POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Implementation of a serverless application

Supervisors
Prof. Maurizio MORISIO

Candidate
Antonio D'Amore

October 2020

A mamma e papà

Table of Contents

Acronyms

ACL

Access Control List

AWS

Amazon Web Services

BaaS

Backend as a Service

B2B

Business-to-business

CDK

Cloud Development Kit

CIDR

Classless Inter-Domain Routing

DSL

Domain-Specific Language

EC2

Elastic Compute Cloud

ECS

Elastic Cloud Service

EKS

Elastic Kubernetes Service

FaaS	Function as a Service
GUI	Graphic User Interface
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
IAM	Identity and Access Management
IP	Internet Protocol
NAT	Network Address Translation
PaaS	Platform as a Service
PCI DSS	Payment Card Industry Data Security Standard
PUE	Power Usage Effectiveness
SECaaS	SECurity as a Service
VPC	Virtual Private Cloud

Chapter 1

Introduction

Today human activities are strictly dependent on the Internet. People can communicate, make money transfers, manage their jobs and a lot more.
Problems with increasing complexity require information technology to find simpler and simpler ways to face against them. The concept of abstraction is what allows to deal with this purpose.
Serverless computing in an abstraction model which allows to execute some code in isolated environments, while hiding the provisioning and management of servers, which are under the cloud provider responsibility. Applications can leverage several paradigms, like Function as a Service, in which code is executed in ephemeral stateless containers, and Backend as a Service, in which the server-side logic relies on third-party services, generally used for providing authentication, cloud storage, and more.
Serverless is meant to help IT teams to build easily scalable architectures, economically convenient and with low effort in maintenance. Furthermore, it allows organizations to reallocate time and people to problems unique to the product.
Serverless is an architectural style brought to the fore a few years ago by leading cloud providers. Today it is mainly spread for executing automation tasks, data-processing pipelines, or as general purpose glue code in operations, but it is difficult to find serverless to manage the core logic of applications. Therefore, despite the fact that serverless promises a lot of benefits in operations, there is still a lack of experience with it.
It is still unclear how serverless affects DevOps, therefore it is necessary to enlarge the general background, and consolidate a real culture, and this can happen thanks to the joint research of universities and organizations.
The main activity on which this thesis focuses is the implementation of a serverless application and the designing of a continuous delivery pipeline on behalf of Consoft Sistemi S.p.A., specialized in systems integration. The project requires the orchestration of multiple and different infrastructure components, even not thought

specifically for serverless environments, therefore they have to be allocated and dismissed as needed, in order to enable a pay-per-use pricing model.

1.1 Motivations

The greatest wish and aim of this thesis, first of all, is to allow that general background, mentioned before, to expand further.
To begin, the term "serverless" will be explored, because it has not an official definition, but is a concept shaped by communities of engineers and developers. It is often interchanged with FaaS, but this thesis will keep these terms separated, treating FaaS as a subset of serverless. The reason is that developing "with no servers" is a promise which is not strictly related to run single stateless functions. Understanding the most subtle differences helps to appreciate the various levels of abstraction.
Serverless architecture can be complicated to design, because of the fragmentation of the model which can bring to a poorly comprehensive overview. It is necessary to detect common and best practices and analyzing a given application in the perspective of microservices application, examining common key features, and how consistency is handled in a distributed environment of this type.
Behind the apparent simplicity of the model, one of the biggest difficulties of serverless is dealing with operations. While promising freedom from managing servers, it still requires DevOps practices to be deepened.
Finally the maturity level of serverless will be examined, in order to understand what today is missing, and what cloud providers need to focus on to fully exploit the benefits that serverless can bring.

1.2 Problem statements

This thesis addresses the following research questions:

- How can a serverless architecture be designed?
- How can a software not meant for serverless be adapted to this cloud model?
- How a serverless application affects operations?
- What is the maturity level of serverless?

1.3 Methodology

Design science research has been chosen for guiding this work, since it aims to develop knowledge that the professionals of the discipline in question can use to

design solutions for their field problems, and focuses on production of an artifact, which makes this methodology suitable for many Engineering and Computer Science disciplines.
The developed artifact must be a viable technology-based solution to a relevant business problem. Its design must be evaluated about its utility, quality and efficiency and its result must be presented effectively to technology-oriented as well as management-oriented audiences.
The artifact enables the researcher to get a better grasp of the problem, which is then re-evaluation in order to improve the designed artifact. Therefore, a build-and-evaluate loop is generated, which makes this methodology a continuous and iterative process.

1.4 Structure of the thesis

This work is developed over several chapters, following the methodology described above, and which try to start from a wide overview, up to the most minute details. In particular:

- chapter 2 makes an analysis of the state of the art of serverless and its ecosystem;
- chapter 3 illustrates the business problem and the core concepts of the designed solution;
- chapter 4 describes the designed architecture, the chosen technical stack and how relevant problems have been addressed;
- chapter 5 evaluates the implemented artifact, with respect to most theoretical and practical aspects.
- chapter 6 exposes the conclusions about the artifact and the serverless vision;

Chapter 2

State of the art

This chapter is meant to give a general knowledge about the serverless computing and the contexts in which it operates. Section 2.1 explores the meaning of the term serverless, benefits and drawbacks of the model. Section 2.2 and 2.3 show the offer of serverless services in AWS, and how the provider handles security and networking. Section 2.4 explains how serverless affects DevOps. Section 2.5 illustrates the Infrastructure as Code approach. Section 2.6 examines the impact of serverless on monitoring, and section 2.7 describes principles for a CI/CD pipeline and the AWS solution for managing it. Finally sections 2.8 and 2.9 give brief theoretical notes on concepts useful for understanding this work.

2.1 Serverless computing

In the last decade, several cloud computing execution models have emerged and one of these is severless computing. There is not an universally accepted definition of this model, therefore the meaning suggested by the word itself will be analyzed in the first place.
It hints that there are not servers, virtual machines or containers, therefore a series of responsibilities are eliminated. While, in reality, physical supports where code is executed, i.e. servers, still exist, the second statement is true, because serverless eliminates the responsibility of their provisioning. In practice, like in the spirit of cloud computing itself, serverless introduces an abstraction layer between the application and the underlying layers.
Being a model delivered as a service, serverless is provided by cloud providers. In a private cloud the consequence of this approach is a more definite decoupling of responsibilities among different teams, e.g. one for providing the cloud, one for the development. But the organization still has costs linked to servers, also for idle resources. Instead, serverless is more beneficial in a public cloud because it gives

the opportunity of reallocating resources to problems unique to the organization, so that the effort in operations can be reduced. Using serverless in the public cloud also means that a series of considerations about costs can be freed, since it allows a pay per use model, to run applications on demand, with no extra costs. Moreover, since cloud providers make available a series of predefined services that many other people use, the Economy of Scale is leveraged in order to offer relatively low prices.
The term "serverless" has been introduced in the cloud context during an AWS re:Invent event in 2014 [1], when Amazon has published **AWS Lambda** [1], a FaaS solution, which allows to run stateless code in an isolated environment.
With FaaS, code is executed in ephemeral stateless "compute containers", which are created and destroyed as needed at runtime. Most important, the vendor handles provisioning and allocation of all underlying resources.
Today the market offers very similar solutions, like Microsoft Azure Functions [2], IBM Cloud Functions [3], Google Cloud Functions [4] and Alibaba Cloud Function Compute [5].
Nevertheless, the term "serverless" can't be reduced only to FaaS. Serverless is a qualifier that can be applied to any software requiring to be consumed as a service [2]. This is similar to PaaS definition, in which there is no possibility to manage or control the underlying cloud infrastructure, but gives the possibility to control over the deployed applications and possibly configuration settings for the application-hosting environment [3]. Note that PaaS does not hide the existence of servers and their environments, but only relinquishes some responsibilities. Conversely, serverless removes user control over hosting and this allows to gain more attractive billing models, based on pay-per-use, and also better scalability. Indeed, ignoring the existence of a server and its environments means that a long-lived server application is not conceived, therefore serverless is supposed to be stateless and this establishes the conditions for a high degree of parallelism.
Serverless gives a very powerful abstraction which intends to link all responsibilities to application development only, trying to speeding up the development itself. According to this perspective applications could significantly or fully incorporate third-party, cloud-hosted applications and services, to manage server-side logic and state. Examples of these could concern cloud-accessible databases (e.g., Parse, Firebase) or authentication services (e.g., Auth0, AWS Cognito). These types of services are called Backend as a Service (BaaS) and can be intended to be serverless,

[1]`https://aws.amazon.com/lambda/`

[2]`https://azure.microsoft.com/en-us/services/functions/`

[3]`https://www.ibm.com/cloud/functions`

[4]`https://cloud.google.com/functions/`

[5]`https://www.alibabacloud.com/products/function-compute`

since they don't require server provisioning or managing [4].
Through the years, the serverless ecosystem has growth, matured and continues to evolve. As mentioned above, serverless is no longer just FaaS, but can take on more varied forms. AWS Fargate [6] allows to launch containers in a serverless environment. It allows fine grained auto-scaling of containers without server provisioning or managing, and offer a pay per use pricing model.
Along with the servers, their limits are also abstracted away. Due to the "as a service" nature, auto-scaling, high availability and fault tolerance are built-in. Horizontal scaling is completely automatic, elastic, and managed by the provider, without any extra configuration by the user.

2.1.1 Benefits

Focus

Moving away from servers and infrastructure means a reduction in responsibilities for operating cloud infrastructure. It provides the opportunity to dedicate resources to the problems closest to the business.

Reduced operational cost

Serverless is an outsourcing solution and it allows to exploit the economy of scale to benefit of affordable prices; companies have to bear only the costs that are really necessary for their applications, while cloud providers have to manage servers, databases, services and pay for idle resources (shared among all their customers).

Reduced development cost

BaaS services aim to solve very common problems for users by offering ready-built functionalities that are easy to integrate, with the added benefit that there are no provisioning or management costs. Development is faster and time to market is shortened.

Scaling cost

Horizontal scaling is completely automatic, elastic, and managed by the provider, and it is requested to pay only for needed resources, since they are allocated when it is really necessary. Traditional infrastructures requires some fixed costs, even for idle resources, and this can be highly inconvenient, especially for very heterogeneous traffic loads over time.

[6]`https://aws.amazon.com/fargate/`

Easier operations management

On one hand, there are not servers to provision or to manage, therefore there is almost zero maintenance, which is a great advantage. To the other hand, there it is requested a greater effort at application level, with monitoring, troubleshooting, tracing vulnerabilities, patching, and so on.

Transitioning from a server-based application

It is easy to adopt, even in an existing application. In fact, the flexibility and stateless nature of serverless allow it to be integrated gradually. If possible, the application can be converted to become completely serverless. If this is not possible, for example because the application uses critical legacy code, it may be better not to intervene directly on that code and use legacy components as services. In general, the transition from a legacy, server-based application to a scalable serverless architecture may take time. It requires attention and a careful DevOps strategy.

Reduction of energy expenditure

The advancement of innovative fields of ICT, like artificial intelligence, big data elaborations, streaming video, IoT (Internet of Things), will increasingly demand more computing power, that means more energy consumption [5].
Power Usage Effectiveness (PUE) is a metric used to determine the energy efficiency of a data center. It is determined by dividing the amount of power entering a data center by the power used to run the computer infrastructure within it [6]:

$$PUE = \frac{Total\ facility\ energy}{IT\ Equipment\ energy}$$

The lower the PUE, the higher the energy efficiency. Moreover, there is no linearity between CPU utilization and power consumption. This means that an idle CPU consumes 66% of the peak power [7]. It is evident that there is a waste of energy, which results in higher carbon dioxide emissions into the environment.
In public cloud all decisions about hardware provisioning are taken by vendors, that can take globally optimal infrastructural choices, with the big players which typically pay attention to values of PUE [7] [8] [9]. This hypothesis could be even

[7]`https://aws.amazon.com/it/about-aws/sustainability/`

[8]`https://www.google.com/about/datacenters/efficiency/`

[9]`https://download.microsoft.com/download/7/3/9/739BC4AD-A855-436E-961D-9C95EB51DAF9/Microsoft_Cloud_Carbon_Study_2018.pdf`

more likely in serverless, because virtual resources are allocated only when really necessary, so the average CPU usage depends on the real needs of customers.

2.1.2 Drawbacks

Serverless is not a silver bullet in all circumstances. It may not be appropriate for latency-sensitive applications or software with specific service-level agreements (SLA). It may not be appropriate also for mission-critical applications, if serverless is executed into a public cloud.

Vendor lock-in

Serverless code is built to run in a specific public cloud, so it is highly dependent on the provider services. It would be costly to migrate to a different solution, because it would be necessary to re-design - at least partially - the application.

Vendor control

As serverless is an outsourcing strategy, applications are bound by the dictates of the vendor. There can be limitations, cost changes, forced component upgrades, loss of functionality, and more.

Multi-tenancy problems

Applications of different customers of a public cloud share the same machine in order to benefit the economy of scale benefits. This is a multi-tenancy solution, so there could be problems with:

- security: one customer being able to see another's data;
- robustness: an error in one customer's software causing a failure in a different customer's software;
- performance: a high-load customer causing another to slow down;
- compliance: some companies may not be able to store data within shared infrastructure, due to regulatory requirements

Decentralization

The distributed nature of serverless can introduce its own challenges because of the need to make remote - rather than in-process - calls and the need to handle failures and latency across a network.

Cold starts

When a function is invoked, a container must be allocated, in order to run the function code. This bootstrap involves a latency that can vary significantly, in a range from a few milliseconds to several seconds. This kind of startup is called **cold start**. After the execution, the container can remain active for some minute, waiting for further function invocations, so subsequent functions can use that **warm container**, where code can be executed immediately.
Cold start can be a problem or not, according to the application nature and requirements. AWS has tried to mitigate the problem through **provisioned concurrency**, described in section 2.2.1.

2.1.3 Future of serverless

Serverless is still a fairly new world. Years of study and experience are needed before it can be consolidated in IT solutions. It is necessary to build a real culture of serverless and this can happen thanks to the joint research of universities and companies. Finding architectural patterns could be useful for this purpose. It is necessary to understand best ways to organize projects, how to organize logic, how the architecture should interact with event-thinking, and more[4].

2.2 Serverless in AWS

The AWS cloud provides many different services that can be components of a serverless application:

- Compute: AWS Lambda;
- APIs: AWS API Gateway;
- Storage: AWS S3;
- Databases: AWS DynamoDB;
- Interprocess messaging: AWS SNS and AWS SQS;
- Orchestration: AWS Step Functions and Amazon EventBridge;
- Analytics: Amazon Kinesis;

In 2018 Amazon introduced **Firecracker** [10], a VMM for executing serverless workloads, through services like AWS Lambda and AWS Fargate [8].

[10]`https://firecracker-microvm.github.io/`

Firecracker is based on KVM and aims to provision secure sandboxes with a minimal footprint, enabling performance without sacrificing security [9].

2.2.1 AWS Lambda

AWS Lambda is the FaaS solution by Amazon introduced at AWS re:Invent in 2014 and it is available in all regions. It is event-driven, so it can run user defined functions in response to events, such as an update to an AWS DynamoDB table or a HTTP call to AWS API Gateway or Lambda API, or it can be invoked directly, synchronously (and wait for the response), or asynchronously [10] from the cloud.
Each Lambda function contains the code to execute, the configuration that defines how code is executed and, optionally, event sources that detect events and invoke the function as they occur.
Users can choose among multiple runtimes, which support languages like Java, Go, PowerShell, Node.js, C#, Python and Ruby [11].
The code is then packaged with all of the necessary assets, like additional files, classes, libraries, binaries, or configuration files. The maximum size of a function code package is 50 MB compressed and 250MB [12].

Concurrency

The key advantage of serverless is that users don't have to handle provisioning to meet traffic demand, but it is managed automatically by the cloud provider. To accomplish this requirement, AWS Lambda uses a parameter called **concurrency**, which refers to the number of executions of function code that are happening at any given time. When a function is invoked, Lambda allocates an instance of it to process the event. When the function code finishes running, it can handle another request. If the function is invoked again while a request is still being processed, another instance is allocated, which increases the function's concurrency [13].

Reserved concurrency

Concurrency is deduced by a pool of concurrency units that is shared by all functions in a Region. To ensure that a function can always reach a certain level of concurrency, it is possible to configure a parameter called **reserved concurrency** to a function. Anyway, all functions with no reserved concurrency have a shared pool of 100 concurrency units.
Reserved concurrency also limits the maximum concurrency for the function, so if the function's concurrency exceeds this value, the function goes in *throttling* and the request is not served. Therefore, reserved concurrency can give an fine-grained control over concurrency. This can be helpful if the function calls an external service which supports a limited number of concurrent requests, in order to be

considered reliable.
Reserved concurrency can give guarantees, but, on the other hand, it reduces the available concurrency capacity in the Region for the given account.

Provisioned concurrency

In some application cold start could be a problem, so AWS Lambda offers an option called **provisioned concurrency**, which has been introduced at AWS re:Invent 2019 [14]. It allows developers to anticipate, in a specified period, the allocation of a certain number of instances of a function and keep them provisioned and warm for incoming requests. In this way performance of all requests are guaranteed to stay below double-digit milliseconds. If this number is exceeded, the standard behaviour is applied.
Provisioned concurrency requires to keep some resources running, even if they're not used, i.e., because the traffic generated by the function is low. Therefore this functionality requires a cost which depends on the Region, size in GB of provided instances and period of allocation in seconds.

Auto scaling

In some application it can be useful to configure **auto scaling** in order to manage provisioned concurrency on a schedule or based on utilization. It is so possible to configure alarms on CloudWatch in conjunction with the growth of traffic to trigger provisioned concurrency, which increases the number of warmed instances.

Security

Lambda API endpoints only support secure connections, encrypted with Transport Layer Security (TLS). Environment variables are always encrypted too, in order to store even secrets. Files uploaded to Lambda, including deployment packages and layer archives, are also encrypted.
Lambda integrates AWS IAM, so there is a strong layer dedicated to authentication and authorization, for handling roles, permissions and so on.
It is possible to configure a Lambda function to connect to private subnets in a virtual private cloud (VPC) in the AWS account [15].
As of March 2019, Lambda is compliant with SOC 1, SOC 2, SOC 3, PCI DSS, U.S. Health Insurance Portability and Accountability Act (HIPAA), etc [16].

2.2.2 AWS DynamoDB

AWS DynamoDB [11] is a key-value and document database, that delivers single-digit millisecond performance at any scale. It's a fully managed, multiregion, multimaster, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications.

Main concepts

There are no servers to provision, patch, or manage and no software to install, maintain, or operate and availability and fault tolerance are built in, so it is a real serverless storage solution.
It has a key-value or wide-column data model. The first one allows to treat tables like a hash map, in which, each element, called item, is uniquely identifiable by a key. It is possible to get, set, update, and delete these elements by referring to its primary key. The second model allows to access tables like if they are B-trees, which allows to easily retrieve multiple records, scan items in a range, and more.
DynamoDB has been thought for infinite scaling with no performance degradation. Most operations in DynamoDB have response times in single-digit milliseconds [17]. Moreover, AWS offers **DynamoDB Accelerator (DAX)**, which is a fully-managed in-memory cache for DynamoDB tables.
DynamoDB is also Infrastructure as Code oriented, being possible to provision, update, or remove infrastructure resources, through AWS CloudFormation, described in section 2.5.1.
Two pricing models are available. The first one allows to separately define read and write throughput. The second one allows to not define throughput, and pay exclusively on-demand, in order to pay per request rather than provisioning a fixed amount of capacity. Per-request price is higher than the provisioned mode, but it can still save money for certain types of workload which don't take full advantage of provisioned capacity. Anyhow it is possible to switch between pricing models over time.

