Politecnico di Torino

Master’s Degree in Computer Engineering, Computer Networks and Cloud Computing

Department of Control and Computing Engineering

Turin, December 2023

Study, design and implementation of infrastructure as code libraries for the provisioning of a resilient cloud infrastructure model in a multi-cloud context

Thesis

By

Davide Manca

In collaboration with

REPLY LIQUID

Supervisor:
Professor Guido Marchetto

Reply Liquid Supervisors:
Dr. Stefano Martinelli,
Dr. Davide Sarais

"A river cuts through a rock not because of its power but because of its persistence."

James Watkins

Table of contents

1. Introduction....3
    1.1 Objective....4
2. Single Cloud vs Multi-cloud....5
    2.1 Single Cloud....6
    2.2 Multi-cloud....7
    2.3 Multi-cloud Adoption: Why Going For It....10
3. Infrastructure Provisioning....12
    3.1 IaC....12
    3.2 Automation....14
    3.3 IaC Tools....15
    3.4 Inner IaC Tools....16
    3.5 Outer IaC Tools....18
4. Pulumi....20
    4.1 Components And Concepts....21
    4.2 Simulation Of An Empty Starter Project....25
    4.4 Backend and State....32
    4.5 IaC tools comparison: Pulumi vs Terraform....35
    4.6 Import....39
    4.7 Templates....40
5. Project: design and implementation of a resilient cloud infrastructure model in a multi-cloud context....43
    5.1 Architecture Design....47
        5.1.1 VPC (Virtual Private Cloud)....49
        5.1.2 AWS Subnet....49
        5.1.3 NAT Gateway....50
        5.1.4 Internet Gateway....50
        5.1.5 Bastion Host....51
        5.1.6 Elastic Kubernetes Service EKS....51
        5.1.7 Load Balancer....52
        5.1.8 Autoscaling Group....52
        5.1.9 Relational Database Service (RDS)....53
        5.1.10 Elastic Container Registry....53
        5.1.11 Web Application Firewall....53
        5.1.12 Lambda....54
        5.1.13 Eventbridge....55
        5.1.14 Resource Group....56
        5.1.15 Virtual Network....57
        5.1.16 Azure Subnet....57
        5.1.17 NAT Gateway....57
        5.1.18 Bastion Host....58
        5.1.19 Azure Kubernetes Service (AKS)....58
        5.1.20 Load Balancer....58
        5.1.21 Autoscale....59
        5.1.22 Azure Database for PostgreSQL Server....59
        5.1.23 Azure Container Registry....59
        5.1.24 Azure Application Insights....60
        5.1.25 Azure Blob Storage....60
        5.1.26 Azure Storage Account....61
        5.1.27 Azure Function....61
    5.2 Implementation settings and tools configurations....65
    5.3 Code implementation for AWS provisioning....66
        5.3.1 Networking module....68
        5.3.2 Bastion Host module....71
        5.3.3 EKS module....73

5.3.4 Load balancer module....75
5.3.5 Relational Database Service module....78
5.3.6 Web Application Firewall module....79
5.3.7 Lambda Function module....80
5.3.8 ECR module....84
5.4 Results....85
5.5 Code implementation for Azure provisioning....88
5.5.1 Networking module....88
5.5.2 Bastion Host module....91
5.5.3 AKS module and Load Balancer module....94
5.5.4 Database module....96
5.5.5 Azure Container Registry module....98
5.5.6 Azure Functions module....99
5.6 Results....104
5.7 Considerations and implementation differences....106
5.8 Disaster Recovery strategy....108
5.9 Template and automation scripts....109
6. Conclusion and future scenarios....114
7. Bibliography....115

List of figures

Fig 1: Differences between single cloud and multi-cloud....5
Fig 2: Infrastructure as code workflow....13
Fig 3: Pulumi workflow....21
Fig 4: Example of bucket object in AWS pulumi code....23
Fig 5: AWS access keys credentials....25
Fig 6: Output of pulumi new aws-typescript command....25
Fig 7: Project folder content....26
Fig 8: Pulumi up output....27
Fig 9: Pulumi up output with "yes" answer....27
Fig 10: Pulumi destroy output....28
Fig 11: Pulumi Cloud online architecture....33
Fig 12: Pulumi Cloud architecture on cloud platform....34
Fig 13: Example of Pulumi import by code....40
Fig 14: Example of a Pulumi AWS EKS template....41
Fig 15: AWS cloud infrastructure project design....48
Fig 16: Azure cloud infrastructure project design....56
Fig 17: AWS configure....66
Fig 18: pulumi new aws-typescript command....67
Fig 19: Pulumi resources preview for AWS provsioning....85
Fig 20: Deployment of AWS resources (pulumi up --yes)....86
Fig 21: Pulumi Cloud resources graph view for AWS....86
Fig 22: Example of resource deployment on AWS: subnets....87
Fig 23: Calls to EKS and LB module functions in index.ts....87
Fig 24: Pulumi resources preview for Azure provisioning....104
Fig 25: Deployment of Azure resources (pulumi up --yes)....105
Fig 26: Content of StartDB folder related to the Azure Function to start the db at 7AM....105
Fig 27: Pulumi Cloud resource graph view for Azure....105
Fig 28: Example of resource deployment on Azure: AKS cluster....106
Fig 29: Example of a bash automation script: deployInfrastructure.sh....111
Fig 30: Pulumi.yaml file for template parameters....111
Fig 31: Azure Pulumi template in execution....112
Fig 32: Azure Pulumi template in execution: bash script involved in installing npm packages for Azure Functions....112
Fig 33: resources.json file containing every resource for the importing....113

1. Introduction

In the latest years a lot of companies and organizations have been considering **cloud computing** and the delivery of **computing services** as one of their top investment priorities for their business. While initial growth may have been slow, in the last 10 years, cloud services have expanded significantly. By 2006, Amazon, Google, Microsoft, and the several big techs decided to launch their own cloud divisions making cloud services (**HaaS, IaaS, PaaS, SaaS**) available to the masses and becoming so the main cloud providers available on the market. The most used and famous providers are indeed **Google Cloud, Amazon Web Services , Microsoft Azure, IBM cloud, Morpheus.**

So businesses of all sizes have adopted cloud services in pursuit of improved services, long-term cost savings and other many advantages such as speed, reliability, performances, productivity. Thanks to this, several industries have been taking into account the cloud transition, better known as cloud adoption and, as a matter of fact, as soon as companies decided to adopt cloud services by the main cloud vendors, the public, private, hybrid clouds started to be seriously considered by them for migrating their data and applications in a more secure, efficient, resilient, scalable and self-managed environment.

Due to this adoption, between 2015 and 2017 various organizations doubled SaaS service offerings and in India new 55 SaaS companies were born. Nevertheless, infrastructure-as-a-service (IaaS) has been the largest area of growth; suffice it to know that in 2018, the IaaS market was dominated by five providers: Google, Amazon, Microsoft, Alibaba, and IBM.

But focusing more on what cloud computing is, it is necessary to precise that we are talking about storing, delivering software on demand, analyzing data, testing and building applications, creating cloud-native applications, infrastructure resiliency, accessing data and programs on remote servers that are hosted on the big facilities containing physical servers grouped by racks and divided in multiple physical region all over the world, reachable through the internet instead of the computer's hard drive or local server. Cloud computing so enables organizations to vastly reduce operational costs, increase efficiency, and become leaner, making them more adaptable to change. As speed and agility become essential for digital economy success, organizations deploy more applications, assets, and workloads to public clouds.

Examples of cloud native applications are the most famous ones everybody knows such as **YouTube**, **Facebook**, **Instagram**, **Google** (and all of its services like Calendar, Gmail, etc..). This means that in daily life under the hood cloud computing is constantly used without people finds it out. Sending emails, editing documents, watching movies or TV, listening to music, playing games, or storing pictures and other files, it is likely that **cloud computing** is making it all possible behind the scenes.

In the near future it will continue to grow providing many benefits thanks to its never ending expansion and to the efficiency it brings to companies which adopts it.

The future of cloud computing is bright and will provide benefits to both the host and the customer.

1.1 Objective

Companies in several kinds of industries (automotive, insurance, banking, e-commerce) has adopted cloud for delivering their services to their customers and the cloud solution used are based on a proper cloud architecture and infrastructure design.

The purpose of this thesis, developed in collaboration with **Reply Liquid** company, is to focus on the study of a specific infrastructure-as-code (**IaC**) tool, **Pulumi**, for designing, creating and making the provisioning of a robust, resilient and redundant cloud infrastructure model for applications deployment which also faces sensitive topics such as disaster recovery and fault tolerance. In particular the purpose is to cover as many use cases as possible in which the main need is to deploy applications for delivering specific services to a specific target of customers. Therefore not only the reasons, the potentialities, the strengths and weakness of the proposed model are demonstrated but also the functionalities, the peculiarities and the Pulumi tool characteristics as well as the comparison with other IaC tools such as Terraform. The infrastructure design will involve the concentration mainly in:

- the study, the implementation of high availability and fault tolerance topics under the considered architecture and the detailed description of how the IaC tool works in order to deploy, to update, to destroy cloud resources;
- the simulation of a possible disaster recovery solution related to the considered infrastructure in order to cover such a relevant a security aspect like infrastructure resiliency and reproducibility;
- the realization, the connection, the software implementation of the cloud resources and the meticulous description of project lines of code;
- the automation of both the provisioning and disaster recovery processes and the eventual publishing to public repositories

2. Single Cloud vs Multi-cloud

Single cloud and multi-cloud are two different important approaches an organization can adopt and they differs each other from the number of cloud service providers a company works with. So far the more widely strategy used by modern enterprises is multi-cloud, although there are some advantages to sustain a single cloud vendor solution.

According to an Oracle research and to a Gartner's article, almost 76% of companies decide to rely on more than just a cloud provider in order to get more resilience for their data and a better management of the company's infrastructure.
Multi-cloud therefore is the solution used by an organization for supporting its applications, and workloads among many cloud vendors (usually no more than 3).

Organizations are increasingly deploying applications, data, workloads, and other assets to the cloud to increase efficiency and agility while reducing operational costs. Single cloud and multi-cloud strategies are predominant cloud storage frameworks in the space, but there are key differences business leaders should know before building their cloud infrastructure.

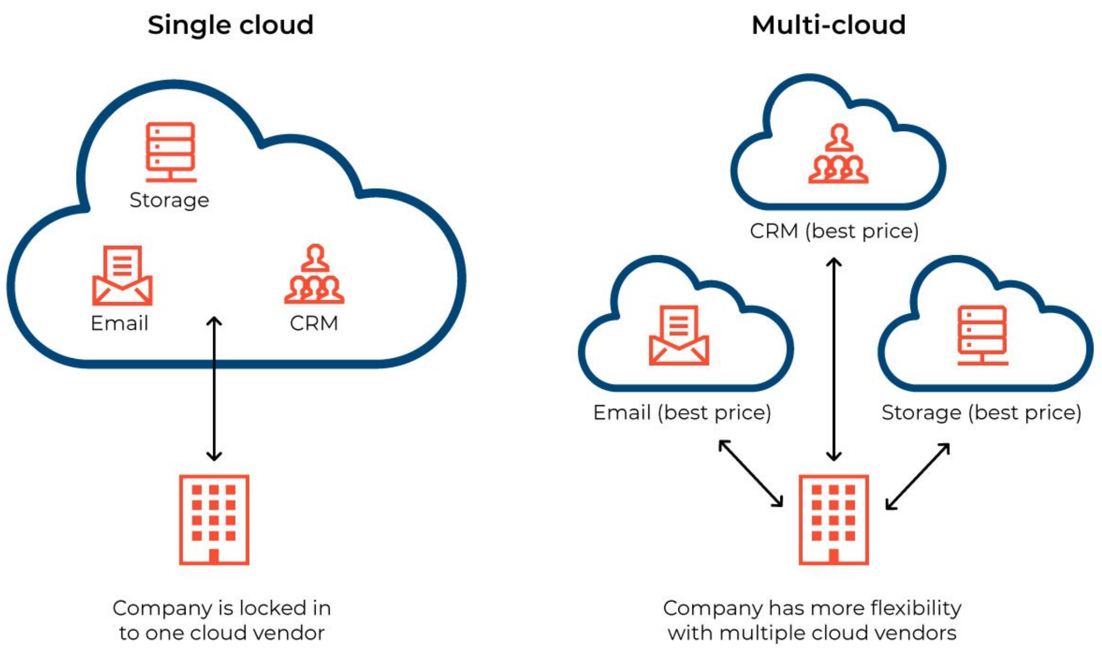

Single cloud
Storage
Email
CRM
Company is locked in to one cloud vendor
Multi-cloud
CRM (best price)
Email (best price)
Storage (best price)
Company has more flexibility with multiple cloud vendors

Fig 1: Differences between single cloud and multi-cloud

2.1 Single Cloud

As mentioned earlier, single cloud is a cloud computing model which consists in relying on a single cloud service provider to deliver cloud services. This strategy is very suitable and useful for companies which has some strict organizational rules for their data and workloads, or a small number of skilled cloud engineers for an efficient cloud management or for cases in which there is a reasonable amount of workloads that can be easily managed by a single cloud provider.

Of course single cloud presents some strengths and drawbacks and below there are the most important ones.

Advantages:

- **Simplification**

  The management is simplified because of the fact that business are supposed to have limited workloads and data deployed to the cloud, so this means a single cloud provider simplifies the creation of the cloud architecture underneath without creating additional drawbacks and limitations.

- **Privacy management**

  Privacy and control are maintained

- **Better management**

  Outsourcing applications, functions, data, and workloads to a single cloud provider vastly simplifies orchestrating the management processes for the enterprise. This can help to eliminate manual administrative tasks and give internal teams more resources to focus on higher priority items.

Drawbacks:

- **Costs**

  It costs more to have all workload managed by a single vendor.

- **Failures**

  Risk of cloud resources unavailability due to any cloud issues that result in a single point of failure.

- **Vendor lock-in**

  Cloud providers usually impose strict contractual terms on the partnership with the organizations and the lock-in clearly creates some troubles when companies would like to stop or to modify the terms of the contract, so there can be the necessity to seek other cloud services belonging to other cloud providers.

- **Inflexibility**

  Every cloud provider has a slightly different set of cloud services and specialties, hence organizations which deal with just a single cloud provider are bounded only to the services of the lone provider; obviously this can represent a relevant limitation to the flexibility of the company when it decide to expand its core and to rely on other innovative solutions.

2.2 Multi-cloud

This is the major used solution among companies when automation, better management of services, cost reducing, are taking into account.

Multi-cloud describes a cloud computing model where organizations use multiple cloud providers for their infrastructure requirements. The name multi-cloud refers to the use of multiple cloud providers, accounts, availability zones, premises, or a combination of them, differently from the single cloud strategy.

It is a special case of hybrid cloud that is defined as the provisioning, the utilization and the management of coordinated services based on a combination between internal and external cloud services.

Multi-cloud offers so much flexibility to the companies about the cloud services choice and as a result it reduces effectively the single vendor lock-in; furthermore it has to be perceived as an approach which consists in more cloud services provided by many public or private cloud vendor, but to be more precise multi-cloud term refers specifically to the use of **multiple public cloud providers.** The multi-cloud strategy is suitable for companies that are unable to fulfill business requirements with a single cloud, for the ones whose workloads are considerably big, varying, and needs to be distributed among many cloud providers.

As single cloud has its own strength and weakness, the multi-cloud solution presents several benefits too but less disadvantages.

Some of them are represented below in a nutshell:

Benefits:

- **Disaster Recovery**

  Multi-cloud facilitates robust disaster recovery strategies. Data and applications can be replicated across clouds to ensure business continuity in case of a disaster.

- **Negotiation**

  Having relationships with multiple cloud providers can provide negotiation leverage when discussing contracts, pricing, and support agreements.

- **Best-of-Breed approach**

  Different cloud providers excel in different areas. With a multi-cloud strategy, it is possible to choose the best-suited services from different providers to optimize performance, cost, and features for each specific task.

- **Avoids vendor lock-in**

  Using a single cloud provider can lead to vendor lock-in, where a customer might become heavily dependent on their services and technologies. Multi-cloud permits to diversify services, reducing the risk of being locked into one provider.

- **Greater flexibility**

  When organizations partner with multiple cloud providers, they can select those whose service offerings best match each specific area of their business. That ensures they gain access to the most appropriate range of capabilities for optimal performance and efficiency.

- **Cost efficiency**

  Similarly, organizations can compare competitive price points between different service providers and opt for those that offer the best balance of pricing and service quality. This enables them to drastically reduce their cloud operational costs and IT spend

Disadvantages:

- **Complexity**

  Managing multiple cloud providers and services can increase complexity in terms of deployment, integration, monitoring, and security. Each provider may have different interfaces, APIs, and management tools.

- **Inconsistency**

  Every cloud service provider has a unique approach to cloud computing, often deploying unique tools, systems, and policies. Relying on multiple cloud providers means learning and engaging with a wide variety of different strategies. This can be disconcerting for employees who have to access multiple clouds.

- **Security challenges**

  Cloud security is usually managed within the platform, using tools built by the cloud service provider. While these tools are highly effective, they do vary between different cloud vendors. This approach can create inconsistency as it relates to cybersecurity for the organization. Ultimately, these inconsistencies may create cybersecurity gaps.

- **Increased costs if mismanaged**

  While multi-cloud can potentially reduce costs through competitive pricing, mismanagement can lead to unexpected expenses due to overprovisioning, inefficient resource allocation, and data movement costs.

- **Data consistency**

  Maintaining data consistency and integrity across multiple clouds can be difficult, especially in scenarios where data needs to be synchronized in real-time.

- **Security and compliance**

  Security becomes more complex in a multi-cloud environment. Each provider may have different security measures, and ensuring consistent security policies across multiple clouds can be challenging. Compliance with data regulations may also become more intricate.

2.3 Multi-cloud Adoption: Why Going For It

While multi-cloud offers numerous benefits, it's important to manage the complexity that can arise from dealing with multiple providers, architectures, and operational practices. Effective governance, management tools, and a well-defined strategy are crucial to make the most of a multi-cloud environment.

Furthermore the requirement of excessive resources and robust strategies to optimize cloud migration is not negligible.

In an alternative way, many organizations opt for a hybrid approach, combining elements of both multi-cloud and single cloud strategies. This allows them to leverage the benefits of both while minimizing the drawbacks. Critical workloads might be hosted on a single cloud for simplicity and reliability, while non-critical workloads are distributed across multiple providers for flexibility.

Evaluating organization's goals, existing infrastructure, risk tolerance, required features, and available resources are a must before making a decision. Many organizations find success in adopting a hybrid approach to balance the advantages and challenges of both multi-cloud and single cloud strategies.

Nevertheless, nowadays businesses are increasingly adopting and moving to the multi-cloud model because it allows to work globally with data and applications spread across various servers and datacenters.

There are five elements to take in consideration to go for multi-cloud model which motivate and justify this choice:

- **Integration**

  extending architectural, network, pattern related to services composition logics (network management, multi-cloud connectivity, NAAS)

- **Automation**

  It becomes central because the will is to abstract as much as possible to avoid the manual management and human intervention minimizing errors and risks of failures (IaC management, DevSecOps automation, app portability, etc..)

- **Security**

  It adjusts to the complexity of heterogeneous environments so delivery and stability are reinforced (security configuration management, identity governance, access management)

- **Governance**

  Technological complexity, financial, economical and eco-sustainability aspects have to be under control

3. Infrastructure Provisioning

3.1 IaC

This paradigm stands for Infrastructure as Code and in the cloud context is a very used instrument which allows to manage infrastructural components through coding instead of manual and separate processes. IaC is applied when the components to be programmed expose APIs to be used for provisioning purposes. It marries with both single cloud and multi-cloud but in the latter context the approach can be totally exploited, with the peculiarities it has.

Provisioning is the cloud environment process of setting up and configuring computing resources, such as virtual machines (VMs), storage, networking, and other services, to create an environment that meets the needs of applications or workloads. It involves allocating and configuring resources based on demand, allowing to quickly deploy and scale infrastructure in the cloud.

We can distinguish two types of provisioning:

- **Manual provisioning**

  Administrators or users manually configure and allocate resources through the cloud provider's management console, command-line interface, or other tools. They specify parameters such as the type of virtual machine, storage capacity, networking settings, and more. Manual provisioning offers flexibility in customization but might be slower and more prone to errors.

- **Automated provisioning**

  Involves using scripts, templates, or orchestration tools to define the desired infrastructure configuration. These scripts or templates can be used to create consistent and repeatable deployments. Automation tools like Infrastructure as Code (IaC) frameworks, as mentioned earlier (e.g. Terraform or AWS CloudFormation), enable to define infrastructure requirements in code and then automatically provision and manage resources according to that code

Infrastructure as Code allows therefore to create and provisioning a designed architecture by using programming languages that involve the use of the APIs for authoring and pushing up the cloud resources. While it enable to build the infrastructure, all of the benefits which come from programming languages can be leveraged in terms of computational efficiency, time of execution, complexity, so this enrich a lot the amount possible reproducible scenarios by using IaC tools and allows at the same time to have a clean, ordered vision of the cloud infrastructure and a shareable, open source, code project which can be pushed in repositories.

Using IaC tools involves to have made some architectural choices in a previous moment; in particular it is needed an automation strategy where choosing what has to be automated on the cloud provider plays a very important role, as well as choosing which IaC tool to use (inner or outer, as explained in a few chapters later), defining a model rich of abstract configurations which can be applied in many environments and evaluating the best solution basing on expecting performances, processes, needed compliance requirements in order to have them homogeneously in a multi-cloud scenario in terms of agility and compliance

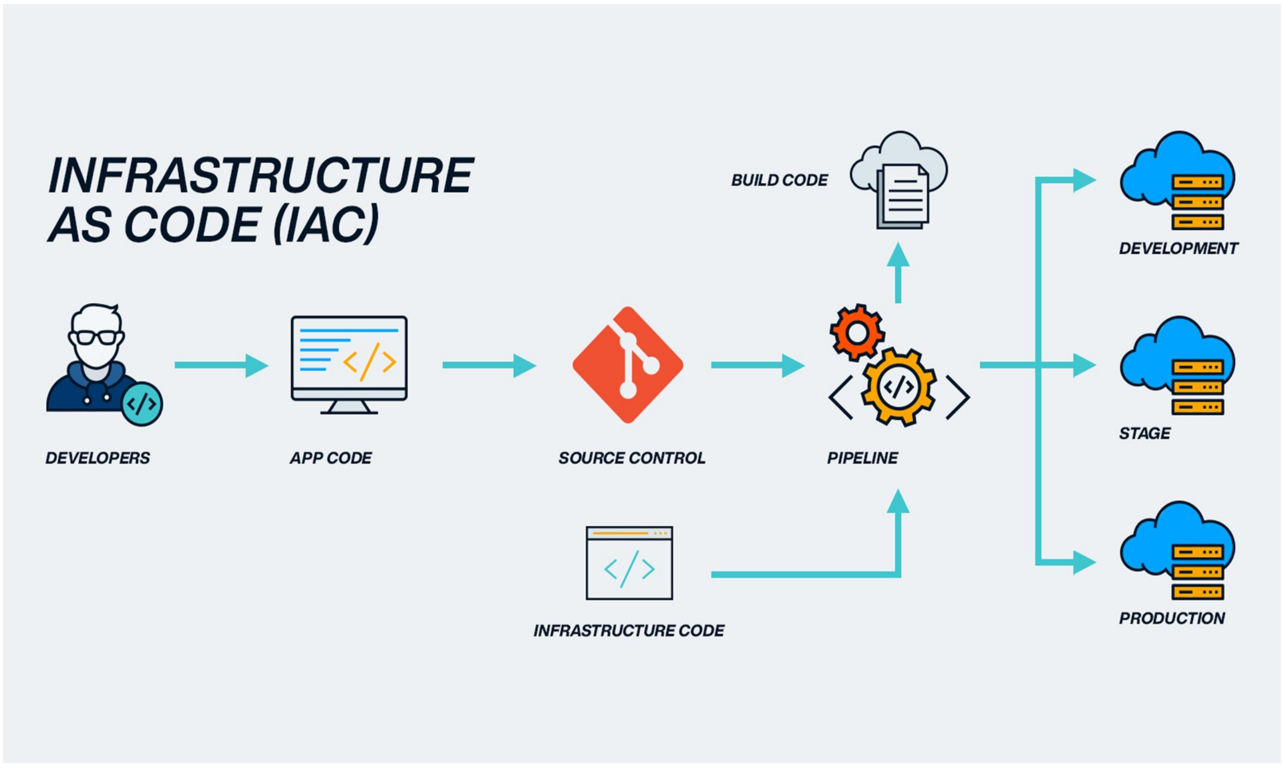


Fig 2: Infrastructure as code workflow

3.2 Automation

Infrastructure as code brings an important aspect to take into account and that is considered in enterprise environments: automation.

Organizations constantly and daily continuously deliver services to their customer, during either the day and the night so there is the need to have stability, efficiency and guarantee of the cloud infrastructures they rely on.

To achieve this behavior automation plays a very important role because it allows to scale up and down cloud resources and services when it is needed by the companies (depending obviously on their necessities and demands from the customers, the market), to manage independently the workloads, ensuring in the meanwhile that failures are handled or in a better perspective even avoided, to spawn, creating, destroying, powering on and off any cloud resource automatically saving hence costs and energy, avoiding wastes and improving performances.

As good as the reasons above, another aspect to highlight is the prevention, as much as possible when it is needed, of the human intervention in processes, data, workloads management in order to avoid highly probable failures and errors due human handling as well.

Previously, installing, configuring, and updating programs for cloud servers was a manual task for IT managers. Similarly, the network teams had to manually store and manage the configuration data. This was not only a time-consuming and tedious process, but also called for the collaboration of several IT personnel. The most pressing problem was scalability because administrators found it challenging to bring up new servers quickly enough to keep up with the speed and scope of continuously evolving business operations. Furthermore, hiring and managing a team increased costs significantly.

It is important to be aware that some IaC tools are involved in the infrastructure setup, while others manage the infrastructure or the applications.

Automation in infrastructure as code tools can be achieved by using the proper tool based APIs useful to automate processes about the creation, deletion, scheduling of the resources; basing on which tools is used, a pool of APIs which can be used to perform the provisioning of a given infrastructure is exposed to the programmers. Those APIs have their own documentation which describes how the function are built, which parameters take in input, which the outputs are, and they go to call commands of cloud provider (other APIs but CP based). Successively many environments will be created, such as testing, development, production in order to have different stages where to deploy and to separate the concepts related to a specific context: this stages usually are known as stacks.

