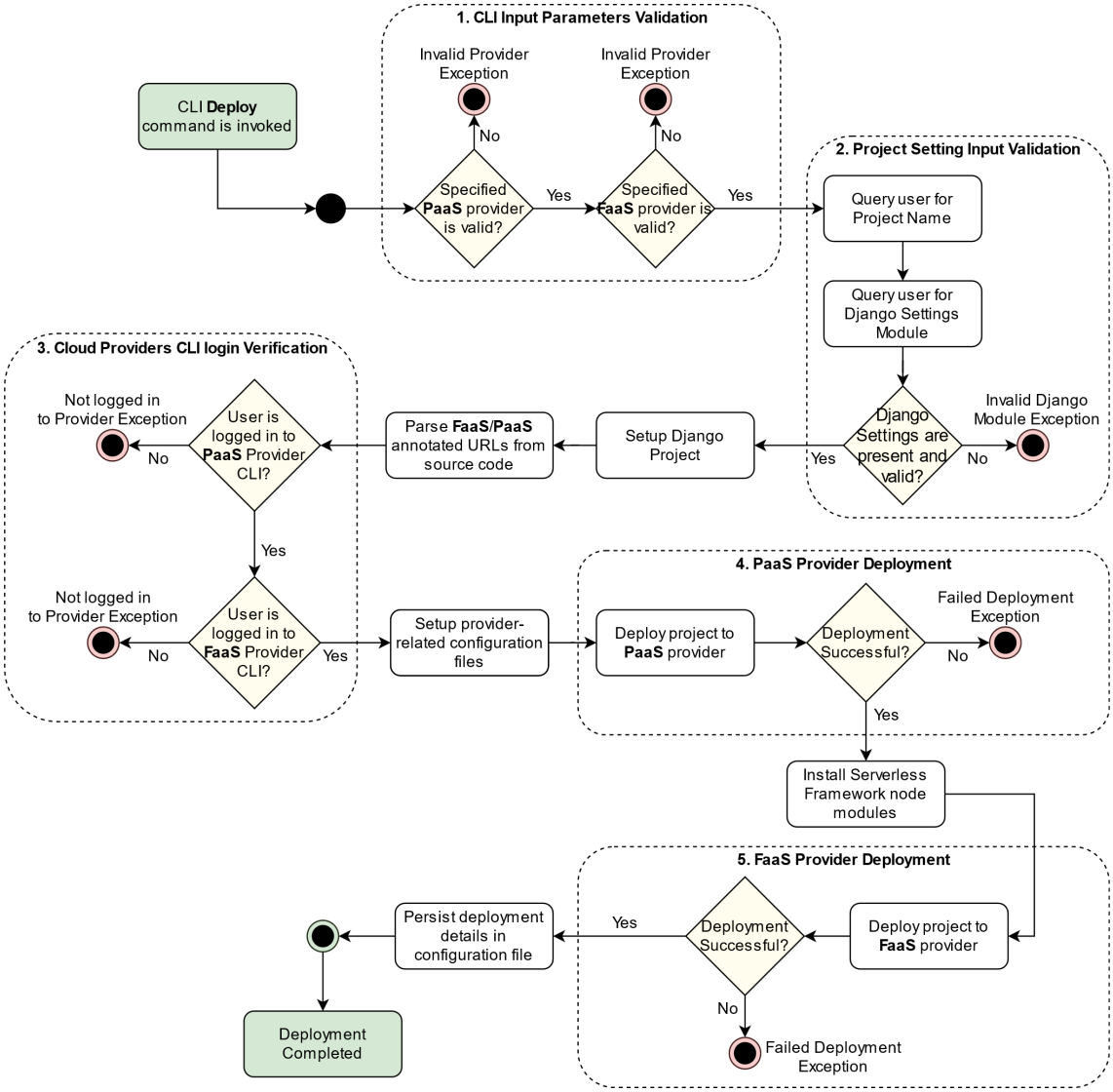

Figure 6.4: Django Cloud Deployer — Deploy Command UML Activity Diagram. The commands executes five main activities before a deployment is completed.

The *destroy* command is used to delete all infrastructure and configurations that were created in a previous deployment using the plugin. Similarly to the *deploy* command, it expects the user to be logged in to both the chosen providers (using their command-line interfaces) — if not logged in, the user will be requested to log in beforehand.

In this command, the plugin simply parses the configuration file (*cf.* Subsection 6.1.4) that was created on a previous plugin deployment, and deletes the deployed resources using the providers' command-line interface tools, as depicted in Figure 6.5.