Stream

Streams are an immutable sequence of records that can be processed by multiple, independent consumers. The combination of immutability plus multiple consumers has propelled the use of streams as a way to asynchronously share data across multiple systems.
When enabled, DynamoDB Streams captures a time-ordered sequence of item-level

[11] `https://aws.amazon.com/dynamodb/`

modifications in a DynamoDB table and durably stores the information for up to 24 hours. Applications can access a series of stream records, which contain an item change, from a DynamoDB stream in near real time [18].
For example, it could be possible to use DynamoDB Stream to capture an update on a table, in order to trigger a function and take some actions.
DynamoDB Streams supports the following stream record views:

- KEYS_ONLY: only the key attributes of the modified item;
- NEW_IMAGE: the entire item, as it appears after it was modified;
- OLD_IMAGE: the entire item, as it appears before it was modified;
- NEW_AND_OLD_IMAGES: both the new and the old images of the item.

2.2.3 AWS Fargate

AWS Fargate [12] is a serverless compute engine for containers that works with both Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS).
The concept of container is extended to an isolated environment which can group more containers and called *task* in ECS (equivalent to *pods* in Kubernetes). The task is then launched in a **cluster**, which represent a Region-specific logical group of tasks.
To launch a Fargate task in an ECS cluster, it is sufficient to set a **task definition**, which is a blueprint which specifies essentially the container images, and optionally resource allocation for each container, such as the memory and CPU values.
Prices depend on CPU and memory usage per hour [13]. Furthermore, it is possible to save up 70% costs in fault tolerant applications by using **spot instances** [19], which use spare capacity of the cloud.

2.3 Security in AWS

The IT infrastructure that AWS provides to its customers is designed and managed in alignment with security best practices and a variety of IT security standards [20]. In addition, the flexibility and control that the AWS platform provides allows customers to deploy solutions that meet several industry-specific standards. Each service provides extensive security features to enable you to protect sensitive data and applications [21].

[12]`https://aws.amazon.com/fargate/`

[13]https://aws.amazon.com/fargate/pricing/

2.3.1 Shared-responsibility model

AWS uses the **shared-responsibility model** [22], meaning responsibility is shared between user and AWS. AWS is responsible for the following

- protecting the network through automated monitoring systems and robust internet access, to prevent Distributed Denial of Service (DDoS) attacks.
- performing background checks on employees who have access to sensitive areas.
- decommissioning storage devices by physically destroying them after end of life.
- ensuring the physical and environmental security of data centers, including fire protection and security staff.

User is instead responsible for the following:

- implementing access management that restricts access to AWS resources to a minimum, using AWS IAM;
- encrypting network traffic to prevent attackers from reading or manipulating data (for example, using HTTPS).
- configuring a firewall for your virtual network that controls incoming and outgoing traffic with security groups and ACLs.
- encrypting data at rest. For example, enable data encryption for your database or other storage systems.
- managing patches for the OS and additional software on virtual machines.

2.3.2 SECurity as a Service

SECaaS (SECurity as a Service) is an outsourcing model for security management and it is a segment of SaaS, so it is provided over the network and on-demand, with a pay per use model [23]. Its main features are:

- the cost varies according to demand;
- ease of management and operations;
- maintenance responsibility of the provider;
- consumer can focus on core competencies;
- outsourcing of administrative tasks, such as log management;

- eliminate or reduce Help Desk calls;
- high scalability;
- faster user provisioning;
- greater security expertise than is typically available within an organization;
- access to specialized security expertise;
- continuous updates;

SECaaS can operate in several categories: identity and access management, network security, web security, encryption, intrusion management, disaster recovery and other.

2.3.3 Least Privilege Principle

The **Least Privilege Principle** states that only the minimum necessary permissions should be assigned to a subject that requests access to a resource and should be in effect for the shortest duration necessary (remember to relinquish privileges). Granting permissions to a user beyond the scope of the necessary rights of an action can allow that user to obtain or change information in unwanted ways. Therefore, careful delegation of access rights can limit attackers from damaging a system [24]. This is recommended by AWS, but it is the user who must respect it. It is not a provider responsibility.

2.3.4 AWS IAM

AWS IAM [14] (Identity and Access Management) is the Amazon SECaaS solution for securely control access to AWS resources. It has no additional price, so it is typically widely used in the infrastructure.
Its main features are [25]:

- shared access to an AWS account, with granular permissions, to grant different permissions to different people for different resources;
- secure access to AWS resources for applications that run on Amazon EC2, to allow them to access other AWS resources;
- Multi-Factor Authentication (MFA), for extra security;

[14]`https://aws.amazon.com/iam/`

- identity federation, in order to allow users who already have passwords elsewhere to get temporary access to your AWS account (e.g., Single Sign On);
- identity information for assurance: If you use AWS CloudTrail, you receive log records that include information about those who made requests for resources in your account. That information is based on IAM identities;
- PCI DSS Compliance, useful especially for companies that directly handle credit cards and online transactions;
- eventually consistent, due to the fact that AWS maintains data in multiple locations, and they can be updated via asynchronous events, which could be delayed, or fail, or be delivered multiple times, so it is necessary to cope with the chance that the copies of retrieved data have become stale [26];
- no additional charge. You are charged only when you access other AWS services using your IAM users or AWS STS temporary security credentials. For information about the pricing of other AWS products, see the Amazon Web Services pricing page.

IAM Identities

Authentication and authorization policies are applied to **IAM Identities**, which can represent people or services; there are several types of Identites:

- **IAM User**: people or application that can access to AWS. It consists of a name and credentials. Credentials can be [27]:
  - a password that the user can type to sign in to interactive sessions such as the AWS Management Console
  - a combination of an access key ID and a secret access key, for API or CLI;
  - SSH Keys that can be used to authenticate with CodeCommit;
  - SSL/TLS certificates that you can use to authenticate with some AWS services;

  In order to easier give same permissions to a set of users, it is possible to create an **IAM Group**;
- **IAM Role**: it is similar to IAM User, but instead of being uniquely associated with one person, it is intended to be assumable by anyone who needs it to assign certain permissions. For example it is possible to assign a role to an AWS Lambda function in order to give it permission to read from an AWS Dynamo table;

2.3.5 AWS VPC

AWS VPC [15] (Virtual Private Cloud) allows to provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network defined by the user.
It allows to create subnets and define security features, such as **Security Groups** and network **Access Control Lists**, to enable inbound and outbound filtering at the resources. AWS VPC is conform to the AWS *shared responsibility model.* AWS is responsible for protecting the global infrastructure that runs all the AWS services.

Networking

There are different types of networks:

- **public subnet**: resources inside it are accessible from the Internet. In this case the subnet's traffic is routed to an internet gateway and public resources must have a public IPv4 address or an Elastic IPv4 address;

- **private subnet**: resources cannot be reached directly from the Internet, since the traffic is routed through a NAT gateway, so that only outbound traffic is allowed;

- **isolated subnet**: traffic is not routed to the Internet (in this VPC). This can be useful, for example, if RDS or Elasticache instances have to be reachable in an exclusive space;

Not all the AWS resources must to be inserted in a VPC. For example, DynamoDB can be used with a serverless approach. It already runs in an isolated space of the AWS public cloud, so it is not necessary to add it to a VPC. The same is for AWS Lambda, even if it can be configured by specifying a reference VPC. The Lambda still runs in the AWS public cloud, but this configuration allows the Lambda to easily connect to subnet of the VPC, in order to use private resources inside it.

Security Group

A security group acts as a virtual firewall, by setting rules of inbound and outbound traffic at instance level (not subnet level). It is possible to assign up to five security groups for each instance, each one containing up 60 inbound rules and 60 outbound rules. A rule is an "allow" sentence, composed of several fields:

[15] `https://aws.amazon.com/vpc/`

- the **source** (for Inbound rules only) of the traffic, which can be another security group, an IPv4 or IPv6 CIDR block, a single IPv4 or IPv6 address, or a prefix list ID;
- the **destination** (for Outbound rules only) for the traffic, which can be can be another security group, an IPv4 or IPv6 CIDR block, a single IPv4 or IPv6 address, or a prefix list ID;
- any protocol that has a standard protocol number [16];
- port range through which traffic must be received (for Inbound rules) or sent (for Outbound rules);

Security Group are *stateful*: this means that responses to allowed inbound traffic are allowed to flow outbound regardless of outbound rules, and vice versa. Hence an Outbound rule for the response is not needed.

2.3.6 AWS Secrets Manager

AWS Secrets Manager [17] aims to securely encrypt, store, and retrieve credentials for any service. Hardcoding credentials in apps is never a good idea, and it is absolutely nefarious if the source code is maintained in a public repository, as it would expose sensitive values to the world. This solution allows to make calls to Secrets Manager to retrieve credentials whenever needed.
Each **secret** has a name and can be composed of multiple key-value pairs. Then is possible to configure secret rotation and attach security policies.

2.3.7 Microservices and serverless

Microservices refer to an architectural style in which the application is made of independent components characterized by certain key principles [26]:

- **Autonomy**: services are loosely coupled, so operate independently and interacts through clearly defined interfaces, or through published events. Then they are independently deployable, so that they can be developed in parallel, often by multiple team, which can ideally enable rapid, frequent, and small releases;
- **Resilience**: as the services are independent of each other, it is possible to isolate failures and to introduce changes gradually. This favours asynchronous

[16]`http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml`

[17]`https://aws.amazon.com/secrets-manager/`

interactions and using provable continuous delivery techniques and robustly monitoring system activity;

- **Transparency**: it is important to be able to detect failure, so the system should be observable. Every service in the application will produce business, operational, and infrastructure metrics, logs, and request traces;
- **Automation**: microservices have introduced an additional complexity which can be reduced, by mean of automation techniques which ensure correctness of deployments and system operation. This necessity led to the spread of Devops techniques, especially IaC, and the rise of infrastructure environments that are fully programmable through APIs, such as AWS or Azure;
- **Alignment**: many traditional SOAs used to practice horizontal decomposition, which is problematic, because new features may require coordinated releases to multiple services and may become unacceptably coupled to others at the same level of technical abstraction. At the opposite, microservices promotes structuring services, and therefore teams, around business concepts, encapsulating all relevant technical layers. This leads to higher cohesion and vertical decomposition.

Today it is not clear how serverless and microservices are really related. They could share a lot or very little. This thesis will go more in deep in the evaluation chapter, section 5.1.

2.4 Serverless in DevOps

DevOps is the process of continuously improving software products through rapid release cycles, global automation of integration and delivery pipelines, and close collaboration between teams [28].

2.4.1 Moving to a new culture

Before DevOps, there was a "silos" approach, in which development and operations teams were clearly separated and their workloads barely grazed each other [29].
With "silos", each team has a precise responsibility: the development team implements new features for the product, QA team checks that requirements and goals have been fulfilled and the operations team ensure that the features are correctly deployed. In this way, each team focuses solely on its tasks, communication is weak and the success of the product becomes out of focus.
Conceptually a project is something that is designed, implemented and completed, but businesses are typically long-lived and continuously and rapidly change over

time. Business problems are long-lived too, so it is necessary to find a mindset which aims to support the business over the entire life-cycle: **products, not projects** [30].
DevOps is not just a set of technical approaches, but it is a new culture which organizations must embrace. Both developers and operations remained separate teams but with some operations responsibility crossing over onto development teams while both operations and development experienced higher than previous levels of communication between each other [31].
In DevOps, everyone in the product pipeline is focused on the customer:

- Product managers measure engagement and retention ratios.
- Developers measure ergonomics and usability.
- Operators measure uptime and response times.

The customer is where the company's attention is. The satisfaction of the customer is the metric everyone aligns their goals against [28].

2.4.2 Serverless breaking changes

With serverless there is no provision and no server management, and this leads to theorize **NoOps** (no operations), which is the concept that an IT environment can become so automated and abstracted from the underlying infrastructure that there is no need for a dedicated team to manage software in-house [32].
Despite there is no server to manage, some responsibilities still exist. "Ops" doesn't mean just server administration, but also monitoring, deployment, security, networking, support, and often some amount of production debugging and system scaling [4]. In practice, serverless is not putting an end to DevOps, but it is redefining it.
The technological stack is moved up. Before serverless, work of operations engineers was done at host level, while it is now done at application level and tools currently involve mostly software for monitoring, troubleshooting, tracing vulnerabilities, patching, etc.
Monitoring in cloud is a complex task, and it is even more so with FaaS because of the ephemeral nature of containers. Generally, cloud vendors give some monitoring support, but it is not taken for granted that these are sufficient.
The fact that there are no servers and operating systems to manage can lead to a false sense of security. Therefore adopting serverless requires awareness and education. Since IT today is called to solve new problems, new disciplines are emerging, like **Chaos Engineering** [18], which consists of experimenting on systems

[18]`https://principlesofchaos.org/`

to understand weaknesses, and build confidence that a system can withstand the variety of conditions it will experience.
With serverless, system changes are tightly coupled with cost. A function which run slightly slower could mean a reasonable jump in cost. This kind of scenarios require much more accurate analysis, that involve finance, product and project management, and engineering. Today this multidisciplinary process is still not very mature, but it is leading to the birth of the figure of **FinDev** [19].

2.5 Infrastructure as Code

Infrastructure as Code (IaC) is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. It allows the automation of handling IT infrastructures, enabling organizations to develop, deploy, and scale cloud applications with greater speed, less risk, and reduced cost [33].

2.5.1 AWS CloudFormation

This service [20] is the IaC solution in AWS. It takes a template file (also called *blueprint*) as input and on the basis of this, it build the stack into the AWS cloud. A stack is a group of resources which is treated as a single unit. It is possible to create, update, and delete a collection of resources by creating, updating, and deleting stacks. All the resources in a stack must be defined by a template.
A CloudFormation template is a JSON or YAML formatted text file and it has full coverage over the capabilities of the AWS cloud. It is possible to specify some conditions, that control whether certain resources are created or whether certain resource properties are assigned a value during stack creation or update. For example, it is possible to take some actions depending on environment (e.g., development, testing, production) [34].
It is also possible to add input parameters to pass to the stack, making the template easier to reuse in different situations [35].
If a running resources needs in a stack need to be modified, it is necessary to update the stack. Then, on the basis of the oldest template and the modified one, it is generated a change set, which is a summary of proposed changes. Change sets allow to see how changes might impact on running resources, especially for critical resources, before implementing them, in order to avoid unwanted issues. For example, it the name of an Amazon RDS database instance has to be modified,

[19]`https://blog.gardeviance.org/2016/11/why-fuss-about-serverless.html`

[20]`https://aws.amazon.com/cloudformation`

AWS CloudFormation will create a new database and delete the old one, so data in the old database will be lost, unless a backup is prepared.

2.5.2 AWS CDK

AWS Cloud Development Kit [21] is an abstraction of CloudFormation, whose output is a CloudFormation template. The difference is that CDK gives the possibility to describe the infrastructure by mean most common programming languages, rather than text files. Currently supported languages are TypeScript, JavaScript, Python, Java and C#. This can be an advantage because:

- a language allows to use cycles and conditions, which helps in more intricate use cases. It could be less prone to errors, because of language features like type-safety, code completion, etc;
- IDEs and editors benefit from modern features like auto-complete, highlighting, built-in syntax checking;
- it is possible to organize code according the application needs. Language constructs and organization allow to favor code reuse;

CDK can potentially be helpful to build a framework with which describing the infrastructure becomes faster and easier.
CDK inherits some concepts from CloudFormation, like stack. More, it introduces the concept of **app**, which is a group of one or more stacks, in order to describe whole complex applications, not just stacks, but allows a centralized management of them.
A **construct** is the basic building blocks of AWS CDK apps. It represents a "cloud component" and encapsulates everything AWS CloudFormation needs to create a component [36]. A construct may require some mandatory parameters for a minimal configuration and set others by default, but it can also be very verbose, giving the possibility to make explicit details.

2.5.3 Serverless Framework

Serverless Framework [22] is a project which aims to offer tool from easily building serverless applications. It is able to work with the most spread cloud providers, like AWS, Microsoft Azure, Google Cloud Platform, Apache OpenWhisk, Cloudflare Workers, or a Kubernetes-based solution. It is written in JavaScript, but supports

[21] `https://aws.amazon.com/cdk/`

[22] `https://www.serverless.com/`

several languages, like NodeJS, Go, Python, Swift, Java, PHP, Ruby.
The template file for IaC is named *serverless.yml*, by default. It is then processed by the framework. When working with AWS, the output is a CloudFormation template. Serverless Framework offers some advantages [37]:

- it provides a configuration DSL which is designed expressly for serverless applications and assists with additional aspects of the serverless application lifecycle;
- it enables IaC while removing a lot of the boilerplate required for deploying serverless applications, including permissions, event subscriptions, logging, etc.;
- its syntax is provider-agnostic, so it reduces vendor lock-in and enables a multi-cloud strategy;

2.6 Monitoring

In a traditional web application, monitoring is a relatively simple task. The backend server will always have roughly the same overhead, synchronous calls will always execute in the same general sequence, etc. It is possible to have a holistic overview and so it is possible to quickly identify bottlenecks.
In serverless, monitoring is much more complex, because of its nature. Being stateless, the application cannot be maintained in terms of discrete multi-event transactions. Furthermore, serverless applications are distributed, so their functionalities are split across multiple disparate machines, and run on almost entirely ephemeral machines. This complicates the logging activity, because it could not be easy to have a ready snapshot of the recent activities, with which investigate when an issue occurs. A distributed tracing system for an application can be as simple as adding a transaction wrapper that ensures every request shares a traceable ID, or implementing a means of aggregating logs from the different resources that govern the application's serverless behavior, or even making use of third-party tools to provide a more coherent view of the application's execution flow. Then resource usage is not predictable. Each function call introduces an additional overhead, so that timing becomes unpredictable, as well costs. Monitoring itself can represent an higher cost with respect to traditional systems, due to the distributed nature of the architecture;

2.6.1 Possible issues in serverless

A lot of things can go wrong in a serverless application:

- the overhead associated to a cold start is generally not a problem, but a high number of cold starts at the same time can result in a significant impact to user experience;
- resource usage is not always clear. In AWS Lambda functions, it is possible to configure amount of memory that should be allocated for running a function. However, depending on the size of the memory, the computational capacity also changes, which may result in higher usage and so in higher bills;
- serverless put at its basis the promise of infinite scaling, but many providers, like AWS, include a concurrency limit in execution. If this limit is exceeded, then unpredictable behavior may occur;
- in on-demand architecture it may happen that a function simply fails to respond, due to network errors, temporary issue on the provider, a bug in the code, and so on;
- costs are very unpredictable, since there could be no linearity with respect so some parameters, like the size of the user base. A misconfigured Lambda function can end up using a processor that is much more powerful than your function actually needs and represent a higher cost than necessary. A pick of traffic, which can be also be of malicious origin, like a DoS attack, can drive costs up exponentially;

All these scenarios need to be handled by monitoring activities, in order to trigger alarms or take countermeasures promptly.

2.6.2 Observability

Observability is a further evolution of monitoring concept. A cloud native application should be constructed as an observable system. In particular, it's normally not possible to know in advance what is causing the problem and should be able to investigate the events occurred to understand it. Events occurring to an application at runtime can be completely described by analyzing three major pillars:

- Metrics Records of execution properties of a single service at a certain time;
- Logs Records of what is happened to the application, using the "words" of developers;
- Traces Records of requests as they traverses through the many services in your micro-service application.

2.6.3 AWS CloudWatch

AWS CloudWatch [23] is a dedicated tool for monitoring AWS resources. CloudWatch is used for logging, aggregating statistics, building metrics, setting alerts, and more. CloudWatch allows to track serverless functions activities, monitor resource usage to identify bottlenecks in the application architecture, and trigger some actions on the basis of what happens: it is used by AWS EventBridge to build application architectures that react in real time to certain data sources, or it is used to launching a stage of a pipeline when there is a commit in a certain repository, and more.