Examining it in more detail, automation permits to achieve also:

- Managing the configuration of cloud resources by ensuring that they adhere to specific configurations and settings. This includes applying security policies, software updates, and patches to maintain a secure and compliant environment.
- Auto-scaling applications to automatically adjust the number of resources based on demand. Load balancers can distribute traffic across instances, ensuring optimal performance and high availability.
- Automated testing, building, and deployment processes ensure that changes to applications are tested and deployed consistently and quickly. Automation is central to modern CI/CD pipelines (continuous integration/continuous delivery)
- Any repetitive task, such as user provisioning, access management, or routine maintenance, can be automated to save time and reduce errors.

3.3 IaC Tools

Nowadays A wide range of Infrastructure as Code tools available help IT Managers address the following infrastructure management tasks: **provisioning, configuration, deployment, orchestration.**

According to IaC definition, IaC tools are software solutions that enable to define, provision, manage, and update the IT infrastructure using code, rather than manual processes. With IaC tools, it is possible to treat infrastructure configurations in a similar way to software code, allowing to automate the setup and maintenance of cloud resources, servers, networks, and more, exactly as discussed and clarified so far.

The most useful idea IaC tools allow to create are templates. It consists in authoring infrastructures as templates so that they can be stored, shared, and reused in other contexts fitting many different use cases in order to solve many different problems with the same solution (or with some edits), maintaining the same configurations in terms of infrastructure's architecture.

Below a list of existing IaC tools is represented by highlighting per each its own advantages, peculiarities and possible drawbacks.

More precisely inner IaC tools and outer IaC tools will be distinguished and compared.

3.4 Inner IaC Tools

Many IaC tools are available to perform the tasks and operations mentioned in the previous paragraph.

In this section proprietary IaC tools are faced, let's say "inner" improperly. These are the ones offered by the cloud providers in order to do provisioning, orchestration, scheduling, deployment.

As any IaC tool the proprietary ones allows to manage an infrastructure under those points of view, they might look working more coordinately with all of the services present in the same cloud provider but any of them has the same more relevant problem that refers to portability, more precisely to the deployment in a multi-cloud context.

An inner IaC tools can only manage the resources and the services of its related cloud platform (through coding) but it cannot affect the management of other environments, so this represent obviously a usage limit.

Let's list the main and well-known ones below:

- **AWS CloudFormation**

  Allows to manage infrastructure and automate any deployments using code. The main difference comes down to how intimate CloudFormation is to AWS in that it only works with AWS IaC. However, it makes up for this by being integrated with the entire platform.

  It is possible to write CloudFormation templates in both YAML and JSON, managing, scaling, and automating AWS resources fast and straightforward. Furthermore, it is possible to preview all the changes before deployment, which helps to visualize the impact a set of changes will have on resources, services, and dependencies.

CloudFormation also offers Rollback Triggers that enables to restore infrastructure to a previous state, guaranteeing controlled deployments in case of any mistakes or issues.

This tool’s close relationship with AWS enables infrastructure stacks to be deployed in several regions and accounts using the same CloudFormation template.

- **Azure Resource Manager (ARM)**

  Another top IaC tool is Azure Resource Manager, which is Microsoft's tool to manage Infrastructure in its platform. It uses the Azure Resource Manager template (ARM templates) to handle dependencies and infrastructure. For example, resources can be organized into groups, delete them, control access levels to resources, just to name a few.

  Controlling access to services and resources is made easy when using Azure, as it supports Role-Based Access Control (RBAC) natively. On the other hand, it is possible to finetune the scope of access with management groups, subscriptions, and resource groups. Additionally, lower levels of hierarchy inherit settings from higher levels, ensuring that policy enforced by higher levels is applied at all desired lower level groups and resources.

  ARM offers templates that can deploy resources in parallel, making it possible for faster deployments. Furthermore, the system comes with great organization tools, letting to attribute tags to resources, organizing the groups, and checking the costs of any resource which shares a specific tag.

3.5 Outer IaC Tools

After having describe IaC tools belonging to cloud providers here will be presented the ones cloud-independent.

Outer adjective wants to underline the fact for which the provisioning, the deployment and the creation of objects related to a cloud infrastructure can be performed upon any cloud provider since these tools are not dependent to a specific vendor. This means that an outer IaC tool can manage resources of AWS rather than Azure or IBM cloud and this is achievable because the software underneath takes care of using the specific IaC tool needed basing on where the deployment is taking place. For instance a project developed for AWS with such a tool will be deployed to AWS by means of CloudFormation, therefore the outer IaC tool will go to use the specific cloud proprietary IaC tool.

The greatest advantage as mentioned before is the vastness of manageable cloud platform, besides other kinds of strength that will be presented later on.

The following ones are the most useful, efficient and famous outer IaC tools:

- **Pulumi**

  IaC tool that sets itself apart from the rest Infrastructure as Code platforms by providing greater flexibility. It supports several programming languages such as Python, JavaScript, C#, Go, and TypeScript. By providing more support for language options, Pulumi can fit a greater variety of IaC DevOps use cases and reach the majority of developers. More languages also mean there are more tools and frameworks readily available for building and testing the infrastructure. A unique aspect of Pulumi as Infrastructure as Code tool is that it does an excellent job keeping core concepts and features of established tools such as Terraform, while offering support for the cloud giants AWS, GCP, and Azure Cloud. Additionally, it has automation options for deployment delivery, quality assurance using policies, easy auditioning, comprehensive identity control.

  All of these capabilities come with high-quality documentation with easy-to-follow tutorials.

- **Terraform**

  It is one of the most popular IaC tools in the market. It's an open-source project with incredible flexibility, supporting all the most prominent cloud platforms, including AWS, GCP, Azure.

  It also offers support to many providers such as DigitalOcean, GitHub, Cloudflare, and many others. Furthermore, Terraform also allows resource destruction through source control. This capability is essential when manipulating hybrid clouds, where plans can be made across multiple cloud providers and infrastructures, all while using the same workflow. Primarily, Terraform improves reliability by ensuring the Infrastructure as Code plan is consistent across all different cloud providers. In addition, the CLI can be used to execute a validation check using the command **terraform plan**, where all configurations are measured and validated. This aspect ensures the result meets expectations to avoid any mistakes, destruction of resources, and potential extra costs.

  Due to Terraform's open-source nature, many essential tools and scripts are designed to improve Terraform's solid foundations.

Infrastructure as Code is the future when it comes to managing cloud resources due to its effectiveness and reliability. The IaC tools we have outlined will significantly improve the efficiency of any project by automating the most laborious tasks while promoting a safer environment and maintaining consistency. Over the past few years, many companies have switched to IaC, which leads to less time spent dealing with the WebUI provided by their cloud platform and inconsistent resources. Many companies are still getting used to using Infrastructure as Code tools in their workflow, which often translates to teams not having a CI implemented for it. Generally, leaving a single developer working with Terraform scales poorly and generates a bottleneck in development.

Reached this point, since the project of this thesis (as it will be explained successively) has been developed with Pulumi outer IaC tool, it is proper to describe how it works, how it is made, its characteristics in detail. For this reason the next chapter is completely dedicated to Pulumi.

4. Pulumi

Pulumi is a free open-source platform that allows developers and operators to define, deploy, and manage cloud infrastructure using programming languages like Python, JavaScript, Go, and .NET. Unlike traditional Infrastructure as Code tools, which often use domain-specific languages, Pulumi enables to use real programming languages to describe the cloud resources.

Pulumi allows to create and manage infrastructure components like virtual machines, databases, and networks using code that closely resembles the code written for applications. This approach makes it more intuitive and accessible for developers to define and manage their cloud infrastructure.

This outer IaC tool supports multiple cloud providers, including AWS, Azure, GCP, and Kubernetes. It offers both declarative and imperative styles of defining infrastructure, by specifying the desired state of the resources or the steps to reach that state.

The platform keeps track of the state of the infrastructure, understanding and managing the differences between the code's intended state and the actual state in the cloud. This makes it easy to apply changes incrementally and maintain a consistent infrastructure configuration. It tracks changes made to the code and applies them to the cloud resources, this helps therefore keeping the infrastructure always aligned with the code and makes updates more manageable.

Moreover it also fosters collaboration among team members and integrates seamlessly with CI/CD pipelines. It provides libraries and modules for common cloud resources, and its capabilities can be extended through custom providers.

The goal of Pulumi is to bridge the gap between developers and operations teams by offering a way to manage infrastructure using the same languages and tools used for building applications. This approach can lead to smoother collaboration, faster deployments, and more efficient infrastructure management overall.

Pulumi programs are written in a general-purpose programming language, among the available ones (Typescript, Javascript, Python, Go, .NET, YAML), describe the composition of the cloud infrastructure to create, so resources are allocated and their properties correspond to the desired state to achieve. Properties themselves can be used among resources to manage dependencies, for instance a resource output can be used by another resource to perform correlated operations.

Let’s have a deep look at the elements which compose this tool in order to understand its properties, its functionalities, its strength with respect to other IaC tools, how it works.

4.1 Components And Concepts

The following illustration highlights the interaction between the components of the model that successively will be explained in detail.

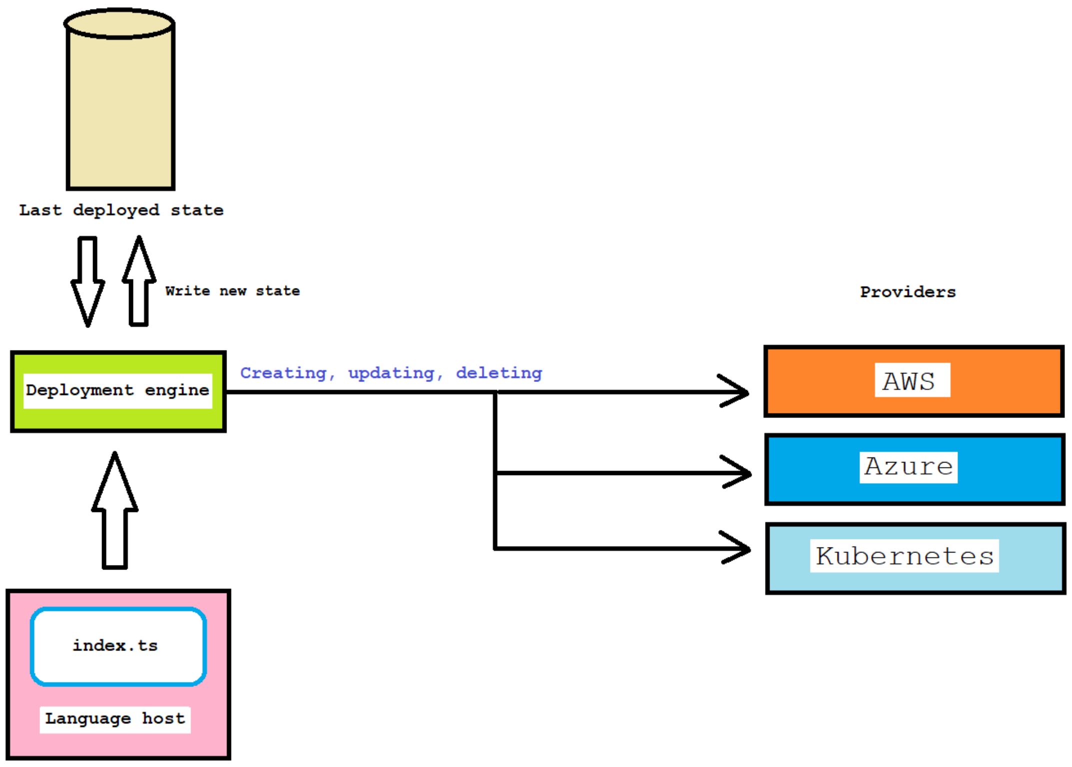


Fig 3: Pulumi workflow

Briefly, after having installed pulumi and then pulumi CLI on the machine, the program resides in a directory which contains the entire project, the program's source code plus the metadata needed to run the program itself. Only when the cloud platform to which deploy the infrastructure has been chosen (by logging on it) the program is ready to be executed; on a terminal command line "pulumi up" command has to be typed in order to start the deploy of the resources in the program and then a preview with the live process will be shown on the same terminal.

A desired state model is using by Pulumi for managing the cloud infrastructure, where current state and desired state are distinguished as well. As it is appreciable on figure 11, the **language host** is the component in charge of running, so executing the pulumi program (identified by index.ts file, just for making an example), and even of setting up an environment used then by the deployment engine to archive and to register resources.

The **language host** is composed of two macro blocks: the language executor, a binary responsible for launching the runtime for the language the program is written in, and the language runtime in charge of either to prepare the program to be executed and to observe the execution itself for detecting resources registrations. The latter block is a regular package, for instance Node is contained in @pulumi/pulumi package available on npm and python is contained into the pulumi package available on PyPI.

The language runtime after the detection of the resource registration requests, send it back to the to the upper block, as shown in the picture.

The **deployment engine** is embedded in the pulumi CLI and it plays the most important role because it is the block responsible for performing all of the operations and procedures needed to set the current state of the infrastructure into the desired state expressed by the program's code.

Current and desired states are two temporal states which are handled by (and depend on) the Last Deployed State block as shown on the figure 11.

In the code resides the calls to the pulumi APIs which can be related to creation, updating, deletion of cloud resources, depending obviously on the used cloud provider. The code so represent the infrastructure to deploy.

When a resource registration request is received from the language host block, the deployment engine verifies the current existing state to determine if that specific resource had been created in a previous moment. If it had not, the block utilizes the resource provider in order to create that resource otherwise the engine collaborates with the resource provider itself to compare the old state of the resource with the new desired one expressed by the program to verify if anything has changed. If some modification has occurred they will perform the resource updating by understanding whether this process can be done just in place or by replace the old resource with the creation of a new version of it deleting so the older one. This decision depends either on which are the properties to change on the resource and on the type of the resource itself.

For the deletion process, when the language host communicates to the deployment engine the completion of the pulumi program execution the engine checks if any of the existing resources has a new resource registration request and for the ones whose this condition   does not match the engine itself schedules their deletion.

The **resource provider**, as the language host, is composed of two blocks: the **resource plugin** which is the binary of a resource and it is used by the deployment engine, and the SDK which establishes bindings for each type of resource manageable by the provider.

SDKs are regular packages such as *@pulumi/aws* package for Node on npm or *pulumi_aws* package available on PyPI. These packages will be added to the project and they will download the resource plugins by executing under the hood *pulumi plugin install* command.

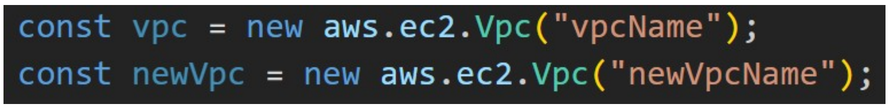
While there are resources which does not depend on others Pulumi can execute operations consisting in resources deployment in parallel, whenever it is possible as well, but when an output of a resource has to be used as input for another resource the deployment engine detects and records the dependency between these two objects and so the deployment might be scheduled sequentially. This is true in particular for those resources whose modification of a parameter involves in advance the deletion and the creation of a new instance, such as a kubernetes cluster. Dependency among resources, under a coding point of view, is manageable by using the *dependsOn* attribute available for each resource, and on its own documentation of course.

The replacing process is more complex than it appears: in fact if a resource must be replaced Pulumi will attempt to create a new copy of the object before removing the older one because the infrastructure in this way can be updated without downtime. If some failures occurred in the creation of the new one and the older copy of the resources had already been deleted this obviously will bring to an irreversible situation which causes relevant issues about current and desired state realignment.

This behavior is automatically standard on Pulumi but it can be forced by specifying *deleteBeforeReplace* option.

Some problems can occurs obviously when different resources or different versions of a same resource take the same name: in this case it has to be changed properly in order to avoid this kind of error.

Let's make a practical example in order to have a better comprehension.



```
const vpc = new aws.ec2.Vpc("vpcName");
const newVpc = new aws.ec2.Vpc("newVpcName");
```

Fig 4: Example of bucket object in AWS pulumi code

In the figure above two aws virtual private cloud objects are created by passing two different names as input parameters, according to the previous motivations about the resources naming.

Here the language host component is called up when the first aws.ec2.Vpc is attempted to be created to send the resource registration request to the deployment engine and then it resumes the code execution of the program. It is important to underline the following aspect: when the call new aws.ec2.Vpc returns it does not mean the resource has been created to the AWS platform but it means the language host has designed it as part of the desired state of the infrastructure to achieve. Language host and deployment engine work concurrently

If the examined resource had not been created previously the last deployed state has no resources so the engine knows that the vpc has to be created for the first time. It uses therefore the AWS resource plugin to create the resource itself and the same plugin will use the AWS SDK in order to create the object upon the cloud platform.

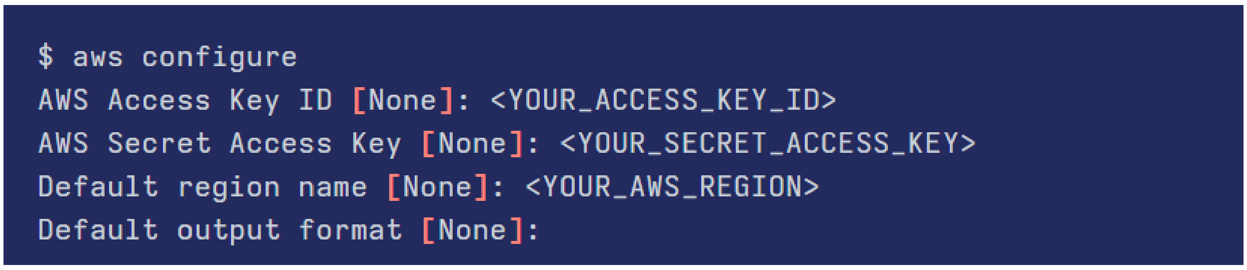
When creating operations are done the deployment engine records the newly created resource information on its state file.

By going on with example, the language host continues to execute the program and the same previous operations are performed so there will be another resource registration request related to the newVpcName object. Since there are no dependencies between the objects, the engine is free to operate and to process the request in parallel.

If one of the resources, let's consider the first one for simplicity, had been modified Pulumi uses the last deployed state component to perform the set of changes needed to update the infrastructure because Pulumi itself rely on a desire state model as specified so far. In this case the language host runs the program and when vpc variable is affected to call the API about the vpc creation a new resource registration request is sent to the deployment engine. Since there is already as a resource with the same name in the current state the engine asks the provider to compare the current state, or the one related to the last run executed, with the desired state expressed by the program. The engine so determines that it is able to update the resource properties without creating a new instance of it. Only when these operation are completed, the current state is then updated with the related changes.

4.2 Simulation Of An Empty Starter Project

This section dedicates a brief paragraph about how to manipulate a Pulumi project and how it works.

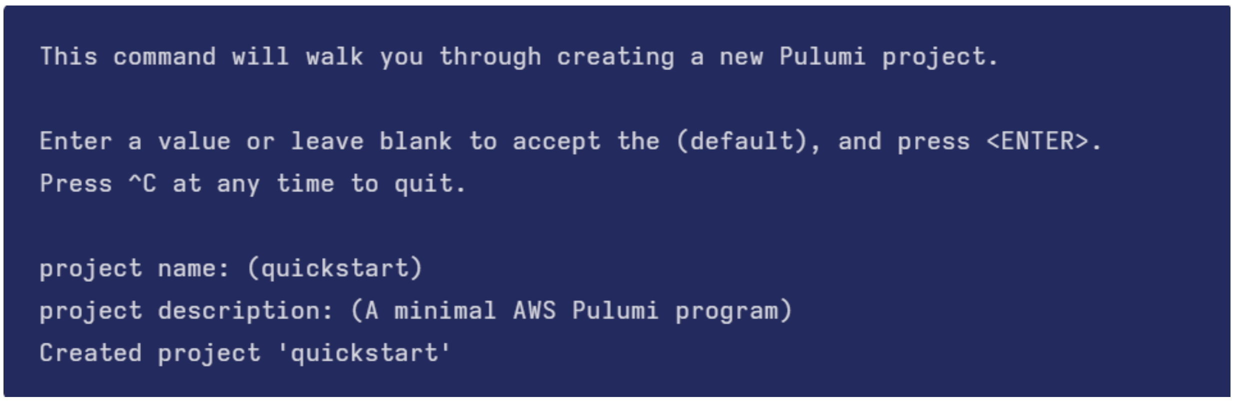


```
$ aws configure
AWS Access Key ID [None]: <YOUR_ACCESS_KEY_ID>
AWS Secret Access Key [None]: <YOUR_SECRET_ACCESS_KEY>
Default region name [None]: <YOUR_AWS_REGION>
Default output format [None]:
```

Fig 5: AWS access keys credentials

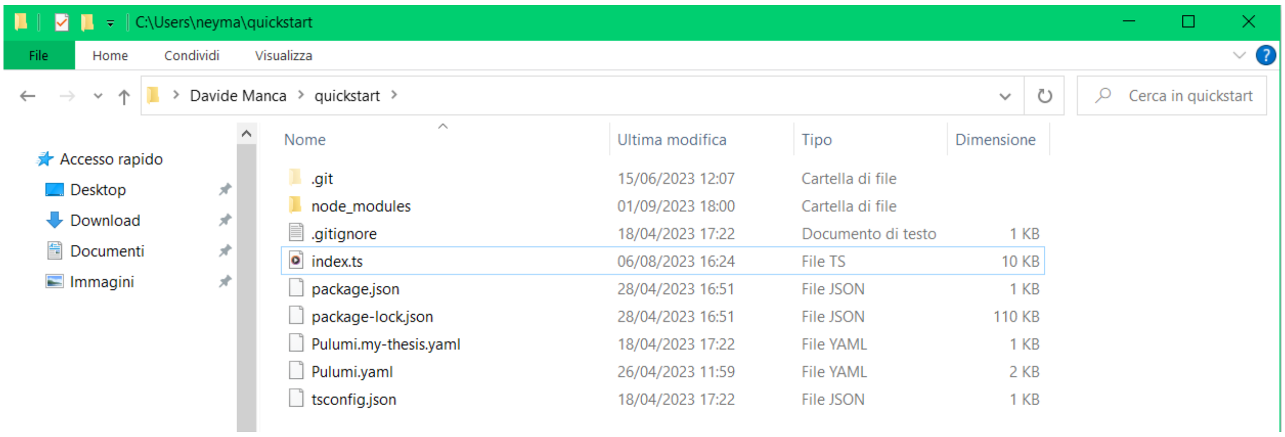
First of all the installer file has to be installed on the machine: in this case by going physically on the pulumi official's webiste https://www.pulumi.com/docs/clouds/aws/getstarted/begin/ or by typing **choco install pulumi** on the Windows command line or Windows Powershell (the last option suppose to have Chocolate already installed). Then a new account on https://app.pulumi.com/signup has to be created in order to manage Pulumi's open source SDK with ease. The account gives the opportunity of having a bult-in state and secret management, integrated with source control and CI/CD, and offers a web console which make it easier to visualize and handle the various clouds infrastructures. Successively a cloud platform on which deploying the infrastructure has to be chosen so this implies signing in to the cloud provider account and this can be performed by command line (for instance if AWS is used the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY credentials has to be set up. These can be done even by using the cloud CLI).

Going on, the next steps are creating an empty folder and to use pulumi new *cloud-language* command, where *cloud* stands for the cloud platform used (e.g. aws) and *language* stands for the used language, to create the project with which programming and authoring the infrastructure (e.g. **pulumi new aws-typescript**).



```
This command will walk you through creating a new Pulumi project.

Enter a value or leave blank to accept the (default), and press <ENTER>.
Press ^C at any time to quit.

project name: (quickstart)
project description: (A minimal AWS Pulumi program)
Created project 'quickstart'
```

Fig 6: Output of pulumi new aws-typescript command



Nome	Ultima modifica	Tipo	Dimensione
.git	15/06/2023 12:07	Cartella di file	
node_modules	01/09/2023 18:00	Cartella di file	
.gitignore	18/04/2023 17:22	Documento di testo	1 KB
index.ts	06/08/2023 16:24	File TS	10 KB
package.json	28/04/2023 16:51	File JSON	1 KB
package-lock.json	28/04/2023 16:51	File JSON	110 KB
Pulumi.my-thesis.yaml	18/04/2023 17:22	File YAML	1 KB
Pulumi.yaml	26/04/2023 11:59	File YAML	2 KB
tsconfig.json	18/04/2023 17:22	File JSON	1 KB

Fig 7: Project folder content

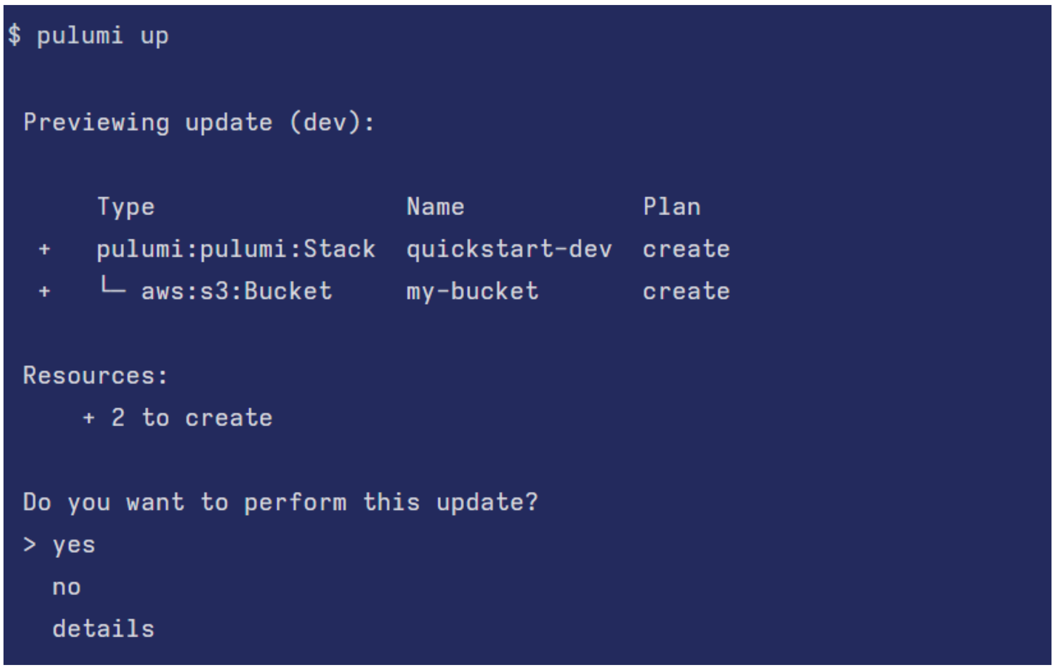
This will create a basic project on the created folder with all the related files about metadata, node modules, stack configuration, and a general code with some resources example provided by pulumi.