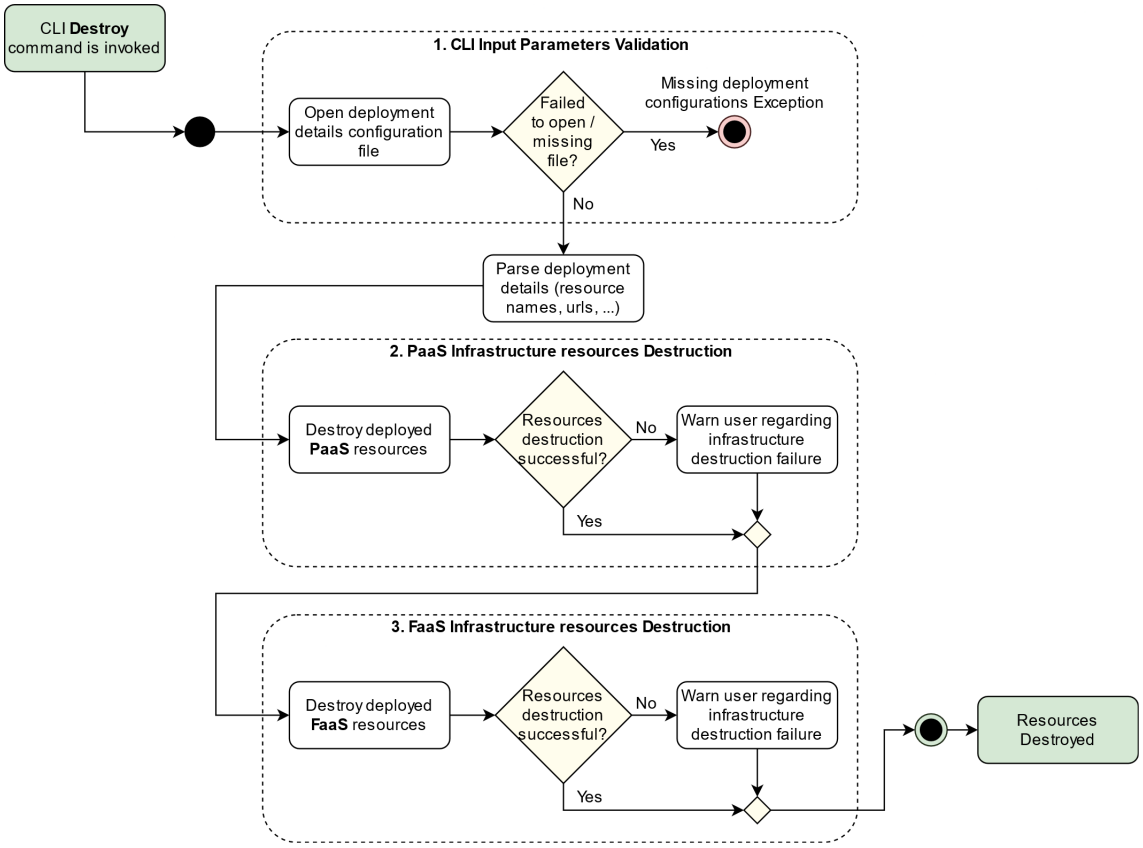

Figure 6.5: Django Cloud Deployer — Destroy Command UML Activity Diagram. The commands executes three main activities before the deployed resources are deleted.

Finally, the *check_deploy* command works as a *"dry run"* of the *deploy* command by setting up the Django project, accessing Django's URL resolver, and reporting to the user which resource URLs were annotated to be deployed and executed in PaaS or FaaS. This process is illustrated in the steps that precede the *"3. Cloud Providers CLI login verification"* activity in Figure 6.4 (p. 63).

6.1.4 Configuration Management

After the plugin successfully completes a deployment, a set of configuration variables are persisted in a JSON (JavaScript object notation) configuration file:

- The name of the project;

- The PaaS provider name, unique resource identifier, and the deployed resource URL;
- The FaaS provider name, unique resource identifier, and the deployed resource URL.

The configuration file should be stored, and can be used to delete the created cloud resources at a later stage by using their unique resource identifier.

6.1.5 Annotation Methodology

Although there are a multitude of ways that resource URLs could be annotated to signal that they should be deployed and executed in PaaS or FaaS, two methodologies were selected for implementation: (a) using source code line comments, and (b) update the source code itself using python wrapper functions (instead of using metadata annotations).

In the first implementation stage, source code line comments were used. This methodology allows a developer to add a single line comment on the line of the resource URL, indicating whether it should be deployed and executed in PaaS or FaaS (using the *cloud.paas* or *cloud.faas* comment strings, respectively), as illustrated in Listing 8.

```
urlpatterns = [
    path('', views.index, name='index'), # cloud.faas
    path('<int:pk>/', views.detail, name='detail'),
    path('info/', include('info.urls')), # cloud.paas
]
```

Listing 8: Django Cloud Deployer — Resource URLs annotations with Comments. The first and third URL path definitions are annotated with a comment that specifies whether the resource should be deployed and executed in FaaS or PaaS, respectively.

With this annotation methodology, the various Django URLs files were statically analysed, and the annotated URLs were extracted using regular expression matching. However, given that these URL patterns could be declared in a multitude of ways, it is cumbersome to ensure that the regular expression matching would work in any given case.

```
urlpatterns = [
    runInFaaS(path('', views.index, name='index')),
    path('<int:pk>/', views.detail, name='detail'),
    runInPaaS(path('info/', include('info.urls'))),
]
```

Listing 9: Django Cloud Deployer — Resource URLs target indication using Wrapper Functions. The first and third URL path definitions are wrapped with a function that specifies whether the resource should be deployed and executed in FaaS or PaaS, respectively.

Thus, in the second implementation stage, the use of python wrapper functions on the declared URLs was explored, as depicted in Listing 9 (p. 65). These wrapper functions, *runInPaaS*() and *runInFaaS*() — which indicate whether the resource should be deployed and executed in PaaS or FaaS, simply store the wrapped targets when the Django project is setup. Then, by accessing the Django's URL resolver, it is possible to traverse the full URLs tree (given that a URL entry can link to another list of various URL entries) without the need to identify and statically analyse all URL source code files. Although this technique does not use metadata annotations to indicate the deployment targets, it achieves the same goal through source code.

It is worth mentioning that in both of the explored methodologies, the lack of a comment annotation or wrapper function indicates that the resource should be deployed and executed in FaaS.

6.2 Evaluation and Validation

The experimentation and validation of the implemented plugin was divided into two distinct phases.

6.2.1 Plugin Experimentation

The plugin was experimented through the deployment of a simple test Django project and two Django open-source projects:

MeetHub: An event management system open-source project[2];

Spirit: A forum open-source project[3];

Django Polls App (DPA): A simple project developed for testing purposes[4].

Each of the projects was analysed and partitioned between PaaS and FaaS, using the plugin's programmatic interface to indicate which resources should be deployed and executed in either service model. The partitioning process was made in an *ad hoc* manner, in a way that seemed to produce a logical separation of resources, and is detailed in Appendix D (p. 83).

The resulting deployments of each project can be accessed in each of the subdomains of the *thesis.ruialves.me* domain, as visible in Table 6.1 (p. 67).

The partitioning, configuration and deployment process of each of the aforementioned processes is thoroughly explained and described in this work's replication package [4].

[2]MeetHub + Django Cloud Deployer, available on GitHub: https://github.com/xRuiAlves/meethub
[3]Spirit + Django Cloud Deployer, available on GitHub: https://github.com/xRuiAlves/Spirit
[4]Django Polls App + Django Cloud Deployer, available on GitHub: https://github.com/xRuiAlves/dpa

Table 6.1: Web Applications deployed using the implemented Plugin. Each deployed project features a subdomain, together with its own specific FaaS and PaaS subdomains.