2.6.4 AWS X-Ray

AWS X-Ray [24] helps developers analyze and debug production, distributed applications, such as those built using a microservices architecture. X-Ray allows to understand how an application and its underlying services are performing to identify and troubleshoot the root cause of performance issues and errors. X-Ray provides an end-to-end view of requests as they travel through the application, and shows a map of the application's underlying components. With X-Ray it is possible to analyze both applications in development and in production, from simple three-tier applications to complex microservices applications consisting of thousands of services.

2.7 CI/CD pipeline

Organizations continuously needs to develop and deploy software releases. Teams can have tens — if not hundreds — of independent services on their own schedule, without explicit coordination or collaboration between teams. Any bad change to a service might have a wide-ranging impact on the performance of other services and the wider application. Deployment should be fast, safe and consistent and these requirement can be achieved by mean a **CI/CD pipeline**, where:

- **Continuous Integration (CI)** means which merge their changes back to the main branch as often as possible. These changes are validated by creating a build and running automated tests against it. This technique is meant to simplify integration among releases;

[23]`https://aws.amazon.com/it/cloudwatch/`

[24]`https://aws.amazon.com/xray/`

- **Continuous Development (CD)** means which every change that passes all stages of production pipeline is released to customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production. This technique is meant to speed up release times and accelerate the feedback loop with customers.

2.7.1 AWS CodePipeline

AWS CodePipeline [25] is a fully managed continuous delivery which allows to model, visualize and automate the steps required to release a software.
A **pipeline** is a workflow construct that describes how software changes go through a release process and consists of a sequence of at least two**stages**. A stage is a group of **actions** and an action is a group of tasks performed on a **revision**, that is a change made to the source location defined for the pipeline. It can include source code, build output, configuration, or data. A pipeline can have multiple revisions flowing through it at the same time.
When an action runs, it acts upon a file or set of files. These files are called **artifacts**. These artifacts can be worked upon by later actions in the pipeline.
Passing from a stage to another is called **transition**.
CodePipeline integrates with AWS services such as CodeCommit, S3, CodeBuild, CodeDeploy, Elastic Beanstalk, CloudFormation, OpsWorks, ECS, and Lambda, external services, like GitHub, Jenkins, BlazeMeter, XebiaLabs and others [26] or custom actions defined by the user [27].
It is possible to create notifications for events impacting the pipelines. Each notification uses AWS SNS and includes a status message and a link to the resources whose event generated that notification. This is important for monitoring purpose.

2.7.2 AWS CodeBuild

AWS CodeBuild [28] is a fully managed continuous integration service that compiles source code, runs tests, and produces software packages that are ready to deploy. There is no need to provision, manage, and scale build servers and there is a pay per use model.

[25]`https://aws.amazon.com/codepipeline/`

[26]`https://aws.amazon.com/codepipeline/product-integrations/`

[27]`https://docs.aws.amazon.com/codepipeline/latest/userguide/actions-create-custom-action.html`

[28]`https://aws.amazon.com/codebuild/`

CodeBuild is integrated with AWS CodePipeline and, when combined, a CI/CD pipeline can be built.
A build project can be configured through the console or the AWS CLI. It is necessary to specify the source repository location, the runtime environment, the build commands, the IAM role assumed by the container, and the compute class required to run the build. Optionally, build commands can be specified in a *buildspec.yml* file.

2.8 Service virtualization

It is a method used in software engineering to emulate the behavior of specific components in heterogeneous component-based applications such as API-driven applications, cloud-based applications and service-oriented architectures.
It addresses what could be a bottleneck for Agile development, that requires rapid capabilities for change, along with modern high-quality composite applications. The software development, testing, and operations teams could not work in synch and this causes delays in workflows, because each team should wait for others to have components ready.
Service virtualization allows to create an isolated environment and simulate APIs or use virtual services instead of production services, so that DevOps teams can test system components and their integration, even when key components are not ready.

Advantages

Service virtualization allows to increase productivity, since it can be integrated easily into the test infrastructure and offers more possibilities to automate tests. Therefore the delivery cycle and end-to-end testing become more efficient and, as a consequence, also time to market could accelerate. This method is expected to improve the testing and product quality, by running realistic, scalable, and secure tests by driving fewer defects into production. The overall expense could decrease, since the test infrastructure is simplified and third-party costs are eliminated.

2.9 Hoverfly

Hoverfly [29] is an open source product developed by SpectoLabs and written in Go. An Hoverfly instance can work according to two schemes:

[29]`https://hoverfly.io/`

1. **as proxy server**: it receives an incoming request, and forwards it to the destination service. In the meantime it waits for the response from the destination. When it arrives, the instance stores the array of request-response pairs in a JSON objects, called **simulations**, and forward the response to the source that made the original request;

2. **as webserver**: it receives an incoming request, and looks among previously captured simulations for a pair having an analogous request. If the pair is found, the instance forwards it to the original source, otherwise it can behave differently according the working mode of Hoverfly that has been set;

These schemes form the basis for service virtualization. Each time Hoverfly receives a request, rather than forwarding it on to the real API, it will respond instead. To make the service virtualization more realistic, it is also possible to simulate network latency, by applying delays to responses based on URL pattern matching or HTTP method.

2.9.1 Simulations

The core functionality of Hoverfly is to capture HTTP(S) traffic to create API simulations which can be used in testing. Hoverfly stores in-memory captured traffic as a JSON object, called **simulations**, that follows the **Hoverfly Simulation schema** [30]. Simulations consists essentially of request-response pairs.

2.9.2 Working modes

Hoverfly can work in different way, accordingly its **mode**. Modes

- Capture mode;
- Simulate mode;
- Spy mode;
- Synthesize mode;
- Modify mode;
- Diff mode;

Latest three modes are not used for this thesis, so they will not be described.

[30]`https://docs.hoverfly.io/en/latest/pages/reference/simulationschema.html`

Capture mode

It is an "as a proxy" mode and the Hoverfly instance is placed in the middle of a client-server application, saving simulations. When in this mode, it is checked the binary flag *stateful*, which determines the instance behaviour when there is an incoming request-response pair:

- if *stateful* is *true*, all pairs are saved, without further checks;
- if *stateful* is *false*, for each pair it is checked whether the request is not already present. It it is not, the pair is saved, otherwise is is checked a further binary flag, called *overwriteDuplicate*. If it is true, the existing pair is replaced by the new one, otherwise nothing happen;

Simulate mode

It is an "as a webserver" mode and the Hoverfly instance represent the destination server. When a new request arrives, the instance searches for a simulation matching the request. If it is not found, an error is returned with status code 502.
For choosing the simulations to return, Hoverfly has to use a **matching strategy**. Actually there is only one valid matching strategy, called **strongest match**, which consists of calculating and assigning a **matching score** to each simulation, and then the one with the highest score is selected.

Spy mode

It extends the behaviour of the *simulate mode*, with the difference that, if a matching is not found, a request to the real service is made and its response is returned to the client. When it happens, the simulation is not persisted.

2.10 Conclusions

This chapter has analyzed the current state of the art of serverless to establish the theoretical basis on which this thesis is founded. The chapter 3 will introduce the case under examination of this thesis, and will detect main components of a draft solution.

Chapter 3

Case study

The case under consideration is an B2B project, currently at a very early stage, on behalf of Consoft Sistemi S.p.A.. Further analysis and validations will be necessary in the future.
This chapter describes the units of the project on which this thesis focuses on. Section 3.1 gives a general overview about the problem and the solution idea. Section 3.2 illustrates a solution draft, and the next sections 3.3 and 3.4 describe main components involved, at a very high level. Section 3.5 divides the projects in two main parts and section 3.6 examines several scenarios, which could have business and technical consequences.

3.1 The case under consideration

Almost all companies are in a continuous evolution and development must support these changes day after day. This is especially true with companies that have adopted an Agile approach. It is not always easy to predict the direction of the business and the application development can face with unexpected changes, while the software is still expected to be resilient to these changes.
To make more robust applications, testing is necessary. Anyway it represent an unwanted and high cost, with 25% of overall project and team effort allocated to testing in this area in the region [38]. Tests must be developed and maintained and the requested effort can be important, because of the person-hours allocated, missing of analytical skills, and delays due to the frequency of releases. Moreover they're useful only if running in an isolated environment, which must be provided.

3.1.1 The idea

The Umarell project aims to automatically generate test lists on the basis of the examination of the navigation paths followed by mobile and desktop traffic. The solution is applied to B2B market, so the Umarell customers are companies that want to adopt this system in their services.
It is based on a waterfall workflow made of four steps:

1. the customer client application sends the HTTP requests toward the customer backend through a proxy;

2. the proxy parses and saves the request-response pairs;

3. the collected data are used for a big data analysis to detect common funnels;

4. funnels are given in input to an AI algorithm to generate the test lists;

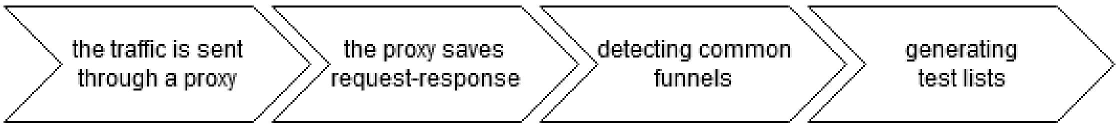
the traffic is sent through a proxy
the proxy saves request-response
detecting common funnels
generating test lists

Figure 3.1: Four steps workflow

3.1.2 The focus of this thesis

The work of this thesis focuses on designing and implementing a fully serverless environment for managing proxies, which corresponds to the step number 2 of the workflow.
It is necessary to determine how customers can handle their proxies, how the platform should manage them, how data have to be collected and more.

3.2 Designing solution

Under subscription to the Umarell platform, customers can create one or more proxies, one for each environment or behaviour they want to test. Each proxy can be personalized and associated to a global unique domain name. Setting up a proxy to make HTTP requests should be sufficient for integrating this part of the Umarell system.

3.2.1 Requirements

- Testing is perceived as an unwanted cost, since it is useful just during the testing activity itself. Therefore there is an economic driver to consider, which should result in a pay-per-use model;
- Companies may want to integrate this system in existing projects, so the system should be easy to adopt;
- Customers may want to group request-response pairs, so that they can study certain situations;
- Customers may want to test different environments and behaviours, so they should be able to create different proxies;
- A proxy should be easy to activate or terminate and it should be easy to consult or manipulate its collected data;
- Scalable, in order to create as many proxies as needed;

3.3 Proxy

Each customer can own one or more proxies, in order to be able to test different environments and services. The term "proxy" is mainly valid from the customers point of view. Indeed, a customer has to set this entity as a proxy in the HTTP communication between client applications and destination. Nevertheless, this component can be used essentially with two alternative behaviours:

- it can intercept request-response pairs of the communication;
- instead of forwarding requests, it acts as a mock server (with some nuances);

Its behaviour changes according the **mode** it is set to.

3.3.1 Modes

From the customer point of view, the proxy is a component that work in front of each HTTP request, but actually it works following different behaviours, depending on its working **mode**:

- capture-stateful
- capture-stateless
- simulate

- spy

When in **capture-stateful** or **capture-stateless** mode, the proxy parses and save request-response pair. The difference between these two modes is that, when in *capture-stateful* mode, the proxy distinguishes each request-response pair made over time. Differently, when in *capture-stateless* mode, the proxy uses the request as a key in order to check if it is already present among the previous collected pairs. This means that, even if the monitored service responds in different ways (e.g.: different HTTP status codes or a different body) to the same request, only one pair will be kept.
When in **simulate** mode, instead of forwarding the requests to the real destination, the proxy uses already stored simulations to act as if it is the real application. In practice, with this mode, the customer can test a virtualized service and simulate a certain behaviour. When a new request arrives, the proxy looks for a pair having the same request. If a match is found, then the pair is returned, otherwise an error occurs.
Finally, when in **spy** mode, the proxy behaves similar to *simulation* mode. The difference is that, if a match for a simulation having the same request is not found, then the real backend service is used as destination of a real request.

3.3.2 Key features

As it will be explained later, in the future some companies could want to host their proxies in an external location, outside the Umarell borders. Alternatively, in the future Umarell may decide to provide different types of proxies, with certain characteristics and strengths. These are possibilities that may never be realized, but taking them into account would allow to not compromise their feasibility. Therefore it is important to define some key features that a proxy should always respect:

- it should be plug and play;
- it should have minimal responsibilities;
- it should be independent and isolated;

Plug and play

In order to be set in HTTP calls, a proxy has to be a network resource, with a public IPv4 address and a port. As one of the requirements is the pay-per-use model, this resource has to be allocated and destroyed as needed, so a proxy must be **plug and play self-contained**.

Minimal responsibilities

As in the future it may be possible that a proxy is outside of the Umarell control, a proxy should have **minimal responsibilities**, in order to prevent abuses over the application logic. In that situation, a proxy could be implemented in a different way, rely on a customized environment and even behave differently, but everything it could affect would be the data collected, which have to be realistic in the interest of the customer.

Independent and isolated

Each proxy should be **independent and isolated** from the others, even if belonging to the same customer. Each one is thought for testing a specific scenario, so the results must be clear and indisputable. Nevertheless, data collected by a proxy may be transferred outside it, in permanent storage. So the isolation has not to be intended as if nothing can enter or exit, but only as a characteristic of non-conditioning from other components. From a more technical point of view this means also that a proxy should be able to accept certain type of requests only by allowed sources.

3.4 Simulation

A request-response pair collected by a proxy is called **simulation**. The behaviour that allows customers to capture simulations and then use collected data for simulating the service is called *service virtualization*, which has been detailed in section 2.8. Simulations are important because they represent the essential unit of this methodology.
When the proxy is in *simulation* mode and a request arrives, it have to look for a simulation having the same *request*. Evaluated fields are:

- scheme, i.e. HTTP or HTTPS protocol;
- HTTP method;
- destination;
- path;
- query parameters;

A response is described essentially by two fields:

- HTTP status code response;

- body, which can be in various formats, i.e. text, HTML, XML, JSON;

Then, as the customer must be able to have a fine-grained control over simulations, a global unique identifier is used for describing a simulation.

3.4.1 Scenarios

The period in which a proxy is running, which means that there is a network resource allocated, is called **testing session**. The simulations collected during this period, ordered as they arrived, form a **scenario**. This concept is important because it allows the customer to test certain situations. For example, a customer may want to reproduce a sequence of requests that has led to certain erroneous situations.
While a proxy is used for testing specific environments, tenants or services of an application, a scenario is used for studying data in the given context.
A scenario can be saved automatically at the end of the current testing session, when the proxy is turned off. Then it can be associated to a name and have a global unique identifier.

3.5 Two units

The Umarell project can be divided in two distinct units, which are the **Umarell Backend** and the **proxy farm**.

3.5.1 Umarell Backend

It is the core of the Umarell application, because it contains:

- an API from which customers can perform CRUD operations on their proxies, simulations, and scenarios;
- a storage in which information are persisted permanently, including data about customers;
- a logic which orchestrates all the involved actors;

In practice, this is the component which customers use for interfacing with the system, therefore any administrative task must be launched through the API.

3.5.2 Proxy farm

It is called **proxy farm** an environment in which proxies live, providing them network connectivity and security.
When a customer uses the API provided by the Umarell Backend to perform an action which affects proxies, the proxy farm is involved. It provides some endpoints which the Umarell Backend can use in order to apply changes on proxies.

3.6 Placement of the proxy farm

Customers have several possibilities for placing their proxies. For each one will be examined pros and cons. Further details, which involve implementation aspects, can be found in sections 4.4.5 and 4.7.

Configuration with the Umarell Proxy Farm

Customers can use a unique shared proxy farm, provided by Umarell, and called **Umarell Proxy Farm**. This solution is currently implemented, as the project is in a early stage, but it could be discarded when Umarell will be launched to the market.
Pros:

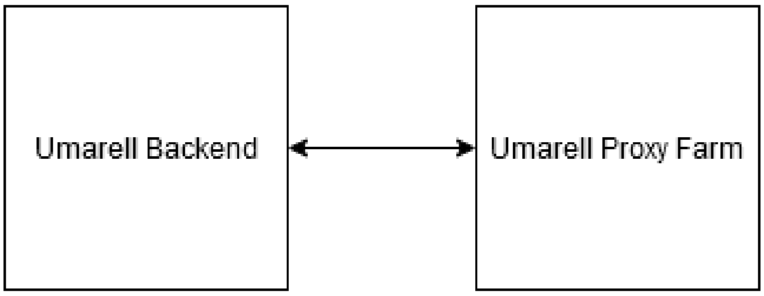

Figure 3.2: Configuration with Umarell Proxy Farm

- Umarell is a SaaS fully-managed platform. Customers only have to create and use their proxies. No extra effort is required, neither monitoring;
- everything is controlled by Umarell, so the communication layer between the Umarell Backend and the proxy farm can be kept simple;
- depending on the implementation and commercial evaluations, launching a proxy could be faster, as this proxy farm could be active for a longer time;

Cons:

- if the client has special needs, it could be a solution with limitations;
- this solution could require some fixed costs, and this doesn't fit with a pay-per-use model;

Configuration with multi-account proxy farms

Each customer have an intermediate isolated space for running proxies. This solution is the one that will be used for the market launch, even if it needs to be analysed in major detail. For example, it is not yet clear how many proxy farms a single customer can own and this depends on commercial evaluations.
Pros:

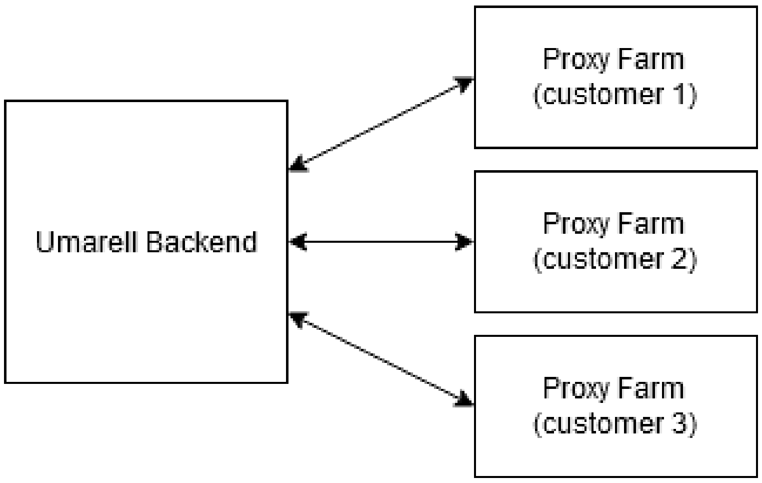

Figure 3.3: Configuration with multi-account proxy farms

- customers have additional flexibility in customizing their environment. Limitations are not yet clear;
- the solutions completely fits with a pay-per-use model;
- Umarell can provide some templates acting like blueprints of the infrastructure (Infrastructure as Code);

Cons:

- handling a subscription requires an addition effort, which should be minimal, although it is not yet well defined;
- customers are required to monitor, even if Umarell may preset monitoring tools;

Configuration with external proxy farms

Further insights and market researches in the future could reveal the desire of customers of managing proxies on their own, keeping them outside the Umarell boundaries, in an external proxy farm, which could be on-premise or a cloud provider. This solution is not and may never be implemented. Nevertheless, during the implementation phase, the possibility described above will be taken into account, in order to take decisions which would not compromise its feasibility.
Pros:

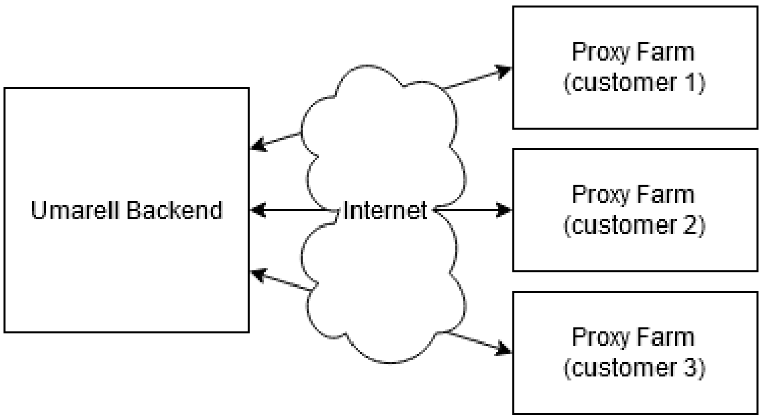

Figure 3.4: Configuration with external proxy farms

- a proxy can be built using any technology, that can behave differently, even better, than ones provided by Umarell;
- the general behaviour could change; e.g., the proxy could perform some extra operations when it captures a new request-response pair;

Cons:

- Umarell could have to provide multiple IaC templates for building a proxy farm in different environments, depending on business requirements. In that case more varied skills would be required;
- no benefit from outsourcing, because the customer must take responsibilities at different levels: from application level - including storage of data, users access, etc. - down to the physical level;
- if something changes in the communication layer between the Umarell Backend and proxy farm, the customer may have to adapt to these changes, which requires an effort. The problem could also exist for Umarell, because it may

be necessary to version the APIs to lighten the customer workload and avoid error scenarios for customers. In any case, customers must be careful to always use stable and not deprecated versions;

- Umarell should pay more attention to integration tests, since the communication layer becomes more complex and shared among different parties;
- monitoring is entirely up to customers;

Note that no consideration about costs has been made. This is a very preliminary study phase, so it is not possible to say if and where costs could be saved, neither for Umarell, nor for the customers. For example, with a customer's proxy farm, Umarell does not have to pay for resources allocated for proxies, but it could result into a different pricing or commercial proposal. Moreover the effort requested to development and DevOps teams could be very different and unpredictable. It is not possible to say anything even about the costs for customers. In their proxy farms, they have to pay for resources they consume, even if idle. But it could be convenient if they exploit their idle servers, or a cheaper space to host their proxies.