When the previous command is performed on terminal an interaction with the user is established where parameters as project name, project description, region to deploy, stack name, are required.

Into the folder an index.ts file is present with a little example of a resource creation and obviously that is the entry point of the program, of the infrastructure, that has to be edited by developing the logic related to the architecture to create, so the resources to program that compose it.

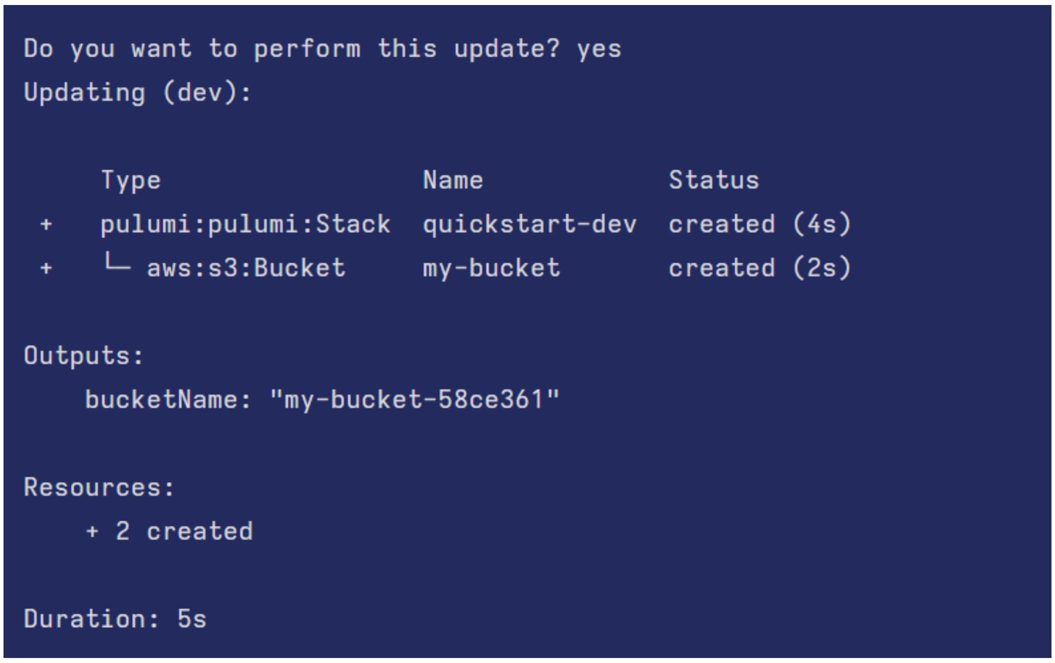
Successively to run the program and performing the deploy **pulumi up** command must be launched and it triggers up the language host component and the operations previously described, including the deployment on the pulumi stack of the project.

This command will result in an interaction with the user because it lists all the resources that has to be deployed (or updated or deleted or replaced) and it asks to the user whether to perform changes or not. To have an automated process **pulumi up --yes** command has to be used. Stacks here play a very relevant role and they will be described afterwards in a dedicated section.



```
$ pulumi up

 Previewing update (dev):

      Type                 Name            Plan
  +   pulumi:pulumi:Stack  quickstart-dev  create
  +   └─ aws:s3:Bucket     my-bucket       create

 Resources:
     + 2 to create

 Do you want to perform this update?
 > yes
   no
   details
```

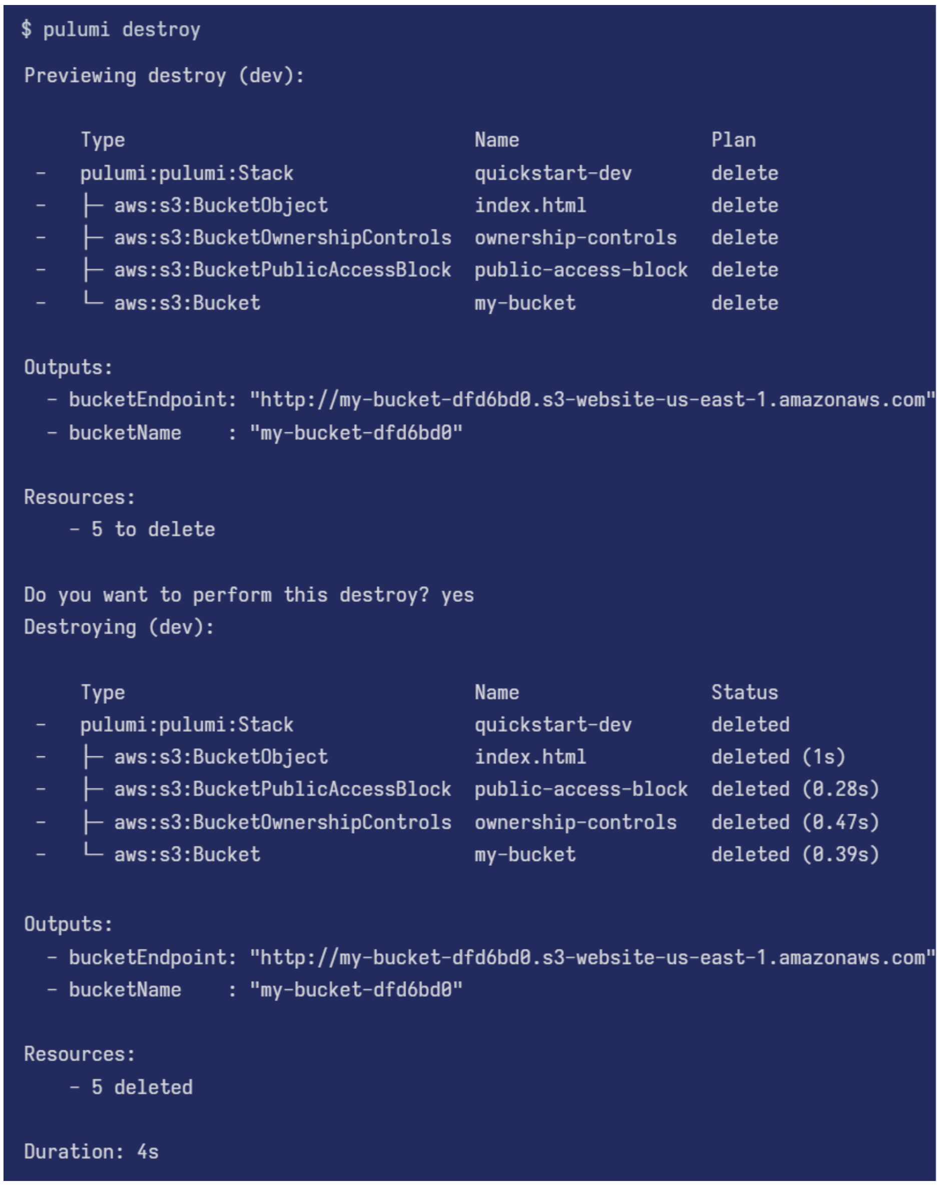
Fig 8: Pulumi up output



```
Do you want to perform this update? yes
Updating (dev):

     Type                 Name            Status
 +   pulumi:pulumi:Stack  quickstart-dev  created (4s)
 +   └─ aws:s3:Bucket     my-bucket       created (2s)

Outputs:
    bucketName: "my-bucket-58ce361"

Resources:
    + 2 created

Duration: 5s
```

Fig 9: Pulumi up output with "yes" answer

To have just a preview of the deployment **pulumi preview** command is enough and to destroy the resources on the cloud platform **pulumi destroy** (with –yes flag if automation is needed) has to be used.



```
$ pulumi destroy
Previewing destroy (dev):

     Type                             Name                 Plan
 -   pulumi:pulumi:Stack              quickstart-dev       delete
 -   ├─ aws:s3:BucketObject           index.html           delete
 -   ├─ aws:s3:BucketOwnershipControls ownership-controls  delete
 -   ├─ aws:s3:BucketPublicAccessBlock public-access-block delete
 -   └─ aws:s3:Bucket                 my-bucket            delete

Outputs:
  - bucketEndpoint: "http://my-bucket-dfd6bd0.s3-website-us-east-1.amazonaws.com"
  - bucketName    : "my-bucket-dfd6bd0"

Resources:
    - 5 to delete

Do you want to perform this destroy? yes
Destroying (dev):

     Type                             Name                 Status
 -   pulumi:pulumi:Stack              quickstart-dev       deleted
 -   ├─ aws:s3:BucketObject           index.html           deleted (1s)
 -   ├─ aws:s3:BucketPublicAccessBlock public-access-block deleted (0.28s)
 -   ├─ aws:s3:BucketOwnershipControls ownership-controls  deleted (0.47s)
 -   └─ aws:s3:Bucket                 my-bucket            deleted (0.39s)

Outputs:
  - bucketEndpoint: "http://my-bucket-dfd6bd0.s3-website-us-east-1.amazonaws.com"
  - bucketName    : "my-bucket-dfd6bd0"

Resources:
    - 5 deleted

Duration: 4s
```

Fig 10: Pulumi destroy output

In these examples a list of aws s3 Bucket objects are created and the process shows up the list of the previewing destroy with all of the resources to be deleted: then it asks the user whether to proceed or not and the "yes" reply triggers the destroying of the whole infrastructure. The number of resources, the execution time and the outputs (additional parameters of the program) are shown in the end of the process.

4.3 Stacks

In Pulumi, a stack is a fundamental concept used to manage and deploy infrastructure as code (IaC) projects. In particular, stacks are a way to organize and manage different configurations and deployments of the infrastructure.

They provide isolation for different environments or configurations. It is likely and very used to have separate stacks for development, staging, and production environments. Each of them can have its own settings, set of configuration parameters and resource definitions. These settings can include things like cloud provider credentials, region, instance sizes, and any other parameters the infrastructure code needs to deploy resources. By using different configurations for each stack, it is easy to switch between environments without modifying the code.

Stacks are especially used to deploy the infrastructure code to the chosen cloud provider. It is possible to target a specific stack when deploying which allows to roll out changes to different environments independently. Moreover, it is possible to organize and to manage many different versions of the infrastructure because changes and rollbacks are traceable for each stack independently; this is very helpful for performing auditing and troubleshooting operations.

Pulumi also supports deploying multiple stacks in parallel. This is particularly useful for complex projects with multiple environments or regions, as it can significantly speed up deployment processes.

Stacks therefore are very useful in maintaining order, consistency, and separation of concerns in the infrastructure code; they provide a structured way to manage and deploy resources for different scenarios or environments, all within the same project.

The content of stacks are but a set of resources which belong to the current state of the infrastructure. More precisely each resource is defined and represented by a specific string, the **URN** (Uniform Resource Name). It is a unique identifier for a resource managed by used to uniquely identify resources across different stacks and deployments. URNs play a crucial role in tracking, referencing, and managing resources in a Pulumi project.

They are stack-independent. This means that a resource's URN remains the same across different stacks and deployments, as long as the resource's attributes (e.g., resource name) remain constant. This allows to reference and manage resources consistently regardless of the stack they belong to. Furthermore, they do include a context that specifies the stack they are associated with. This context helps Pulumi distinguish resources across different stacks with the same name. The stack context is embedded in the URN to ensure uniqueness. In the next chapter it is appreciable to observe how URNs are important especially if operations like importing have to be performed.

When resources are deployed and deposited in their related stack, it is possible even to manage manually the current state (of the Pulumi stack): in particular operations like resources deletion, renaming, unprotecting, can be performed. These can be useful to troubleshoot a stack or for performing specific modifications that otherwise would require editing the state file by hand.

In order to distinguish a stack from others it is possible to assign them tags which consist in a name and a value. A set of built-in tags are automatically assigned and updated each time a stack gets updated. The command to assign tags is the following: **pulumi stack tag set <name> <value>**. This turns out to be useful when for instance development, staging, testing, production stacks have to be conceptually separated in terms of semantic and environments.

A best practice establishes not to give any stack name among *pulumi:*, *gitHub:*, *vcs:*, in order to avoid conflicting with built-in tags that are assigned and updated each a stack gets updated.

The command to create a stack is **pulumi stack init <name>**. This will create a new stack with the passed name and if there are other stacks present in the project it is possible to check all of them by typing **pulumi stack ls**, visualizing also the number of resources per each stack. To select any it is necessary **pulumi stack select <name>**.

A relevant point to underline is the generated file after the stacks creation. As soon as a new project gets created, Pulumi provides by default the creation of a stack because each resource belonging to the code, to the project and so to the infrastructure must reside in a Pulumi stack in order to be managed and deployed. Therefore as mentioned previously when pulumi new <cloud-platform>-<language> command is attempted, the process spawns a stack and the users give it a name. From here onwards the user is free to spawn up as many stack as needed, by certainly assigning it a proper corresponding name, but per each stack created Pulumi takes care of creating a YAML file with the related configuration. The file name of the default stack is **Pulumi.yaml** while the other stacks file names are structured as **Pulumi.<stackName>.yaml**

Pulumi stacks can even export values as stack output which are shown during an update and not only they can be easily retrieved by leveraging the Pulumi CLI but also they are displayed in the Pulumi Cloud dashboard online. These outputs can be used for many purposes such as DNS names, resource IDs, IP addresses. For each language there is a way to implement the exporting of a value, for instance in typescript it can be achievable with **export let variable = resource.id**

The CLI provides the **pulumi stack outputs** command to get the value and to incorporate it into other scripts or tools if needed.

Outputs can be of many types: a regular value, an Output, a Promise. The actual values are resolved after the pulumi up process gets completed, and exports are effectively JSON serialized.

Stack outputs take care of secret annotations and are properly encrypted. If an output contains a secret value it is not shown up by default, in terms of plaintext but they are displayed as secret objects. It is necessary to append the –**show secret** flag when using **pulumi stack output** if plaintext needs to be displayed as well.

Moving on another concept related to this topic, it is appropriated to cite the importance of **references**. In particular stack references allow to access the outputs of a stack from another one, so a stack can reference the outputs of another one. To reproduce this, from a code point of view, creating an instance of StackReference (object of the pulumi library) type is needed, using the fully qualified name of the stack as an input, and then reading the exported stack outputs by their name.

Fully qualified name means including the organization, project, stack name in the order **<organization>/<project>/<stackName>**.

Pulumi programs exchange and communicate information for external consumption by using stack exports.

Concluding this chapter, the last topic is dedicated to stack deletion: more precisely when a **pulumi destroy** is performed all of the resources available in a pulumi stack gets deleted but still the stack itself remains active and empty. In order to remove the stack **pulumi stack rm <stackName>** command is necessary: this will ask for stack deletion to the user so human interaction is considered here otherwise it is sufficient to add –yes flag to perfor it automatically. Very important to take into account is the fact that this operation takes care also to delete the stack configuration saved in the file created by Pulumi with name *Pulumi.<stackName>.yaml* (previously cited).

4.4 Backend and State

Metadata, known as **state**, which manages the infrastructure cloud resources is stored by Pulumi in a backend. Each stack has its own state and Pulumi, thanks to it, can recognize how and when to do the CRUD operations upon cloud resources.

The state is stored in transactional snapshot known as **checkpoint** which are recorded by Pulumi in order to operate securely and reliably such as database transactions work. The state functions permits to recover from failures, to accurately destroy resources.

Regarding backend, it is an API storage endpoint to coordinate updates by the CLI which means writing and reading the stack's state each time there is an incoming update as well.

There are two backend options: Pulumi Cloud, a secure and reliable hosted application, and simple storage objects such as AWS S3, Microsoft Azure Blob Storage, or a local filsesystem.

Usually the Pulumi Cloud hosted application is used because it provides accurate details about the state and backend even in a graphic way. In particular it records each checkpoint so that it result easy to recover and performing steps undo from unusual failure scenarios. Instead, whenever a cloud storage or a local filesystem is used as backend there is certainly more control over where the state is located but this point implies to supervise manually security, state management and other related concerns that Pulumi Cloud application instead would handle on its own.

It is possible to distinguish two classes of state backends for storing the infrastructure state:

- **Service**

  A managed experience provided by the online Pulumi Cloud hosted application. By default Pulumi itself uses this option, hosted at https://app.pulumi.com/ as it provides usability, safety, and security combined with a robust state management with transactional checkpointing for disaster recovery and fault tolerance, a concurrent state locking to prevent infrastructure corruptions, an encrypted state, a managed encryption for secrets (including key management), a secure access to cloud resources metadata, and finally a way of defining policies through Policy as Code and RBAC (Role Based Access Control).

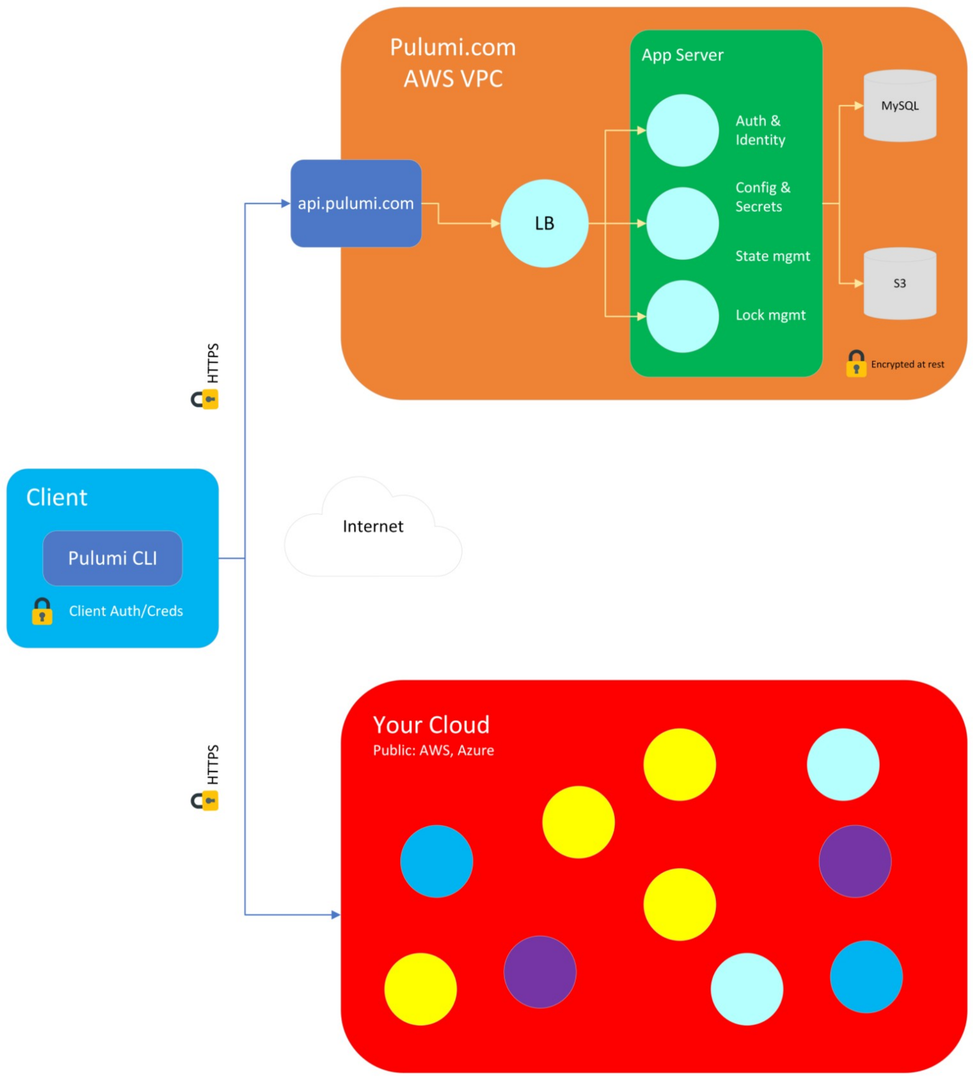


Fig 11: Pulumi Cloud online architecture

In the figure 19 above, the Pulumi Cloud does not acquire any cloud credential and there is not a direct communication with the related cloud provider. CLI instead interacts with either Pulumi Cloud API and cloud provider's API in order to ensure key management has not the necessity to change while using the tool. Since server (violet shape in the fig 19) does not have direct access to the cloud credentials, so thanks to the client/server separation of concerns, Pulumi Cloud has been used in the organizations even with advanced compliance needs.

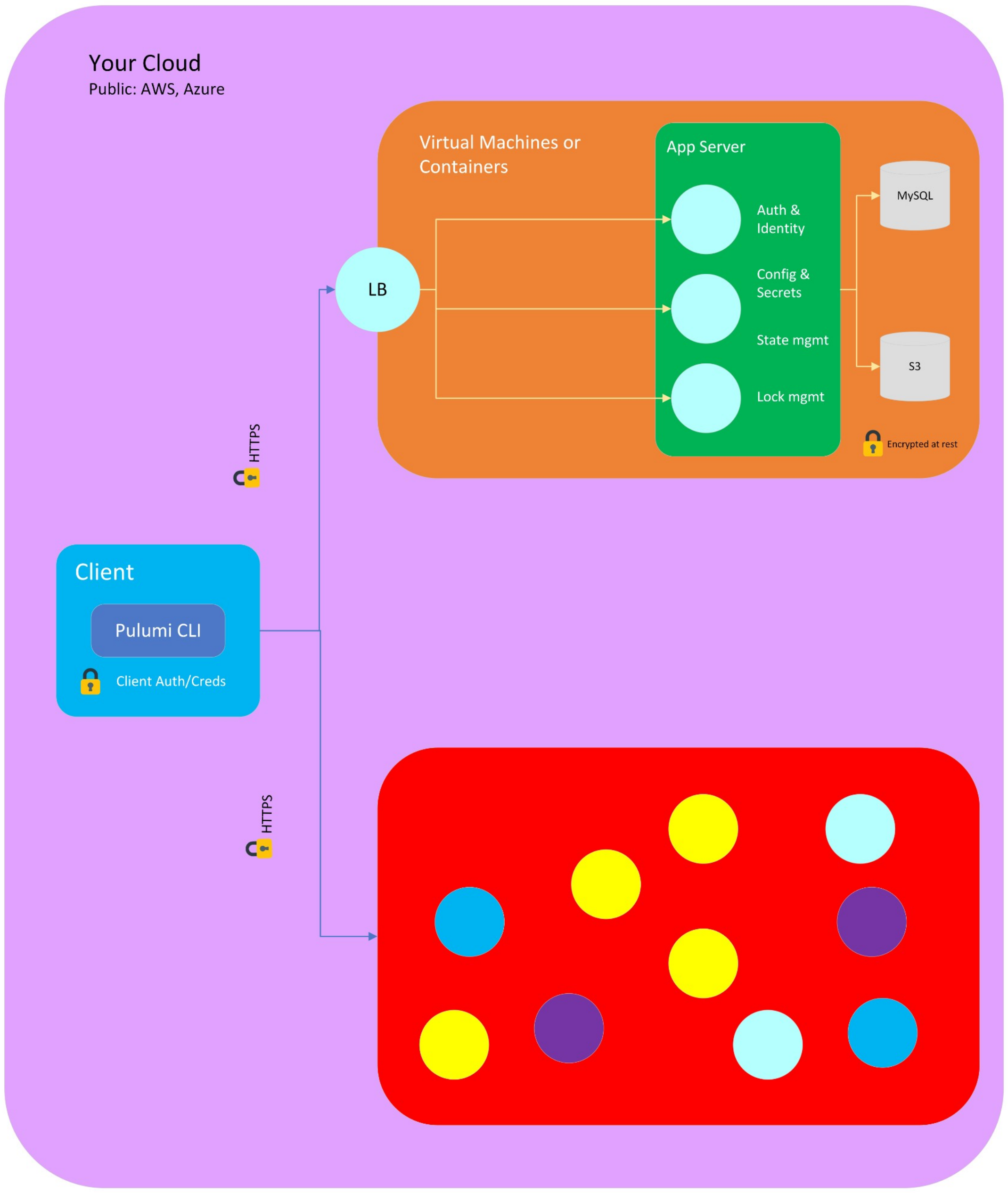


Fig 12: Pulumi Cloud architecture on cloud platform

In the other hand whenever there is the necessity to have the Pulumi Cloud architecture represented in the previous figure on a private cloud environment it is achievable thanks to the versions Pulumi itself offers which can run natively on AWS, Azure, Kubernetes, Google Cloud or even in hybrid cloud environments. The architecture above hence (fig 20) is very similar to the previous one because the only aspect changed is the context underneath: it does not depend on public access on the internet.

- **Self-Managed**

  A manually managed cloud storage, such as AWS S3, Google Cloud Storage, Microsoft Azure Blob Storage, or a local filesystem.

  Pulumi's SDK works perfectly with both the backend options, but in self-managed the state is stored as a simple JSON file into one of the cloud storage objects mentioned just above. Using this backend option means exchanging some amount of security and reliability with more control over the position of the metadata stored. In this case encryption, key management for secrets, have to be manually configured as well as the concurrency and the recovery capabilities (fault tolerance, disaster recovery).

As mentioned previously the **Service** backend is the one used by default by Pulumi; this does not mean it is strictly necessary to use it during the utilization of the tool. In fact it its possible to migrate from a backend to another: this is a common behavior when the project has been started with a self-managed backend and later it comes the decision of using the Pulumi Cloud for a easier management of the team which leads the project itself.

The stack state includes information about its backend as well as its encryption provider and other unique information. So moving a stack between backend is not that easy like barely copying its state file. However Pulumi supports this migration and it supplies **pulumi stack export** and **pulumi stack import** commands which are able to understand the way to perform the translations needed.

Nevertheless Pulumi has been designed to abstract the state management from the human interaction in order to operate just in terms of declarative infrastructure as code.

4.5 IaC tools comparison: Pulumi vs Terraform

As mentioned in the chapters 3.3 Iac tools and 3.5 Outer Iac Tools, many are the tools that can be suitable for making provisioning, infrastructure designing, creation, orchestration. All of them can overlap the capabilities of each other and many of them might be used to cooperate together as complementary

As regards Pulumi and Terraform, they are present similarities and differences. From here onwards let's show and discuss about their peculiarities. In essence, Pulumi is a tool which permits to handle a multi-cloud environment by using the most popular programming and markup languages: Typescript, Go, .NET, Python, Java, YAML, CUE.

Terraform, is a IaC tool which manages multi-cloud contexts too and enables users to define and provision cloud infrastructure and other resources using declarative configuration files. As a matter of fact, both the tools include the possibility of creating, deploying and managing infrastructure as code and cloud architecture models on many cloud providers as well as they work on a desired state model so the concept about the comparison, made by the deployment engine, between the current and the desired state in order to realign them with the proper operations whenever they unmatch, is the same. For instance either Pulumi and Terraform supports more or less even the same cloud providers such as AWS, Azure, Google Cloud, but clearly they present also differences.