Project	Deployed Application	FaaS part	PaaS part
MeetHub	mhub.thesis.ruialves.me	faas.mhub.thesis.ruialves.me	paas.mhub.thesis.ruialves.me
Spirit	spirit.thesis.ruialves.me	faas.spirit.thesis.ruialves.me	paas.spirit.thesis.ruialves.me
DPA	dpa.thesis.ruialves.me	faas.dpa.thesis.ruialves.me	paas.dpa.thesis.ruialves.me

6.2.2 Plugin Validation

A plugin's resulting deployment is deemed as valid and successful if (a) every resource was correctly deployed to its specified service model (PaaS or FaaS), and (b) if traffic is being properly routed and requests are being executed in its specified service model.

Thus, to validate if the plugin operated successfully in each of the aforementioned deployments, every resource URL of each project was iteratively accessed, verifying if the requests were routed to the expected PaaS or FaaS targets. This validation is automated by a script that receives the testing rules through a configuration file, as visible in Listing 10.

```
{
    "domain": "basedomain.example.org",
    "paasDomain": "paas.basedomain.example.org",
    "faasDomain": "faas.basedomain.example.org",
    "paasTargetUrls": [
        "sample/paas/url/1",
        "sample/paas/url/2",
        "sample/paas/url/3"
    ],
    "faasTargetUrls": [
        "sample/faas/url/1",
        "sample/faas/url/2",
        "sample/faas/url/3"
    ]
}
```

Listing 10: Plugin validation script input testing rules JSON configuration. It features a list of URLs that should be redirect to PaaS, and a list of URLs that should be redirect to FaaS.

Then, for each of the specified target URLs, a set of requests is made to the base domain, appended with the resource URL. If the request is redirected to the expected resource, the test case is deemed as successful (and vice-versa), as visible in Listing 11 (p. 68).

```
{
    "domain": "basedomain.example.org",
    "paasDomain": "paas.basedomain.example.org",
    "faasDomain": "faas.basedomain.example.org",
    "successRate": "98.50%",
    "results": [
        {
            "url": "http://basedomain.example.org/url/1/",
            "expectedDomain": "faas.basedomain.example.org",
            "obtainedDomain": "faas.basedomain.example.org",
            "success": true
        },
        {
            "url": "http://basedomain.example.org/url/2/",
            "expectedDomain": "faas.basedomain.example.org",
            "obtainedDomain": "paas.basedomain.example.org",
            "success": false
        },
        ...
    ]
}
```


Listing 11: Plugin validation script output results JSON report. Each result entry features a *success* status, that represents whether the *obtainedDomain* matches the *expectedDomain* for that tested URL. The rate of successful tests results in the overall *successRate*.

The results showed a 100% success rate in all three deployments, and the scripts used to automate these tests and the full results data, as well as a comprehensive guide on how to replicate this process, are available in this work's replication package [4].

Finally, the results obtained in this phase demonstrate the first premise of our hypothesis. We show that the implemented plugin allows the partitioning of a web application to be executed and deployed in PaaS or FaaS, thus allowing the optimisation of the web application without significant development effort or infrastructure manual configuration.

6.2.3 Threats to Validity

The infrastructure-related constraints and limitations that were present in this validation process pose a set of external (if there is threat that the results can't be generalised the scope of this work) and internal (if there is threat that the results are caused by internal treatment anomalies) threats to validity [37]:

The number of experimented projects may influence the results (external threat)
The plugin's experimentation was limited to a small number of Django projects (only three). Although this consists of a qualitative evaluation process that does not necessarily require a large number of samples, it is not clear if the plugin would achieve valid results in projects of a different nature, featuring other software versions or distinct dependencies.

The selected application partitioning may influence the results (internal threat)
Although the way that each project was partitioned does not directly influence the validity of the resulting deployments, it is worth pointing out that, since their partitioning was made in an *ad hoc* manner that did not include the thorough analysis of the project, the chosen partitioning could influence the achieved results.

6.2.4 Replication Package

To allow the full replication of the experimentation and validation steps presented in this chapter, as well as detailed instructions on how to replicate them, an experiments replication package [4] was built, featuring the following content:

Django Cloud Deployer plugin: Source code, usage instructions and its documentation;

Plugin-deployed open-source Django projects: Source code, a comprehensive guide on how to replicate the deployments, the automation scripts used to validate the deployments, and their results reports.

6.2.5 Desiderata Revisited

The implemented plugin achieves most of the *desiderata* proposed in Section 4.3 (p. 34), providing an interface to annotate where resources are deployed and executed (**D1**), deploying the annotated web application resources to the cloud (**D2**), having commands to manage the deployed web application cloud resources (**D3**), and featuring capabilities to traffic between PaaS and FaaS (**D4**).

However, due to limited time constraints, the plugin does not feature automatic reconfiguration of the cloud resources, thus not achieving the last identified *desiderata* **D5**. That being said, this improvement has been identified as future work in Section 7.4 (p. 74).

6.3 Summary

This chapter describes the implementation of a plugin, which consists of a python package that facilitates the deployment of Django web applications to both PaaS and FaaS simultaneously. After deployed, the *RT3* routing technique (*cf.* Chapter 5, p. 39) is used to route traffic between PaaS and FaaS.

The plugin was developed with a flexible architecture to allow its extension to multiple cloud providers with minimal development effort. This plugin offers both a routing annotations API and a CLI interface to manage the Django application cloud deployment.

The routing annotations API allows developers to indicate through the use of two functions — *runInPaaS* and *runInFaaS* — which resource should be deployed and executed in PaaS or FaaS, respectively. As for the CLI interface, it features a set of commands (*deploy*, *destroy*, and *check_deploy*) that can be used to manage plugin deploys. In this work, support for the Heroku and Azure cloud providers was implemented.