3.6.1 Flexibility

Proxy farms in multi-account configuration are designed to be created and destroyed with agility, in order to not consume resources. When a customer wants to use the Umarell platform, first of all the proxy farm is created. When it is up, then proxies are launched. Figure 3.5 shows an example in which all the customers have their account, which is a private space they can manage. In this case *customer 1* and *customer 2* have their proxy farms active, with some proxy inside. They will be charged for resources they're consuming. It can be supposed which *customer 1* will pay more, due to the higher number of proxies used. Instead, *customer 3* has not a proxy farm active, so there is no cost.

3.7 Conclusions

This chapter has illustrated the case study and the key component and concepts of a draft solution. The chapter 4 focuses on the production of the artifact which implement the described case study. The application will be designed, the technical stack and some implementation aspects will be analyzed.

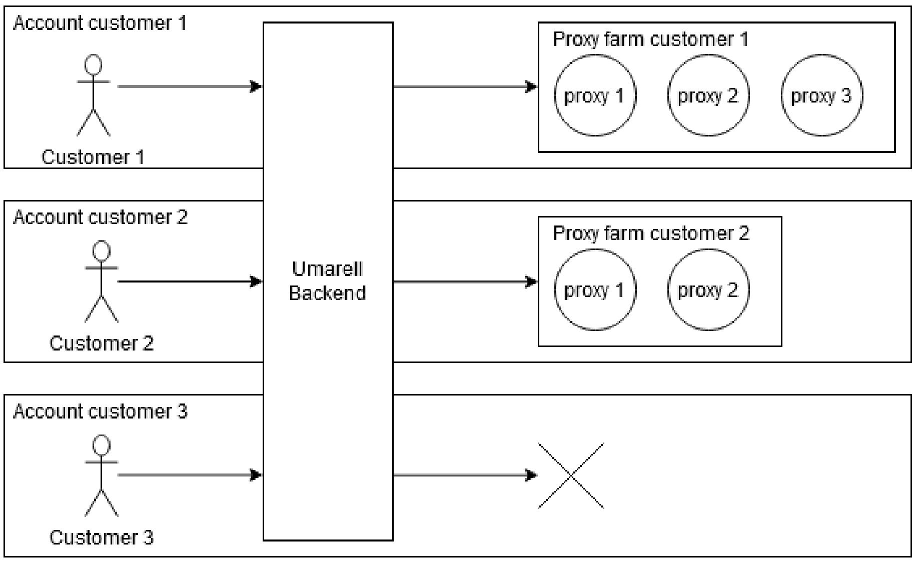

Figure 3.5: Proxy farms flexibility

Chapter 4

Implementation

This chapter describes the key point of the implemented solution, starting from minimal technical requirements, in section 4.1. Section 4.2 illustrates the architecture principles and the main components which participate. Then the REST API for customer will be exposed in section 4.3. Section 4.4 goes down to a lower level, examining the technical stack. Section 4.5 shows how the project has been organized. Sections 4.6 and 4.7 and 4.8 illustrate some application processes. Section 4.9 explains some detail for security and networking. Sections 4.10 and 4.11 deepen aspects concerning Infrastructure as Code and the construction of a DevOps pipeline.

4.1 Requirements

- Easy to manage: customers should be able to create, launch and stop proxies with minimal effort;
- Easy to maintain, with minimal operation tasks;
- Payment per usage model: customers are not charged for resources they don't use;
- Proxy farms can be created and destroyed flexibly;
- Security by design and by default;

4.2 Architecture overview

According to requirements, a pay-per-use model is what customers would have. Moreover, the solution needs to be scalable and with a low maintenance needing.

A serverless architecture perfectly fits with these requirements, since it has auto-scaling features built-in and easily adopts a pay-on-demand pricing model, so that customers only have to face the costs related to the resources they consume. This also means that if customers are not using their proxies, there must be no resource allocated that requires cost, therefore some resources may need to be created and destroyed dynamically. To take full advantage of it, it will be exploited the elasticity, reliability, and agility of a public cloud provider.

SaaS

If the Umarell Proxy Farm is used, everything customers need is provided by Umarell. In this case customers have only to use the Umarell platform by mean of a graphic interface, i.e. a web app which exploits the API provided by the Umarell Backend behind the scenes. In practice customers can enjoy the benefits of a fully SaaS solution. Instead, if customers want to use their proxy farms, they need to build and manage a custom solution which is compliant to a possible standard defined by Umarell. Therefore, in that case the solution would be a SaaS just in part. Finally, in the multi-account solution, customers may exploit a template proposed and provided by Umarell, to use as a blueprint for building the infrastructure. Customers have a working solution that is ready to build, but they still have monitoring responsibilities.

Components

Main components of the implementation are:

- Proxy;
- Umarell Backend;
- Proxy farm;
- Communication layer;
- Testbox;

4.2.1 Proxy

A **proxy** can be intended as a virtual network device made available for a single customer. It must be referenced in HTTP requests, so it must be reachable over the internet through a public IPv4 address and a port. This means which a proxy is an entity that consumes resources and this involves costs. Since customers have to pay per usage according to requirements, a proxy must be created or destroyed when the customer requests it.

A proxy is described essentially by a name and a working mode, but in the future there may be other attributes. Furthermore, a proxy collects simulations. All these data are interesting in the long run, because customers may want to use the same proxy in different moments. In practice, the proxy is a stateful component, and its state must not be deleted after use, but persisted in a permanent storage, like a database.
For each proxy a name, a mode, and a status must be stored. The status can be used to determine if a proxy is running or not. Also simulations and scenarios data must be stored.

REST resource

Each proxy is described by several fields:

- *name*: it is the name of the proxy and it is specified during the creation of the resource;
- *id*: it is an alphanumeric token which is built by appending a global unique integer identifier to the name of proxy, in order to grant uniqueness in the system;
- *mode*: it is the working mode of the proxy and can assume one value among *capture-stateful*, *capture-stateless*, *simulate* and *spy*. Their meaning have been described in the previous chapter and it will be described later in this chapter;
- *status*: it is the current status of the proxy and can assume one value among *creating*, *running*, *stopping*, *stopped*, *updating*. They're self-descriptive, but it is interesting to note the presence of temporary statuses, such as *creating*, *stopping*, and *updating* statuses, which hint that some operations are not immediate;

4.2.2 Umarell Backend

The **Umarell Backend** is the core of the whole system, being responsible for managing data about proxies, simulations, scenarios, customers, and more.

Event-driven

This part of the application is implemented on a serverless architecture which strongly relies on FaaS paradigm, which is event-driven by definition. An event is the representation of a situation which occurs under certain conditions and which is monitored, in order to take some actions. When an event is intercepted, an information representing it is pushed to a queue. Then, an event listener, which

continuously looks at this queue and waits for new events, triggers a specific callback function.
This mechanism can be summarized in three steps:

1. the customer performs an action, which is registered into a store;
2. the update of the store triggers another action, in order to react;
3. when the latter action finishes its job, it gives a feedback about the operation;

This is the behavior behind the proxy management. First, there is an API service, which exposes the endpoints for implementing a CRUD for proxies. The arrival of a request for the creation of a new proxy is an event, for which the specific function for the creation of the proxy is invoked. This function stores proxy data in a database table. The creation of a new record is itself another event, which triggers another function that is in charge of creating - indirectly - the virtual network resource associated with the proxy, that has to be launched in a proxy farm.
Instead, if a proxy needs to be stopped, there will be another API endpoint which triggers a function which change the field *status* for stopping, and this change invokes another function which destroys the virtual resource in the proxy farm where it is running.

Client application

Requests for creating or stopping proxies have to be handled by customers through the Umarell Backend API, but, in order to provide a better user experience, a client application will be provided in the future. It could consists of a GUI served via web and from which it would be possible to perform any administrative task, like creating new proxies, modifying existing ones, starting or stopping them and more.

4.2.3 Proxy farm

A **proxy farm** is the execution environment where one or more proxies run. It is responsible to provide all the network facilities in which proxies can live, and more, including all the needed security measures, like firewall rules, ACLs, and permission policies.
Proxies must run in isolation, because they must not be able to influence each other, even if they may run in the same virtual subnet. It is due to security and privacy reasons, but also to not compromise the application logic.
Role of the proxy farm is creating a safe environment for proxies: it should have a virtual private cloud with at least one public and one private subnet. In the public one there are resources that need to be accessible through the Internet, whereas

in the private one there are resources that don't need to be exposed. A proxy must be directly accessible, because it has to be used in HTTP calls of customer's applications, so it is placed into the public subnet. Nevertheless, administrative actions must be taken through the Umarell Backend, in order to have a fine-grained control on the overall state of the system.
In conclusion a proxy farm should be able to receive commands (e.g., for starting and stopping proxies) and react by triggering some actions (e.g., for reporting that a proxy has been successfully started).

4.2.4 Communication layer

Any administrative action that a customer wants to make, must be taken by interacting directly or indirectly with the API provided by the Umarell Backend. If the action involve allocate resources which represent a proxy, it needs to interact with the proxy farm in which the proxy is hosted.
Therefore it is essential to provide a communication layer, that should have certain features. Being the system based on serverless, synchronous communication should be avoided, because it is blocking and resource-consuming, therefore it is costly. Instead, asynchronous communication should be favoured, therefore both parts should expose an API to the other, so that each one can reach the other at any time. Moreover, in the future, the proxy farm could be external to Umarell borders, so the two main part of the platform should use an agnostic API to communicate. This communication layer consists of REST APIs (over HTTP), as shown in figure 4.1. It can be seen in more detail in figures 4.3, 4.4, 4.5:

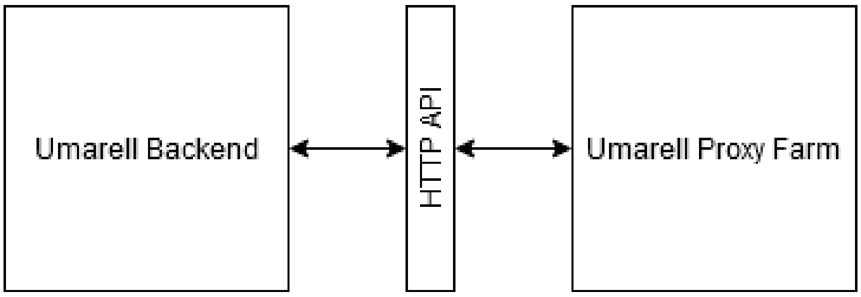

Figure 4.1: Communication layer between the Umarell Backend and a generic proxy farm

4.2.5 Testbox

Customers don't need to be aware about the proxy farm, or that their proxies are running alongside those of others. They only need to know, for each proxy, where it

is or has to be executed. This information is made essentially of an IP address and a port, but it can be made of other data, as it will be mentioned in section 4.4.4. For this reason the concept of **testbox** is introduced. This information is distributed between the Umarell Backend and the Proxy Farm. The ID of the testbox must be unique among all the testboxes, so it must be generated by the Umarell Backend, which centralizes data. The ID will be used as query parameter in a HTTP requests which uses the API URL, to reach a certain proxy inside the Proxy Farm. On the other side, the Proxy Farm uses the generated ID as key in a key-value cache to maintain data about the exact location of the proxy inside the environment, which is transparent of the Umarell Backend. This configuration allows maximum flexibility in implementation of proxy farms and proxies themselves.
A proxy farm is a concept which lives at a lower level. It may contain a network with multiple running proxies which share resources, functions, and so on. A testbox is an abstraction of it and represents a toolkit containing all the facilities necessary to a single proxy in order to run. It gives the impression that each testbox is an isolated environment, containing virtual replicas of those functions contained in the proxy farm. To be clearer, the proxy farm is more significant at the infrastructural level, whereas a testbox is closer to the domain data.

REST resource

Each testbox is described by several fields:

- *id*: a global unique identifier;
- *ip*: the public IPv4 address which identify the network resource associated with the proxy;
- *port*: the port used along with the IP. Currently it is fixed (the 8500 is used);

- $api_url : isthebaseURLtouseforreachingtheproxyforadministrativepurposes.Inpracticeitisassocia$

4.3 Customer REST API

Customers can hendle their proxies and data through a REST interface. This architectural style has been chosen because it is very spread (it is easier to find skilled people and it is encouraged by many providers), for its consolidation in web environments, and because the organization behind Umarell is already experienced. In the first place an API specification has been defined, in order to write APIs with the following characteristics:

- be RESTful;

- use JSON in request and response body;
- favour automatic serialization and deserialization for marshalling and unmarshalling;
- simple, with few concepts easy to follow;
- flexible, with few limitations due to semantic constraints, in order to be able to withstand the continuous changes of Agile methodologies;
- consistent representation of the data;
- correct use of HTTP status codes and HTTP verbs;
- error handling at application layer;

Defining a specification adds constraints which helps developers to have a reference for an organized implementation. The following class diagrams illustrates how data are related at an high view:

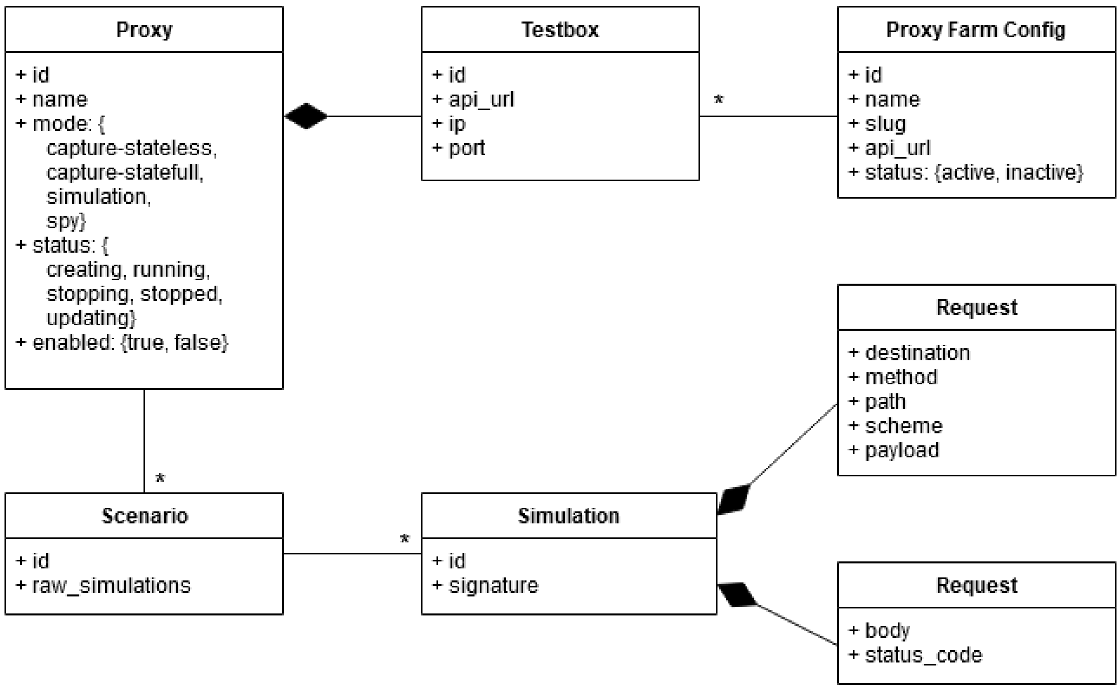

Figure 4.2: UML class diagram

Method	**Path**	**Body**	**Response**	**Status**
POST	/proxies	object	object	202, 400, 405
GET	/proxies		array of objects	200, 400, 405
GET	/proxies/{id}		object	200, 400, 405
PATCH	/proxies/{id}	object	object	200, 202, 400, 404, 405
DELETE	/proxies/{id}			204, 400, 404, 405
GET	/proxies/{pid}/testbox			200, 404, 405
GET	/proxies/{pid}/scenario			200, 404, 405
PUT	/proxies/{pid}/scenario	object	object	202, 400, 404, 405
DELETE	/proxies/{pid}/scenario			204, 400, 404, 405
GET	/scenarios/{id}		object	200, 404, 405
DELETE	/scenarios/{id}			204, 400, 404, 405
GET	/scenarios/{id}/simulations		array of objects	202, 404, 405
GET	/scenarios/{id}/simulations/{id}		array of objects	202, 404, 405

4.4 Technical stack

There is an economic driver among defined business requirements and it leads to prefer a serverless architecture, as this model allows to decrease operational costs. Reduction of costs is reflected on the pricing for customers. Moreover, if a customer is not using a proxy, then no resource is allocated, therefore there are no costs at all. To take full advantage of serverless cost saving, a public cloud is more convenient. In 2019 market share was contended as follow [39]:

- AWS, 32.3%;

- Microsoft Azure, 16.9%;
- Google Cloud, 5.8%;
- Alibaba Cloud, 4.9%;
- Others, 40.1%;

Amazon Web Services (AWS) has been chosen, being a leader in the public cloud provider market, with annual sales close to $34.9 billion [40]. Amazon has been the first player to offer a FaaS solution like AWS Lambda, which was used by 70% of serverless users at the end of 2017 [41]. Lambda is constantly evolving, trying to solve any kind of problem, such as cold start (through *provisioned concurrency*). Furthermore, AWS offers solutions that matches with the Umarell needs, such as advanced serverless solutions - like containers deployment and databases -, networking, world-wide datacenters in which deploy and run applications, for scalability and high availability, and more.
Finally, the decision has been also driven by the organization behind Umarell, already experienced in AWS.

4.4.1 Umarell Backend

As already said, the Umarell Backend is the core of the whole application. Indeed, each action in the system is taken on the basis of modifications on existing tables. First of all, the API service is provided by AWS API Gateway, which allows to define routes, HTTP methods, parameters, and other. When a request arrives at a certain endpoint, a specific Lambda function is invoked to respond to that HTTP event. It's the case of functions for creating, modifying, reading and deleting proxies. Their implementation is pretty clean, consisting of simple modification of a table named *proxies*, but complex actions are still required. For example, when a proxy is created, a correspondent virtual resource has to be created and launched in a proxy farm. For this reason the *proxies* table has DynamoDB Stream enabled, which allows to invoke a function whenever a change in the table is detected.
Then AWS IAM is used in order to defines roles and policy, which assign specific permissions to each Lambda function.