In fact Terraform is not open source with the Business Source License, requires a domain-specific language while Pulumi instead is open source with Apache 2.0 License, and allows general-purpose programming languages.

By accurately analyzing the supported languages it surely figures out that Terraform is based on HashiCorp Configuration Language, better known with its acronym HCL. In the other hand, having the possibility of using a wide range of languages gives benefits in terms of using blocks and constructs such as functions, classes, loops, conditions for cutting logic complexity, getting a high level complex infrastructure, and doing IaC as it was an usual general-purpose programming language project.

HCL, by adopting just a declarative approach, allows to copy and paste different pieces of code among projects; Pulumi permits it too but relying especially on languages that have been constructed over decades offers more security and stability in terms of code and complexity reduction with the aim at operating at global scale. The best approach in Pulumi is to leverage the existing package created either by Pulumi itself or by the Pulumi community.

From an IDEs point of view, the ones available for Terraform are rather limited while for Pulumi it is quite conversely since general-purpose programming languages are supported by the majority of IDEs, which provide clearly automatic code completion, strong typing, detailed resources documentation.

From a state management point of view, instead, the engine of Terraform is responsible of performing resources provisioning, updating, and Pulumi engine gives the possibility to update the infrastructure similarly, but the very key difference is the handing of the state. In fact with Terraform concurrency and state has to be managed manually by means of the state files while Pulumi (as explained in 4.4 Backend and state chapter) relies on Pulumi Cloud service to have an automated management, and this clearly increments the Pulumi score. However the other face of the Pulumi backend, self-management, looks more similar to what Terraform uses, so manually configuration of the state.

Moreover, talking about cloud providers (as its official website highlights), Pulumi is even capable to adapt any Terraform provider allowing so to handle, with Pulumi programs, any cloud infrastructure supported by Terraform providers. This leads to have the possibility of converting Terraform HCL to Pulumi, by using Pulumi CLI via **pulumi convert --from terraform** command. Another concept related to the conversion is consuming local or remote Terraform state from programs of Pulumi: it sparks interest because it lets to manage the infrastructure with Terrafrom while at the same time it allows to incrementally move towards Pulumi.

Facing dynamic resource providers topic, Pulumi provides the possibility of creating custom resources through their opportunity of directly coding the **C**reation **R**ead **U**pdate **D**elete methods which results very useful when complex logics and operations has to be implemented such as virtual machines configuration management, database migration, etc. In contrast to Pulumi, Terraform does not present an equivalent, neither a similar, peculiarity and this turns in having more complex and proprietary modules, in term of code, needed for reproducing the CRUD operations.

Infrastructure as code tools, generally, provides several aspects which are common among each other: one of those is certainly the modularity and the reuse of the infrastructure. In this instance both the tools present significant differences if it is considered that Pulumi, in contrast to Terraform, promotes the creation of reusable and modular components. This is a crucial key to explain what is next, that is the thesis project and goals, because standard and well designed architectures can be templatized in order to be reused and fitted on many use cases. Let’s just think about the Pulumi packages: they enable to create components in whatever language and making them accessible in all the other available Pulumi languages differently from Terraform which limits, with HCL, programmers to author proprietary modules and Go-based providers to exploit infrastructure reuse. More precisely Pulumi registry is a large packages collection where packages themselves can be directly installed from.

As regards testing and validation, Pulumi offers advantages in terms of native testings frameworks and performed automated tests of the infrastructure. There are several automated testing including end-to-end testing of the complete application, unit testing, integration testing of system components. Instead of Terraform which supports just integration testing, Pulumi do provides unit tests, property tests and also integration tests. More accurately unit tests perform an evaluation of the code behavior in isolation and are very useful for fast feedbacks during the development, such as in Test-Driven Development, since they run in memory without being affected by any out of process call.

Property tests, instead, are policy as code based; in particular each policy is an invariant evaluated and asserted by the property test itself. This kind of tests run inside the Pulumi CLI either previously and successively to the infrastructure provisioning process. Those policies have access to all of the input and outputs of the resources in the Pulumi stack. Property tests are cloud independent and differ from unit tests in evaluating real returned from the cloud provider values instead of the mocked ones, since unit tests takes replace those inputs/outputs with mocks.

Integration tests acts differently from unit tests because they deploy the resources and then they validate their actual behaviour. More specifically resources are deployed into an ephemeral environment: the test gets infrastructure endpoints from the stack outputs, typically an IP address, for verifying infrastructure acts as expected. The real strength which comes out from here is testing the actual cloud infrastructure and its real values but it takes more time than unit tests.

Another characteristic which distinguish a tool from another is what it is known as **automation API**. It consists in the possibility of embedding the Pulumi program into the code of an application (the one related to the service or to what a company is developing), such as a web server, through a programmatic interface which allows to use the programs safely without interacting forcefully with the CLI. For instance Terraform does not have something similar. In this way it result easy to create experiences on top of the tool personalized to the use case.

Finally the comparison between the two tools might lead to perceive Pulumi as the best one but in essence both present advantages and weakness so there is not a choice which considers a tool better than the other but, in spite of the large use of Terraform in enterprise environments, it has to be highlighted the fact whereby Pulumi present many interesting characteristics that Terraform does not have and this brings Pulumi itself to be a more interesting tool to be studied and to be used also by organizations. Moreover by leveraging the potentiality of Pulumi and its peculiarity to be open source, my personal opinion, as a computer engineering student, is that this tool may overlap almost completely Terraform due to the benefits it brings, previously explained.

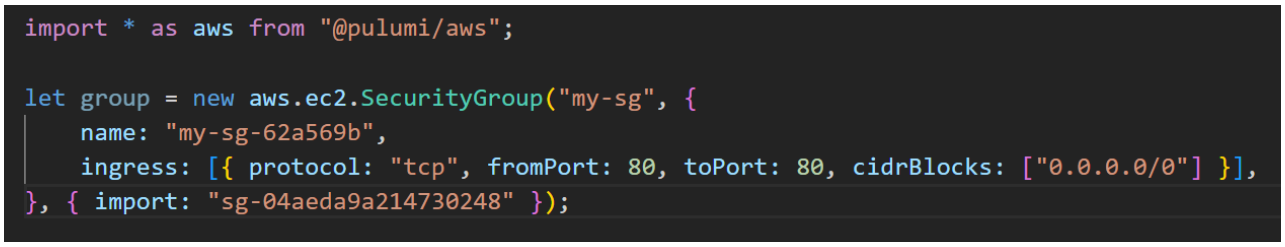
4.6 Import

Infrastructure provisioning, either performed by infrastructure as code or by cloud CLI/dashboard, leads to have many scenarios of the architecture which can be handled in many ways. By focusing on the IaC approach it is possible to integrate the code of the program containing the cloud resources with other resources which are the existing one on the cloud provider. For instance many projects require to build on top of existing resources or to manage existing resources for a more resilient solution.

Pulumi offers two different ways for importing cloud resources, and so the entire cloud infrastructure:

- The first one refers to use a special command which allows to import either a single resource or a batch of resources described in a JSON file: this is the pulumi import [type] [name] [id] command, where type stands for the kind of resource (storage, network security group, virtual machine), name is the name of the resource and id for its identification. This command checks out the resource on the cloud platform in which it resides, with the indicated id and the Pulumi engine import (so generates the code of) the resource into the program including all its properties.
- The second option available is to declare by code a specific key used for importing a selected resource. In this case it is deducible that the whole infrastructure cannot be imported into the own program. Infact the usage of this option, in the code, forces the user to call the resource object API and to specify inside of it the property **import** by assigning it just the id of the resource already present in the cloud platform so that the

declared one in the code could take the properties and characteristics of the imported one. The following picture shows an example of how this kind of import can be done.



```
import * as aws from "@pulumi/aws";

let group = new aws.ec2.SecurityGroup("my-sg", {
    name: "my-sg-62a569b",
    ingress: [{ protocol: "tcp", fromPort: 80, toPort: 80, cidrBlocks: ["0.0.0.0/0"] }],
}, { import: "sg-04aeda9a214730248" });
```

Fig 13: Example of Pulumi import by code

As it can be noticed, it this tiny piece of code a security group object of AWS cloud provider is going to be created and Pulumi is going to adopt the existing resource by querying AWS for that specific resource with that specific id, sg-04aeda9a214730248, so it will not perform the creation of a new instance of it.

Taking into account the first importing option, instead, it has to be created a JSON file with a list of all of the resources needed to be imported. In particular each of them must contain the type of the resource, a name which describes the resource itself, and its id. Obviously this file can be created by hand or generated dynamically as the resources gets deployed into the cloud provider. For the project of this thesis it has been implemented a solution of resources importing in order to face a sensitive topic such as the disaster recovery and fault tolerance, and the approached used has been the automatic and dynamic generation of a JSON file containing resources description.

4.7 Templates

Pulumi offers the possibility of authoring templates which represent a powerful approach to infrastructure as code allowing so developers and operators to define and manage cloud resources and infrastructure in a programmatic and declarative manner. These templates are essentially code-based blueprints that describe the desired state of the cloud infrastructure to create and they enable to create, configure, and manage cloud resources across various cloud providers.

First of all, they are written in familiar programming languages Pulumi has available. This language flexibility allows to leverage user’s programming skills and tools to define infrastructure, making it more accessible and adaptable to the working team's preferences and expertise.

Furthermore templates provide a high level of abstraction over cloud resources; in this expressing the infrastructure needs in a concise and human-readable manner results easier. Not only the infrastructure components like virtual machines, databases, and networks can be defined but also the relationships and dependencies between them. This declarative approach therefore ensures a consistent infrastructure reducing the risk of configuration drift or inconsistencies. These templates also support dynamic values and logic, allowing so to incorporate conditions, loops, and data transformations within the infrastructure definitions. This flexibility comes from the high level programming languages offered and so the infrastructure can be tailored to meet specific requirements, such as scaling resources based on load, handling regional failovers, or integrating with external services.

A very relevant trait to consider is the convenience of using version control systems like Git to track changes, collaborate with team members, and apply best practices for software development, such as code reviews, testing, and continuous integration/continuous deployment (CI/CD) processes. This point is what make the difference in a flexible context where the projects reuse, the cooperation become useful as well as comfortable.

Moreover, the multi-cloud support results to be very convenient to target multiple cloud providers within the same template, enabling hybrid and multi-cloud deployments.

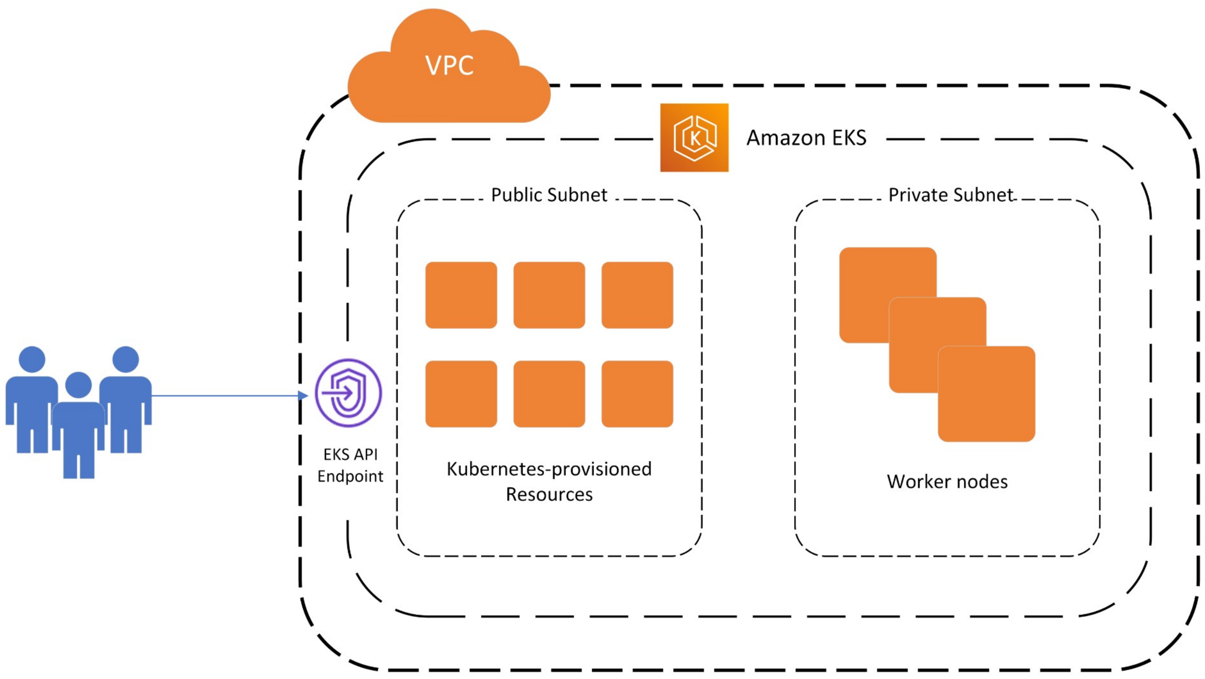


Fig 14: Example of a Pulumi AWS EKS template

This versatility is particularly valuable when the need is to distribute workloads across different cloud platforms or migrate between providers without rewriting your infrastructure code.

In addition, the tool also provides a rich ecosystem of libraries and plugins, which extend its capabilities and simplify the management of complex cloud resources. These libraries offer built-in abstractions and templates for common cloud patterns and services, accelerating the development and deployment of your infrastructure. The following pictures, taken from the Pulumi official website, give a deeper comprehension of the characteristics just mentioned. Libraries and APIs offered by Pulumi are the key concepts to introduce the next chapter which represent the main point of the study: the project realization.

5. Project: design and implementation of a resilient cloud infrastructure model in a multi-cloud context

The purpose, the thesis project aims to, is to build a consistent cloud infrastructure for suiting several kind of use cases in which organizations, as well as private teams, could deploy their applications and workloads by guaranteeing the implementation of the best practices needed to provide **high availability, scalability, reliability, robustness, resiliency, redundancy.**

In essence, organizations are increasingly adopting solutions whereby services are deployed in a public cloud, private cloud, or hybrid cloud in order to benefit from advantages like automation and orchestration, so the goal is to provision a flexible, reusable infrastructure for hosting applications securely, efficiently, robustly and redundantly. In addition it is crucial and decisive to take care of aspects such as data protection and controlled accesses as well as disaster recovery and fault tolerance.

According to Liquid Reply internal tutors, the idea was to realize a very common enterprise use case which allow to take into account all of the best practices needed to project the architecture: this refers to a generic company's web application. It is composed by a lot of services, so a microservices architecture has involved, and a service such as the frontend one is exposed publicly for sending the requests to the backend private services; hence the presence of a load balancer, an autoscaling group and other cloud objects that guarantee the high availability, redundacy, resiliency, scalability, have been included and they are faced in the next chapters.

As regards the best practices for building an optimal cloud infrastructure, instead, the **AWS Well-Architected Framework** Amazon Web Services paper affirms, "the AWS Well-Architected Framework helps you understand the pros and cons of decisions you make while building systems on AWS. By using the Framework you will learn architectural best practices for designing and operating reliable, secure, efficient, and cost-effective systems in the cloud. It provides a way for you to consistently measure your architectures against best practices and identify areas for improvement. The process for reviewing an architecture is a constructive conversation about architectural decisions, and is not an audit mechanism. We believe that having well-architected systems greatly increases the likelihood of business success."

In particular the same paper continues with "creating a software system is a lot like constructing a building. If the foundation is not solid, structural problems can undermine the integrity and function of the building. When architecting technology solutions, if you neglect the five pillars of operational excellence, security, reliability, performance efficiency, and cost optimization, it can become challenging to build a system that delivers on your expectations and requirements. Incorporating these pillars into your architecture will help you produce stable and efficient systems. This will allow you to focus on the other aspects of design, such as functional requirements"

Obviously the earlier citations are valid for each cloud provider used.

According to AWS Well-Architected Framework, defining and designing a good cloud architecture (basing on the characteristics described at the beginning of this chapter) requires to identify a set of principles, known as the **five pillars,** which facilitate the tasks. For each pillar the related best practices will be highlighted. In particular they consist in:

- **Operational excellence**

  It refers to support the development, to run workloads efficiently, by gaining insights and continuing to improve the supporting process. This practically means to define the infrastructure by using code in order to automate the execution and to limit human error, but also designing workloads to be updated regularly, as well as evolving the workloads effectiveness procedures and performing what the AWS Well-Architected Framework defines as *pre-mortem* exercise to identify potential failures to remove or mitigate in advance.

  For this pillar some **best practices** have been identified; expressly they result in **determining the priorities of building a certain cloud infrastructure, understanding responsibility and who can make decisions, supporting who is in charge of the designing and implementing processes in order to support the outcomes, designing workloads including all of the info useful to learn more about their state (e.g. metrics, logs, traces), reducing deployment issues with reactive approaches which improve the flow of changes so enabling as fast as possible recovery from unexpected changes/failures, understanding the health of workloads by their metrics.**

- **Security**

  Guaranteeing security means a set of tasks such as data and systems protection, access control, centralizing identity management, deleting dependency to long-term static credentials as well as considering the insights of the environment to monitor changes in real time. Moreover it is necessary to adopt what is called **defense in depth** approach which consists in applying several security and automation controls in each cloud resource of the infrastructure (e.g. virtual network, load balancer, virtual machines) in order to prevent from human errors risks when managing sensitive data; this solution includes more rapid and cost-effective scale mechanisms which can be implemented with the security as code paradigm.

  The best practices for this pillar include **automating security processes, testing and validation with the aim of scaling security operations, managing human identities (related to administrators, developers, operators) and machine identities (related to cloud resources, application, workloads) under permissions based on conditions, analyzing events from logs and metrics, protecting public and private networks as well as servers, databases, functions and all of the resources with multiple layers of defense against threats, classifying data basing on their vulnerabilities, managing the authorized accesses, planning a recovery solution from security attacks for minimizing infrastructure (and company) disruption.**

- **Reliability**

  Reliability embraces the ability of controlling, testing, performing a workload functions and verifying its correctness as well as consistence. The behavior of a workload is observed in order to check whether it performs as expected.

  Several points are relevant for this phase: in particular it is impossible not to underline the importance of monitoring workloads and planning an automatical recover from possible failures. This means also to include testing operations which usually are not used to validate recovery strategies in an on-premises environment while in the cloud it becomes a regular procedure so designing recovery solutions results more facilitated. Furthermore in addition to testing and recovers, it is smart to split workloads in smaller pieces to reduce the impact of a single failure on the overall workload, as cited in the considered paper, but also monitoring demands and workloads utilization is proper for not exceeding their capacity. For this reason automation could be useful to automate the addition or removal of cloud resources to satisfy demand without under or overprovisioning.

Reliability pillar makes available the following **best practices** which must be considered: **take advantage of service quotas to prevent over-provisioning or under-provisioning, planning the network topology by considering aspects such as internal and external connectivity as well as security rules and IP addresses management, building a highly available, scalable and reliable workloads using a microservice architecture for simpler and smaller components, involving a stable network communication among microservices and operating reliably despite network latency or data loss, monitoring workloads by means of their metrics, designing scalable resources for gaining more elasticity and an easier autoscaling, including high availability and MTTR (Mean Time To Recovery) for resilency, performing backups and projecting redundant workloads with a focus on a disaster recovery plan.**

- **Performance efficiency**

Computer resources are involved either to accomplish infrastructure requirements and to guarantee efficiency as demand increases. Cloud vendors offers technologies as services so there is no need of special expertise of some techonologies. Moreover to get better performances it is proper to deploy the workloads in multiple regions so that latency can be lowed and a better cloud experience can be provided to the customer, with minimum costs. Performance efficiency includes also to involve serverless architecture to replace physical servers such as a web server which hosts a website that can be replaced with a serverless storage. This strategy can benefit especially in an easier and better scaling because it is a cloud resource, so very different from managing physical machines, as well as faster and optimized configurations.

For this pillar several **best practices** have been considered, in particular **selecting the well-architected system by using multiple architectural solutions, selecting optimal computer solution (resources hardware profile) for improving workloads performance efficiency, using multiple storage solutions, databases for an efficient use of resources, configuring network profile basing on the workloads constraints and requirements, evolving workloads to take advantages of new releases, monitor system to identify degradation.**

- **Cost optimization**

  Designing a well-architected infrastructure means also taking care of a very important point such as the cloud financial management. In order to build and deliver an efficient product it is necessary to provide a cost-effective solution so a good consumption model should be adopted like the pay-per-use one. This model provides for paying just the required computing resources and increasing or decreasing the usage depending on the business requirements. So the perspective is not about forecasting the amount of usage in terms of time for resources but is about paying for how long resources are used. The practical example could be having virtual machines in development and test environments powered on just from 9 A.M. to 5 P.M. and powered off for the remaining time; in this case the cost saving gets around 75%. The cloud hence facilitates identifying the system cost and usage and it helps so to measure what it is defined as **return of investment (ROI)** which contributes to optimize resources and to reduce their costs.

  Cost optimization pillar identifies a set of **best practices** to take into account including **<u>implementing cloud financial management for optimizing costs and usages, avoiding overspending by establishing policies and mechanisms that allow to monitor costs and to improve the cost-efficiency of the workloads (checks-and-balance approach), implementing change control and resource management for ensuring the termination of unused resources, relying on managed services instead of custom services for cost optimization, minimizing the waste by selecting the most cost effective type, size, and number of resources for the task, using the most appropriate pricing model to minimize expenses, ensuring that paid resources are used and avoiding significantly underutilized instances, comparing the current architecture with new services in order to certify the current one still is the most cost-effective.</u>**

5.1 Architecture Design

For the project, according to **Liquid Reply** internal decision, **AWS** and **Microsoft Azure** have been chosen as the cloud providers for which performing the infrastructure as code provisioning so two different ecosystems and their differences have been studied.

In the light of the earlier considerations regarding the five pillars, designing a cloud infrastructure have to be done by following the indicated best practices in terms of architecture design and implementation.

The conducted studies therefore led to follow a step-by-step approach starting off to consider the most relevant guide lines previously described; in particular it has been necessary to identify the key elements to design conceptually the infrastructure that has been implemented secondarily by means of Pulumi. In principles those are referred to design resources scalably (so that autoscaling can be easily performed), to take care of high availability for granting resiliency and redundancy, to plan and configure a network topology which considers control access, reliability, protection from threats and workloads isolation, to involve a microservices architecture for an effective workloads split, to minimize the costs by managing the amount and the type of resources basing on the demands and scaling up or down consequently if needed, to plan a possible strategy for disaster recovery and fault tolerance.

After this identification, the next task has been to design the architecture including all of the previous factors so the following illustration is related to AWS.

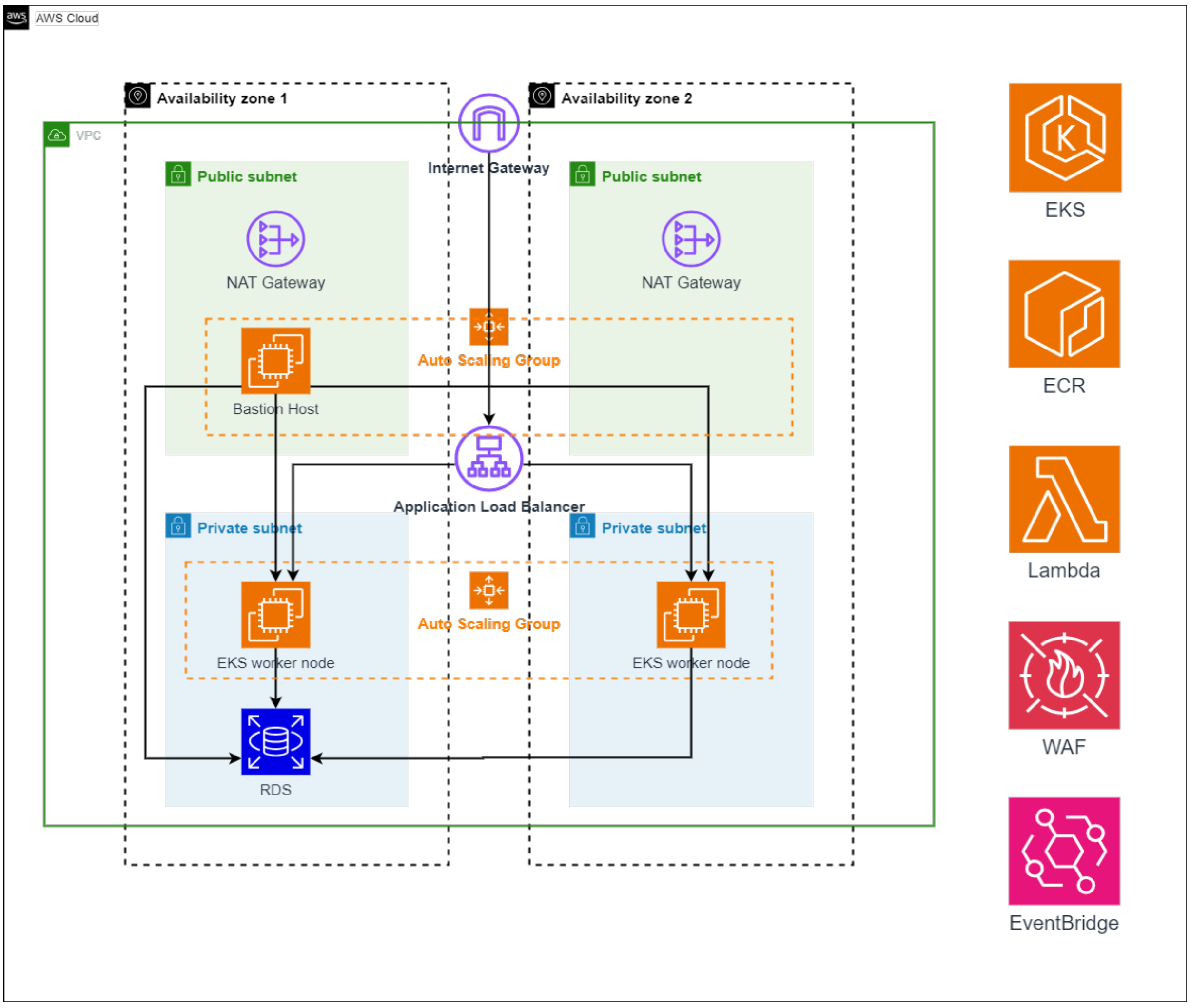


Fig 15: AWS cloud infrastructure project design

For the AWS architecture the followings resources have been involved, in terms what they are generically used for, according to the **AWS official website documentation.**

5.1.1 VPC (Virtual Private Cloud)

"*A virtual private cloud (VPC) is a virtual network dedicated to AWS account. It is logically isolated from other virtual networks in the AWS Cloud. You can specify an IP address range for the VPC as well as to add subnets, to add gateways, and to associate security groups.*" AWS resources such as Amazon EC2 instances, RDS databases, and more, can be launched in a highly customizable and secure network environment. It enables to define a custom network topology, to connect to on-premises data centers or to other VPCs using AWS Direct Connect or VPN connections, and to apply security measures like network security Groups (NSGs) and route tables to control traffic flow. This resource plays a fundamental role to building scalable and secure cloud architectures.