Firstly, with regards to the plugin's experimentation and validation, the plugin was used to deploy a test project and two open-source Django projects (MeetHub and Spirit). In each of these deployments, the plugin was also used to partition the application between FaaS and PaaS, as detailed in Appendix D (p. 83). Secondly, to validate the success of the plugin deployments, every resource URL of each project was iteratively accessed, verifying if the requests were routed to the expected PaaS or FaaS targets — the results showed a 100% success rate in all three deployments. Thus, in this chapter we demonstrate the first premise of our hypothesis.

Finally, the results, automation scripts, source code, as well as replication and usage guides were comprised in a replication package [4] for future usage. Additionally, a comprehensive guide on how to install, set up, and use the plugin can also be found in the replication package.

On the following chapter we describe the key conclusions and final remarks of the work developed in this dissertation.

Chapter 7

Conclusions

On the previous chapter we discuss the implementation and validation of the reference plugin prototype. This chapters describes the key conclusions and final remarks of the work developed in this dissertation (*cf.* Section 7.1). The contributions of this work are described in Section 7.2. Then, the main difficulties that arose during the course of this process are listed in Section 7.3. Finally, Section 7.4 identifies the possible opportunities for future work.

7.1 Conclusions

Cloud Computing democratised access to large scale computation resources, offering developers an on-demand solution to deploy their applications at vast scales (often across multiple regions), with guaranteed flexibility and Quality of Service (QoS). It provides different types of service models, such as PaaS and FaaS, each with its own set of advantages and drawbacks.

The ability of leverage the complementary benefits of each of these models leads to the optimisation of web applications and their deployment. However, these types of deployments that feature multiple service model layers (hybrid deployments) are not popular, since they feature a multitude of challenges.

The literature review showed that these challenges lie in managing deployed systems' observability, monitoring, reconfiguration, distribution decision, and others. Additionally, there are shortcomings in allowing developers to partition their applications without significant development effort, and there is a lack of investigation on using dynamic strategies to route traffic within these hybrid deployments.

In this dissertation, we address these shortcomings by conceiving and implementing a set of techniques for dynamic routing between PaaS and FaaS, and validating them with a reference implementation of a plugin prototype that facilitates the programmatic partitioning of web applications into PaaS and FaaS and their deployment to the cloud. This implemented work, validated through its experimentation on open-source web projects, aims at demonstrating our hypothesis:

> *PaaS and FaaS provide complementary benefits for web applications. Developers can leverage them to optimise their applications, by (1) partitioning their applications to be deployed to the best service model for each functionality, (2) using a dynamic routing strategy to forward requests to the proper resource, and (3) allow the application to dynamically reconfigure itself by moving code between PaaS and FaaS (and vice-versa) in a way that optimises infrastructure costs, client latency, and response time.*

The implemented plugin shows that the source code level annotations, through the use of wrapper functions, allow the partitioning of the web application between PaaS and FaaS in a programmatic and effortless way (answering **RQ1**), while also allowing its subsequent deployment to the cloud in a completely provider-agnostic manner (answering **RQ2**), thus demonstrating the first premise of the hypothesis.

Additionally, the evaluation process of the conceived routing techniques showed that moving the routing service to the FaaS layer results in the optimal configuration, due to this model's wider geographical distribution (answering **RQ3**), minimising client latency when accessing the hybrid application, and thus demonstrating the second premise of the hypothesis.

The plugin implementation also fulfills the identified *desiderata* from **D1** to **D4** (*cf.* Section 4.3, p. 34).

However, due to research and development time constraints, we did not demonstrate the last premise of our hypothesis, which is related to the application's optimisation through dynamic reconfiguration based on runtime self-inspection. Moreover, although the developed work does not fully answer research questions **RQ4** and **RQ5**, the literature review we present in Chapter 3 (p. 15) provides a good basis towards their response. This shortcoming has also been identified as a possibility for future work (*cf.* Section 7.4, p. 74).

Finally, we believe that *Django Cloud Deployer* is a valuable contribution towards practical hybrid deployments, since it addresses a multitude of open problems on this subject. Although the automation of dynamic system reconfiguration was not addressed, the literature review we presented, together with the various implementation aspects of this work, is considered to be a strong basis towards this step.

7.2 Contributions

The work developed in this dissertation resulted in the following contributions to the software engineering state of the art, and the web application development communities:

Pseudo-systematic Literature Review on Cloud Computing and FaaS Leveraging: An investigation was performed to the state of the art of cloud computing, applications refactoring and partitioning, FaaS leveraging, and dynamic routing in the cloud (*cf.* Chapter 3, p. 15);

Investigation on Dynamic Routing Strategies: We conceived a set of strategies for dynamic traffic routing between PaaS and FaaS, and presented the achieved conclusions and results from their validation and evaluation, together with comprehensible deployment and replication guidance.

Reference Implementation: We implemented *Django Cloud Deployer*, a plugin that facilitates the programmatic partitioning of web applications into PaaS and FaaS and their deployment to the cloud, which was published on the Pypi python package public repository[1];

Publishable Contribution: The parts of this dissertation that focused on the techniques for dynamic routing between PaaS and FaaS, together with the plugin's reference implementation, were synthesised into a paper, that will be submitted to a domain-related conference in the near future.

7.3 Challenges

Throughout this work, we came across a set of roadblocks that made progress more challenging and, although some were overcome or mitigated, others could not be alleviated or solved, and were identified as future work.

To begin with, there was a lack of cloud provider resources. In the beginning of this work, multiple attempts were made to receive sponsorship from AWS and GCP — however, these attempts proved to be unsuccessful. Although AWS, which is the leader when it comes to the documentation of the offered services, offers a free student subscription, it proved to be quite limited and did not allow any kind of programmatic management of resources. As a result, we used Azure in the course of this work, given that it was the only market-leading provider that offered an amount of credits that could suffice the infrastructure needs.

However, Azure proved to have quite a lot of limitations. Even though the web management dashboard lacked documentation resources and was not intuitive to navigate, most of the challenges arose from the lack of programmatic support of basic functionalities such as DNS and network interface management.

Finally, even though it didn't have a significantly negative impact on the progress of this work, the Serverless framework lacked official documentation for integration with Azure, which led to various steps of its utilisation relying on *"trial and error"* techniques.