4.4.2 Proxy

The proxy's implementation is based on Hoverfly, an open source product which is able to be used in the middle of a client-server communication for capturing HTTP request-response pairs. Each proxy corresponds to an Hoverfly instance and it is logically located between the caller who makes HTTP requests and the destination,

which is the service to test.
Hoverfly provides two ports:

- one for administration, 8888 by default;
- one for proxying, 8500 by default;

It will be examined later, in section 4.8.

4.4.3 Proxy farm

The proxy farm plays a key role, because it is responsible of networking and security of proxies. To provide this safe environment, it uses several services.

AWS Fargate

Hoverfly is not an AWS service, so it needs some auxiliary services to run in the AWS cloud. It is convenient to use containers, in order to benefit from an isolated and easy to deploy environment. The more sophisticated way to do it in AWS is to use one of the following services: **Elastic Cloud Service (ECS)** or **Elastic Kubernetes Service (EKS)**. They're very similar solutions for orchestrating Docker containers, but the first one is a native Amazon solution, while the second one is a managed service to run Kubernetes clusters. The choice fell on ECS, since it is better integrated with other Amazon services and it is a little bit cheaper than EKS, since the latter has some extra costs.
ECS, as well as it would have been with EKS, it is possible to run containers using two different approaches. The first one is **EC2**, that is a virtual machine which needs to be provisioned and managed. The second one is **Fargate**, which allows to run containers without any provisioning or extra operations. In practice, Fargate is a serverless solution for running containers.
For running Hoverfly in Fargate, it will be used a Docker image, that has already made available by SpectoLabs on an image registry.

Application

Some Lambda functions are defined in this project in order to receive commands. One of these is the function *RunProxy*, which is invoked from the Umarell Backend for launching a new proxy. Along with the request, some parameters are passed, like the ID and the working mode of the proxy. Then a new Fargate task is launched on ECS, by using the *task definition* specified as environment variable of the function. This operation requires some time (in the order of seconds), so, when this function terminates, the task is in state *PROVISIONING*. It is the configured an AWS Event Bridge rule, which invokes the function *ProxyFeedback* when the task has both parameters *lastStatus* and *desiredStatus* equal to *RUNNING*. The triggered function the invokes the function *UpdateProxy*, placed in the Umarell Backend and previously described.

Networking

Each proxy must have a public IP address and a port through which it can be reached over the Internet by customers. The aforementioned public IP address is the one that customers have to set as proxy in their HTTP requests.
Since they must be reachable through the Internet, proxies are placed in a public subnet in the proxy farm's VPC.

Security

Security of containers is handled by using some features available in AWS. First of all it is used **AWS VPC**, which allows to have a logically isolated section of the AWS cloud where AWS resources can live. AWS VPC offers a virtual network that can be configured to benefit from networking and some security features, like **Security Groups**, which are virtual stateful firewalls that are able to set inbound and outbound rules for controlling traffic.
Hoverfly must be reachable from the Internet on the port 8500, because it has to be used for proxying in the customer's requests. The port 8888, instead, it is used only for administrative purposes of Hoverfly. Using the latter port from the Internet must be denied, since any administrative operation should happen through the Umarell platform.

VPC quotas

In order to design a suitable solution, it is necessary to understand the AWS VPC quotas [1]. In particular, for each region, only five VPC per account can be created, each one containing at maximum two hundreds subnets. Moreover it is possible to have only five Elastic IP per region. These limits are definitely binding, because without them each proxy farm could have been built with a dedicated VPC, with its own Lambda function and API. Therefore each customer could have benefited from a dedicated subnet, with a major isolation, but with some drawbacks, as mentioned in section 5.2.1.

[1]`https://docs.aws.amazon.com/vpc/latest/userguide/amazon-vpc-limits.html`

4.4.4 Testbox

In more elaborated scenarios, in the future, more subnets, VPCs, ECS clusters, or even AWS accounts, could be used. The Umarell Backend must always be able to reach a proxy, even if it is not fully aware of the infrastructure it is running on. A testbox is a container of all the necessary data about where the proxy is running. As already said, this information is distributed. The Umarell Backend contains:

- *id*: a global unique identifier;
- *testbox_api_url*: the URL of the testbox in which the proxy is running. Currently it coincides with the URL of the proxy farm;
- public_ip

The Umarell Proxy Farm contains:

- *cluster_arn*: the ARN of the ECS cluster;
- *task_definition_arn*: the ARN of the task definition, which is a blueprint common to all the tasks. It defines the Hoverfly image and the port mapping;
- *task_arn*: the ARN which uniquely identify the task in the cluster;
- *task_public_ip*: the public IPv4 of the task, which is used by customers;
- *task_port*: the port of the task, which is used by customers for proxying activity;
- *task_private_ip*: the private IPv4 of the task, which is used by resources which have access to the VPC;
- *subnet_id*: the subnet identifier of the VPC in which the task is running;

In both parts, these data are collected in DynamoDB tables. The difference is that the Umarell Proxy Farm is a volatile unit, because it can be destroyed along with the table, at a certain point. Therefore the DynamoDB table can be referenced as a cache containing information about the running proxies and which strictly depends on the environment (AWS in the case of Umarell Proxy Farm). In case of external proxy farms, these data are going to change if AWS is not used.

4.4.5 Placement of proxies

The configurations for placing proxies are illustrated in major detail below. Arrows represent requests, if they are continuous, and responses, if they are dashed.

Configuration with the Umarell Proxy Farm

The configuration with the Umarell Proxy Farm is shown in figure 4.3. Units communicate through their APIs.

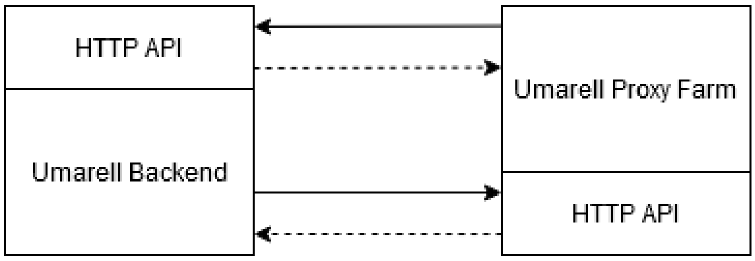

Figure 4.3: Configuration with Umarell Proxy Farm

Configuration with multi-account proxy farms

The possibility of assigning a proxy farm to each customer is enabled by the cloud. This configuration puts in evidence the great advantage that comes along the cloud, which is the flexibility in allocating and deallocating resources with the minimal effort.
The cloud allows to use a text file for describing an entire infrastructure, i.e. Infrastructure as Code, so the proxy farm infrastructure could be easily reproduced through a parameterized template file. Therefore customers could have their proxy farms in separated AWS subscriptions. this multi-account configuration is show in figure 4.4. For simplicity only requests have been shown. Pros and cons of this

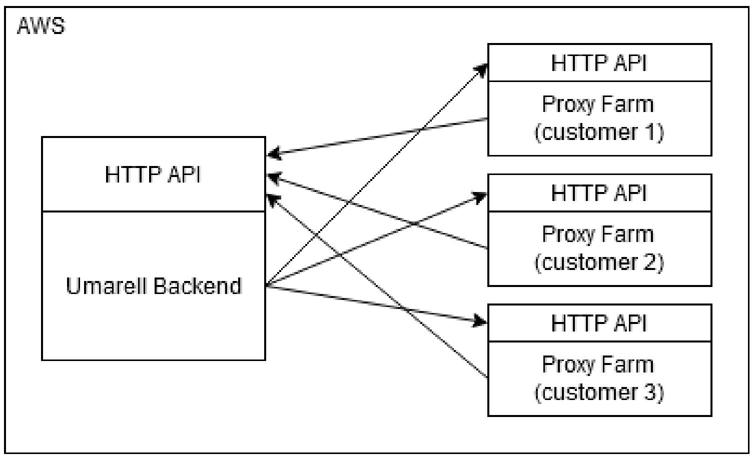

Figure 4.4: Configuration with multi-account proxy farms

solution have already been exposed in section 3.6. Further benefits due to the technical stack are listed below:

- the proxy farm (the testbox) can be created and destroyed on the basis of the customer real needs. It is not necessary to keep it alive if no one is using it;
- using the Umarell Proxy Farm should already ensure isolation, anyway using different subnets, VPCs and accounts gives an higher grade of isolation from other proxies and customer activities;
- it is possible to calculate the exact costs due to each user, as there is a very high traceability of expenses, because each proxy farm is associated to a single customer;
- some components, like ECS, have some quotas which could affect scalability [2], e.g. each cluster can contain at maximum 2000 container instances. Even if these quotas should be already difficult to reach, this solution makes reaching these limits even more difficult. Therefore this solution has a better scalability;

Configuration with external proxy farms

In the latest possible configuration customers can place their proxy farms anywhere (even in AWS). Anyhow they have to expose an API which adheres to a standard defined by Umarell. This configuration is show in figure 4.5. For simplicity only requests have been shown.

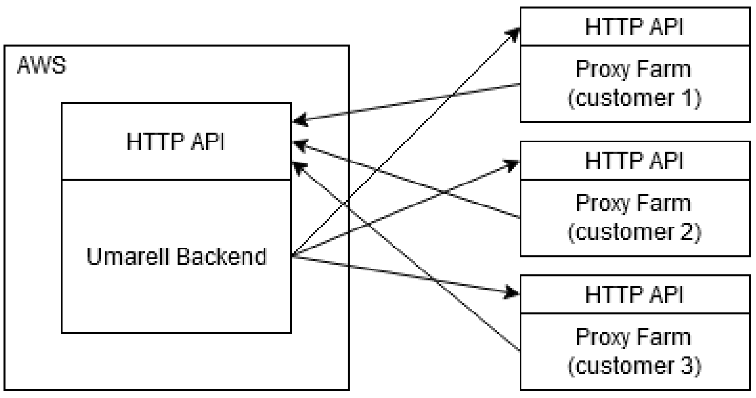

Figure 4.5: Configuration with external proxy farms

[2]https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-quotas.html

4.5 Projects organization

As already said, Umarell has been divided in two macro areas which are essentially autonomous, but only need to communicate, so each part can call the functions of the other. This independence brings a lot of benefits, like explained in section 5.1, and allows to keep the two parts in two different Git repositories, hosted on AWS CodeCommit. There is a further source location, which is DockerHub, an external image registry, and it is used to pull the Hoverfly image from the address *spectolabs/hoverfly*.
The code has been written in **Python** and consists mostly in functions for AWS Lambda. Designed functions involve less code as possible, in order to minimize the size and speedup the Lambda instance. Common libraries and utilities have been placed in Lambda layers, so that sharing them is easier.

4.5.1 Umarell Backend project

It is a Serverless Framework project, so there is a configuration file, called *serverless.yml* that is the template that describes the serverless architecture.
This part of the application is pure serverless. It makes a large use of FaaS and event-driven paradigm. Essentially customers have an API, based on **AWS API Gateway** [3] and each endpoint is mapped to an AWS Lambda function. This is the ideal use case for **Serverless Framework** [4]. All the Lambda functions have been defined in the folder *functions*. Each file contains only one handler. This helps to have a unique and clear workflow per file, to subdivide the handler function in modules and to manage smaller functions, increasing maintainability.These functions are:

- **ReadProxy**, **CreateProxy**, **ModifyProxy**, and **DeleteProxy** that represents the CRUD for a proxy;
- **ReadSimulation**, **CreateSimulation**, **ModifySimulation**, and **DeleteSimulation** that represents the CRUD for a simulation stored in the database;
- **ReadScenario** and **LoadScenario**, which allow to read a scenario (from the database) or load a previously stored scenario on the proxy, that is the remote Hoverfly instance;
- **ProxyWatcher** is triggered whenever a change is detected on the table *proxies* and, on this basis, certain actions are taken;

[3]`https://aws.amazon.com/api-gateway/`

[4]`https://www.serverless.com/`

- **UpdateProxy** is invoked - through the Umarell Backend API - by proxy farms for giving feedbacks about the Fargate tasks;

The key *functions* in the template specifies the list of functions that needs to be created. For example:

```
functions:
  # ...
  CreateProxy:
    handler: functions/api/proxy/create.handler
    role: CreateProxyRole
    package:
      exclude:
        - ./**
      include:
        - functions/api/proxy/create.py
        - proxy/**
        - storage/**
        - utils/**
        - api_builder/**
      individually: true
    events:
      - http:
          path: proxies
          method: post
```

The nested key *events* specifies that the function must be invoked when a POST request to the path */proxies* arrives. In practice this HTTP request is linked to a Lambda function call. The key *handler* specifies the path in the project of the function which has to respond to the request.
The key *package* establishes files and folders that has to be included into the package, the deployable unit of this Lambda function. The term function may not be entirely appropriate, since it can be a set of libraries, classes, assets, and more. For each function the package has been specified, in order to include only the essential.

4.5.2 Umarell Proxy Farm project

The entrypoint file is named *app.py* and it mainly instantiates the stack for CloudFormation inherent to this part of the platform. The stack is described in a OOP class which sequentially declares all components of the infrastructure. CDK allows to do it by facilities based on abstract factory, which brings advantages on ease of use and development speed.
For a greater separation of responsibilities, proxies - therefore Fargate tasks and

Hoverfly instances - are handled only by Lambda functions defined in this project. These functions receive some parameters from the Umarell Backend (more precisely, from an event), take some environment variables and then operate on the resources:

- **RunProxy** receives a proxy ID and a proxy mode, then runs a task with the corresponding Hoverfly mode;
- **StopProxy** receives the task ARN of a proxy and then stops the corresponding task;
- **ProxyIsRunningFeedback** is triggered by an Event Bridge rule when the task has been started. Then it notifies the Umarell Backend about the event using its API;
- **ProxyHasStoppedFeedback** is triggered by an Event Bridge rule when the task has been stopped. Then it notifies the Umarell Backend about the event using its API;
- **GetProxySimulation** retrieves the private IP of the proxy by the event, then it uses the Hoverfly API to get in-memory stored simulations;
- **LoadProxySimulation** retrieves the private IP of the proxy and a set of simulations by the event, then it uses the Hoverfly API to load simulations in the instance;

4.6 Asynchronous workflows

To exploit the advantages of serverless and cloud environments, asynchronous techniques are used. In particular, the Umarell application relies completely on the even-driven paradigm.

4.6.1 Creating, starting and stopping a proxy

A snippet in *serverless.yml* configures DynamoDB Stream to trigger the Lambda function *ProxyWatcher* whenever a change on the table *proxies* is detected:

```
functions:
  # ...
  ProxyWatcher:
    handler: functions/proxy_watcher.handler
    role: ProxyWatcherRole
    package:
      exclude:
        - ./**
      include:
        - functions/proxy_watcher.py
        - proxy/**
        - scenario/**
        - testbox/**
        - simulation/**
        - utils/**
        - storage/**
        - api_builder/**
      individually: true
    events:
      - stream:
          type: dynamodb
          arn:
            Fn::GetAtt:
              - ProxiesTable
              - StreamArn
```

The function receives the event as argument and is able to retrieve the event type, called *eventName*, whose value is checked in order to behave differently according to it:

```
stream_source = event['Records'][0]
event_type = stream_source['eventName']
stream = stream_source['dynamodb']

if event_type == 'INSERT':
    return handle_insert(stream)

elif event_type == 'MODIFY':
    return handle_modify(stream)
```

When the event type is *INSERT*, it means that a new proxy is being registered and *ProxyWatcher* calls an endpoint of the Umarell Proxy Farm API, which invokes the lambda *RunProxy* in the Umarell Proxy Farm. *RunProxy* launches a Fargate task into the ECS cluster of the proxy farm. This operation requires some seconds. When the task is created, it passes through several phases, in which its status is initially equal to *PROVISIONING*, and then becomes *PENDING*, and finally *RUNNING*. An AWS EventBridge rule has been defined in the Umarell Proxy Farm, which triggers the lambda *ProxyIsRunningFeedback* whenever the attributes *lastStatus* and *desideredStatus* of the event are both equal to *RUNNING*. Therefore, this function calls an endpoint of the Umarell Backend API, which invokes the lambda *UpdateProxy*, passing some parameters about the launched task, more precisely:

- the base URL of the testbox API;
- the public IPv4 and port of the proxy;
- the proxy ID, which is necessary because all the steps are asynchronous and stateless, so the response is decoupled from the request;
- the status of the task, which is a summary of attributes *lastStatus* and *desiredStatus* of the ECS event.

The lambda *UpdateProxy* use the received proxy ID to find the proxy that has been referred in the table *proxies* and then updates its testbox configuration with the IPv4, port and API URL, and set the field *status* equal to *running.* The workflow is illustrated in figure 4.6.

The sequence diagram of the creation process is shown in figure 4.7. It has been

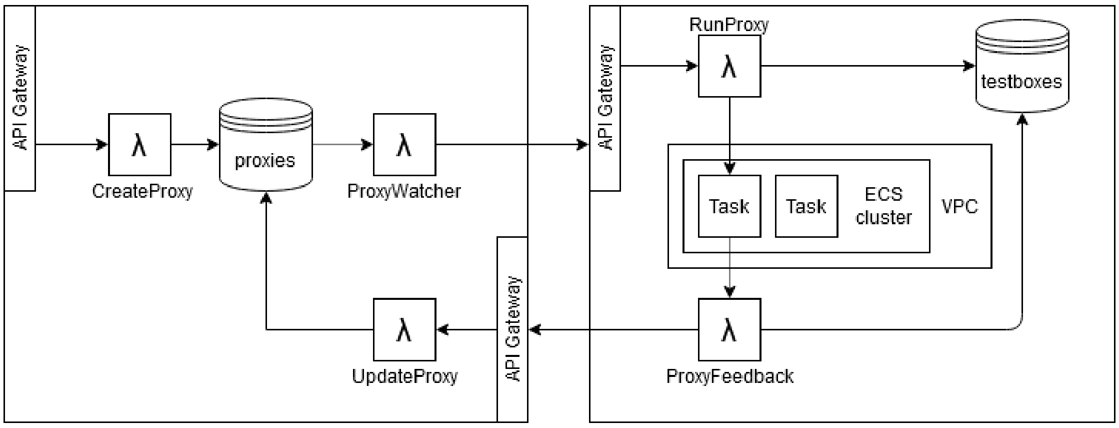

Figure 4.6: Workflow for creating a proxy

simplified to bring out the most important passages. For example, logging and tracing activities have been omitted (especially for Lambda and API Gateway services). Then, the diagram considers a multi-account scenario, so AWS services have been grouped by account.