5.1.2 AWS Subnet

"*A subnet is a range of IP addresses in the VPC. You launch AWS resources, such as Amazon EC2 instances, into your subnets. You can connect a subnet to the internet, other VPCs, and your own data centers, and route traffic to and from your subnets using route tables.*" They are essentially address ranges within the VPC's IP address space and their functionality is to group AWS resources like EC2 instances, RDS databases, and Lambda functions. A subnet has its own route table, which can be used to control the flow of traffic to and from resources within the subnet itself, and plays a crucial role in controlling network traffic and providing both security and availability. For the project two public and two private subnets were required.

- **Public subnet**: "the subnet has a direct route to an internet gateway. Resources in a public subnet can access the public internet."
- **Private subnet**: "the subnet does not have a direct route to an internet gateway. Resources in a private subnet require a NAT device to access the public internet."

5.1.3 NAT Gateway

"*A NAT gateway is a Network Address Translation (NAT) service. You can use a NAT gateway so that instances in a private subnet can connect to services outside your VPC but external services cannot initiate a connection with those instances. When you create a NAT gateway, you specify one of the following connectivity types:*

- ***Public (Default)**: instances in private subnets can connect to the internet through a public NAT gateway, but cannot receive unsolicited inbound connections from the internet. You create a public NAT gateway in a public subnet and must associate an elastic IP address with the NAT gateway at creation. You route traffic from the NAT gateway to the internet gateway for the VPC. Alternatively, you can use a public NAT gateway to connect to other VPCs or your on-premises network. In this case, you route traffic from the NAT gateway through a transit gateway or a virtual private gateway.*

- ***Private**: instances in private subnets can connect to other VPCs or your on-premises network through a private NAT gateway. You can route traffic from the NAT gateway through a transit gateway or a virtual private gateway. You cannot associate an elastic IP address with a private NAT gateway. You can attach an internet gateway to a VPC with a private NAT gateway, but if you route traffic from the private NAT gateway to the internet gateway, the internet gateway drops the traffic.*"

5.1.4 Internet Gateway

"*An internet gateway is a horizontally scaled, redundant, and highly available VPC component that allows communication between your VPC and internet. It supports IPv4 and IPv6 traffic. It does not cause availability risks or bandwidth constraints on your network traffic. An internet gateway enables resources in your public subnets (such as EC2 instances) to connect to the internet if the resource has a public IPv4 address or an IPv6 address. Similarly, resources on the internet can initiate a connection to resources in your subnet using the public IPv4 address or IPv6 address. For instance, an internet gateway enables you to connect to an EC2 instance in AWS* *using your local computer. An internet gateway provides a target in your VPC route tables for internet-routable traffic. For communication using IPv4, the internet gateway also performs network address translation (NAT). For communication using IPv6, NAT is not needed because IPv6 addresses are public.*"

5.1.5 Bastion Host

"A bastion host is a server whose purpose is to provide access to a private network from an external network, such as the Internet. Because of its exposure to potential attack, a bastion host must minimize the chances of penetration. For example, you can use a bastion host to mitigate the risk of allowing SSH connections from an external network to the Linux instances launched in a private subnet of your Amazon Virtual Private Cloud (VPC)." For this project it has been used as single entry point to the application from the outside of the infrastructure, by connecting to the service the application load balancer exposes.

5.1.6 Elastic Kubernetes Service EKS

"*Amazon Elastic Kubernetes Service (Amazon EKS) is a managed Kubernetes service to run Kubernetes in the AWS cloud and on-premises data centers. In the cloud, Amazon EKS automatically manages the availability and scalability of the Kubernetes control plane nodes responsible for scheduling containers, managing application availability, storing cluster data, and other key tasks. With Amazon EKS, you can take advantage of all the performance, scale, reliability, and availability of AWS infrastructure, as well as integrations with AWS networking and security services. On-premises, EKS provides a consistent, fully-supported Kubernetes solution with integrated tooling and simple deployment to AWS Outposts, virtual machines, or bare metal servers.*"

5.1.7 Load Balancer

"*Elastic Load Balancing automatically distributes your incoming traffic across multiple targets, such as EC2 instances, containers, and IP addresses, in one or more Availability Zones. It monitors the health of its registered targets, and routes traffic only to the healthy targets. Elastic Load Balancing scales your load balancer as your incoming traffic changes over time. It can automatically scale to the vast majority of workloads.*

A load balancer serves as the single point of contact for clients. The load balancer distributes incoming application traffic across multiple targets, such as EC2 instances, in multiple Availability Zones. This increases the availability of your application. You add one or more listeners to your load balancer.

A listener checks for connection requests from clients, using the protocol and port that you configure. The rules that you define for a listener determine how the load balancer routes requests to its registered targets. Each rule consists of a priority, one or more actions, and one or more conditions. When the conditions for a rule are met, then its actions are performed. You must define a default rule for each listener, and you can optionally define additional rules.

Each target group routes requests to one or more registered targets, such as EC2 instances, using the protocol and port number that you specify. You can register a target with multiple target groups. You can configure health checks on a per target group basis. Health checks are performed on all targets registered to a target group that is specified in a listener rule for your load balancer."

5.1.8 Autoscaling Group

"*An Auto Scaling group contains a collection of EC2 instances that are treated as a logical grouping for the purposes of automatic scaling and management. An Auto Scaling group also lets you use Amazon EC2 Auto Scaling features such as health check replacements and scaling policies. Both maintaining the number of instances in an Auto Scaling group and automatic scaling are the core functionality of the Amazon EC2 Auto Scaling service.*

The size of an Auto Scaling group depends on the number of instances that you set as the desired capacity. You can adjust its size to meet demand, either manually or by using automatic scaling."

5.1.9 Relational Database Service (RDS)

"*Amazon Relational Database Service (Amazon RDS) is a collection of managed services that makes it simple to set up, operate, and scale databases in the cloud.*". Not only it allows to "*support growing apps with high availability, throughput, and storage scalability*" but also to "*take advantage of flexible pay-per-use pricing to suit various application usage patterns.*"

5.1.10 Elastic Container Registry

"*Amazon Elastic Container Registry (Amazon ECR) is a fully managed container registry offering high-performance hosting, so you can reliably deploy application images and artifacts anywhere.*"

It allows to "*meet your image compliance security requirements using the tightly integrated Amazon Inspector vulnerability management service to automate vulnerability assessment scanning and remediation ticket routing*"

5.1.11 Web Application Firewall

"*AWS WAF helps you protect against common web exploits and bots that can affect availability, compromise security, or consume excessive resources. With AWS WAF, you can create security rules that control bot traffic and block common attack patterns such as SQL injection or cross-site scripting (XSS).*"

It allows also to "*monitor your application's login page for unauthorized access to user accounts using compromised credentials*" as well as "*Create and maintain rules automatically and incorporate them into the development and design process*"

5.1.12 Lambda

"*AWS Lambda is a compute service that lets you run code without provisioning or managing servers. Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, and logging. With Lambda, all you need to do is supply your code in one of the language runtimes that Lambda supports.*

You organize your code into Lambda functions. The Lambda service runs your function only when needed and scales automatically. You only pay for the compute time that you consume—there is no charge when your code is not running.

Lambda is an ideal compute service for application scenarios that need to scale up rapidly, and scale down to zero when not in demand. For example, you can use Lambda for:

- ***File processing**: Use Amazon Simple Storage Service (Amazon S3) to trigger Lambda data processing in real time after an upload.*

- ***Stream processing**: Use Lambda and Amazon Kinesis to process real-time streaming data for application activity tracking, transaction order processing, clickstream analysis, data cleansing, log filtering, indexing, social media analysis, Internet of Things (IoT) device data telemetry, and metering.*

- ***Web applications**: Combine Lambda with other AWS services to build powerful web applications that automatically scale up and down and run in a highly available configuration across multiple data centers.*

- ***IoT backends**: Build serverless backends using Lambda to handle web, mobile, IoT, and third-party API requests.*

- ***Mobile backends****: Build backends using Lambda and Amazon API Gateway to authenticate and process API requests. Use AWS Amplify to easily integrate with your iOS, Android, Web, and React Native frontends.*"

5.1.13 Eventbridge

"*Amazon EventBridge is a service that provides real-time access to changes in data in AWS services, your own applications, and software as a service (SaaS) applications without writing code*". It allows to "*remove the need to coordinate across service teams with decoupled microservices using AWS, SaaS apps, or your own custom apps*", to "*monitor and audit your AWS environments, and respond to operational changes in your applications in real time to prevent infrastructure vulnerabilities*".

For the project the bastion host has been used as a virtual machine on a public subnet in order to connect to the application (its exposed service by the application load balancer) residing into the kubernetes cluster inside the private subnet. Whenever that VM broke up for failures or downtime an autoscaling group is in charge to spawn a new replica of it in one of the two public subnet, so one of the two zones. The application owns a link to a relational database for storing and managing app data; the cluster is managed redundantly thanks to the presence of another autoscaling group object.

ECR service can be used to store and pull images that could be used by kubernetes services and the database power on/off is handled through a Lambda function. This tasks performs so automation by powering on the RDS on 9 AM and shuting it down at 7 PM each day excluding the weekend. The same argument worth with the bastion host with the difference about the power on/off automation which is managed by the autoscaler itself.

As regards Azure, instead, the design below represents the architecture's shape in the Azure context

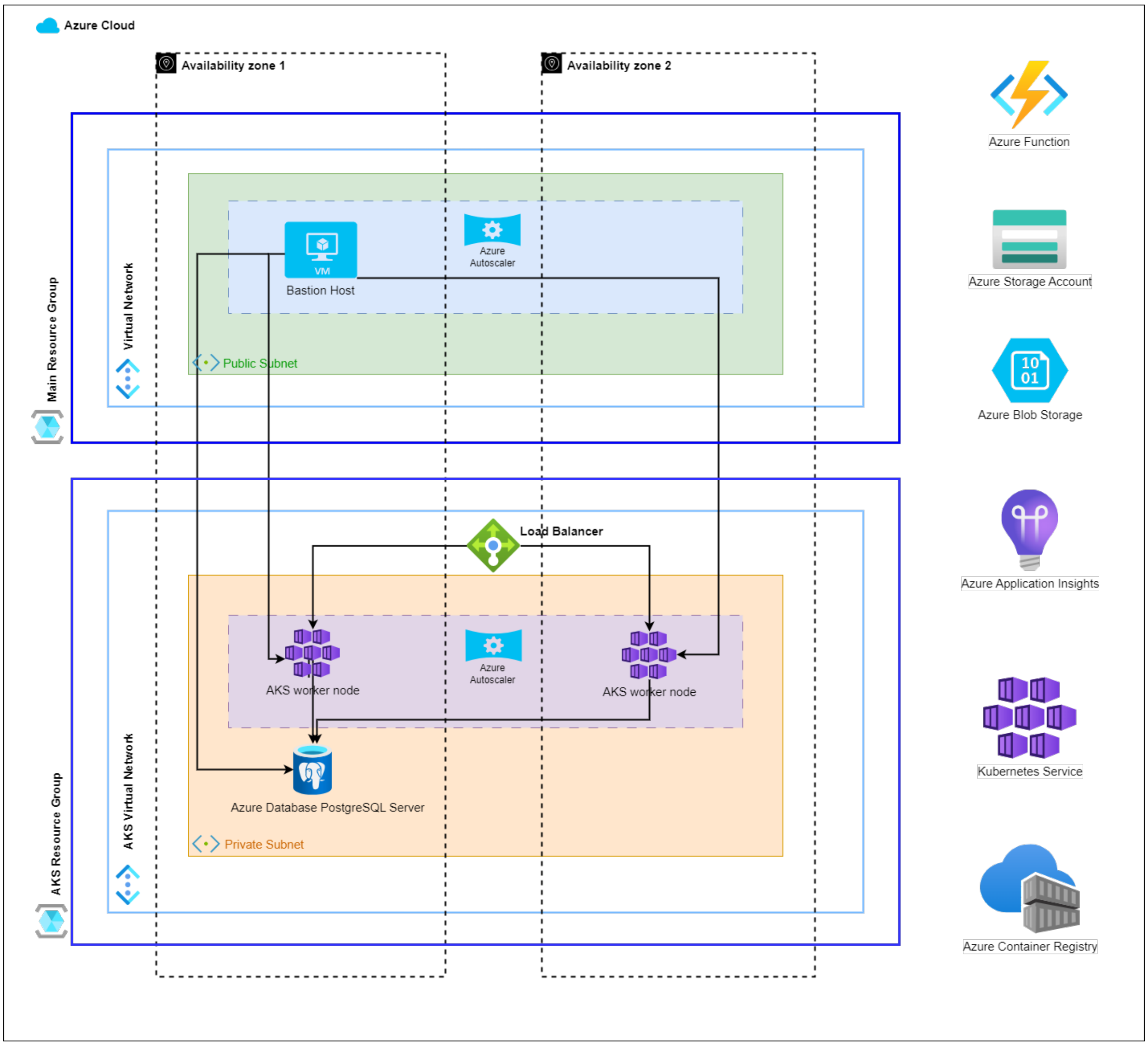


Fig 16: Azure cloud infrastructure project design

In addition, it is noticeable how the architecture and so the infrastructure present some changes with respect to AWS, due to some differences which distinguish both the cloud providers.

But with a step back, let’s have a look on the microsoft azure cloud objects, **referring to the official azure website documentation:**

5.1.14 Resource Group

“A resource group is a container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your

organization. Generally, add resources that share the same lifecycle to the same resource group so you can easily deploy, update, and delete them as a group.

The resource group stores metadata about the resources. Therefore, when you specify a location for the resource group, you are specifying where that metadata is stored. For compliance reasons, you may need to ensure that your data is stored in a particular region.“

5.1.15 Virtual Network

“*Azure Virtual Network is a service that provides the fundamental building block for your private network in Azure. An instance of the service (a virtual network) enables many types of Azure resources to securely communicate with each other, the internet, and on-premises networks. These Azure resources include virtual machines (VMs).*

A virtual network is similar to a traditional network that you'd operate in your own datacenter. But it brings extra benefits of the Azure infrastructure, such as scale, availability, and isolation”.

5.1.16 Azure Subnet

“*A subnet is a range of IP addresses in the virtual network. You can divide a virtual network into multiple subnets for organization and security. Each NIC in a VM is connected to one subnet in one virtual network. NICs connected to subnets (same or different) within a virtual network can communicate with each other without any extra configuration.*“

5.1.17 NAT Gateway

“*Azure NAT Gateway is a fully managed and highly resilient Network Address Translation (NAT) service. You can use Azure NAT Gateway to let all instances in a private subnet connect outbound to the internet while remaining fully private. Unsolicited inbound connections from the internet aren't permitted through a NAT* *gateway. Only packets arriving as response packets to an outbound connection can pass through a NAT gateway.*"

5.1.18 Bastion Host

"*Azure Bastion is a service you deploy that lets you connect to a virtual machine using your browser and the Azure portal, or via the native SSH or RDP client already installed on your local computer. The Azure Bastion service is a fully platform-managed PaaS service that you provision inside your virtual network. It provides secure and seamless RDP/SSH connectivity to your virtual machines directly over TLS from the Azure portal or via native client. When you connect via Azure Bastion, your virtual machines don't need a public IP address, agent, or special client software.*" For the project a standard VM has been used as bastion host.

5.1.19 Azure Kubernetes Service (AKS)

"*Azure Kubernetes Service (AKS) simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance. When you create an AKS cluster, a control plane is automatically created and configured. This control plane is provided at no cost as a managed Azure resource abstracted from the user. You only pay for and manage the nodes attached to the AKS cluster. When you deploy an AKS cluster, you specify the number and size of the nodes, and AKS deploys and configures the Kubernetes control plane and nodes.*"

5.1.20 Load Balancer

"*Load balancing refers to efficiently distributing incoming network traffic across a group of backend servers or resources.*

Azure Load Balancer operates at layer 4 of the Open Systems Interconnection (OSI) model. It's the single point of contact for clients. Load balancer distributes inbound flows that arrive at the load balancer's front end to backend pool instances. These flows are according to configured load-balancing rules and health probes. The *backend pool instances can be Azure Virtual Machines or instances in a Virtual Machine Scale Set.*"

5.1.21 Autoscale

"*Autoscale is a service that you can use to automatically add and remove resources according to the load on your application.*

When your application experiences higher load, autoscale adds resources to handle the increased load. When load is low, autoscale reduces the number of resources, which lowers your costs. You can scale your application based on metrics like CPU usage, queue length, and available memory. You can also scale based on a schedule. Metrics and schedules are set up in rules. The rules include a minimum level of resources that you need to run your application and a maximum level of resources that won't be exceeded."

5.1.22 Azure Database for PostgreSQL Server

"*Azure Database for PostgreSQL - Flexible Server is a fully managed database service designed to provide more granular control and flexibility over database management functions and configuration settings. The service generally provides more flexibility and server configuration customizations based on user requirements. The flexible server architecture allows users to collocate the database engine with the client tier for lower latency and choose high availability within a single availability zone and across multiple availability zones. Flexible servers also provide better cost optimization controls with the ability to stop/start your server and a burstable compute tier ideal for workloads that don't need full compute capacity continuously.*"

5.1.23 Azure Container Registry

"*Azure Container Registry is a managed registry service based on the open-source Docker Registry 2.0. Create and maintain Azure container registries to store and manage your container images and related artifacts.*

Use Azure container registries with your existing container development and deployment pipelines, or use Azure Container Registry Tasks to build container images in Azure. Build on demand, or fully automate builds with triggers such as source code commits and base image updates.”

5.1.24 Azure Application Insights

“Application Insights is an extension of Azure Monitor and provides application performance monitoring (APM) features. APM tools are useful to monitor applications from development, through test, and into production in the following ways:

- *Proactively understand how an application is performing.*
- *Reactively review application execution data to determine the cause of an incident.*

Along with collecting metrics and application telemetry data, which describe application activities and health, you can use Application Insights to collect and store application trace logging data. The log trace is associated with other telemetry to give a detailed view of the activity. Adding trace logging to existing apps only requires providing a destination for the logs. You rarely need to change the logging framework.”

5.1.25 Azure Blob Storage

“*Azure Blob Storage is Microsoft's object storage solution for the cloud. Blob Storage is optimized for storing massive amounts of unstructured data. Unstructured data is data that doesn't adhere to a particular data model or definition, such as text or binary data.*

Blob Storage is designed for:

- *Serving images or documents directly to a browser.*
- *Storing files for distributed access.*
- *Streaming video and audio.*

- *Writing to log files.*
- *Storing data for backup and restore, disaster recovery, and archiving.*
- *Storing data for analysis by an on-premises or Azure-hosted service.*

Users or client applications can access objects in Blob Storage via HTTP/HTTPS, from anywhere in the world. Objects in Blob Storage are accessible via the Azure Storage REST API, Azure PowerShell,Azure CLI, or an Azure Storage client library. Client libraries are available for different languages, including .NET, Java, Node.js, Python, Go.”

5.1.26 Azure Storage Account

“An Azure storage account contains all of your Azure Storage data objects: blobs, files, queues, and tables. The storage account provides a unique namespace for your Azure Storage data that's accessible from anywhere in the world over HTTP or HTTPS. Data in your storage account is durable and highly available, secure, and massively scalable.“

5.1.27 Azure Function

“Azure Functions is a serverless solution that allows you to write less code, maintain less infrastructure, and save on costs. Instead of worrying about deploying and maintaining servers, the cloud infrastructure provides all the up-to-date resources needed to keep your applications running.

You focus on the code that matters most to you, in the most productive language for you, and Azure Functions handles the rest.

With Functions, you write your function code in your preferred language using your favorite development tools and then deploy your code to the Azure cloud. Functions provides native support for developing in C#, Java, Javascript, PowerShell, Python, plus the ability to use more languages, such as Rust and Go.

Functions integrates directly with Visual Studio, Visual Studio Code, Maven, and other popular development tools to enable seemless debugging and deployments.

Functions also integrates with Azure Monitor and Azure Application Insights to provide comprehensive runtime telemetry and analysis of your functions in the cloud.”

From the project profile that specific architecture design has been assumed to accomplish the goal of building a cloud infrastructure for a company which wants to deploy its (web) application robustly and securely.

The concept so is that incoming requests arrive from the internet to the bastion host in the public subnet which is used to manage access to the private network from the internet; in this way access control becomes easier to manage while minimizing the potential attack surface. Then requests are forwarded to the worker nodes of the kubernetes cluster inside the private network, where the application resides, which are responsible to process them and to provide the response back to the NAT gateway so to where the request were coming, the internet.

The idea is to expose a frontend microservice of the application by means of the load balancer to a server of the public subnet (it can also be exposed on the bastion host considering the architecture of this project but this is not recommended in an enterprise use case because it is typically used for secure remote access to the private network and its primary purpose is to provide a controlled and secure entry point into your network from the public internet) and then this frontend spans requests to the rest of the microservices in the cluster basing on which ones are needed to process the requests (e.g. database microservice, payment microservice, or others).

A postgresql database has been used for managing the application backend and data persistence and bastion host as well as the AKS worker nodes have been managed by providing automation by means of both autoscale and Azure functions (5.1.27). Specifically they are powered on at 9 AM and powered off at 7 PM for each working day in order to avoid human presence in performing this kind of tasks but especially to automate the processes, to save servers energy and costs.

Azure Container Registry has been used as a resource in which images for AKS pods and services can be pushed in, and storage account, blob storage, application insights have been used in order to create the function app which manages the automated database booting and shutdown.

Either for AWS and Azure, from a security perspective, network security groups and route table have been configured according to allow just a set of IP address from the outside to reach the internal resources of the infrastructure (the application microservices) and to permit the outcoming traffic to reach the internet without particular restrictions. More precisely the set of IP addresses, according to an internal choose taken by Liquid Reply tutors, refers to 12 public IP address related to 12 “entrypoints” to the infrastructure which are basically owned by Liquid Reply itself. This way allow to manage the access flow by detecting who has to be denied and who can access the infrastructure.

It is appreciable, from figure 29 and figure 30, how the design principles and architecture best practices have been respected and utilized for the design. Two factors have to be consider: the way for which the best practices have been implemented and the structural (and architectural) differences of the two cloud providers considered.

First of all, both the topologies illustrate a strong example of high availability, resiliency, robustness, redundancy, scalability, cross-zone communication; in fact these properties are guaranteed by many elements but especially by the presence of two availability zones which ensure applications to be available whether failures such as power outage rather than network or hardware issues occurred.

The chosen region, in which successively these infrastructures have been deployed, refers to **eu-south-1** (Milan) for AWS and **West Europe** for Azure.

Usually a physical region have three availability zones available which consist in three geographically separated datacenters but for implementation and simplicity reasons two availability zones have been chosen.

By designing and deploying applications in a cloud infrastructure, as the ones represented in the pictures above, with multiple availability zones high availability is achieved as well as scalability and so resiliency. For instance a zone that experiences problems, the traffic can be redirected to resources in the other zone without significant downtime so also redundancy is guaranteed as a matter of fact. Moreover zones enable the scaling of resources to meet the changing demands, by distributing workloads in the two areas, as well as planning a disaster recovery strategy for quickly recovering from disasters or outage in a zone but this point is explained later. In addition with low-latency and high-bandwidth connections applications can communicate with each other (cross-zone communication) so it represents a good solution for distributed system.

From the pictures it is noticeable both the availability zones are affected by a VPC (V**irtual Private Cloud**) in AWS and by a **Virtual Network** in Azure. This cloud resource assumes two different names depending on the cloud provider but it allows the main action which regulate the whole infrastructure which is the networking management; it encloses all of the resources included in the architecture and permits to manage incoming/outcoming traffic of the infrastructure itself. While in the AWS case there is just a VPC which spans across both the availability zones, regulates the connections among resources by IP and security groups management, in the Azure case instead two virtual networks have been necessary because of the infrastructure needs and of the Azure internal self objects management/disposition which is explained in the next paragraphs.

A first key difference which emerges is the **resource group** resource. This is present just in Azure, in fact AWS does not have it, for having a better resources management. More specifically it acts like a collector which encloses all of the resources so even the virtual network belongs to a distinct resource group.

The reason why two resource groups and so two virtual networks have been needed is due to how Azure manage its kubernetes service. As a matter of fact, creating an AKS means Azure is going to create a separate resource group completely dedicated to the cluster, including the virtual network on which the cluster itself resides in and all of the resources related to the AKS such as load balancer, virtual machine scale set, subnet.

VPC and Virtual Networks basically contain public and private subnets which in turn include all of the remaining cloud resources.

At this point it is compulsory to highlight one of many structural differences between AWS and Azure in terms of networking, especially of subnets. In both the cloud providers subnets are created inside the VPC or Virtual Network but considering amazon environment firstly, subnets are associated with a specific availability zone so they cannot span multiple AZs; this mechanism allows to distribute resources across multiple zones for a better resilience. Furthermore, to achieve high availability it is necessary to create separate subnets in each AZ and then set up networking and load balancing between them (as it has been done by observing the fig 29 and fig 30.

As regard Azure, availability zones are managed similarly to AWS but, conversely to itself, subnets can span across multiple AZs within the same physical region providing so more flexibility in designing highly available architectures as well as in distributing resources across AZs while still keeping them within the same network. It is observable therefore how

Azure offers more flexibility in terms of subnet design compared to AWS where subnets have to be created zone-independently for isolation and fault tolerance.

Another difference to underline is how the distinction between public and private subnet is managed by both the cloud providers. In AWS it is possible to specify whether a subnet has to be public or private, either by means of the AWS portal or, for this project, by means of specific parameters (*mapPublicIpOnLaunch* set on true for the aws.ec2.Subnet() Pulumi API), while in Azure public subnets are not inherently different from private subnets in terms of how they are represented within a virtual network. The distinction between a public subnet and a private subnet is not determined by Azure's representation but rather by the network and resource configurations within the subnet. Public and private subnets are both defined as regular subnets within a VNet, but their characteristics are determined by their configuration and purpose as well. A public subnet is configured to have a route that directs traffic to the internet, usually via an Azure Nat Gateway. This allows resources within the subnet to send outbound traffic to the internet and receive inbound traffic from the internet. Additionally, public IP addresses may be associated with resources within the public subnet to make them directly accessible from the internet. In fact the bastion host represented in the fig. 30 has been assigned a public IP address in order to be directly accessible from the outside of the infrastructure.