[1] Django Cloud Deployer package, available on Pypi: https://pypi.org/project/django-cloud-deployer/

7.4 Future Work

In the routing techniques presented in this work, we used temporary redirects (*"302 Found"* and *"307 Temporary Redirect"*). However, the exploration of permanent redirects (*"301 Moved Permanently"* and *"308 Permanent Redirect"*), which aim to reduce request latency, was not a part of this work. In permanent redirects, since requests subsequent to the first one are accessed directly without visiting the original resource, this period of time is saved in future requests. Even though they pose challenges when the redirect resource location changes, a problem known as *"Cache Invalidation"*, which is cumbersome in scenarios where a system needs to reconfigure itself, there are known strategies that may be used to reset permanent redirects [81]. Thus, this could be a path towards optimising the routing techniques presented in this work.

It would be of interest to expand the implemented plugin with the support of more providers — Although the available infrastructure constraints led to the plugin supporting only Heroku PaaS and Azure FaaS, supporting more providers would be helpful for the Django web development community.

The lack of access to performance metrics and detailed billing costs (which resulted from the limitations of the available Azure student subscription) limited the metrics used for the evaluation process of the studied routing techniques - repeating the experiments with access to these performance metrics

would provide valuable insights that could weigh on the evaluation of each of the considered routing techniques.

Finally, provided that access to the previously mentioned metrics is available, it would prove interesting to make use of real-time performance analysis to automate moving parts of a deployed web application from PaaS to FaaS, with the aim of optimising the deployment — either in terms of cost-efficiency, client latency, or others.

Appendix A

PaaS Machine Number of Users Threshold Discovery Results

Table A.1: PaaS machine number of users threshold discovery — Average response time results from 5 experiment trials.

Users	Reqs.	Fails	F. Rate	Response Time (ms)					
				Avg	Min	Q1 25%	Med	Q3 - 75%	Max
100	9893	0	0.00%	258	116	120	180	220	5602
200	16116	0	0.00%	597	119	130	350	580	7086
300	12076	0	0.00%	2347	157	300	1800	2700	10002
400	16518	0	0.00%	2388	452	900	2000	2600	13918
500	15767	0	0.00%	3428	546	2100	3000	3600	13349
600	15513	0	0.00%	4489	1454	1700	3900	5000	10594
700	15512	0	0.00%	5472	2188	2200	4800	6900	10886
800	20790	0	0.00%	4483	2064	3000	4000	4500	12883
900	21807	0	0.00%	4890	2472	3400	4600	5400	7801
1000	23769	0	0.00%	4944	2917	3900	4700	5200	9440
1100	21805	0	0.00%	6297	3724	5700	6300	6700	11937
1200	46313	0	0.00%	2566	1152	1900	2500	2800	5601
1300	19622	4	0.02%	8550	5035	5800	8400	9700	15555
1400	21462	2	0.01%	8460	5419	5900	7800	8800	15765
1500	46595	0	0.00%	3425	2072	2800	3300	3600	5964
1600	22881	0	0.00%	9084	6004	7800	8900	9600	13753
1700	49909	0	0.00%	3714	2240	3100	3600	3900	5927
1800	48179	1	0.00%	4224	65	2500	3800	4500	7767
1900	21419	0	0.00%	12117	8408	8000	11000	13000	19486
2000	21305	0	0.00%	12544	8641	10000	12000	13000	18027

2100	21717	2851	13.13%	13135	57	11000	13000	14000	22320
2200	22210	2485	11.19%	13329	65	12500	13000	14000	30272
2300	23720	4840	20.40%	13093	60	11000	13000	14000	27709
2400	23117	6664	28.83%	13658	55	12000	13000	16000	37701

Appendix B

Routing Techniques Response Time Results

B.1 Response Time including both PaaS and FaaS results

Table B.1: Routing Techniques — Average response time results from 5 experiment trials.

RT	Users	Reqs.	Fails	F. Rate	Response Time (ms)					
					Avg	Min	Q1 25%	Med	Q3 - 75%	Max
RT1	500	60144	2	0.00%	430	113	115	230	590	17095
	1000	121957	1	0.00%	399	119	120	340	600	10152
	1500	180849	2	0.00%	425	124	120	370	620	10188
	2000	205520	2	0.00%	696	128	430	600	770	10810
RT2	500	64861	0	0.00%	310	40	80	310	540	2365
	1000	124231	2	0.00%	374	42	210	380	550	10950
	1500	182768	0	0.00%	404	45	250	410	570	7667
	2000	223221	29	0.01%	527	44	410	540	670	10941
RT3	500	65310	0	0.00%	299	25	110	270	430	10957
	1000	123445	1	0.00%	384	31	190	360	530	10482
	1500	181111	9	0.00%	425	35	240	410	580	10925
	2000	236684	0	0.00%	439	40	310	440	570	9784
RT4	500	63602	6	0.01%	330	9	40	230	420	1994
	1000	126470	9	0.01%	343	48	200	240	440	4962
	1500	157035	390	0.25%	683	27	320	600	880	4114
	2000	189742	39691	20.92%	874	26	660	880	1100	6144

B.2 Response Time including PaaS results only

Table B.2: Routing Techniques — PaaS Average response time results from 5 experiment trials.

RT	Users	Reqs.	Fails	F. Rate	Response Time (ms)					
					Avg	Min	Q1 25%	Med	Q3 - 75%	Max
RT1	500	30284	0	0.00%	338	126	150	160	590	1970
	1000	61234	0	0.00%	341	127	160	170	590	5668
	1500	90297	0	0.00%	360	127	210	230	590	2458
	2000	102009	0	0.00%	486	128	400	500	640	7271
RT2	500	32394	0	0.00%	291	50	60	240	540	2225
	1000	61957	0	0.00%	299	52	60	250	540	2292
	1500	91002	0	0.00%	331	53	30	290	550	4547
	2000	111608	13	0.02%	465	54	320	460	600	10941
RT3	500	32688	0	0.00%	334	53	20	270	550	10957
	1000	61294	0	0.00%	390	54	140	360	580	6277
	1500	90420	4	0.00%	422	59	230	420	610	10003
	2000	118081	0	0.00%	455	64	300	460	620	8139
RT4	500	31536	4	0.01%	331	9	40	230	420	1994
	1000	63238	2	0.00%	342	56	100	240	440	2271
	1500	78255	171	0.22%	683	27	320	600	880	2841
	2000	95176	20026	21.04%	871	26	640	870	1100	6144

B.3 Response Time including FaaS results only

Table B.3: Routing Techniques — FaaS Average response time results from 5 experiment trials.