When the event type is *MODIFY* and the status of the proxy has changed, the new value of this property has to be checked, to understand what kind of change it is: a proxy being stopped, an existing proxy being started, a change of mode, a modification of the proxy fields, etc. For this reason, it is necessary to have both the new and the old values stored in the table. In order to do it the DynamoDB Stream has been configured with the value *NEW_AND_OLD_IMAGES* in *serverless.yml.*

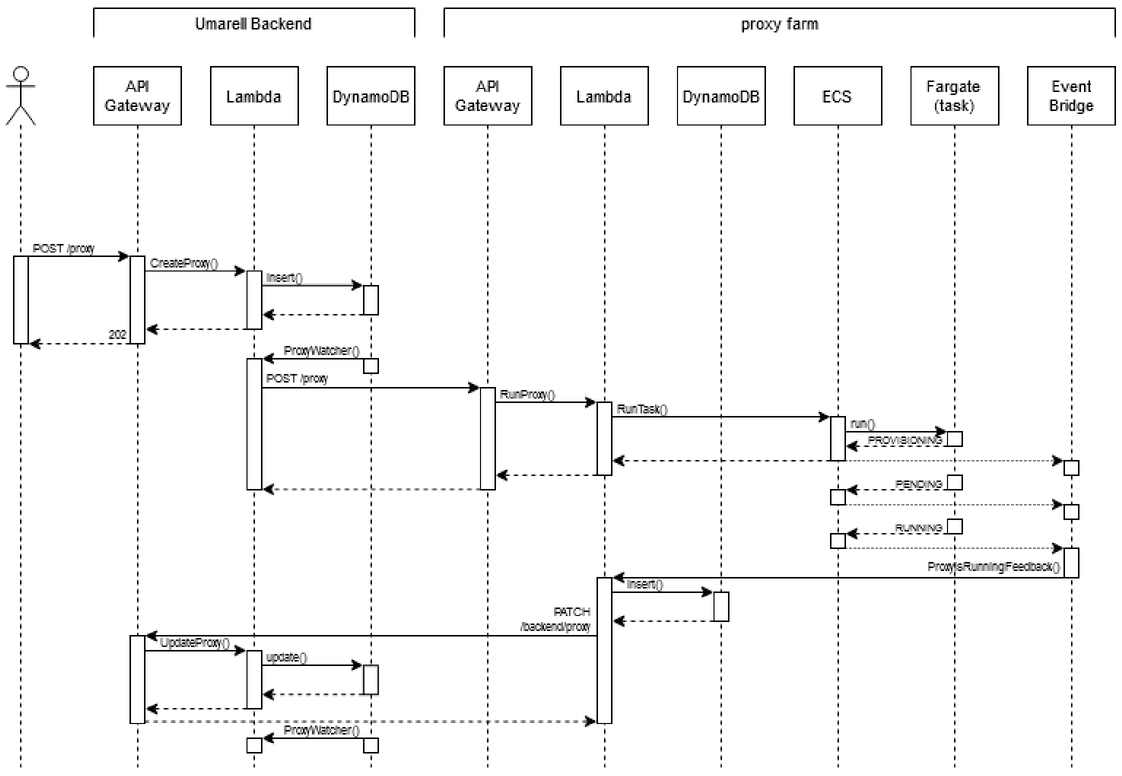

Figure 4.7: Proxy creation sequence diagram in a multi-account scenario

```
ProxiesTable:
        Type: AWS::DynamoDB::Table
        Properties:
            TableName: proxies
            ...
            StreamSpecification:
                StreamViewType: NEW_AND_OLD_IMAGES
```

Figure 4.8 shows the update of the proxy mode, which requires an update of the Hoverfly instance.

If the status has become *running*, it is changed to *creating* before being persisted, and it's the case of an existing proxy which has to started. The workflow behaves like for creation, so the task is launched by invoking the Lambda function *RunProxy* and when it becomes *RUNNING*, the Lambda function *ProxyIsRunningFeedback* is triggered, which performs the same operations as before.

Instead, if the status has become *stopped*, it is changed to *stopping* before being persisted, it means that there is a running Fargate task that has to be stopped. The behaviour is very similar to the previous one, like figure 4.9 illustrates.

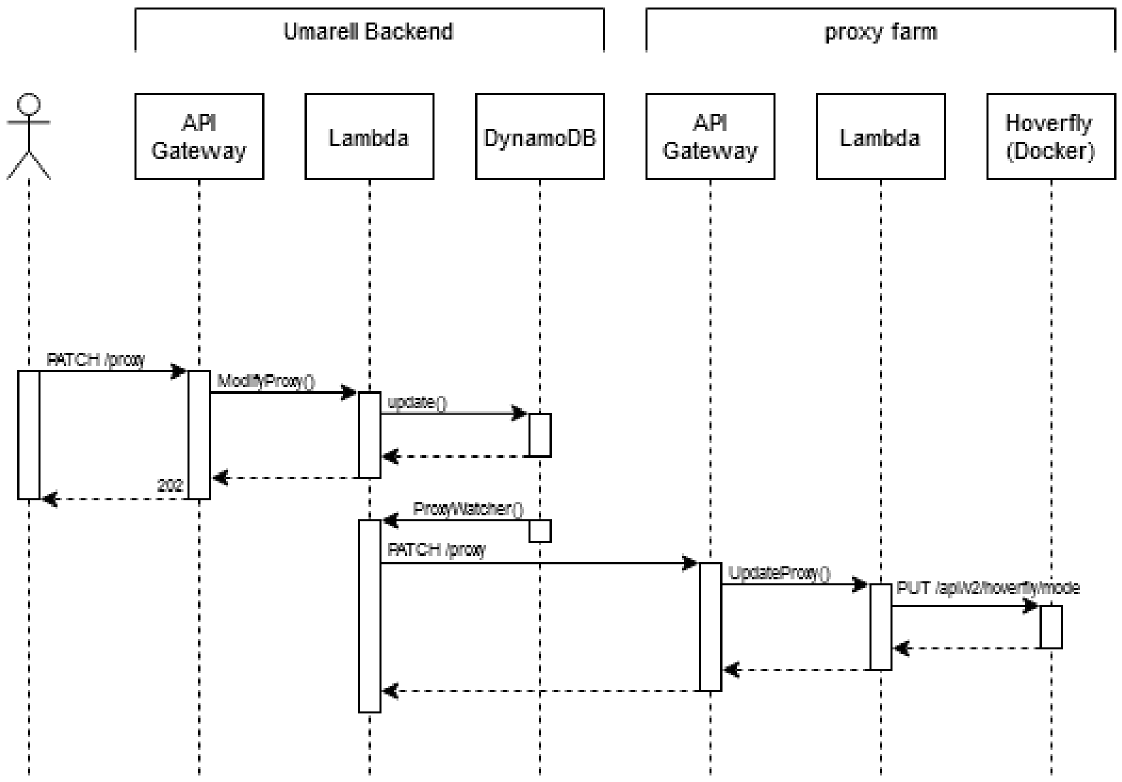

Figure 4.8: Updating proxy mode sequence diagram in a multi-account scenario

4.6.2 Handling scenarios

A scenario is a set of simulations initially stored in-memory inside a Hoverfly instance. It is necessary to download simulations from the instance and save them in the database, in order to make the scenario available to the user.
The main idea is that a scenario makes sense at the end of the current testing session, when the behavior of the customer service can be determined. Therefore, when the customer request a proxy for stopping, simulations are downloaded from the instance through the Hoverfly administrative API, on the port 8888. Simulations are then stored with a scenario ID, and finally the Fargate task is requested to stop.
Afterwards, in a new testing session, a customer may want to choose one of the stored scenarios and upload it in an empty proxy, for simulating to simulate the service or for capturing new request-response pairs, in order to append new simulations to the selected scenario.

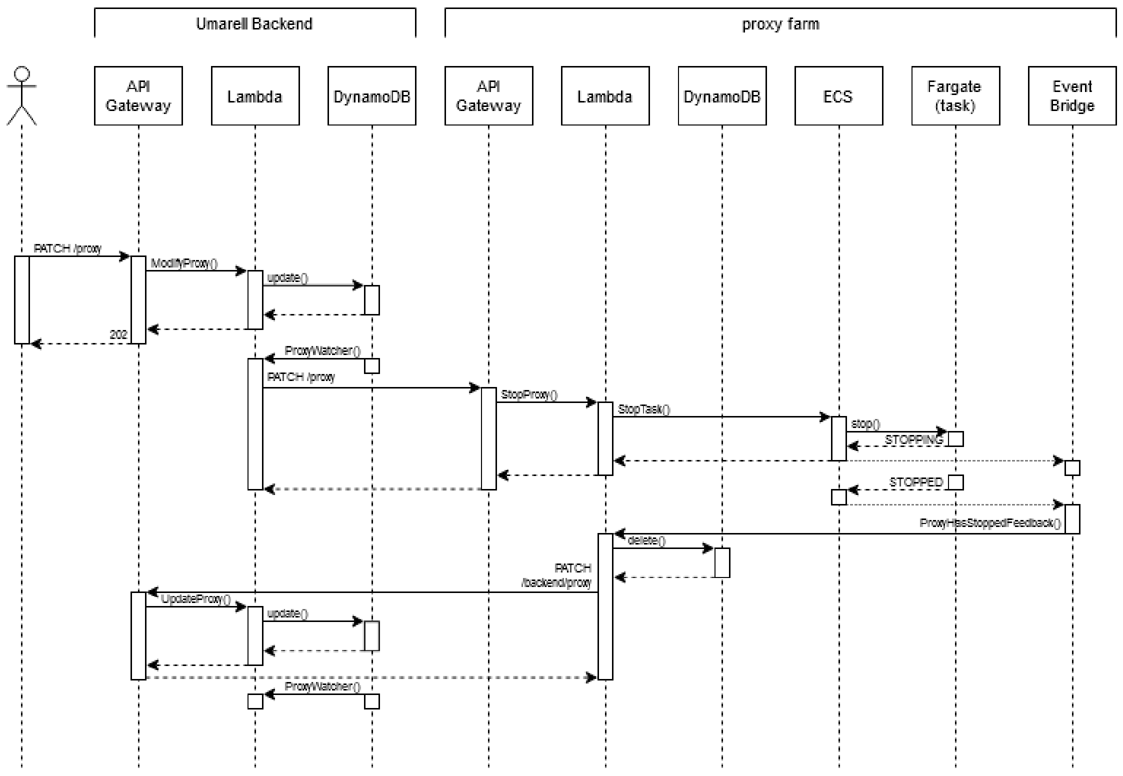

Figure 4.9: Proxy stopping sequence diagram in a multi-account scenario

Security measures

Hoverfly containers runs in a public subnet, so Hoverfly API could be potentially used by customers. For example, for retrieving simulations could be sufficient to query the endpoint */api/v2/simulation* on port 8888. However, if in the future the organization behind Umarell decides to replace Hoverfly with another tool and the customer attempts to query the same endpoint, barring pure coincidence, an error will be returned. For this reason the presence and use of Hoverfly should be transparent. In order to avoid customer to use the API, it has been defined a security group which denies access on the port 8888 from the Internet, so that only resources inside AWS can use that API.

4.7 Adapt to pay-per-use model

One of the defined requirements is to eliminate unnecessary costs and enable a pay-per-use model. The Umarell Backend uses serverless services, like Lambda and DynamoDB, which don't require fixed costs. Instead, the proxy farm uses components like Fargate and AWS VPC, which may require fixed costs. Therefore

it is necessary to adapt the application, in order to benefit from the mentioned pricing model. It means that a resource which incurs a cost even if it is idle, should be destroyed, if not used.
The Hoverfly instance representing a proxy runs in Fargate, which calculates the bill on the basis of CPU and memory consumption, from the time in which the image is downloaded up the task termination [5], therefore, it should exist only in the meanwhile a customer want to use it.
The life-cycle of a proxy farm can also be evaluated. Ideally, if it has not running proxies inside, i.e. it is unused, it can be destroyed. However, it must be taken into account that the creation of a CloudFormation Stack is not immediate and can require some minute. In the case of the Umarell Proxy Farm it could be considered which it is shared among many customers, so it is plausible that there is a constant use of it. For these reason it could be kept always active. If further business analysis will put in evidence that this proxy farm is not used for a long period (e.g., outside office hours, if the service is active in countries with the same time zone, or during public holidays), then the proxy farm could be created and destroyed with greater dynamism.
Proxy farms managed by customers are supposed to be destroyed after use, so building time would result in a less pleasant user experience. However, some compromises might be found. For each customer, customized times of the day in which to activate the proxy farm could be scheduled.

4.8 Study of Hoverfly

Hoverfly behaves like a proxy should do, therefore it is very convenient for this use case.

4.8.1 Working modes

Hoverfly can work with several attributes (like *mode* and *stateful*), already discussed in section 2.9.2, and proxy implementation can leverage what this tool already offers. In practice there is mapping:

Proxy	**Hoverfly**	
capture-stateful	mode: capture, stateful: true	
capture-stateless	mode: capture, stateful: false	8500
simulation	mode: simulate	
spy	mode: spy	

[5]`https://aws.amazon.com/fargate/pricing/`

Actually *capture-stateful* and *capture-stateless* summarize what Hoverfly does through multiple attributes. This has been a choice which finds benefits at REST level. The attribute *stateful* in Hoverfly makes sense only when it works in *capture* mode. When it is switched to *simulate*, that attribute disappears. This is a problem in REST, where the resource should be represented always in the same way, therefore it has been preferred to use a single value. The decision has been applied since the storage level.

4.8.2 How to run it

SpectoLabs offers different ways for using Hoverfly:

- by using Hoverfly Cloud;
- by using the CLI application;
- by building a container starting from a Dockerfile;
- by building a container starting from a Docker image;

Hoverfly Cloud

Hoverfly Cloud [6] is a test environment offered "as a Service" for API simulation. It does not fit with the Umarell use case, because of several reasons:

- Hoverfly is used to shape the behaviour of a proxy, but with the cloud solution there is no full control over the Hoverfly instance and this could be a limitation in the future;
- tariff plans do not adapt to the serverless logic, as they don't offer a pay per use pricing model, but only fixed monthly rates;
- Umarell could require as many Hoverfly instances as needed, but, in order to achieve this possibility it is necessary an "Enterprise" pricing plan that leads to high fixed costs which may be unsustainable in certain scenarios;

CLI application

The second alternative is using the CLI application, but it does not fit to the Umarell use case, since it is necessary a Docker container that has to be launched in the ECS cluster (as a Fargate task). Building a new image and using the executable inside it requires an effort that does not take any advantage.

[6]`https://cloud.hoverfly.io/`

Building a Dockerfile

Using the official Dockerfile available on GitHub [7] for building a container starting from the Hoverfly project brings further responsibilities for maintaining the project.

Using a Docker image

The last possibility is using an image available on DockerHub [8]. It is ready to use and when a container based on this image is launched, it is possible to set one or more Docker commands. The commands available are the same as those accepted by Hoverfly CLI application [9]. This is the solution that best suits the Umarell scenario.

4.8.3 Commands coverage problem

As already said before, the proxy modes are not identical to Hoverfly ones, but a mapping is required. Although the Hoverfly container allows to use commands to specify the working mode, there is no full coverage. In particular, it is not possible to specify whether the *capture* mode has to be launched with the *stateful* flag set to *true* or not. Instead, there are no problems in specifying *simulation* (the corresponding command is named *webserver* on Hoverfly CLI) and *spy* modes.
Hoverfly exposes an administrative API which is available on the port 8888 by default. It allows to change the Hoverfly mode and to set the *stateful* flag, so it can be used for setting the wanted configuration. When asked to launch a proxy, its status is set to *creating*. At this point an event is triggered and a new Hoverfly instance is launched within a new Fargate task. Whether the proxy has been requested to have status *capture-stateful* or *capture-steless*, the Hoverfly instance is launched to have *capture* mode. When the Fargate task status becomes *RUNNING*, a call to the Hoverfly instance API is made, in order to update the *stateful* flag. After that, the proxy is updated with status *running*. From that moment the proxy is effectively up from the point of view of the customer. This means which a proxy is essentially an abstraction that uses an Hoverfly instance, but it is not the instance.

[7]`https://github.com/SpectoLabs/hoverfly/blob/master/Dockerfile`

[8]`https://hub.docker.com/r/spectolabs/hoverfly/`

[9]`https://docs.hoverfly.io/en/latest/pages/reference/hoverfly/hoverflycommands.html`

4.9 Networking and security management

The proxy is a network resource which must be accessible through the Internet, therefore it is placed in a public subnet. In AWS, it means which the **route table** contains an entry as follows:

Destination	Target
0.0.0.0/0	igw-12345678901234567

The destination for the route is 0.0.0.0/0, which represents all IPv4 addresses. The target is the internet gateway that's attached to the VPC.
The private subnet is currently used for hosting the network interfaces of Lambda functions which need to connect to resources. Being within the same network, the proxies can be accessed by mean of their private IPv4. This happens, for example, when the simulations in the Hoverfly instance must be retrieved. To be more precise, Lambda is a serverless service which runs in Amazon's cloud space and does not require networking. Therefore, asserting that a Lambda function is placed into a subnet is incorrect. Instead, only its network interface is. In AWS, the route table of a private subnet is configured as follows:

Destination	Target
0.0.0.0/0	nat-12345678901234567

This time, the target is a NAT (Network Address Translations), which prevents resources from being directly reachable.
Subnets are managed with CDK, through the package *aws_ec2*. It is straightforward, consisting of using an abstract factory which requires few arguments, like the name of the subnet, a CIDR number and a constant of the same package, which specifies whether the network is public or private.

Securing administrative port

As already mentioned, each proxy has two ports. The 8500 is used directly by customers, for the proxy functionality, whereas the 8888 is used for administrative purposes and its use has to be filtered by the application, therefore this port must prevent access from the Internet. At this purpose two security groups have been created. The first one is pretty simple. It has been named *ProxyAdministratorSecurityGroup* and it's assigned to Lambda functions that need to access the proxy through the port 8888. The second one is named *HoverflySecurityGroup* and the following **ingress rules**:

Protocol	Port	Source
TCP	8500	0.0.0.0/0
TCP	8888	sg-12345678901234567 (ProxyAdministratorSecurityGroup)

While on port 8500 is allowed any IPv4, for port 8888 only resources which have the specified security group are allowed. Therefore this is a firewall rule which consents only properly selected internal resources.

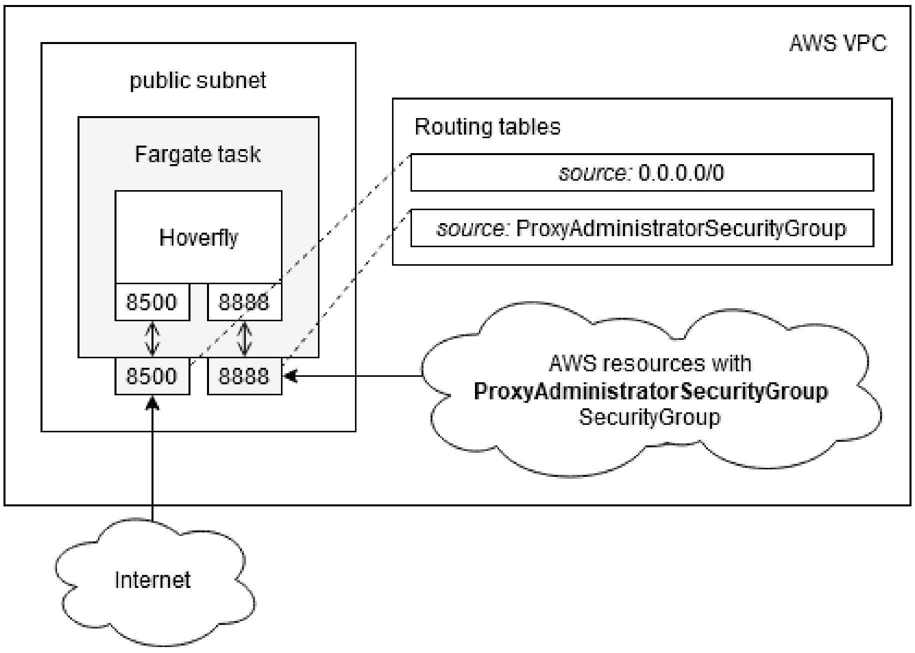
AWS VPC
public subnet
Fargate task
Hoverfly
8500
8888
8500
8888
Routing tables
source: 0.0.0.0/0
source: ProxyAdministratorSecurityGroup
AWS resources with **ProxyAdministratorSecurityGroup** SecurityGroup
Internet

Figure 4.10: Securing administrative port schema

4.10 Infrastructure as Code

The proposed solution completely relies on the **Infrastructure as Code (IaC)** approach. It consists of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.
AWS CloudFormation [10] is the IaC service that Amazon uses to build resources in its cloud. The alternative ways to build resources in AWS would be using the AWS console [11] or the CLI application [12], which are valid approaches, but they are not tools created with the purpose of being automatable, deployable or versionable. CloudFormation offers is a complete tool for creating resources and shaping the

[10]`https://aws.amazon.com/cloudformation/`

[11]`https://aws.amazon.com/console/`

[12]`https://aws.amazon.com/cli/`

infrastructure, but it has a notable drawback. Writing a complete and optimal template can take some time, because the simplest infrastructure may take hundreds of lines, so it is required experience and a deep knowledge of CloudFormation concepts and syntax. For this reason some workaround to avoid writing CloudFormation templates directly have been considered.
As already said, the Umarell Backend and the Umarell Proxy Farm projects are based respectively on Serverless Framework and CDK and these are IaC tools.