In addition Azure does not have a dedicated resource called an Internet Gateway as other cloud providers have, like AWS. Instead, it uses a different approach for outbound internet connectivity, primarily relying on NAT gateways or network virtual appliances (NVAs) for this purpose. To enable outbound communication from resources in a private subnet to the internet, typically NAT Gateway, resources with a public IP address directly associated or which communicate to the NAT gateway itself, a route table which includes the default route 0.0.0.0/0 that directs traffic to the NAT Gateway as the next hop, network security groups to control inbound and outbound traffic at the subnet or network interface level by configuring rules to allow or deny specific traffic based on security requirements, are used.

5.2 Implementation settings and tools configurations

At the beginning of the practical development the following tools, softwares and configurations have been needed.

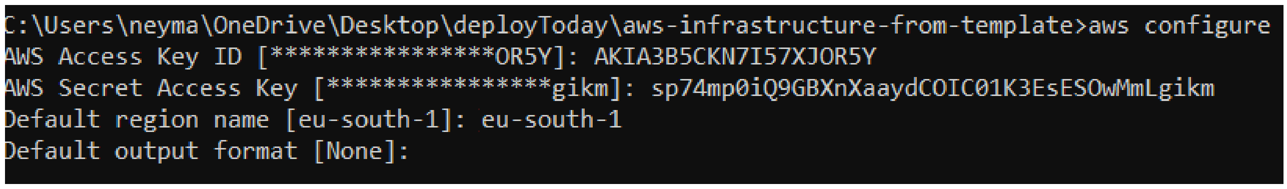
The project has been developed in Typescript language as a personal choice for more confidentiality and to leverage the potentialities this language offers in terms of computational speed, logic complexity, simple, intuitive and efficient constructs. So Node.js framework has been installed on the local machine and Visual Studio Code has been used as IDE editor to code.

The operating system used has been Windows 10 and the installation of Pulumi has been performed by means of the **choco install pulumi** command, as suggested by the Pulumi official website https://www.pulumi.com/docs/clouds/aws/get-started/begin, which led to finally have the Pulumi CLI. This has been used for performing commands related to the deployment, the resources, the stack, and other pulumi elements.

After that AWS and Azure have been considered separately and sequentially so it is proper to explain both the solutions separately as well.

5.3 Code implementation for AWS provisioning

At the beginning of all the processes, the first thing to carry out has been to configure the connection with my personal AWS profile. Tutors in the company had created an AWS identity by means of the IAM service: it allows to manage permissions that establish which resources a user can access to. So a key pair has been provided, related to my identity, composed by an ACCESS_KEY_ID and a SECRET_ACCESS_KEY which have been used by Pulumi to connect to AWS platform in order to perform the next deployment tasks.

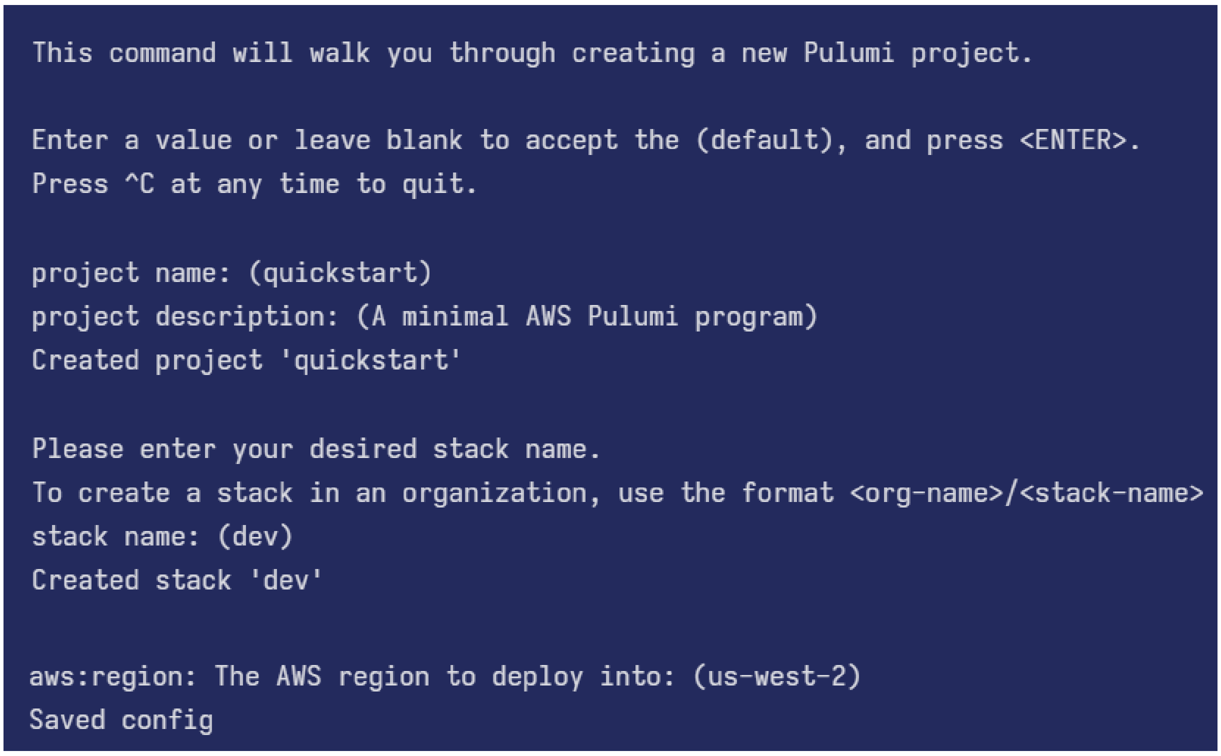


```
C:\Users\neyma\OneDrive\Desktop\deployToday\aws-infrastructure-from-template>aws configure
AWS Access Key ID [****************OR5Y]: AKIA3B5CKN7I57XJOR5Y
AWS Secret Access Key [****************gikm]: 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
Default region name [eu-south-1]: eu-south-1
Default output format [None]:
```

Fig 17: AWS configure

Hence the Windows command line has been used to perform commands and in addition to those parameters, also the deploying region (**eu-south-1** → **Milan**) was required by **aws configure** command for establishing the connection between Pulumi and my AWS identity.

After this passages the next step has been to create the starter project by means of **pulumi new aws-typescript** command which has created a folder with all the starter files including the YAMLs of the environment configuration. Here in the CLI, a set of standard parameters has been requested: the **project name**, the **project description**, the **name of the stack in which resources are based**, and the **region**.



```
This command will walk you through creating a new Pulumi project.

Enter a value or leave blank to accept the (default), and press <ENTER>.
Press ^C at any time to quit.

project name: (quickstart)
project description: (A minimal AWS Pulumi program)
Created project 'quickstart'

Please enter your desired stack name.
To create a stack in an organization, use the format <org-name>/<stack-name>
stack name: (dev)
Created stack 'dev'

aws:region: The AWS region to deploy into: (us-west-2)
Saved config
```

Fig 18: pulumi new aws-typescript command

In the end the automated process creates the folder containing a set of files including the Pulumi.yaml related to the variables just cited, the package.json and package-lock.json for the pulumi aws libraries, and the index.ts which contains the code of the whole infrastructure.

The latter file has been the one which has gone through modification. According to the AWS architecture design explained in the **section 5.1,** modules related to networking, bastion host, EKS cluster, load balancer, Lambda function, RDS database, WAF, ECR, has been created and programmed separately in a different folder (modules) located inside the project directory. So let's have a sequential look firstly to how these modules have been coded by using the proper Pulumi APIs, to provision the needed cloud resources, which can be found in their **official online documentation,** and eventually to how these wrapped APIs have been called inside index.ts.

Each module acts as a library and contains a special-purpose class which includes the definitions of several functions whose task is to return the Pulumi API related to the cloud resources to provision for the infrastructure creation. Functions parameters are passed to the called Pulumi API. The project has been pushed on this github repository: https://github.com/dev9815/pulumiAWSTemplateThesis/

5.3.1 Networking module

In this typescript module file, called *vpc-networking.ts*, resources about the networking management have been created. More precisely the task here, as in each module, is to wrap the APIs, one at a time, into custom functions to be called inside index.ts. So a dedicated class has been created containing all of these functions.

In order the following wrappers are presented:

-
  ```
  public createVpc (
      vpcName: string,
      vpcNetworkCidr: string,
      resources: any[]
  )
  ```

  This is the first function of the class and it takes care of creating a vpc, as it can be understood by the function name, and the following parameters have been passed: **the name of the vpc**, **the address space of the vpc (10.0.0.0/16)**, **a list of resources** which is explained later in a dedicated chapter.

  This function is a wrapper because its body is just a call to a pulumi API which is in charge of creating a vpc. The parameters this API accepts are the ones passed to createVpc(). Here it is: **aws.ec2.Vpc**

-
  ```
  public createSubnet (
      subnetName: string,
      subnetCidr: string,
      idVpc: pulumi.Output<string>,
      az: string,
      ipType: boolean,
      resources: any[]
  )
  ```

  The task of this function is certainly to create a subnet which respects the passed parameters about subnet **name**, **address space**, **referred vpc**, **availability zone** in which residing in, **type of IP (public or private)**, and the **list of resources**.

According to the AWS design, 4 subnets had to be created so this function has been called four times in the index.ts in order to have just a wrapper which could suit both the public and private subnets.

One public subnet and one private have been designed per each availability zone so the value of the **az** parameter changes as well and the public/private is indicated by the **ipType** parameter.

This wrapper calls inside the Pulumi API responsible for creating a subnet: **aws.ec2.Subnet**

-
  ```
  public createInternetGateway ( name: string,
          idVpc: pulumi.Output<string>,
          resources: any[]
  )
  ```

  An internet gateway is created. It is needed to establishing connection from the outside to the inside of the infrastructure and viceversa. In AWS this cloud resource is needed differently from Azure. Specifically this custom function needs **the name of the IG**, **the vpc in which it should stay in**, and **the list of resources** parameter.

  The pulumi API called inside is the following: **aws.ec2.InternetGateway**

-
  ```
  public createEip (nameEip: string, resources: any[])
  ```

  Each public subnet, in the first as well as in the second AZ, includes a NAT Gateway needed for the instances in the private subnets to connect to the internet, as it can be appreciable from the AWS infrastructure design. So a public IP address is needed to be associated to the NAT Gateway; the only parameter to pass to the returned Pulumi API **aws.ec2.Eip** is just the name of the public IP address, resources param is used for another purpose.

-
  ```
  public createNatGateway (
       name: string,
       eipAllocationId: pulumi.Output<string>,
       subnetId: pulumi.Output<string>,
       resources: any[]
  )
  ```

  The creation of a NAT gateway provide for calling **aws.ec2.NatGateway** API. The parameters refer to **the name of the resource**, **the public IP address** to associate to the NAT gateway, as just seen earlier, **the id of the subnet i**n which this resource should be located, and **the list of resources**.

-
  ```
  public createRouteTable ( name: string,
       idVpc: pulumi.Output<string>,
       idGateway: pulumi.Output<string>,
       resources: any[]
  )
  ```

  Subnets have configured either by considering their route tables in order to route the traffic properly to the destinations to reach, which are the worker nodes in the private subnets. So routes for the private subnets and for the external internet (0.0.0.0/0) have been configured by creating this function whose task is to return the **aws.ec2.RouteTable** Pulumi API. The parameters refer to **the name of the resource**, **the vpc id** the resource refers to, **the id of the gateway** to associate to the route table, **the list of resources**. A route table for both the public subnets is needed and two distinct ones for the private subnets. In the public route table so the id of the gateway refers to the id of the internet gateway while in the first and second private route tables it stands for the id of the first and second NAT gateways.

-
  ```
  public subnetAssociation (name: string,
       SubnetId: pulumi.Output<string>,
       idRouteTable: pulumi.Output<string>,
       resources: any[]
  )
  ```

  This resource is needed for associating the proper route table to the proper subnet.

So it is not a distinct cloud resource but let’s say an “association resource” where the id of the subnet, the id of the route table, and the list of the resources are passed to **aws.ec2.RouteTableAssociation** API.

5.3.2 Bastion Host module

Module for the cloud resources related to the bastion host. The typescript file is named *bastionhost.ts* and it contains the **BastionHost** class with all the related functions which calls the proper Pulumi APIs.

In this module there are the following functions:

-
  ```
  public securityGroup ( name: string,idVpc:
  pulumi.Output<string>,resources: any[])
  ```

  aws.ec2.SecurityGroup Pulumi API is called here by passing to it the name of the security group, the id of the vpc, and the list of resources. Security group is needed to the bastion host in order to manage the control access by specifying which are the public IP addresses allowed to access the infrastructure. This is a security mechanism, such as kind of firewall, to prevent undesired access which will be denied. A list of public IPs has been provided and it is shown at the end of the 5.3 chapter.

-
  ```
  public instanceTemplate (
        name: string,
        idSG: pulumi.Output<string>,
        resources: any[]
  )
  ```

  The bastion host virtual profile is defined inside this function. More precisely this wrapper calls the **aws.ec2.LaunchTemplate** API where the **OS image**, **the IAM permissions**, **the instanceType**, **the id of the vpc security group** are specified inside the API. The external function in addition passes also **the name of the LaunchTemplate resource**, and **the list of resources**. In order to reduce costs *t3.micro* instance type is used.

-
  ```
  public createBastionHost (
      name: string,
      firstPublicSubnetId: pulumi.Output<string>,
      secondPublicSubnetId: pulumi.Output<string>,
      instanceTemplateId: pulumi.Output<string>,
      resources: any[]
  )
  ```

  With this function the creation of the bastion host is performed. In details, as mentioned in some previous paragraph, the provisioning of this resource has been made by taking care of automation aspect. In fact this wrapper function calls aws.autoscaling.Group Pulumi API which scales up the number of instances at 1. Obviously the indicated parameters about **the bastion host name**, **subnet ids**, **the list of resources** and **the reference to the LaunchTemplate** resource previously described are passed to that API.

-
  ```
  public scheduledAction (
          name: string,
          action: string,
          recurrence: string,
          asName: pulumi.Output<string>,
          minSize: pulumi.Input<number>,
          maxSize: pulumi.Input<number>,
          desiredCapacity: pulumi.Input<number>,
          resources: any[]
  )
  ```

  Here the automate bastion host booting and shutdown is handled. The autoscaling increase the number of instances at 1 but with this function the type of the action and its recurrence have been specified. This library function so is called twice in the index.ts, for booting and for shutdown. It works by declaring the minSize, maxSize and desiredCapacity parameters to 1 for booting and to 0 for shutdown. The bastion host gets powered on at 9AM and gets over at 7PM. With this behaviour automation is achieved as well as performance efficiency. The called Pulumi API is aws.autoscaling.Schedule**.**

5.3.3 EKS module

Kubernetes cluster functions have been programmed in this module within the *eks.ts* file. EKS resource has been used in order to manage the company's effective application; worker nodes are in charge to host the workloads as well as the microservices and their connections in order to provide the requested service. A relational database has been further attached in order to manage the backend of the hosted application with a persistence of for data. Obviously since the goal of the project was to build the implementation of the architecture, the actual application implementation is not a responsibility of the current project but it is proper to mention it as software solutions which can fit this infrastructure proposal in order to have a better comprehension of the modalities the infrastructure itself has been raised up. A class including functions about provisioning of security group, permissions, cluster, nodegroup, automated action scheduling, has been handled.

-
  ```
  public securityGroup (name: string,
          idVpc: pulumi.Output<string>,
          vpcCidr: pulumi.Output<string>,
          resources: any[]
  )
  ```

  aws.ec2.SecurityGroup has been configured differently from the one used for the bastion host; in fact here it has been specified which is the allowed incoming and outcoming traffic as well as the allowed ports. More precisely incoming traffic from ports 0 and 443, and from the vpc (so from IP addresses belonging to the vpc address space) is accepted but for the outcoming the there had not been restrictions so the default route 0.0.0.0/0 has been used.

-
  ```
  public createRole (name: string, resources: any[])
  ```

  In order to create the EKS cluster, since my IAM identity on AWS did not has all of the needed permissions to perform actions to EKS it has been necessary to use the aws.iam.Role API from Pulumi to create the permissions for provisioning the cluster. The policies in question are the following: *arn:aws:iam::aws:policy/AmazonEKSClusterPolicy*, *arn:aws:iam::aws:policy/AmazonEKSServicePolicy*,

arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy,
arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly

-
  ```
  public createCluster (name: string, roleArn:
       pulumi.Output<string>, firstPrivateSubnetId:
       pulumi.Output<string>, secondPrivateSubnetId:
       pulumi.Output<string>, idSG:
  pulumi.Output<string>,   resources: any[]
  )
  ```

  As the function name suggests, this is the function in charge to provision the kubernetes cluster. Since the id of the first and second private subnets are passed as parameters, the cluster spans across both the availability zones. The API used is aws.eks.Cluster and the parameters refers to **the cluster name**, **the** previous **role** to have permissions to operate with the EKS, **the private subnet ids**, **the id of its security group**, and **the list of resources**. The used address space is 172.20.0.0/16.

-
  ```
  public createNodeGroup (
       name: string,
       clusterId: pulumi.Output<string>,
       roleArn: pulumi.Output<string>,
       firstPrivateSubnetId: pulumi.Output<string>,
       secondPrivateSubnetId: pulumi.Output<string>,
       instanceType:
  pulumi.Input<pulumi.Input<string>[]>,
       nodeGroupName: string,
       resources: any[]
  )
  ```

  While the previous function was responsible for creating the cluster this one creates the EKS worker nodes, known as **nodegroup.** So the parameters are **the name of the resource**, **the cluster id** in which it has been located, **the ARN of the IAM role**, **the private subnet ids**, **the instance type**, **the nodegroup name**, and

the list of resources. The instance type is *t3.micro*, the cheapest one in order to remain cost-effective; since the goal of the thesis project is to focus on the infrastructure and not on the functional application, the cheapest resources have been chosen. This means that whether more computational resources would be needed, a powerful instance type has to be selected, such as *t3.medium* or *t3.large* but of course these solution are also more expensive. The wrapped API is **aws.eks.NodeGroup**

-
  ```
  public scheduledAction (
      name: string,
      action: string,
      recurrence: string,
      asName: pulumi.Output<string>,
      minSize: pulumi.Input<number>,
      maxSize: pulumi.Input<number>,
      desiredCapacity: pulumi.Input<number>,
      resources: any[]
  )
  ```

  This looks equal to the function of the Bastion Host module but it is used for automating the task of EKS booting and shutdown, more precisely this refers to the worker nodes which are *t3.micro* virtual machines.

  The operation is the same of the Bastion Host scheduledAction(). The only particular to add is that the previous function createNodeGroup() returns a set of parameters, as every called API of each module, including the reference to the worker nodes autoscaling group; this is passed in the **asName** parameter because the called Pulumi API aws.autoscaling.Schedule needs it.

5.3.4 Load balancer module

The load balancer library deals with provisioning the resources related to the load balancer. It is in charge to route and to separate the traffic among the microservices (nodegroup of the cluster) in the cluster and it can also be used, according to the use case meant to be reproduced, for exposing a service publicly by assigning it a public IP address and a port.

Basing on this premise, the API used have been the following ones:

-
  ```
  public securityGroup (name: string,
        idVpc: pulumi.Output<string>, resources: any[]
  )
  ```

  It is the same securityGroup function used so far, which calls the same used API. The same list of public IPs of the bastion host security groups has been used but instead of port 443, ports 80 and 31803 have been used.

-
  ```
  public createLoadBalancer (
    name: string,
    lbPort: number,
    lbProtocol: string,
    idSG: pulumi.Output<string>,
    firstPublicSubnetId: pulumi.Output<string>,
    secondPublicSubnetId: pulumi.Output<string>,
    resources: any[]
  )
  ```

  As the name suggest here the load balancer is provisioned. In particular it has been assigned the port, the protocol, the id of its security group, the subnet ids, the list of resources. The load balancer type is "application" and the used API is **aws.lb.LoadBalancer.**

-
  ```
  public createTargetGroup (name: string,
        idVpc: pulumi.Output<string>,
        port: number, protocol: string,
        targetType: string, resources: any[]
  )
  ```

  Since the application in the cluster is served by two worker nodes, these instances are registered into the target group. The load balancer distributes the incoming traffic across the instances to balance the load. So the needed parameters are the name of the resource, the port, the protocol the type of the target (instance) and the list of resources which is independent from the purpose. The Pulumi API used is **aws.lb.TargetGroup**.

-
  ```
  public createListener (
        name: string,
        tg: pulumi.Output<string>,
        lbArn: pulumi.Output<string>,
        port: number,
        protocol: string,
        resources: any[]
  )
  ```

  A listener is used to define the protocol and port the load balancer uses to route traffic from clients to the registered targets in the target group. It determines how incoming traffic is handled and where it should be directed (to the worker nodes of the cluster obviously).

  The parameters so are referred to the resource name, the referred target group, the ARN of the load balancer to refer it, the port, the protocol, and a list of resource which has been passed for another purpose. The pulumi API is **aws.lb.Listener**.

-
  ```
  public createAttachment (
        name: string,
        autoscalingGroupName:pulumi.Output<string>,
        lbTargetGroupArn: pulumi.Output<string>,
        resources: any[]
  )
  ```

  This resource, just like the subnetAssociation() function of the networking module, has been provisioned to associate the autoscaling group of the worker nodes to the target group for the load balancer, by means of **aws.autoscaling.Attachment**

5.3.5 Relational Database Service module

As mentioned in the previous chapter (5.3.5) the AWS RDS db has been used for backend management purposes. Each organization needs to handle data persistently, methodically and tidily, so in general it is necessary to provide for a solution which involves the provisioning of a database. Among the several db services offered by AWS, for this project **RDS** has been chosen and the following ones are the library functions implemented for managing it:

-
  ```
  public securityGroup (
          name: string,
          idVpc: pulumi.Output<string>,
          ingressFromPort: number,
          ingressToPort: number,
          vpcCidrBlock: pulumi.Output<string>,
          resources: any[]
  )
  ```

  Such as each security group described so far, this element is needed to prevent from undesired access. So it filters the requests directed to the database basing on where they come from. Only the ones from within the vpc are accepted. **aws.ec2.SecurityGroup**

-
  ```
  public createSubnetGroup (
          name: string,
          firstPrivateSubnetId: pulumi.Output<string>,
          secondPrivateSubnetId: pulumi.Output<string>,
          resources: any[]
  )
  ```

  The RDS database has to be deployed in a list of subnets (at least one), so this function is in charge to call aws.rds.SubnetGroup in order to specify the related subnets (the two private ones). Since one of the main goals of the whole project is to provide high availability, more than one subnet has to be specified. When the RDS is created a group of subnet is associated to it and the database will pick a subnet from that group to launch an instance into it. In this case if a subnet goes down the RDS instance can failover to a different subnet in the group, hence providing better reliability and uptime.

-
  ```
  public createInstance (
        name: string,
        instanceClass: string,
        dbName: string,
        engine: string,
        az: string,
        idSG: pulumi.Output<string>,
        username: string,
        password: string,
        port: number,
        subnetGroupName: pulumi.Output<string>,
        resources: any[]
  )
  ```

  Here the effective instance creation is performed. All the related parameters are indicated: the name, the type of instance, the amount of storage in GB, the engine (postgresql), the AZ, the related security group, username, password, port, the related group of subnets, and the list of resources. aws.rds.Instance is called.

5.3.6 Web Application Firewall module

The reason why a WAF has been involved on the project is to take care also of the security aspect. In particulare this cloud resource has been provisioned for protecting hosted web applications from various online threats and attacks; in fact it results useful against SQL injection, cross-site scripting, cross-site request forgery, DDoS attacks, and others. The related library functions have been implemented inside the *webAcl.ts* typescript file and they consists in:

- `public createWebAcl (name: string, resources: any[])`

  The creation of the web access control list performed by the aws.wafv2.WebAcl Pulumi API

-
  ```
  public createAssociation (
        name: string,
        resourceArn: pulumi.Output<string>,
        webAclArn: pulumi.Output<string>,
        resources: any[]
  )
  ```

  A WAF is associated to the load balancer (resourceArn param) in order to allow the WAF to inspect and filter traffic before it reaches the application servers behind the load balancer. This is useful when there are multiple instances of the application running and each one must be protected. **aws.wafv2.WebAclAssociation** API has been involved.

5.3.7 Lambda Function module

The file on the project is *lambda.ts*.

The lambda function resource has been provisioned for performing a specific operation automatically: the booting/shutdown of the RDS instance. It allows to define a function in whatever programming languages, typescript has been chosen to remain coherent to the whole development, that is triggered to execute a specific task.

Three main resources have to be considered: **EventRule, EventTarget, Lambda Function**. These are components of the EventBridge serverless service.

The first one defines the conditions that trigger the execution of a specific action when an event matching those conditions occurs;

the second one receives events when an EventRule triggers. When an event matches the conditions specified in an EventRule, EventBridge sends that event to the associated EventTarget for further processing;

the third one is a type of EventTarget. When a Lambda function is configured as an EventTarget for an EventBridge rule, it means that the Lambda function will be invoked (triggered) when an event matching the rule's conditions is generated.

So the correlation is:

1. Creating an EventRule in EventBridge, specifying the conditions (event pattern or schedule expression) that define when the rule should trigger.
2. Configuring one or more EventTargets for that EventRule which can be a Lambda function. So when the conditions specified in the EventRule are met, EventBridge will send the event data to the Lambda function as an input.
3. The Lambda function receives the event data and can execute custom code in response to the event. Lambda functions are used to perform tasks such as data processing, integration with other AWS services, sending notifications, and more.

The implemented functions are the following:

- `public createRole (name: string, resources: any[])`

  Since the IAM identity related to my AWS account missed some permission in terms of policies and roles, there were the need to add the permission to manage lambda functions. This has been solved by implementing this function which calls aws.iam.Role Pulumi API. In particular the called pulumi function wants some fields such as *Action* and *Service* (the most relevant ones) to be specified (respectively to *sts:AssumeRole* and *lambda.amazonaws.com*). In practice this allows one IAM entity (mine *d.manca*) to assume the permissions and role of another IAM entity (the lambda role) temporarily. This operation is commonly used to delegate access and permissions within AWS services and resources securely. The AssumeRole concept is a fundamental aspect in AWS identity and access management, enabling secure delegation of permissions and adhering to the principle of least privilege. It helps organizations manage and control access to AWS resources while maintaining security and compliance.