RT	Users	Reqs.	Fails	F. Rate	Response Time (ms)					
					Avg	Min	Q1 25%	Med	Q3 - 75%	Max
RT1	500	29860	2	0.01%	523	113	130	240	600	17095
	1000	60723	1	0.00%	458	119	180	420	660	10152
	1500	90552	2	0.00%	489	124	210	450	690	10188
	2000	103511	2	0.00%	903	130	510	720	930	10810
RT2	500	32467	0	0.00%	328	40	120	330	540	2365
	1000	62274	2	0.00%	449	42	260	440	620	10950
	1500	91766	0	0.00%	477	45	300	480	660	7667
	2000	111613	16	0.02%	590	44	490	610	730	10188
RT3	500	32622	0	0.00%	265	25	130	260	390	9798
	1000	62151	1	0.00%	379	31	240	360	480	10482
	1500	90691	5	0.01%	428	35	270	400	530	10925
	2000	118603	0	0.00%	422	40	320	420	520	9784
RT4	500	32066	2	0.01%	328	45	50	230	420	1816
	1000	63232	7	0.01%	344	48	50	240	440	4962
	1500	78780	219	0.28%	682	28	320	600	880	4114
	2000	94566	19665	20.80%	876	26	660	880	1100	4818

Appendix C

Literature Review Process Exemplification

This appendix intends to provide an exemplification of the iterative process used for selecting literature of interest for this work. The exemplification includes the iterative process steps for a set of queries that were conceived in regards to **LRQ2 — How do applications refactor themselves at runtime?**

Table C.1: Literature Review Process Exemplification — Results for the query set 1 for SQR2.

Iteration	Query	Results	Captured Keywords
1	system dynamic refactoring	[43, 83, 73]	runtime
2	system (dynamic OR runtime) refactoring	[43, 83, 73, 107]	reformulation, reconfiguration
3	system (dynamic OR runtime) (refactoring OR reformulation OR reconfiguration)	[43, 83, 73, 107, 41, 66]	...

Table C.2: Literature Review Process Exemplification — Results for the query set 2 for SQR2.

Iteration	Query	Results	Captured Keywords
1	system self-inspection	[89, 100]	monitoring, application
2	(system OR application) (monitoring OR self-inspection)	[89, 100, 102]	dynamic analysis, introspection
3	(system OR application) (monitoring OR self-inspection OR introspection OR dynamic analysis)	[89, 100, 102, 78, 69]	...

It is worth pointing out that, depending on the research databases used, additional iterations and specific fine-tuning were added in each query set.

Appendix D

Test Web Applications Partitioning

This appendix intends to detail how each of the Django web applications used for the plugin's experimentation (*cf.* Section 6.2, p. 66) was partitioned for deployment.

D.1 Spirit

Partitioned for deployment in PaaS:

- $user/*$
- $topic/*$
- $comment/*$

Partitioned for deployment in FaaS:

- $admin/*$
- $category/*$
- $search/*$

D.2 MeetHub

Partitioned for deployment in PaaS:

- $comments/*$
- $event/*$
- $password/*$

Partitioned for deployment in FaaS:

- $admin/*$
- $auth/*$
- $category/*$
- $notifications/*$
- $accounts/*$
- $filebrowser/*$

D.3 Django Polls App

Partitioned for deployment in PaaS:

- $polls/ : id/results/$
- $info/contacts/*$

Partitioned for deployment in FaaS:

- $info/help/*$
- $polls/*$

References

[1] Gojko Adzic and Robert Chatley. Serverless computing: Economic and architectural impact. ESEC/FSE 2017, page 884–889, New York, NY, USA, 2017. Association for Computing Machinery.

[2] Lucas F Albuquerque Jr, Felipe Silva Ferraz, RF Oliveira, and SM Galdino. Function-as-a-service x platform-as-a-service: Towards a comparative study on faas and paas. In *ICSEA*, pages 206–212, 2017.

[3] Lucas F Albuquerque Jr, Felipe Silva Ferraz, RF Oliveira, and SM Galdino. Function-as-a-service x platform-as-a-service: Towards a comparative study on faas and paas. In *ICSEA*, pages 206–212, 2017.

[4] Rui Alves. Cloud routing tests - experiments validation package. Available at https://doi.org/10.5281/zenodo.5021443, 2021.

[5] L Arockiam, S Monikandan, and G Parthasarathy. Cloud computing: a survey. *International Journal of Internet Computing*, 1(2):26–33, 2011.

[6] R. N. Avula and C. Zou. Performance evaluation of tpc-c benchmark on various cloud providers. In *2020 11th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, pages 0226–0233, 2020.

[7] AWS. Amazon Elastic Block Store EBS). Available at https://aws.amazon.com/ebs/. Accessed in January 2021.

[8] AWS. Amazon Web Services EC2. Available at https://aws.amazon.com/ec2/. Accessed in January 2021.

[9] AWS. Amazon Web Services Elastic Beanstalk. Available at https://aws.amazon.com/elasticbeanstalk/. Accessed in January 2021.

[10] AWS. Amazon Web Services Lambda, Serverless Compute. Available at https://aws.amazon.com/lambda/. Accessed in January 2021.

[11] AWS. AWS CLI: Tools for working with Lambda - AWS Lambda. Available at https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-tools.html. Accessed in January 2021.

[12] AWS. AWS Lambda - FAQs. Available at https://aws.amazon.com/lambda/faqs/. Accessed in February 2021.

[13] AWS. Serverless Web App, Amazon Web Services. Available at https://aws.amazon.com/serverless/build-a-web-app/. Accessed in January 2021.

[14] Azure. App Service, Microsoft Azure. Available at https://azure.microsoft.com/en-us/services/app-service/. Accessed in January 2021.