4.10.1 Umarell Backend

Serverless Framework is able to work with several cloud providers for building serverless event-driven applications and it is integration-ready: it is sufficient specify the desired provider to get it working.
To set the wanted environment it is sufficient to specify few parameters in *serverless.yml*:

```
provider:
  name: aws
  region: eu-west-1
  runtime: python3.8
  memorySize: 128
   profile: umarell
```

AWS has been set as provider, so Serverless Framework will output a CloudFormation template, that will be deployed on region *eu-west-1* using the credentials specified into the profile named *umarell*. For Lambda functions are reserved 128MB with a runtime based on Python3.8.
Under the key *functions* are defined the FaaS functions, implemented with Lambda in this case. Under the key *resources*, instead, any kind of resource can be described, by using the CloudFormation format.
The whole infrastructure is described through YAML, which makes management simple on the one hand, and hides complexity on the other. Indeed, behind the scenes it creates an API, based on AWS API Gateway, and maps it to the specified functions.

4.10.2 Proxy farm

This part of the application does not contain the classic servereless services, like in the previous case, therefore Serverless Framework is not a good choice. CDK has been chosen in this case, because it allows to describe the infrastructure by using the most common programming languages, exploiting their power and flexibility.

CDK evaluation

This tool is not very spread yet, so it has been evaluated with particular attention, but resulting in a positive verdict. Key points are as follows:

- CDK is an abstraction of CloudFormation. The latter is a service, while the first one is a framework that generates a template. Being the same thing, there is no change regarding the price;
- If problems linked to CDK arise, it is possible to migrating to "plain" CloudFormation in several ways: using CDK as a wrapper to inject "plain" templates, using automatic tools, which may be not perfect, or rewriting the template from scratch;
- It is easy to use and relies on Abstract Factory style for resource creation. It is intuitive and well documented, although there is a lack of practical examples;
- it allows to exploit the great expressiveness of programming languages, like cycles and conditions. It is possible to organize code and build libraries, giving the ability to create custom, modular and reusable constructs;
- It is potentially less prone to errors, thanks to features of languages and modern IDE: type-safety, code completion, etc.

4.11 DevOps Pipeline

Git [13] is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Moreover it is well documented, has a large community and it is very known among developers.
AWS offers all the needed services for building a CI/CD pipeline, like **AWS CodeBuild** [14] and **AWS CodePipeline**.

4.11.1 Stages

The Umarell project is in a very early stage, so CI/CD pipelines are quite simple and they will be better evaluated in the future. Essentially the Umarell Backend and the Umarell Proxy Farm can be seen as two microservices, like noted in section 5.1. They must be independent each other, which means also that when one of

[13]`https://git-scm.com/`

[14]`https://aws.amazon.com/codebuild/`

them is updated, the other must not be rebuild. For this reason, two pipeline have been implemented, one for each project. Nevertheless they are very similar, being formed by analogues stages.
Both the pipelines have been created through the AWS console and consist mainly of three steps:

- preparing the project;
- building and testing;
- deploying the application;

preparing the project
buinding and testing
deploy

Figure 4.11: Main steps of the pipeline

Nevertheless, two stages currently exist. Building, testing, and deploy is made by mean of a unique AWS service, which will be described below.

First stage: preparing the project

The first stage is very simple, as the pipeline has only to download the project from the source location, that consists of cloning the project from the repository on CodeCommit in both projects. Integration of this service in the pipeline is very easy, since it is sufficient to select an item in a drop down menu in the console. If the operation is successful, then the pipeline passes to the next stage.

Second stage: building and testing

This stage aims to execute unit tests, build and deploy the project in a test environment, and then to execute cross-service tests. It has been build using AWS CodeBuild, so in both projects a *buildspec.yml* file has been defined, which defines four steps:

- *install* specifies the runtime environment, which is based on Python 3 and installs the specific IaC tool: Serverless Framework for the Umarell Backend project and CDK for the Umarell Proxy Farm. In both cases *npm* has been used;
- *pre_build* specifies Python dependencies, which currently consists only of *pytest* library, for running unit tests;

- *build* configures an AWS profile (access key, secret key, region) to be used with the selected IaC tool (Serverless Framework or CDK) and executes tests by mean of *pytest*;
- *post_build* launches the commands for deploying the application, which is *sls deploy* for the Umarell Backend project and *cdk deploy* for the Umarell Proxy Farm project;

If at the end of the stage there are not errors, the pipeline passes to the next stage. The pipeline implementation for one single project is summarized in figure 4.12.

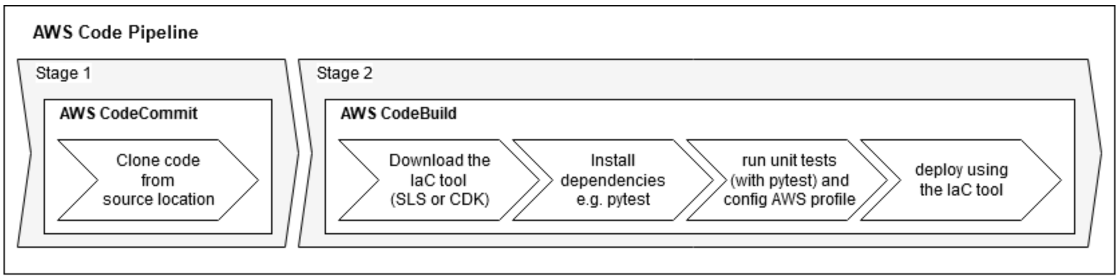
AWS Code Pipeline
Stage 1
AWS CodeCommit
Clone code from source location
Stage 2
AWS CodeBuild
Download the IaC tool (SLS or CDK)
Install dependencies e.g. pytest
run unit tests (with pytest) and config AWS profile
deploy using the IaC tool

Figure 4.12: Implementation schema

4.11.2 Security

Managing secrets

It is not advisable to hardcode secrets, like security keys and tokens, directly in the code, for several reasons:

- if a secret change, it should be manually replaced over all the occurrences, also in a different code base;
- the secret would be seen by all the employees that can access that code; this is a problem, because different persons could require different permissions (and have different responsibilities);
- if the code is in a public repository, secrets would be exposed to the world, with harmful consequences;

In Umarell, secrets are handled by mean **AWS Secrets Manager**. It allows to create a virtual resource called **secret** which contains essentially a name and an ARN and handles some keys, which are encrypted.
In the CodeBuild Action, before, some AWS credentials have been handled, in

ordupadteer to deploy the stack in CloudFormation. AWS Secrets Manager has been used, so in the AWS console it has been created a secret named *dev/Umarell/AwsAccount*, which handles two key-value items, where the keys have been named *KeyID* and *AccessKey*, and it has been specified that Amazon should encrypt them with a *DefaultEncryptionKey.*
In this way, only employees with specific permissions can see and handle these secrets. In *buildspec.yml*, instead, it is sufficient to declare some *environment* parameters, like:

```
env:
  secrets-manager:
    AWS_KEY_ID: dev/Umarell/AwsAccount:key_id
    AWS_ACCESS_KEY: dev/Umarell/AwsAccount:access_key
```

and then reference them with the symbol $:

```
post_build:
     commands:
       - serverless config credentials --provider aws --key
   $AWS_KEY_ID --secret $AWS_ACCESS_KEY --profile umarell
```

As already said, a secret is a resource, so CodeBuild needs specific permissions to retrieve it, therefore, the role assigned to CodeBuild must have attached a policy with refers to the service *Secrets Manager*, with *Read* access at the resource *arn:aws:secretsmanager:eu-west-1:138538731418:secret:dev/Umarell/AwsAccount-** (the secret) and without request conditions.

4.11.3 CI/CD pipeline

Currently, it is requested that each pipeline relies on CD practice, so the pipeline is executed every time a change in the source location is detected. This means that, if a developer pushes some code on the repository, then the pipeline is triggered. The sequence diagram in figure 4.13 illustrates the interactions among the AWS services when a developer pushes some code on a repository hosted by CodeCommit.

4.11.4 Monitoring and observability

As the project is in an early stage, the pipeline is quite simple, but it is sufficient to show the needs of a serverless application.
Distributed infrastructure components operate through multiple abstraction layers of software and virtualization, which make makes controllability impractical and

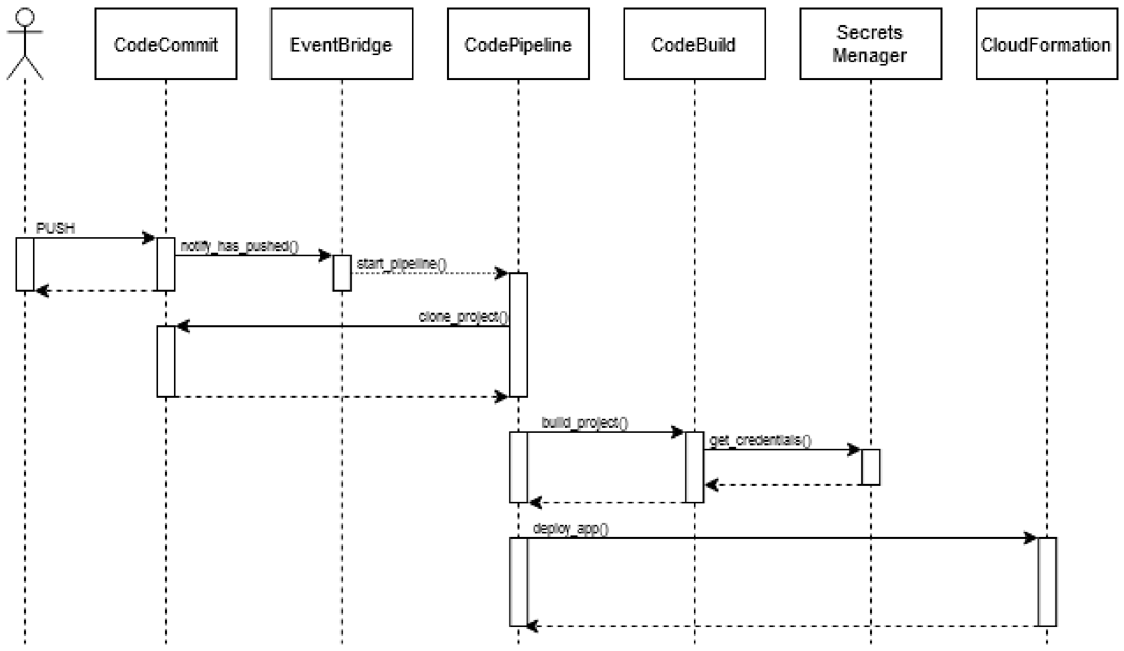

Figure 4.13: Starting a CI/CD pipeline sequence diagram

challenging. Instead, common practice is to observe and monitor infrastructure performance logs and metrics to understand the performance of individual hardware components and systems.
At the moment, two main AWS services have been used for these tasks: CloudWatch and X-Ray. The first one is enabled by default on Lambda. Indeed, it is sufficient writing on standard output for logging application messages. These are then collected by the CloudWatch services and grouped in the dashboard.
Then, X-Ray has been introduced, in order to trace functions calls, check performances, debug and identify and troubleshoot performance issues and errors. Enabling it for Lambda and API Gateway services in the Umarell Backend is straightforward, since it is sufficient a minimal configuration in *serverless.yml*:

```
provider:
    iamRoleStatements:
        - Effect: Allow
          Action:
              - xray:PutTraceSegments
              - xray:PutTelemetryRecords
          Resource: "*"
    tracing:
          lambda: true
          apiGateway: true
```

In CDK X-Ray can be enabled in Lambda function by passing the construct prop to Function construct, like below:

```
aws_lambda.Function(self, 'LambdaId',
    function_name='LambdaName',
    # other construct parameters
    tracing=aws_lambda.Tracing.ACTIVE)
```

4.12 Conclusions

The next chapter focuses on the evaluation of the work done during the implementation phase.

Chapter 5

Evaluation

This chapter is meant to help perceive the quality of the implementation and highlight the contribution of serverless, as well as its level of maturity. Section 5.1 compares the implementation to a microservices application, discussing common principle that have been respected. Section 5.2 goes into detail, by discussing aspects concerning consistency, one of the most important talking points in cloud native applications. Section 5.3 is about DevOps practices that can be still implemented. Section 5.4 discusses costs required by the application.

5.1 A microservices application

In this thesis, serverless has been defined as an elaboration environment which does not require provisioning or managing servers, whereas microservices have been referred to an architectural style in which the application is made of independent components, characterized by certain key principles, already described in section 2.3.7. According to this, microservice and serverless concepts live at different levels, and one does not exclude the other. Indeed, a microservice application could be made of Docker containers launched in a serverless environment, as it happens with AWS Fargate.
Things seem to be different comparing microservices, which are usually intended as continuously running containers, to FaaS, that executes some code in an ephemeral container. Nevertheless, these two approaches may come close.
The Umarell Backend and the Umarell Proxy Farm have been organized in two distinct Git repositories hosted on AWS CodeCommit. This gives a greater separation of responsibilities and allows them to be handled, packaged, built and deployed separately. The communication layer between the two parts of the project is made of calls to HTTP API, which is platform agnostic. These characteristics allow to deal with two independent parts, loosely coupled and independently deployable.

This enables the **autonomy principle** of microservices.
The two parts of the application have separated responsibilities and points of failure. The Hoverfly instances are handled entirely by the Umarell Proxy Farm, whereas all the tables interact with the Umarell Backend (it is a good practice which a storage is accessible by only one microservice). So **resilience principle** is respected.
Observability is generally considered a difficult task in cloud environments, due to the distributed environment and the complexity of the additional abstraction layers. In FaaS it may be even more complicated, because of the ephemeral nature of containers. Nevertheless, improving traceability, debugging, or monitoring, is one of the main goals of cloud providers. In Umarell, DevOps practices are still unripe due to the early state of the project and monitoring mostly consists of logging at Lambda level, but there is no technical impediment in order to respect the **transparency principle**.
As already said, two pipelines have been implemented, which rely on IaC approach, so the **automation principle** is already enabled.
Finally, FaaS have a fine granularity over the decomposition, which can be led near to business problems. Like for classic microservices, scoping is a key issue which has to be addressed before any implementation, and it can be made around business capabilities. In conclusion, also the **alignment principle** can be respected.
This implementation wants to make clear that serverless and microservices can overlap and share the same problems, the same difficulties, but also the same strengths, and the same theory. It is not a fixed rule, because a different implementation might have failed to adhere to one or more key principles of microservices, but this twinning should be taken into account in order to build maintainable fully serverless applications.

5.2 Handling consistency

Umarell is a distributed application since it is executed in a cloud environment which in the future could be delivered through several data centers around the world, therefore it has to handle with characteristic problems of this type of architecture.
In the first place, let's evaluate the application. The Umarell Backend has been built by using serverless services, like AWS Lambda and AWS DynamoDB. Along with DynamoDB Stream they represent the implementation of an event-driven application, in which each action is taken on the basis of changes to the data stored in the tables. Other events are triggered as defined by AWS EventBridge rules. When a AWS Fargate task enters in running state or has to stop, then a Lambda function is invoked. These mechanisms are event-driven too.
Asynchronous events aid in decoupling components from each other and increase

overall system availability, but they also bring to scenarios in which to grant consistency is difficult. It is especially true in distributed environments. At the opposite synchronous blocking communications should be avoided, since they decrease the availability, occupy resources and increase costs, especially in serverless FaaS, where a pay per use model is used.
Some processes require transactions, which can be defined as a sequence of information exchange and related work that is treated as a unit for the purposes of satisfying a request and for ensuring consistency. One possibility is using distributed transactions, which use a two phase commit protocol, and can use a lock in order to be able to access resources in mutual exclusion. Nevertheless distributed transactions are problematic, as they bring to synchronous communications, lower the availability of resources and increases the risk of contention and deadlock. Like in microservices, transactions and general workflows are based on choreographed interactions. Each component performs some work and emits one or more events. So other components are triggered. The process is asynchronous, increases the availability of resources and results with a last event that notify the success to other components, otherwise it's invoked a rollback procedure in all involved services.
Umarell behaves like this in several situations. For example, for the proxy creation, a new item into the database is inserted with the field *status* equals to *creating*, which represents an intermediate state. A sequence of events and actions follows and, at the end, if the Fargate task is successfully created, the proxy status is changed to *running*, which represent the final state. If an error occurs in the middle of the process, the creation is invalidated. Another case is when a proxy has to be launched in *capture-stateless* or in *capture-stateful* mode. Before setting the proxy in status *running*, making it usable by the customer, a further HTTP call is made, in order to set the Hoverfly instance and avoids problems due to its limitations, as described in section 4.8.3. Even in this case, the process passes through different intermediate states and a chain of interactions, before it can be considered completed by the system.

5.2.1 Proxy farms managed by customers

If only the scenario with a single shared proxy farm had been allowed, the communication layer could have consisted of invoking Lambda functions by mean of an SDK, like **boto3** in Python, and IAM identities and permissions would be used.
The will to put heterogeneous systems in communication has made mandatory to use a platform agnostic protocol, like HTTP, but it requires a major effort. Indeed, it is necessary to ensure that this API layer works correctly. Therefore Umarell should spend more resources (person-hour) on some kind of testing, like contract testing or end-to-end testing.
Another issue is that, if Umarell Backend changes something in its API, customers

have to update their proxy farms. Realistically, this would be a gradual process, therefore API versioning would be necessary, which is another effort required to both parties.

5.3 DevOps pipeline

Keeping the two main parts in separated repositories means having two independent source locations, which are used in two different pipelines. Thanks to Continuous Development, when a change is detected inside one of the two repositories, the related pipeline is launched, while the other one is not affected.
As already said, DevOps practices adopted are not very sophisticated yet, so pipelines are very simple.

5.3.1 Monitoring and observability

As the project is in an early stage, the pipeline is quite simple, but it is sufficient to show the needs of a serverless application.

Alarms

In the future there will be a dedicated team to implement DevOps practices. For example, CloudWatch alarms will be inserted, in order to notify the arrival of unexpected events, e.g. functions returning a 5xx status code, or for ensuring stored data corresponding to what is in the proxy farm, e.g. if the status of a proxy is *running* there must be a task in execution.

External services

Then strategies for having an end-to-end overview will be necessary. CloudWatch logging and insights could be better combined with trace data from AWS X-Ray. Then external services could be used for having a different manipulation of collected data, and hopefully better information.

Responsibility

Responsibility for monitoring proxy farms depends on the placement of proxies. If the Umarell Proxy Farm is used, then monitoring is up to Umarell. In the case of the multi-account solution, Umarell could provide templates with preset features for facilitating the task. In the case of external proxy farms, directly handled by customers, monitoring is a responsibility of them.

5.3.2 Testing

Currently, testing consists of running unit tests at microservice level, when a new change is released. It is undoubtedly useful, but it is not sufficient to test the overall application behaviour, because interactions between the two parts of the application are not evaluated. To do this, a further pipeline could be implemented. This could be made of two concurrent jobs, which build the existing pipeline, but resulting in two testing environments. Therefore integration testing, system testing, and acceptance testing, can be executed. However this kind of testing is very difficult, especially in distributed applications like in this case, because it requires creating complex environments, creating lengthy integration suites and managing test data. An alternative method to consider - but which is not necessarily sufficient - is contract testing. Unlike end-to-end testing, it is easy to setup and scale. It consists of testing an integration point by checking each application in isolation to ensure the messages it sends or receives conform to a shared understanding that is documented in a "contract". In the case of Umarell, it should be implemented to test the communication layer between the two main parts.