- `public createPolicy(name: string, resources: any[])`

  After assuming a role the next step is to indicate which are the policies needed to perform the needed operations: starting and stopping the database. So the AWS actions needed (in the *Action* filed of the aws.iam.Policy Pulumi API used) are “*rds:DescribeDBInstances*", "*rds:StopDBInstance*", "*rds:StartDBInstance*". The parameters of the wrapper function createPolicy have been also passed to the called API, excluding *resources*. It is the same for the previous function.

-
  ```
  public createAttachment ( name: string,
          policyArn: pulumi.Output<string>,
          role: pulumi.Output<string>,
          resources: any[]
  )
  ```

  A role contains a policy or a set of them; so by calling **aws.iam.RolePolicyAttachment** API the created policy is attached to the created role. This is performed through the ARN of both the policy and the role (a unique AWS resource identifier).

-
  ```
  public createLambdaFunction(
          name: string,
          role: pulumi.Output<string>,
          action: string,
          instanceId: pulumi.Output<string>,
          resources: any[]
  )
  ```

  After having created policies and role for having the needed permission to manage lambda function resources, here it is the creation of the lambda that takes in input the name, the just created role, the action, the id of the instance in which it has to operate, and the latter parameter which is the same for each module function. Since this resource has been used for starting and stopping the RDS instance automatically, differently from the automated mechanism used for the worker nodes or for the bastion host, the *action* parameter correspond to a string whose values can be aws.lambda.Function "*start*" (at 9AM of each working day) or "*stop*" (at 7PM of each working day).

  The *instanceId* parameter obviously refers to the id of the RDS db. The relevant point to underline here is that the called Pulumi API in its definition allows to define the body of the function which performs the start/stop operation.

The code of the function depends on the needs to apply and can be specified in whatever allowed programming language: in order to be coherent typescript has been used. This function is called twice: for starting and stopping.

-
  ```
  public scheduleAction (
        name: string,
        time: string,
        resources: any[]
  )
  ```

  Creation and scheduling of the lambda are two distinct processes: in fact by means of `aws.cloudwatch.EventRule` the lambda gets triggered at a specific indicated time. The related parameter, as it can be noticeable, is a CRON string which is the standard format dealing with time scheduling. It is represented by *cron(0 7 ? * MON-FRI *)* which in this specific example means it has to trigger the lambda from Monday to Friday at 7AM. Even this function is called twice, once for starting and once for stopping.

-
  ```
  public createEventTarget (name: string, rule:
  pulumi.Output<string>,
  lambdaArn: pulumi.Output<string>,
  eventRule: aws.cloudwatch.EventRule,resources: any[]
  )
  ```

  aws.cloudwatch.Event`Target` resource is provisioned twice for the same previous reason. The EventTarget stands for the lambda function for which the EventRule has to be applied to. Hence the name of the rule, the ARN of the lambda and the EventRule are parameters needed to pass the Pulumi API. Here there is the association between the scheduling rule which triggers the lambda and the lambda itself.

-
  ```
  public createPermission (name: string,
        action: string, principal: string,
        sourceArn: pulumi.Output<string>,
        func: pulumi.Output<string>,
        resources: any[]
  )
  ```

  Not only createRole() and createPolicy() are needed but also aws.lambda.Permission for giving an external resource such as the EventRule the proper permission to access the lambda functions. So *action* is equal to "*lambda:InvokeFunction*", *principal* to "*events.amazonaws.com*", *sourceArn* refers to the start and stop rules (even this function is called twice in the *index.ts*), and *func* stands for the lambda function (start or stop).

5.3.8 ECR module

Kubernetes cluster is in charge to manage the supposed application, its microservices, by means of the kubernetes orchestration objects. If a microservice is considered, it is handled through kubernetes service objects (SVC), which demand the requests to the backend kubernetes deployment and so to the pods. Since each pod represent a container running typically a docker image, the presence of a registry service from/to which pulling/pushing images has been considered a necessary cloud resource to provision. In AWS this service is provided by the Elastic Container Registry service and by means of the related Pulumi API this object has been provisioned into a private subnet. So the file is *repository.ts* and the related class contains just a function corresponding to the creation of the repository.

-
  ```
  public createRepository(
        name: string,
        resources: any[]
  )
  ```

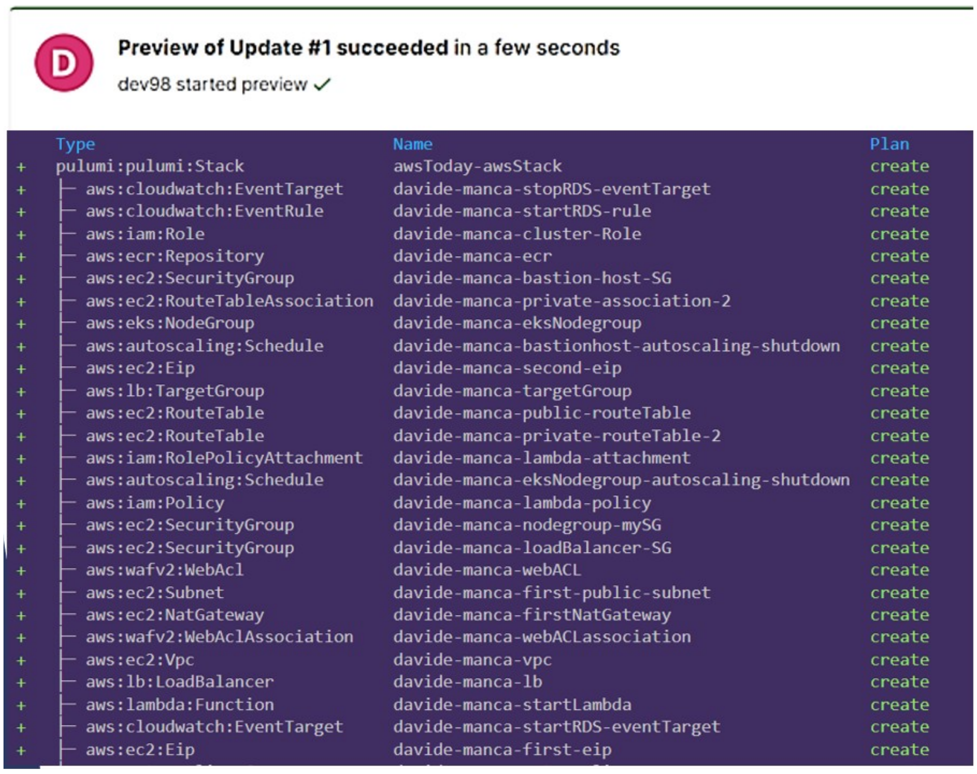
  As just mentioned, this wrapper function is in charge to call `aws.ecr.Repository` Pulumi API which creates the registry on the cloud with the name passed as a parameter to createRepository() function.

Images can be pushed and pulled from this object in order to be served for instance to the worker nodes related to the application in the EKS cluster.

resource parameter is explained later in a dedicated chapter.

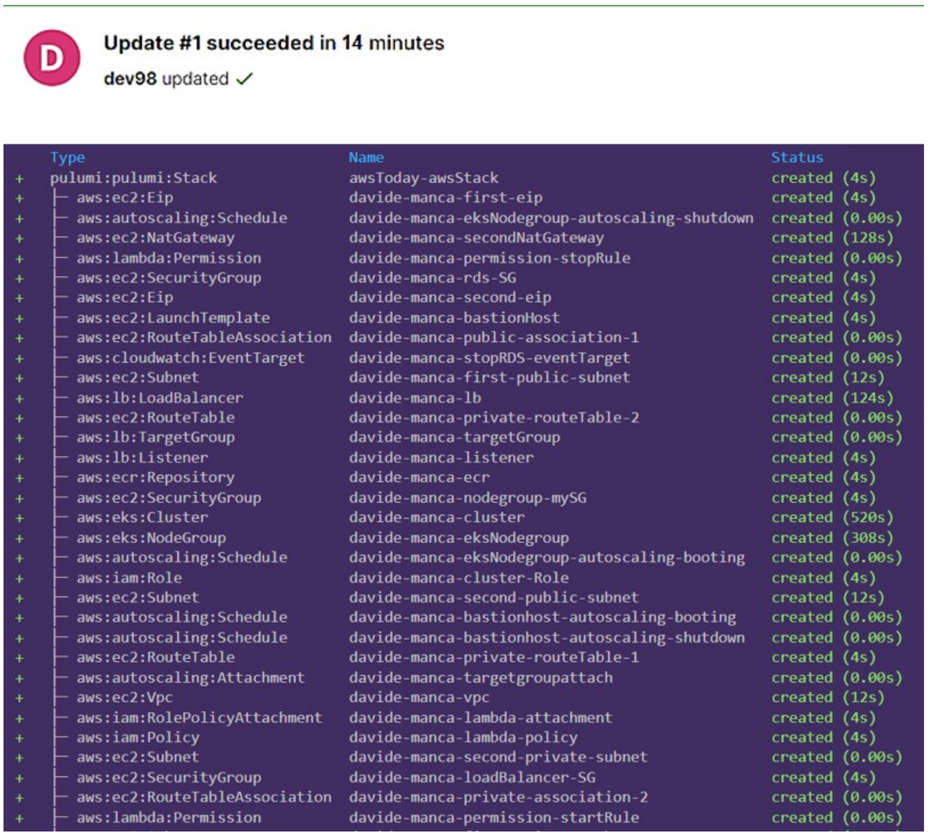
5.4 Results

With ECR module the implementation and the resources provisioning of the cloud infrastructure on AWS has been concluded. The next steps have been the deployment of the whole infrastructure on AWS by means of pulumi. All the resources are placed inside a pulumi stack so with **pulumi up --yes** command has been performed the deployment which has pushed each resource from the pulumi stack to the cloud platform, in a batch processing approach. The following figures illustrate the processes.


Preview of Update #1 succeeded in a few seconds
dev98 started preview ✓

```
    Type                              Name                                          Plan
+   pulumi:pulumi:Stack               awsToday-awsStack                             create
+   ├─ aws:cloudwatch:EventTarget     davide-manca-stopRDS-eventTarget              create
+   ├─ aws:cloudwatch:EventRule       davide-manca-startRDS-rule                    create
+   ├─ aws:iam:Role                   davide-manca-cluster-Role                     create
+   ├─ aws:ecr:Repository             davide-manca-ecr                              create
+   ├─ aws:ec2:SecurityGroup          davide-manca-bastion-host-SG                  create
+   ├─ aws:ec2:RouteTableAssociation  davide-manca-private-association-2            create
+   ├─ aws:eks:NodeGroup              davide-manca-eksNodegroup                     create
+   ├─ aws:autoscaling:Schedule       davide-manca-bastionhost-autoscaling-shutdown create
+   ├─ aws:ec2:Eip                    davide-manca-second-eip                       create
+   ├─ aws:lb:TargetGroup             davide-manca-targetGroup                      create
+   ├─ aws:ec2:RouteTable             davide-manca-public-routeTable                create
+   ├─ aws:ec2:RouteTable             davide-manca-private-routeTable-2             create
+   ├─ aws:iam:RolePolicyAttachment   davide-manca-lambda-attachment                create
+   ├─ aws:autoscaling:Schedule       davide-manca-eksNodegroup-autoscaling-shutdown create
+   ├─ aws:iam:Policy                 davide-manca-lambda-policy                    create
+   ├─ aws:ec2:SecurityGroup          davide-manca-nodegroup-mySG                   create
+   ├─ aws:ec2:SecurityGroup          davide-manca-loadBalancer-SG                  create
+   ├─ aws:wafv2:WebAcl               davide-manca-webACL                           create
+   ├─ aws:ec2:Subnet                 davide-manca-first-public-subnet              create
+   ├─ aws:ec2:NatGateway             davide-manca-firstNatGateway                  create
+   ├─ aws:wafv2:WebAclAssociation    davide-manca-webACLassociation                create
+   ├─ aws:ec2:Vpc                    davide-manca-vpc                              create
+   ├─ aws:lb:LoadBalancer            davide-manca-lb                               create
+   ├─ aws:lambda:Function            davide-manca-startLambda                      create
+   ├─ aws:cloudwatch:EventTarget     davide-manca-startRDS-eventTarget             create
+   ├─ aws:ec2:Eip                    davide-manca-first-eip                        create
```

Fig 19: Pulumi resources preview for AWS provsioning



Update #1 succeeded in 14 minutes

dev98 updated ✓

```
    Type                              Name                                         Status
+   pulumi:pulumi:Stack               awsToday-awsStack                            created (4s)
+   ├─ aws:ec2:Eip                    davide-manca-first-eip                       created (4s)
+   ├─ aws:autoscaling:Schedule       davide-manca-eksNodegroup-autoscaling-shutdown created (0.00s)
+   ├─ aws:ec2:NatGateway             davide-manca-secondNatGateway                created (128s)
+   ├─ aws:lambda:Permission          davide-manca-permission-stopRule             created (0.00s)
+   ├─ aws:ec2:SecurityGroup          davide-manca-rds-SG                          created (4s)
+   ├─ aws:ec2:Eip                    davide-manca-second-eip                      created (4s)
+   ├─ aws:ec2:LaunchTemplate         davide-manca-bastionHost                     created (4s)
+   ├─ aws:ec2:RouteTableAssociation  davide-manca-public-association-1            created (0.00s)
+   ├─ aws:cloudwatch:EventTarget     davide-manca-stopRDS-eventTarget             created (0.00s)
+   ├─ aws:ec2:Subnet                 davide-manca-first-public-subnet             created (12s)
+   ├─ aws:lb:LoadBalancer            davide-manca-lb                              created (124s)
+   ├─ aws:ec2:RouteTable             davide-manca-private-routeTable-2            created (0.00s)
+   ├─ aws:lb:TargetGroup             davide-manca-targetGroup                     created (0.00s)
+   ├─ aws:lb:Listener                davide-manca-listener                        created (4s)
+   ├─ aws:ecr:Repository             davide-manca-ecr                             created (4s)
+   ├─ aws:ec2:SecurityGroup          davide-manca-nodegroup-mySG                  created (4s)
+   ├─ aws:eks:Cluster                davide-manca-cluster                         created (520s)
+   ├─ aws:eks:NodeGroup              davide-manca-eksNodegroup                    created (308s)
+   ├─ aws:autoscaling:Schedule       davide-manca-eksNodegroup-autoscaling-booting created (0.00s)
+   ├─ aws:iam:Role                   davide-manca-cluster-Role                    created (4s)
+   ├─ aws:ec2:Subnet                 davide-manca-second-public-subnet            created (12s)
+   ├─ aws:autoscaling:Schedule       davide-manca-bastionhost-autoscaling-booting created (0.00s)
+   ├─ aws:autoscaling:Schedule       davide-manca-bastionhost-autoscaling-shutdown created (0.00s)
+   ├─ aws:ec2:RouteTable             davide-manca-private-routeTable-1            created (4s)
+   ├─ aws:autoscaling:Attachment     davide-manca-targetgroupattach               created (0.00s)
+   ├─ aws:ec2:Vpc                    davide-manca-vpc                             created (12s)
+   ├─ aws:iam:RolePolicyAttachment   davide-manca-lambda-attachment               created (4s)
+   ├─ aws:iam:Policy                 davide-manca-lambda-policy                   created (4s)
+   ├─ aws:ec2:Subnet                 davide-manca-second-private-subnet           created (0.00s)
+   ├─ aws:ec2:SecurityGroup          davide-manca-loadBalancer-SG                 created (4s)
+   ├─ aws:ec2:RouteTableAssociation  davide-manca-private-association-2           created (0.00s)
+   ├─ aws:lambda:Permission          davide-manca-permission-startRule            created (0.00s)
```

Fig 20: Deployment of AWS resources (pulumi up --yes)

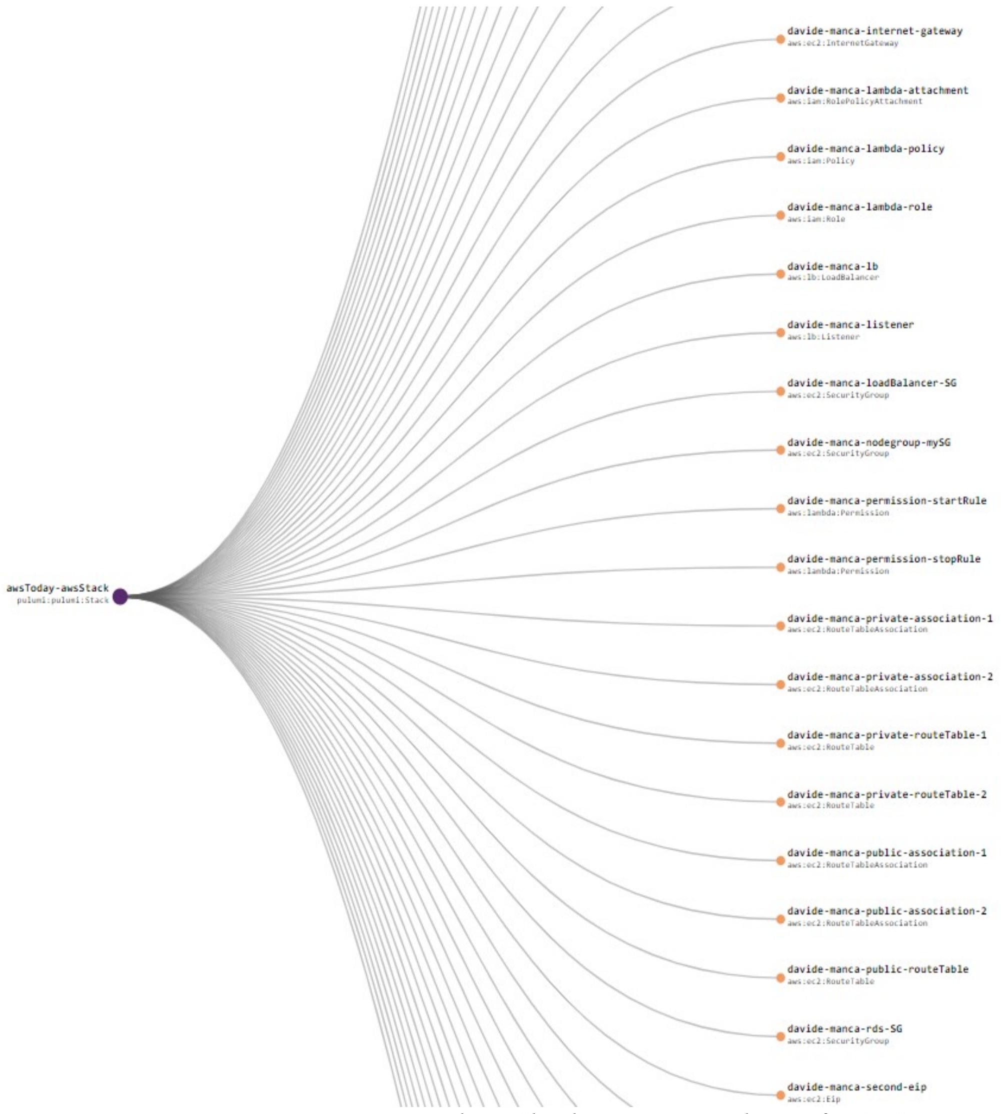
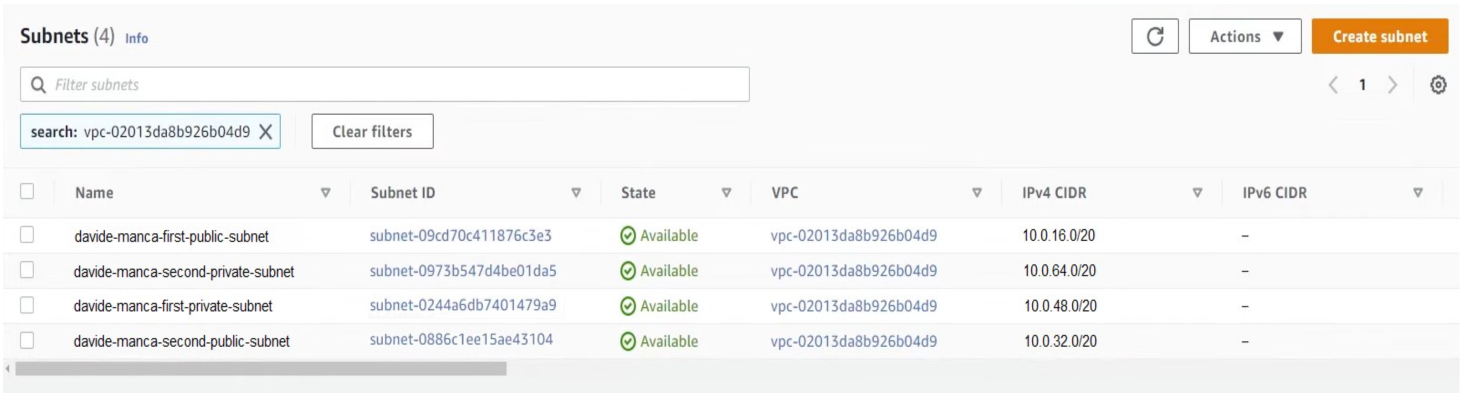

awsToday-awsStack
pulumi:pulumi:Stack
davide-manca-internet-gateway
aws:ec2:InternetGateway
davide-manca-lambda-attachment
aws:iam:RolePolicyAttachment
davide-manca-lambda-policy
aws:iam:Policy
davide-manca-lambda-role
aws:iam:Role
davide-manca-lb
aws:lb:LoadBalancer
davide-manca-listener
aws:lb:Listener
davide-manca-loadBalancer-SG
aws:ec2:SecurityGroup
davide-manca-nodegroup-mySG
aws:ec2:SecurityGroup
davide-manca-permission-startRule
aws:lambda:Permission
davide-manca-permission-stopRule
aws:lambda:Permission
davide-manca-private-association-1
aws:ec2:RouteTableAssociation
davide-manca-private-association-2
aws:ec2:RouteTableAssociation
davide-manca-private-routeTable-1
aws:ec2:RouteTable
davide-manca-private-routeTable-2
aws:ec2:RouteTable
davide-manca-public-association-1
aws:ec2:RouteTableAssociation
davide-manca-public-association-2
aws:ec2:RouteTableAssociation
davide-manca-public-routeTable
aws:ec2:RouteTable
davide-manca-rds-SG
aws:ec2:SecurityGroup
davide-manca-second-eip
aws:ec2:Eip

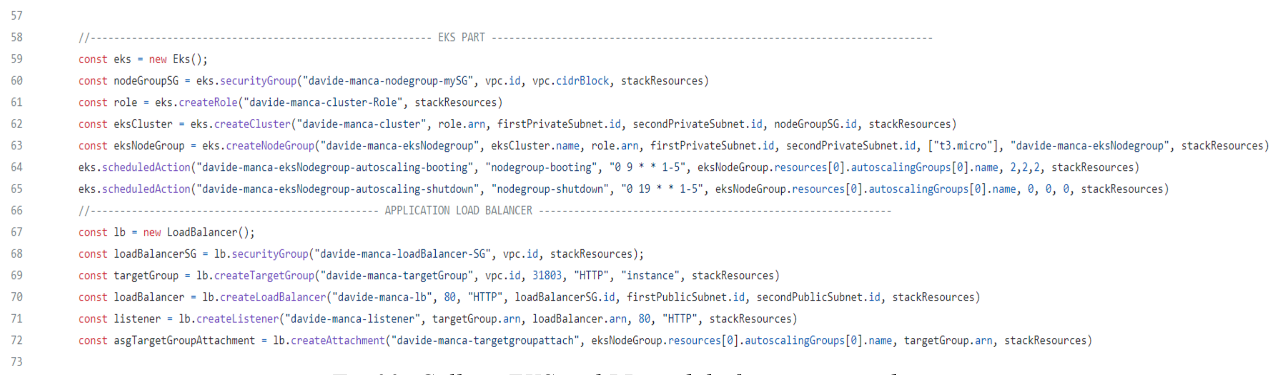
Fig 21: Pulumi Cloud resources graph view for AWS

The figure 21 is a picture taken from the Pulumi Cloud account. As already explained, when the resources are created they are contained into a pulumi stack. All of these can be visualized within the own Pulumi account where there is also either the possibility of manage them by dashboard and having a graph view of them. In particular the graph shows that each resource belongs to a specific stack (the purple point on the left).



Name	Subnet ID	State	VPC	IPv4 CIDR	IPv6 CIDR
davide-manca-first-public-subnet	subnet-09cd70c411876c3e3	Available	vpc-02013da8b926b04d9	10.0.16.0/20	–
davide-manca-second-private-subnet	subnet-0973b547d4be01da5	Available	vpc-02013da8b926b04d9	10.0.64.0/20	–
davide-manca-first-private-subnet	subnet-0244a6db7401479a9	Available	vpc-02013da8b926b04d9	10.0.48.0/20	–
davide-manca-second-public-subnet	subnet-0886c1ee15ae43104	Available	vpc-02013da8b926b04d9	10.0.32.0/20	–

Fig 22: Example of resource deployment on AWS: subnets



```
//---------------------------------------------------------- EKS PART -------------------------------------------------------------------------
const eks = new Eks();
const nodeGroupSG = eks.securityGroup("davide-manca-nodegroup-mySG", vpc.id, vpc.cidrBlock, stackResources)
const role = eks.createRole("davide-manca-cluster-Role", stackResources)
const eksCluster = eks.createCluster("davide-manca-cluster", role.arn, firstPrivateSubnet.id, secondPrivateSubnet.id, nodeGroupSG.id, stackResources)
const eksNodeGroup = eks.createNodeGroup("davide-manca-eksNodegroup", eksCluster.name, role.arn, firstPrivateSubnet.id, secondPrivateSubnet.id, ["t3.micro"], "davide-manca-eksNodegroup", stackResources)
eks.scheduledAction("davide-manca-eksNodegroup-autoscaling-booting", "nodegroup-booting", "0 9 * * 1-5", eksNodeGroup.resources[0].autoscalingGroups[0].name, 2,2,2, stackResources)
eks.scheduledAction("davide-manca-eksNodegroup-autoscaling-shutdown", "nodegroup-shutdown", "0 19 * * 1-5", eksNodeGroup.resources[0].autoscalingGroups[0].name, 0, 0, 0, stackResources)
//------------------------------------------------- APPLICATION LOAD BALANCER -------------------------------------------------------
const lb = new LoadBalancer();
const loadBalancerSG = lb.securityGroup("davide-manca-loadBalancer-SG", vpc.id, stackResources);
const targetGroup = lb.createTargetGroup("davide-manca-targetGroup", vpc.id, 31803, "HTTP", "instance", stackResources)
const loadBalancer = lb.createLoadBalancer("davide-manca-lb", 80, "HTTP", loadBalancerSG.id, firstPublicSubnet.id, secondPublicSubnet.id, stackResources)
const listener = lb.createListener("davide-manca-listener", targetGroup.arn, loadBalancer.arn, 80, "HTTP", stackResources)
const asgTargetGroupAttachment = lb.createAttachment("davide-manca-targetgroupattach", eksNodeGroup.resources[0].autoscalingGroups[0].name, targetGroup.arn, stackResources)
```

Fig 23: Calls to EKS and LB module functions in index.ts

The figure above instead shows a little extract of the *index.ts* file in which all the functions of each module previously explained have been called by passing explicitly the proper parameters. The entire AWS project has been pushed on the github repository **https://github.com/dev9815/pulumiAwsTemplateThesis/**.