[15] Azure. Azure Functions Serverless Compute, Microsoft Azure. Available at https://azure.microsoft.com/en-us/services/functions/. Accessed in January 2021.

[16] Azure. Azure Infrastructure as a Service (IaaS), Microsoft Azure. Available at https://azure.microsoft.com/en-us/overview/what-is-azure/iaas/. Accessed in January 2021.

[17] Azure. Supported languages in Azure Functions | Microsoft Docs. Available at https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages. Accessed in February 2021.

[18] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: An experience report. In Antonio Celesti and Philipp Leitner, editors, *Advances in Service-Oriented and Cloud Computing*, pages 201–215, Cham, 2016. Springer International Publishing.

[19] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. Springer Singapore, Singapore, 2017.

[20] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[21] R Barga. Serverless computing: Redefining the cloud. In *First International Workshop on Serverless Computing (WoSC'2017)*, volume 5, 2017.

[22] Daniel G Bobrow, Richard P Gabriel, and Jon L White. Clos in context-the shape of the design space. *Object Oriented Programming: The CLOS Perspective*, pages 29–61, 1993.

[23] M. U. Bokhari, Q. M. Shallal, and Y. K. Tamandani. Cloud computing service models: A comparative study. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 890–895, March 2016.

[24] Diogo Bortolini and Rafael R Obelheiro. Investigating performance and cost in function-as-a-service platforms. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 174–185. Springer, 2019.

[25] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[26] Walter Cazzola. Evaluation of object-oriented reflective models. In *European Conference on Object-Oriented Programming*, pages 386–387. Springer, 1998.

[27] Vasian Cepa. *Attribute Enabled Software Development*. VDM Verlag, 2007.

[28] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[29] Rui Chen, Shanshan Li, and Zheng Li. From monolith to microservices: a dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475. IEEE, 2017.

[30] Matt Cooper. *azf-wsgi — WSGI apps on Azure Functions*, 2020. Package version 0.3.1.

[31] Diego AA Correia, Eduardo M Guerra, Fábio F Silveira, and Clovis T Fernandes. Quality improvement in annotated code. *CLEI Electron. J*, 13(2), 2010.

[32] Matt Crane and Jimmy Lin. An exploration of serverless architectures for information retrieval. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*, ICTIR '17, page 241–244, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Tiago Boldt Pereira de Sousa. *Engineering Software for the Cloud: A Pattern Language*. PhD thesis, Faculty of Engineering, University of Porto, 2020.

[34] Danny Dig. A refactoring approach to parallelism. *IEEE software*, 28(1):17–22, 2010.

[35] T. Dillon, C. Wu, and E. Chang. Cloud computing: Issues and challenges. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 27–33, 2010.

[36] Dropbox. Dropbox - File Hosting. Available at https://www.dropbox.com/. Accessed in January 2021.

[37] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research-an initial survey. In *Seke*, pages 374–379, 2010.

[38] Hugo José Sereno Lopes Ferreira. *Adaptive object-modeling: patterns, tools and applications*. PhD thesis, Faculty of Engineering, University of Porto, 2010.

[39] Hugo Sereno Ferreira, André Restivo, and Tiago Boldt Sousa. Towards a pattern language for the masters student. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, pages 1–12, 2019.

[40] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Rfc2616: Hypertext transfer protocol–http/1.1, 1999.

[41] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE software*, 23(2):62–70, 2006.

[42] Django Software Foundation. The Web framework for perfectionists with deadlines, Django. Available at https://www.djangoproject.com/. Accessed in January 2021.

[43] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[44] Geoffrey C Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028*, 2017.

[45] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[46] Alireza Goli, Omid Hajihassani, Hamzeh Khazaei, Omid Ardakanian, Moe Rashidi, and Tyler Dauphinee. Migrating from monolithic to serverless: A fintech case study. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 20–25, New York, NY, USA, 2020. Association for Computing Machinery.

[47] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong. The characteristics of cloud computing. In *2010 39th International Conference on Parallel Processing Workshops*, pages 275–279, Sep. 2010.

[48] Google. App Engine Application Platform, Google Cloud. Available at https://cloud.google.com/appengine/. Accessed in January 2021.

[49] Google. Cloud Functions, Google Cloud. Available at https://cloud.google.com/functions/. Accessed in January 2021.

[50] Google. Cloud Run: Container to production in seconds, Google Cloud. Available at https://cloud.google.com/run/. Accessed in January 2021.

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

[52] Google. Google Docs: Free Online Documents for Personal Use. Available at https://www.google.com/docs/about/. Accessed in January 2021.

[53] Google. Quickstart: Using the gcloud Command-Line Tool. Available at https://cloud.google.com/functions/docs/quickstart. Accessed in January 2021.

[54] Google. Writing Cloud Functions | Cloud Functions Documentation | Google Cloud. Available at https://cloud.google.com/functions/docs/writing. Accessed in February 2021.

[55] Eugene Gorelik. *Cloud computing models*. PhD thesis, Massachusetts Institute of Technology, 2013.

[56] Dan Gunter, Brian Tierney, Keith Jackson, Jason Lee, and Martin Stoufer. Dynamic monitoring of high-performance distributed applications. In *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, pages 163–170. IEEE, 2002.

[57] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *European Conference on Service-Oriented and Cloud Computing*, pages 185–200. Springer, 2016.

[58] Matthias Heinrich, Franz Josef Grüneberger, Thomas Springer, and Martin Gaedke. Exploiting annotations for the rapid development of collaborative web applications. In *Proceedings of the 22nd international conference on World Wide Web*, pages 551–560, 2013.

[59] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[60] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[61] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web*, pages 148–159, 2003.

[62] IBM. IBM Cloud Functions. Available at https://cloud.ibm.com/functions/. Accessed in February 2021.

[63] IBM. IBM Cloud Infrastructure as a Service. Available at https://www.ibm.com/cloud/infrastructure/. Accessed in January 2021.

[64] IBM. What is the IBM Cloud Platform? Available at https://cloud.ibm.com/docs/overview?topic=overview-whatis-platform. Accessed in January 2021.