5.3.3 Consequences of serverless

Developed pipelines are very close to application layer. It does not have to provision or manage servers or virtual machines. Then it can leverage Infrastructure as Code approach in order to build portions of the infrastructure that can be huge. This is a great advantage, because it moves the focus of operations at higher levels, toward business problems, which favours having a product view, not a project view, in which success is determined by business metrics. The responsibility of the operations side of a DevOps team will be the health of the services and systems. While software developers remain focused on the individual components, operations engineer focuses on the system as a whole.
On the more technical side, in developed pipelines observability is a complex and coveted goal to achieve. As said before, logging is particularly difficult in serverless, because FaaS functions run in ephemeral containers, which are destroyed at the end of the code execution and, therefore, logs need to be collected by an external aggregator and furthermore network delays can make information dirty.

5.4 Costs

As already said, Umarell is an embryonic project, so any hypothesis on final costs would be a gamble. However some assumptions could be made about cost sharing in the two scenarios described in this thesis.
Knowing costs is essential, because it allows to shape tariffs for customers (e.g.,

based on usage), detect commercial strategies, manage money accurately, and so on.

5.4.1 Umarell Backend

The Umarell Backend is easier to analyze, because it uses simple services, with clear pricing models, at least apparently.

AWS Lambda pricing

AWS Lambda has fixed costs, which depends on the number of invocations and duration, defined as the time required by the code to be execute. Nevertheless there are multiple factors that come into play. First of all, the price depends on the amount of memory allocated to a function. An increase in memory size triggers an equivalent increase in CPU. More power means more speed, so a function which requires more memory could be cheaper than a function requiring less memory, because it is sufficiently faster. In that case a function could be faster and cheaper than it would be with a different configuration. This behaviour could change drastically with a minimal change.
The execution speed depends also by the runtime. A language may be faster than another in performing some types of operations, but slower in others. Furthermore, a runtime could be much faster than another in code execution, but it could be slower in providing the environment, i.e. building the container. The allocation time is especially important when a cold start occurs, then it must be taken into account the cold start frequency and it should be convenient to properly configure a provisioned concurrency for some Lambda function. Cold starts can greatly affect costs if a functions calls invokes a second function, waiting for a response, and there is not an available warm container. The first function could wait several seconds before it can finish running. More generally, this is the risk associated with functions that make outgoing calls, chains of function calls, and sequences of synchronous calls to other functions. In these cases, in addition to raising costs, a timeout can occur, which causes the requested task to fail and unnecessary costs. When it happens, asynchronous mechanisms could bring some benefit.
Another cost can be associated with the development speed. Some languages may allow faster debugging, or faster loading of new changes, especially for interpreted languages, because of the absence of compiling phase.
At the end, it must be considered that event-driven applications may be unpredictable. An action may trigger several events, and each of these may trigger other events. Therefore a tree of events may be generated with a single happening. Moreover it could be not said what events are generated when some code is executed, because they depend on the inputs and the state of the system, which can be

different at any execution. A simple case of this type concerns the modification of the proxy. If the change involves the status, then a Fargate task must be launched or stopped, therefore an EventBridge rule triggers a Lambda, which makes further calls. Instead, if the change involves the mode, then there is a chain of Lambda and an HTTP request to the Hoverfly container is made. Finally, if the change doesn't affect neither the status nor mode, no furthers events are generated, apart from the invocation of ProxyWatcher, which is there in any case.
For the reasons described above, costs are not easily predictable.

DynamoDB pricing

DynamoDB offers two pricing models [1]. *Pricing for on-demand capacity mode* does not require the customer for the data reads and writes on tables. It is a solution without provisioning. It is a convenient solution if the workload is unknown, if the traffic is unpredictable , or if paying on demand is preferred for other reasons.
Pricing for provisioned capacity mode requires to specify the expected number of reads and writes per second. It is possible to set auto scaling to automatically adjust tables' capacity based on the specified utilization rate to ensure application performance while reducing costs. It is a convenient solution if traffic workload is predictable, if traffic is consistent or ramps gradually, or if it is wanted to better control costs. This mode is cheaper, but requires information which is not always easy to forecast.
Umarell is used by business customers, which have presumably office hours. If they're all in Europe, it would be easy to predict a useful time range, therefore it could be convenient a provisioned capacity mode with autoscaling, in order to reduce costs.

5.4.2 Umarell Proxy Farm

It must be considered which creating the Umarell Proxy Farm takes some minute, and long waits could negatively affect the customer experience. Moreover this proxy farm hosts proxies for multiple customers, so it could presumably be occupied more or less continuously over time. In light of this, it could be convenient to keep this proxy farm alive, so that the startup time of proxies is short and the experience remains pleasant for all customers. Although the platform can be still considered fully serverless, as there is not provisioning or managing of servers and customers have to pay only for what they use, it must be considered that in this situation there may be fixed costs that the organization behind Umarell has to bear. In particular, the VPC has a NAT Gateway, which incurs costs on an hourly basis.

[1] `https://aws.amazon.com/dynamodb/pricing/`

This means that there are costs even if no customer is using the Umarell platform. Anyway the rate is very low, around 0.05 USD per hour, which is a cost that can be easily drowned in other expenses, even outside of IT. In summary, the Umarell Proxy Farm can be left up over time, or at most it can be dismissed during the night, assuming that the customers are mostly active during the day.
If only the Umarell Proxy Farm is used, then all the components consume resources in the Umarell cloud, so each one contributes to the final cost. Instead, in the scenario in which customers can create, manage, and destroy on their own, outside of the Umarell boundaries, then what happens in these proxy farms should incur no cost.

Cost estimation

Calculating cloud costs is never easy, but some providers offer cost estimation tools. **AWS Pricing Calculator** [2] allows to select and configure services used by an application in order to have an estimate on the monthly bill.
What follows is a purely demonstrative example that gives an idea of the costs that Umarell might have to face.

AWS API Gateway

The configuration used for the estimation is:

- 1 million requests per month;
- 1.6 GB for cache memory size;

The calculation was done as follows:
1 requests x 1,000,000 unit multiplier = 1,000,000 total REST API requests
Tiered price for: 1000000 requests
1000000 requests x 0.0000035000 USD = 3.50 USD
Total tier cost = 3.50 USD (REST API requests)
Tiered price total for REST API requests: 3.50 USD
0.20 USD per hour x 730 hours in a month = 146.00 USD for cache memory
Dedicated cache memory total price: 146.00 USD
3.50 USD + 146.00 USD = 149.50 USD
REST API cost (monthly): 149.50 USD

AWS Lambda

The configuration used for the estimation is:

[2] `https://calculator.aws`

- 10 million requests per month;
- 300 ms average duration;
- 138 MB of reserved memory;

The calculation was done as follows:
Amount of memory allocated: 128 MB x 0.0009765625 GB in a MB = 0.125 GB
RoundUp (300) = 300 Duration rounded to nearest 100ms
10,000,000 requests x 300 ms x 0.001 ms to sec conversion factor = 3,000,000.00 total compute (seconds)
0.125 GB x 3,000,000.00 seconds = 375,000.00 total compute (GB-s)
375,000.00 GB-s x 0.0000166667 USD = 6.25 USD (monthly compute charges)
10,000,000 requests x 0.0000002 USD = 2.00 USD (monthly request charges)
6.25 USD + 2.00 USD = 8.25 USD

5.5 Conclusions

The aim of this work is to be an academic contribution to broaden knowledge, therefore this thesis tried to be objective. The next chapter draws conclusions about it.

Chapter 6

Conclusions

Serverless promises very interesting benefits, such as easy scalability, high availability, fault-tolerance, pay-per-use model, changing of operations management effort, and more. However it is not perfect, and needs to acquire more maturity. In particular, serverless supply a very high level of abstraction, but under the hood there are containers, virtual machines, servers, clusters, physical appliances, which still need to be handled. The ability of the cloud provider in doing this is the key for bringing serverless to the next level.

Maturity of serverless

To configure a function which must be executed in AWS Lambda, which is at the forefront of the market, it is required to set a memory amount for the function, which affects also the CPU computing power. These are are characteristic linked to hardware, therefore don't fit with the idea of abstraction which serverless promises. Despite the fact which serverless promises unlimited scaling, Lambda allocates a certain budget, called "concurrency", which is to the number of function executions that are happening at any given time. It can be limited, re-arranged, reserved, provisioned, changed dynamically (auto-scaling), but it is still a limit with respect to the idea of infinite invocations.
FaaS states that a function can run somewhere in the cloud with the minimal effort. Anyway it is required to instantiate a container and this operation can take some time (cold-start). It is a flaw which cloud providers have to minimize.
One of the greatest advantages of serverless is that it does not require payment for idle resources. Nevertheless, to handle the problem of cold-starts, Amazon has devised provisioned concurrency. Its cost is affordable, but it requires fixed fees for maintaining some resources even if they're not used. Therefore it is a trade-off solution between costs and performances.

Cold-start is one of the most critical technical problems, because decreasing the warm-up times is challenging. Beyond solutions like provisioned concurrency, solutions outside the box are still needed, which could be based on re-utilization of multiple generic stateless layers, which is a further abstraction of the model. Distributed infrastructure components have to operate through multiple abstraction layers, and this makes controllability impractical and challenging. At the same time, this complexity makes observability fundamental and takes a lot of effort.

Monitorability

Since serverless applications are event-driven, it is highly recommended defining metrics, generating insights, analyzing real-time traffic, and adopting automation techniques, in order to set thresholds and alerting and react to unpredictable events. If well designed, an application can lead to cost savings if compared to traditional architecture, but the final billing depends on a multiplicity of factors, like traffic homogeneity, resource usage, ability of the provider in managing resources, optimizations, monitoring capabilities, and so on. Moreover a serverless application requires a certain sophistication in some activities, like monitoring and maintenance, which come with an effort, even economical.

Costs predicting

Costs in serverless are difficult to predict, since they depend on the commitment requested to the cloud, that is the duration of execution, the memory allocated, the computing power used, and so on. Each event can trigger several actions, therefore a tree of events is generated when something happens. Furthermore, different events can be generated in different executions, as they depend on inputs and application state. The combination of all these factors makes costs prediction challenging. This is the reason why new figures, like FinDev, need to be better defined, and even more efficient cost monitoring tools need to be implemented.

Public and hybrid cloud

Companies like Amazon, Microsoft, Google, Alibaba, IBM, etc., are making huge investments on serverless. Competitiveness can be the root for a myriad of new solutions that lead to the improvement of the existing ones. On one hand providers offer good products, useful in solving business problems of companies, and increase the maturity of those cloud models. On the other hand they create vendor lock-in, which is a well known problem with public cloud providers, because each provider has its own version of serverless architecture. Nevertheless, the concepts behind paradigms like FaaS are common, therefore platform agnostic frameworks can mitigate the problem. More in general, tools which doesn't depend on a single

cloud provider can enforce industrialized practices, and enrich the whole serverless culture.
Cloud is an opportunity to enable multiple environments optimized for specific purposes, distribute content geographically near to users, use software which best fits in a given use case, use certain architectural styles, and so on. This means which many applications may not be dependent on a single cloud provider, but may want exploit the strengths of a hybrid architecture.
The problems mentioned above can be even more complicated in this context. Platform agnostic tools are needed to monitor and aggregate logs, metrics, and insights coming from different providers, in order to build a centralized and meaningful analysis interface for observing costs, performances, scaling, and more. Then, tools for deploy are required in order to put a strategy in practice.

Thinking in serverless

Industrialization of practices is one of the most precious driver for the growth of serverless, because it could mitigate the lack of patterns and consolidated approaches.
FaaS tends to make serverless seem like a set of discrete functions, therefore it's required to enforce a mindset oriented to macro functionalities, and identify application apparatuses, rather than focusing only on individual functions. For doing this, event-thinking is fundamental. Moreover, asynchronicity can be difficult to handle, and integrating multiple and different services and making them interacting can be challenging. When architecting a serverless application, it could be useful to refer to more familiar models and practices.
First of all it can be useful to design workflows and components interactions through UML diagrams, in order to have a clean and clear view of the application and the components involved. Regarding to the implementation, the microservices theory can be helpful. Serverless functions can be seen as part of wider bounded components which communicate by mean of protocols with external representation and can be deployed independently from each other. Moreover each storage can be accessed by only one of these units and each workflow can be based on eventual consistency approaches.
Compared to a classic microservices container-based application, a serverless application contains further smaller isolated stateless independently deployable units, which are functions, which are closer to the nanoservices philosophy.

Bibliography

[1] Castro P., Ishakian V., Muthusamy V., and Slominski A. *The Rise of Serverless Computing.* URL: `https://cacm.acm.org/magazines/2019/12/241054-the-rise-of-serverless-computing/fulltext` (cit. on p. 6).

[2] Sbarski P., Cui Y., and Nair A. *Serverless Architecture on AWS, Second Edition.* Manning Publications Co, 2020. ISBN: 9781617295423 (cit. on p. 6).

[3] Mell P. and Grance T. *Platform as a Service (PaaS) definition.* URL: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf` (cit. on p. 6).

[4] Mike Roberts. *Serverless Architectures.* URL: `https://martinfowler.com/articles/serverless.html` (cit. on pp. 7, 10, 21).

[5] Radoslav Danilak. *Why Energy Is A Big And Rapidly Growing Problem For Data Centers.* URL: `https://www.forbes.com/sites/forbestechcouncil/2017/12/15/why-energy-is-a-big-and-rapidly-growing-problem-for-data-centers/#56c6ca985a30` (cit. on p. 8).

[6] Margaret Rouse. *Power Usage Effectiveness (PUE).* URL: `https://searchdatacenter.techtarget.com/definition/power-usage-effectiveness-PUE` (cit. on p. 8).

[7] Hermann De Meer Robert Basmadjian Florian Niedermeier. *Modelling and Analysing the Power Consumption of Idle Servers.* URL: `https://www.fim.uni-passau.de/fileadmin/dokumente/fakultaeten/fim/lehrstuhl/meer/publications/pdf/Basmadjian2012c.pdf` (cit. on p. 8).

[8] *Introducing Firecracker.* URL: `https://aws.amazon.com/about-aws/whats-new/2018/11/firecracker-lightweight-virtualization-for-serverless-computing/` (cit. on p. 10).

[9] *Firecracker: lightweight virtualization for serverless applications.* URL: `https://blog.acolyer.org/2020/03/02/firecracker/` (cit. on p. 11).

[10] *Invoke AWS Lambda.* URL: `https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html` (cit. on p. 11).

[11] *AWS Lambda runtimes.* URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html (cit. on p. 11).

[12] *AWS Lambda quotas.* URL: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html (cit. on p. 11).

[13] *Managing concurrency for a Lambda function.* URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html (cit. on p. 11).

[14] *Provisioned Concurrency for Lambda Functions.* URL: https://aws.amazon.com/it/blogs/aws/new-provisioned-concurrency-for-lambda-functions/ (cit. on p. 12).

[15] *Serverless Architectures with AWS Lambda.* 2019. URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-vpc.html (visited on 08/24/2020) (cit. on p. 12).

[16] *Security Overview of AWS Lambda.* URL: https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf (cit. on p. 12).

[17] DeBrie A. *The DynamoDB Book.* 2020 (cit. on p. 13).

[18] Balasubramanian G. *DynamoDB Streams Use Cases and Design Patterns.* 2017. URL: https://aws.amazon.com/blogs/database/dynamodb-streams-use-cases-and-design-patterns/ (visited on 08/31/2020) (cit. on p. 14).

[19] *AWS launches Fargate Spot.* URL: https://aws.amazon.com/about-aws/whats-new/2019/12/aws-launches-fargate-spot-save-up-to-70-for-fault-tolerant-applications/ (visited on 08/24/2020) (cit. on p. 14).

[20] *AWS security compliance.* URL: https://aws.amazon.com/compliance/services-in-scope/?nc1=h_ls (cit. on p. 14).

[21] *AWS: Overview of Security Processes.* URL: https://d1.awsstatic.com/whitepapers/aws-security-whitepaper.pdf (cit. on p. 14).

[22] *Shared Responsibility Model.* URL: https://aws.amazon.com/compliance/shared-responsibility-model/ (cit. on p. 15).

[23] Margaret Rouse. *Definition of Security as a Service.* URL: https://searchsecurity.techtarget.com/definition/Security-as-a-Service (cit. on p. 15).

[24] Gegick M. and Barnum S. *Least Privilege Principle definition.* URL: https://us-cert.cisa.gov/bsi/articles/knowledge/principles/least-privilege (cit. on p. 16).

[25] *IAM Features.* URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html (cit. on p. 16).

[26] Bruce M. and Pereira P. *Microservices In Action.* Manning Publications Co, 2019. ISBN: 9781617294457 (cit. on pp. 17, 19).

[27] *IAM Credentials.* URL: `https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html` (cit. on p. 17).

[28] Vehent J. *Securing DevOps.* Manning Publications Co, 2018. ISBN: 9781617294136 (cit. on pp. 20, 21).

[29] Vester J. *DevOps vs. Siloed Cultures.* URL: `https://dzone.com/articles/devops-vs-siloed-cultures` (visited on 08/24/2020) (cit. on p. 20).

[30] Narayan S. *Products Over Projects.* 2018. URL: `https://martinfowler.com/articles/products-over-projects.html` (visited on 08/24/2020) (cit. on p. 21).

[31] McLaughlin T. *Serverless DevOps.* 2019 (cit. on p. 21).

[32] Bride L. *Serverless Computing: Moving from DevOps to NoOps.* URL: `https://devops.com/serverless-computing-moving-from-devops-to-noops/` (cit. on p. 21).

[33] *Infrastructure as Code (IaC).* URL: `https://www.ibm.com/cloud/learn/infrastructure-as-code` (cit. on p. 22).

[34] *AWS CloudFormation template anatomy.* URL: `https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-anatomy.html` (cit. on p. 22).

[35] *AWS CloudFormation concepts.* URL: `https://aws.amazon.com/cloudformation/resources/` (cit. on p. 22).

[36] *CDK constructs.* URL: `https://docs.aws.amazon.com/cdk/latest/guide/constructs.html` (cit. on p. 23).

[37] *Serverless Framewokr compared.* URL: `https://www.serverless.com/learn/comparisons/` (cit. on p. 24).

[38] *World Quality Report 2019-20: Top software testing trends for CIOs.* URL: `https://www.capgemini.com/research/world-quality-report-2019/` (cit. on p. 31).

[39] Canalys. *Cloud market share Q4 2019 and full-year 2019.* 2020. URL: `https://www.canalys.com/static/press_release/2020/Canalys---Cloud-market-share-Q4-2019-and-full-year-2019.pdf` (visited on 08/24/2020) (cit. on p. 50).

[40] Dignan L. *Top cloud providers in 2020.* URL: `https://www.zdnet.com/article/the-top-cloud-providers-of-2020-aws-microsoft-azure-google-cloud-hybrid-saas/` (cit. on p. 51).

[41] Hecht L. *AWS Lambda Still Towers Over the Competition, but for How Much Longer?* URL: `https://thenewstack.io/aws-lambda-still-towers-competition-much-longer/` (cit. on p. 51).

Vorrei dedicare questo spazio dell'elaborato a tutti coloro che mi hanno donato qualcosa durante questo percorso.

Ad Antonio Pessolano per i suoi consigli illuminanti, i libri e i caffè.

Alla mia famiglia, che mi ha permesso di diventare chi sono oggi.

A Naomi, la mia roccia, compagna di vita e migliore amica.

Grazie!