[65] Intel. Annotations, Intel Advisor. Available at https://software.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/optimize-cpu-usage/threading-perspective/annotate-code-for-deeper-analysis/annotations.html. Accessed in January 2021.

[66] Florian Irmert, Thomas Fischer, and Klaus Meyer-Wegener. Runtime adaptation in a service-oriented component model. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 97–104, 2008.

[67] Aman Jain, Ata F Baarzi, George Kesidis, Bhuvan Urgaonkar, Nader Alfares, and Mahmut Kandemir. Splitserve: Efficiently splitting apache spark jobs across faas and iaas. In *Proceedings of the 21st International Middleware Conference*, pages 236–250, 2020.

[68] Pooyan Jamshidi, Claus Pahl, and Nabor C Mendonça. Pattern-based multi-cloud architecture migration. *Software: Practice and Experience*, 47(9):1159–1184, 2017.

[69] Jazzband. jazzband/django-silk: Silky smooth profiling for Django. Available at https://github.com/jazzband/django-silk. Accessed in January 2021.

[70] Gabor Kecskemeti, Attila Csaba Marosi, and Attila Kertesz. The entice approach to decompose monolithic services into microservices. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 591–596. IEEE, 2016.

[71] K. Kritikos and P. Skrzypek. A review of serverless frameworks. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168, Dec 2018.

[72] Manoj Kumar. Serverless architectures review, future trend and the solutions to open problems. *American Journal of Software Engineering*, 6(1):1–10, 2019.

[73] Ling Lan, Gang Huang, Weihu Wang, and Hong Mei. A middleware-based approach to model refactoring at runtime. In *14th Asia-Pacific Software Engineering Conference (APSEC'07)*, pages 246–253. IEEE, 2007.

[74] Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. A mixed-method empirical study of function-as-a-service software development in industrial practice. *Journal of Systems and Software*, 149:340 – 359, 2019.

[75] Sulav Malla and Ken Christensen. Hpc in the cloud: Performance comparison of function as a service (faas) vs infrastructure as a service (iaas). *Internet Technology Letters*, 3(1):e137, 2020.

[76] Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188. IEEE, 2018.

[77] Kyle Marvin, David Remy, David Bau, Roderick A Chavez, and David Read. Systems and methods for creating network-based software services using source code annotations, April 27 2010. US Patent 7,707,564.

[78] Tiago Matias, Filipe F. Correia, Jonas Fritzsch, Justus Bogner, Hugo S. Ferreira, and André Restivo. Determining microservice boundaries: A case study using static and dynamic software analysis. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann, editors, *Software Architecture*, pages 315–332, Cham, 2020. Springer International Publishing.

[79] Klaus Meffert. Supporting design patterns with annotations. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*, pages 8–pp. IEEE, 2006.

[80] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, USA, 2011.

[81] Peter J Meyers. Can you reverse a 301 redirect?, May 2019.

[82] Filipe Rui Rocha Oliveira. Exploring the scala macro system for compile time model-based generation of statically type-safe rest services. Master's thesis, Faculty of Engineering, University of Porto, 2015.

[83] William F Opdyke. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[84] OpenFaaS. OpenFaaS - Serverless Functions Made Simple. Available at https://www.openfaas.com/. Accessed in January 2021.

[85] Cüneyt M Özveren, Alan S Willsky, et al. Observability of discrete event dynamic systems. 1989.

[86] Dana Petcu. Multi-cloud: expectations and current approaches. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 1–6, 2013.

[87] Duarte Pinto, João Pedro Dias, and Hugo Sereno Ferreira. Dynamic allocation of serverless functions in iot environments. In *2018 IEEE 16th international conference on embedded and ubiquitous computing (EUC)*, pages 1–8. IEEE, 2018.

[88] F. Ponce, G. Márquez, and H. Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7, Nov 2019.

[89] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul Grünbacher. A comparison framework for runtime monitoring approaches. *Journal of Systems and Software*, 125:309–321, 2017.

[90] Derrick Rountree and Ileana Castrillo. *The Basics of Cloud Computing: Understanding the Fundamentals of Cloud Computing in Theory and Practice*. Syngress Publishing, 2013.

[91] Miroslav Sabo and Jaroslav Porubän. Preserving design patterns using source code annotations. *Journal of Computer Science and Control Systems*, (1):53, 2009.

[92] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1063–1075, 2019.

[93] Tiago Boldt Sousa, Ademar Aguiar, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Engineering software for the cloud: patterns and sequences. In *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming*, pages 1–8, 2016.

[94] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. Patterns for software orchestration on the cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, pages 1–12, 2015.

[95] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Overview of a pattern language for engineering software for the cloud. In *Proceedings of the 25th Conference on Pattern Languages of Programs*, pages 1–9, 2018.

[96] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Messaging systems and logging. In *Proceedings of the 22Nd European Conference on Pattern Languages of Programs*, pages 1–14, 2017.

[97] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Automated recovery and scheduler. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–8, 2018.

[98] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: External monitoring and failure injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–8, 2018.

[99] Matúš Sulír, Ján Juhár, et al. Source code annotations as formal languages. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 953–964. IEEE, 2015.

[100] Sysstat. sysstat/sysstat: Performance monitoring tools for Linux. Available at https://github.com/sysstat/sysstat. Accessed in February 2021.

[101] Blesson Varghese and Rajkumar Buyya. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79:849 – 861, 2018.

[102] Martti Vasar, Satish Narayana Srirama, and Marlon Dumas. Framework for monitoring and testing web application scalability on the cloud. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, pages 53–60. 2012.

[103] Adbys Vasconcelos, Lucas Vieira, Ítalo Batista, Rodolfo Silva, and Francisco V Brasileiro. Distributedfaas: Execution of containerized serverless applications in multi-cloud infrastructures. In *CLOSER*, pages 595–600, 2019.

[104] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182. IEEE, 2016.

[105] Lizhe Wang, Gregor von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu. Cloud computing: a perspective study. *New Generation Comput.*, 28:137–146, 04 2010.

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

[107] Olaf Zimmermann. Architectural refactoring for the cloud: a decision-centric view on cloud migration. *Computing*, 99(2):129–145, 2017.