As regards the deployment time execution it has not been very fast but better than Azure because it has been registered 14 minutes.

5.5 Code implementation for Azure provisioning

In a similar manner with respect to AWS, the procedures for performing the operations on the Azure cloud have been quite the same but with different configurations. The first step has been to authenticate to the cloud platform by logging in. In this case the passage has been different because there had not been keys nor secret keys but just the login into the Azure portal has been necessary.

So first of all the Azure command line was to be installed and in the Windows command line, **az login** has been the first command performed which has allowed to do the login to the portal.

After that a new typescript project has been created and all the modules related to the designed Azure infrastructure architecture have been implemented.

So as before let's introduce each module by describing all of the implemented functions. As in the AWS case, even here each module is composed of a typescript class related to the part of the cloud infrastructure to provision and which contains the wrapper functions whose bodies are calls to Pulumi APIs in order to perform the provisioning of resources. Each function is in charge to provision a resource. The project has been pushed on this github repository: https://github.com/dev9815/pulumiAzureTemplateThesis/

5.5.1 Networking module

This part is related to the creation of the virtual network and its related route tables and subnets. Here is also present the creation of the resource group. Each resource of the cloud infrastructure in Azure belongs to a resource group. The file in question is *networking.ts* which contains a typescript class (Networking) that involves several functions in charge to provide the needed resources, the following ones:

-
  ```
  public createResourceGroup (
          name: string,
          resources: any[]

  )
  ```

  This function is in charge to call the `azure-native.resources.ResourceGroup` API which creates a resource group by the passed name. The resource group is a list containing cloud resources; for this project two instances were needed and the motivations are explained in the AKS module.

-
  ```
  public createVirtualNetwork (
        name: string,
        resourceGroupName: pulumi.Output<string>,
        addressPrefix: string, dnsServer: string,
        resources: any[]
  )
  ```

  The virtual network is the equivalent of AWS VPC so it handles and contains all of the resources in terms of networking connections. It has been assigned its address prefix and the resource group it belongs to. The called API is azure-native.network.VirtualNetwork**.**

-
  ```
  public createPublicIpAddress(
        name: string,
        resourceGroupName: pulumi.Output<string>,
        location: pulumi.Output<string>,
        addressVersion: string,
        allocationMethod: string,
        skuName: string,
        resources: any[]
  )
  ```

  As the EIP of Amazon Web Services, this function allows to create an instance of a public IP address. The purpose is the same of before: this IP is given to the NAT gateway in order to allow the traffic from (what they are supposed to be) the private subnets to reach the internet. The Pulumi API used is azure-native.network.PublicIPAddress and the passed parameters are **the name, the resource group reference**, **the region**, **the address version**, **the allocation method**, **the sku** which refers to the Azure type of the object in question, and (as for AWS) a **list of resources**.

-
  ```
  public createNatGateway (name: string,
          resourceGroupName: pulumi.Output<string>,
          location: pulumi.Output<string>,
          idIP: pulumi.Output<string>,
          skuName: string, resources: any[]
  )
  ```

  The azure-native.network.NatGateway API performs the creation of the NAT gateway. The parameters needed are **the resource name**, **the resource group reference**, **the region**, **the id of the public IP address** of the previous function, **the sku** (**S**tock **K**eeping **U**nit) of the object, **the list of resources**.

-
  ```
  public createRouteTable(name: string,
          resourceGroupName: pulumi.Output<string>,
          location: pulumi.Output<string>,resources: any[]
  )
  ```

  `azure.network.RouteTable` Pulumi API enables to create the route tables needed for routing the traffic from the outside towards the hosted application and viceversa. The name, the reference to the resource group route tables belong to, the region and the list of resources are the parameter needed to pass the Pulumi API (except the last one). A public and a private route tables have been created.

-
  ```
  public createRoute (name: string,
          resourceGroupName: pulumi.Output<string>,
          routeTableName: pulumi.Output<string>,
          addressPrefix: string, nextHopType: string,
          idIP: pulumi.Output<string>, resources: any[]
  )
  ```

  A route table is made out of routes so this function provides the needed routes for forwarding the traffic inside and outside the cloud infrastructure subnets. The Pulumi API used is **azure.network.Route** where the most important parameters to consider are *addressPrefix*, *nextHopType* and *idIP* because the first one indicates the destination to which the route applies, the second one is related to the type of Azure hop the traffic should be sent to, and the third one is the IP address the traffic should be forwarded to. In order to use the latter parameter the next hop type value has to be *VirtualAppliance*.

-
  ```
  public createSubnet(
        name: string,
        resourceGroupName: pulumi.Output<string>,
        vNetName: pulumi.Output<string>,
        addressPrefix: string,
        idRouteTable: pulumi.Output<string>,
        location: pulumi.Output<string>,
        resources: any[]
  )
  ```

  azure-native.network.Subnet API creates subnets. The arguments are related to the subnet name, the referred virtual network, the address space, the id of the previously created route tables related to that specific subnet, the region, and a list of resources which assumes a role for another purpose. In the networking module file is also present *createNatSubnet()* function which acts equally with the difference of the addition of the NAT gateway.

5.5.2 Bastion Host module

The bastion host module in Azure has involved mainly three elements: the network security group to associate to the bastion host itself, the creation of the instance, and the autoscaling object to schedule its booting/shutdown. The functions have been implemented into *bastionHost.ts* typescript file and they are the following ones:

- `public createSecurityGroup (name: string, resourceGroupName: pulumi.Output<string>, location: pulumi.Output<string>, access: string, direction: string, protocol: string, nameRule: string, sourcePort: string, destinationPort: string, destinationAddressPrefix: string, priority: number, resources: any[])`

  As in AWS, the network security group is needed to filter the access to a specific set of IP addresses (which consists in some Liquid Reply public IP addresses). The resource group reference, the region (known as location), the direction (inbound/outbound traffic), the protocol, the source and destination ports, the *destinationAddressPrefix* which refers to the IP address space related to the instances which has to receive the traffic ("*" for indicating each resource), the priority and a list of resource have been the parameter used for this function responsible to call azure-native.network.NetworkSecurityGroup API.

- `public createVmScaleSet(name:string, resourceGroupName: pulumi.Output<string>, location: pulumi.Output<string>, skuName: string, adminUsername: string, disablePassword: boolean, username: string, publicKey: pulumi.Output<string>, caching: string, storageAccountType: string, nameNetInt: string, primary: boolean, nameIPConfig: string, offer: string, idSubnet: pulumi.Output<string>, namePublicIPAddress: string, idSecurityGroup: pulumi.Output<string>, publisher: string, skuImage: string, version: string, instances: number, firstZone: string, secondZone: string, resources: any[])`

  As the name of this function lets imagine, the resource to be provisioned here is the Virtual Machine Scale Set. In order to create the bastion host instance the steps have been to create firstly the security group, then the VMSS, and finally the autoscaling resource. Here each parameter about the virtual machine virtual profile are indicated such as the public key for the ssh connections, the id of the related subnet, the id of the security group, the used image, the OS, the deployment zone, the type of the instance (the cheapest one for this solution in order to reduce costs) the associated network interface and others. The Pulumi API involved is **azure.compute.LinuxVirtualMachineScaleSet**.

-
  ```
  public createAutoScaling(name: string,
          resourceGroupName: pulumi.Output<string>,
          location: pulumi.Output<string>,
          idVmScaleSet: pulumi.Output<string>,
          nameProfileBoot: string,
          defualtCapacityBoot: number,
          minCapacityBoot: number,
          maxCapacityBoot: number, days: string[],
          hoursBoot: number,minutes: number,
          nameProfileShutdown: string,
          defualtCapacityShutdown: number,
          minCapacityShutdown: number,
          maxCapacityShutdown: number,
          hoursShutdown: number, timezone: string,
          resources: any[]
  )
  ```

  The bastion host instance has to be powered on per each working day by 9 AM to 7 PM so to achieve this behavior, as in AWS, an autoscale resource has been provisioned; in particular the most important parameter here are the number of instances desired and wanted (*minCapacity Booting/Shutdown* parameters) and the time indicated in terms of days, hours, minute so CRON format has not been involved by the **azure.monitoring.AutoscaleSetting**.

5.5.3 AKS module and Load Balancer module

Conversely to AWS, the kubernetes cluster in Azure is managed quite differently. First of all the cloud platform, at the creation of the resource, dedicates an entire resource group to the cluster and to all of its relative resources; more precisely a new resource group related to the AKS, whose name is structured as **MC_resourcegroupname_clustername_location**, gets created including other resources such as internal subnet in which the nodes resides in, a load balancer, the Virtual Machine Scale Set which contains the VMs related to the worker nodes and other objects. So the worker nodes are represented by a nodepool (the equivalent AWS nodegroup) and the managedCluster API in charge to provision the cluster already provides the creation of a nodepool, related to the control plane, while the addition of other ones can be done but it requires separate commands to execute CRUD operation on an individual nodepool. The file of the AKS module is *aksCluster.ts*.

- `public createCluster (name: string, resourceGroupName: pulumi.Output<string>, location: pulumi.Output<string>, dnsPrefix: string, privateCluster: boolean, serviceCidr: string, dnsServiceIP: string, adminUsername: string, publicKey: pulumi.Output<string>, version: string, nodeResourceGroupName: string, enableRBAC: boolean, nameNodes: string, firstZone: string, secondZone: string, autoscaling: boolean, count: number, maxCount: number, minCount: number, mode: string, vmSize: string, type: string, osType: string, vmScaleSet: LinuxVirtualMachineScaleSet, bastionHostSG: NetworkSecurityGroup, autoscalingBH: AutoscaleSetting, resources: any[])`

  The relevant parameters to consider are definitely the **serviceCidr** for the

  address space related to the internal microservices, the public key to allow SSH connection to the worker nodes, the zones of deployment, allowing the autoscale of nodes, the min and max number of instances, the security group to filter accesses and the VM scale set to indicate how the worker nodes hardware profile should be (linux VMs). The Pulumi API used is **azure-native.containerservice.ManagedCluster**

-
  ```
  public createAutoScalingNodePool(
          name: string, type: string,
          resourceGroupName: string, location: string,
          nameProfileBoot: string,
          defualtCapacityBoot: number,
          minCapacityBoot: number,
          maxCapacityBoot: number, days: string[],
          hoursBoot: number, minutes: number,
          nameProfileShutdown: string,
          defualtCapacityShutdown: number,
          minCapacityShutdown: number,
          maxCapacityShutdown: number,
          hoursShutdown: number, timezone: string,
          resources: any[]
  )
  ```

  The nodes are managed by means of autoscaling so they are powered up and down like the bastion host follow the used coherence. The Pulumi API used is **azure.monitoring.AutoscaleSetting**.

So not even a new nodepool (the equivalent AWS nodegroup) has been necessary because the effective creation of the AKS cluster provides by default an agent pool related to the control plane. So for convenience, by considering to remain into an academic study, application are deployed inside the control plane of the kubernetes cluster.

Moreover, thanks to mangedCluster APIs, the cluster is easier to manage rather than AWS and the relative resources are already associated and handled within the same resource group by means of Azure itself. In the other side, managing the AWS cluster and the nodegroups requires more attention in involving all the needed resources having so a less automated process but it is helpful to understand how the cloud resources are/have to be linked each other.

As regards the load balancer, the only resource to be created has been an additional rule in order to distinguish where the traffic has to be redirect. So the only function the *loadbalancer.ts* module contains is:

-
  ```
  public createRule(
          name: string, resourceGroupName: string,
          loadBalancerName: string, frontendPort: number,
          backendPort: number, protocol: string,
          disableOutboundSnat: boolean, resources: any[]
  )
  ```

  The frontend and backend ports as well as the protocol are the most relevant parameters passed to azure.lb.Rule API in order to manage the direction of the traffic towards the nodepool worker nodes

5.5.4 Database module

The relational database used for managing the backend of the application workloads inside the AKS cluster is Azure Database for PostgreSQL. In principle the concept is slightly different from AWS because instead of creating just a resource corresponding to the relational db instance, in Azure the workflow provide for creating a server (flexible server) which contains the db instance inside. **As the azure official website** reports, "*flexible Server is a fully managed database service designed to provide more granular control and flexibility over database management functions and configuration settings. The service generally provides more flexibility and server configuration customizations based on user requirements. The flexible server architecture allows users to collocate the database engine with the client tier for lower latency and choose high availability within a single availability zone and across multiple availability zones.*"

The functions library in question is *dbServer.ts* where the content is the following:

-
  ```
  public createFlexibleServer(name: string,
          resourceGroupName: string, location: string,
          passwordAuth: string,
          activeDirectoryAuth: string,
          createMode: string, adminLogin: string,
          adminPassword: string, skuName: string,
          skuTier: string, version: string,
          storageSize: number, cluster: ManagedCluster,
          resources: any[]
  )
  ```

  The tasks of this function is to provision the flexible server on which then creating the postgresql db instance. So the login credentials, the stock keeping unit informations related to the model of the db, the storage size are the most relevant parameter passed to the called API **azure-native.dbforpostgresql.Server**. The managed cluster has nothing to do with the server provisioning, it has been given just for a resource provisioning purpose that is to establish a resource deployment order: in this case the server has been created just after the AKS cluster deployment.

-
  ```
  public createDB(
          name: string, resourceGroupName: string,
          serverName: pulumi.Output<string>,
          charset: string, collation: string,
          cluster: ManagedCluster, resources: any[]
  )
  ```

  The db gets created inside the flexible server thanks to azure-native.dbforpostgresql.Database API.

-
  ```
  public createFirewallRule(
        name: string, resourceGroupName: string,
        serverName: pulumi.Output<string>,
        startIpAddress: string, endIpAddress: string,
        cluster: ManagedCluster, resources: any[]
  )
  ```

  The flexible server needs a rule into its firewall in order to filter the access from and to the virtual network; more precisely only traffic that comes from the AKS virtual network address space is accepted because, as mentioned in the 5.4.3 chapter, the db has been pushed into the same virtual network as the cluster since it should manage the application backend and persistence. So because of Azure manages AKS cluster by dedicating a completely isolated resource group, as well as the virtual network, the needed IP addresses to be passed to the called azure-native.dbforpostgresql.FirewallRule are the ones of the related virtual network, for filtering the incoming/outcoming traffic.

5.5.5 Azure Container Registry module

Such as the AWS ECR resource, the *registry.ts* file has been implemented for basically creating a registry from/to which pulling/pushing images that can be used by the worker nodes, actually by the microservices based on the deployed application.

So the only function this library contains is the following:

-
  ```
  public createRegistry(
        name: string,
        resourceGroupName: pulumi.Output<string>,
        skuName: string, adminUser: boolean,
        location: pulumi.Output<string>,
        resources: any[]
  )
  ```

  azure-native.containerregistry.Registry is the Pulumi API called to provision the registry resource.

5.5.6 Azure Functions module

The module containing the library functions implemented is *azureFunctions.ts.* As Lambda Functions, Azure Function resource has been useful to manage the database booting and shutdown automation. Two separate functions have been implemented and for making them working other resources have been necessary to provision, the following ones:

-
  ```
  public createAppServicePlan(
          name: string, resourceGroupName: string,
          location: string, skuName: string,
          skuTier: string, kind: string, resources: any[]
  )
  ```

  As the Azure official documentation claims, "*An App Service plan defines a set of compute resources for a web app to run. When you create an App Service plan in a certain region (for example, West Europe), a set of compute resources is created for that plan in that region. Whatever apps you put into this App Service plan run on these compute resources as defined by your App Service plan. Each App Service plan defines Operating System (Windows, Linux), Region, Number of VM instances, Size of VM instances, Pricing tier*".

  azure-native.web.AppServicePlan is the Pulumi API used.

-
  ```
  public createApplication(name: string,
  currentClient:Promise<azuread.GetClientConfigResult>
  , description: string, resources: any[])
  ```

  "***An Azure AD Application is a digital identity and some associated configuration which informs Azure AD about how to treat software which uses that digital identity. The software itself may be running anywhere, on any platform, written in any language*** "

  The Pulumi API used is azuread.Application

- `public createServicePrincipal (name: string, applicationId: pulumi.Output<string>, description: string, accountEnabled: boolean, currentClient:Promise<azuread.GetClientConfigResult>, resources: any[])`

  "***An Azure service principal is an identity created for use with applications, hosted services, and automated tools to access Azure resources. This access is restricted by the roles assigned to the service principal, giving you control over which resources can be accessed and at which level.***" This resource and the previous one, coupling with the next one have been necessary to manage the control of the azure function resources. These are preparatory steps. The API used is `azuread.ServicePrincipal`

-
  ```
  public createApplicationPassword(name: string,
       appObjId: pulumi.Output<string>,
       endDateRelative: string, resources: any[]
  )
  ```

  Function related to the client secret. "*A client secret is an authentication technique that uses a string value in the Azure application instead of a certificate for identity. It is also called an application password that is used for the authentication of tokens for Azure app access. The Azure AD issues a token to access the desired resource upon successful verification of the client secret.*". So the Pulumi API used is **azuread.ApplicationPassword**.

-
  ```
  public createStorageAccount(name: string,
       resourceGroupName: string, location: string,
       accountName: string, resources: any[]
  )
  ```

  The storage account is an essential component for Azure Functions because it provides data storage, triggers, state management, and other features that are critical for building serverless applications and processing data in a scalable and reliable manner within the Azure cloud environment. The parameters passed are the resource name, resource group reference, location, name of the storage account, and a list of resources used in a different moment. The used Pulumi API is **a**zure-native.storage.StorageAccount**.**

-
  ```
  public createBlobContainer(
          name: string,
          resourceGroupName: string,
          account: StorageAccount,
          resources: any[]
  )
  ```

  createBlobContainer() is responsible for provision a blob container. It is designed for storing unstructured data, such as images, and more. This makes it suitable for storing a wide range of data that Azure Functions might need to process. Moreover Azure Functions often require input data to perform their tasks so the creation of the blob container can be used to store input data that triggers a function's execution. The Pulumi API used is azure-native.storage.BlobContainer**.** The storage account on which operating is passed on the parameter as a reference

-
  ```
  public createBlob (
          name: string,
          resourceGroupName: string,
          containerName: pulumi.Output<string>,
          source: pulumi.asset.Asset,
          account: StorageAccount,
          resources: any[]
  )
  ```

  This object is needed by the Azure Functions for various purposes: it can store data, process data, and serve as a trigger. It has been used because it facilitates logging and audit trail management, while being cost-effective for data storage. In this instance it goes to retrieve the function code in which there is the task to perform (booting/shutdown), in fact the *source* parameter refers to the directory path in which there is the folder containing the function code ("./startDB"). The API used is azure-native.storage.Blob

-
  ```
  public createInsights(
          name: string,
          resourceGroupName: string,
          location: string,
          appType: string,
          resources: any[]
  )
  ```

  This resource allows to monitor your Azure Functions in real-time; it provides a comprehensive view of the application's performance, including metrics, traces, and logs. So in order to use it for diagnosing issues, tracking down errors, and understanding how the azure functions are behaving, azure.**appinsights.**Insights Pulumi API has been used.

-
  ```
  public createWebApp (
          name: string, resourceGroupName: string,
          location: string, farmId: pulumi.Output<string>,
          instrumentationKey: pulumi.Output<string>,
          applicationId: pulumi.Output<string>,
          clientSecretValue: pulumi.Output<string>,
          tenantId: pulumi.Output<string>,
          extensionVersion: string, kind: string,
          workerRuntime: string, nodeVersion: string,
          codeContainer: BlobContainer, codeBlob: Blob,
          account: StorageAccount, resources: any[]
  )
  ```

  The webApp is quite the equivalent of the AWS Lambda function resource. In particular it takes a set of parameters (*instrumentationKey, applicationId, clientSecret, tenantId, codeContainer, nodeVersion*) that are individuated as environment variable of the azure function. So in order to work those variable have to be set properly and it takes the references about the previously created resources about azuread application, client secret, version of node.js used for the code of the function, etc. It needs all of the previous resource to allow the function performing its task. The Pulumi API **azure-native.web.WebApp.**

-
  ```
  public createRoleAssignment ( name: string,
       principalId: pulumi.Output<string>,
       idRole: string, idSubscription: string,
       description: string, principalType: string,
       applicationClient: Application,
       resources: any[]
  )
  ```

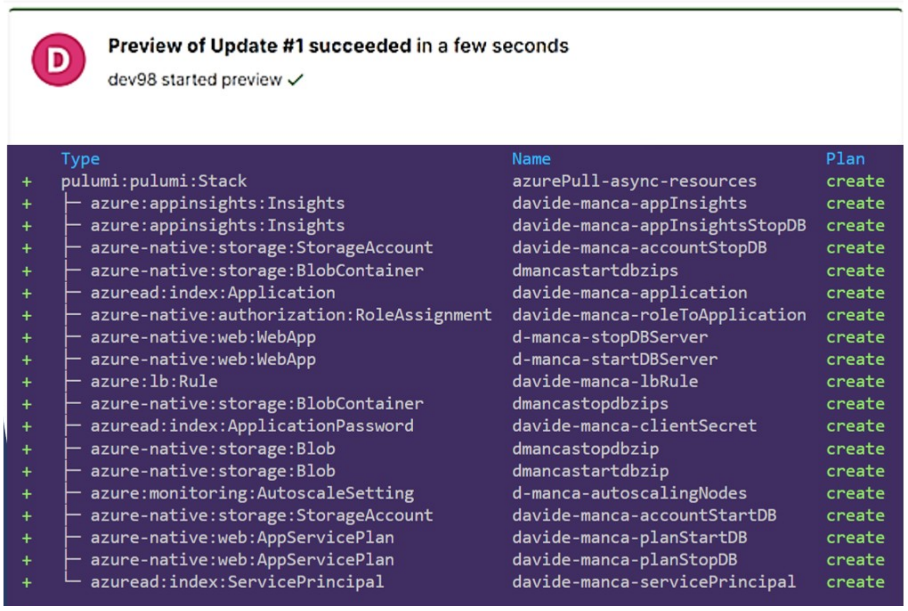
  In order to manage the Azure Functions, the proper permissions are needed. So since my IAM identity on Azure was limited in allowed actions, I have assigned the **Contributor** role in order to have permission to execute the function app. **azure-native.authorization.RoleAssignment** API is used.

-
  ```
  public createAzureFunction ( planName: string,
  resourceGroupName: string, location: string,
  planSkuName: string, planSkuTier: string, planKind:
  string, storageName: string, storageAccountName:
  string, blobContainerName: string, blobName: string,
  source: pulumi.asset.Asset, insightsName: string,
  appType: string, webAppName: string, webAppKind:
  string, applicationClientId: pulumi.Output<string>,
  clientSecretValue: pulumi.Output<string>, tenantId:
  pulumi.Output<string>, extensionVersion: string,
  workerRuntime: string, nodeVersion: string,
  resources: any[])
  ```

  In order to make just a call in the *index.ts* this function is the wrapper of the wrappers. It is in charge to call all of the previous described functions to actually spawn an Azure function and make it works. So two instances of this functions have been involved: for booting and for shooting. All the passed parameters are used to call all the previous needed functions. So this does not call any Pulumi API.

5.6 Results

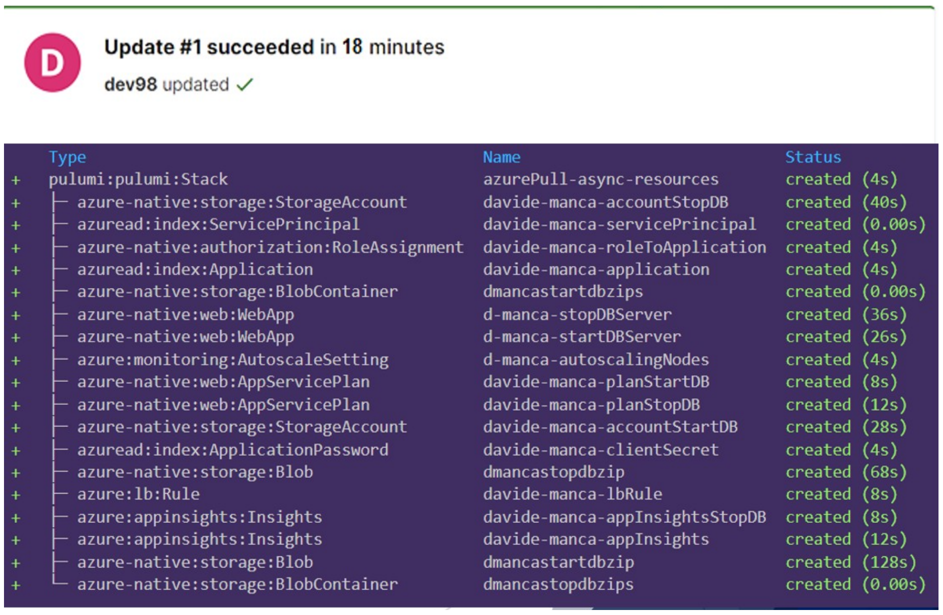
As it can be noticeable, Azure differs quite a lot from AWS with respect to Lambda and azure functions. While in a case it was enough just to make a call to the proper API where the code related to the task to perform (start/stop db) were specified as a parameter of the called API, in the other case a lot of auxiliary resources were needed to perform the creation and execution of the final function. As a matter of fact Azure in this perspective has required a lot of additional work which is definitely not necessary in AWS context. The code related to the db start/stop has not been specified as parameter of the API in charge to create the azure function but it has been separated into two distinct folders (StartDB, StopDB). Only by means of the Blob resource the typescript code of the booting/shutdown has been taken and "passed" to the WebApps resource in order to have a reference of where the azure functions code actually reside. Furthermore the scheduling of this processes at 7AM and 9PM per each function is a part related to the StartDB and StopDB folders; in fact both contains an *index.ts* file where the task is implemented and a JSON file containing the information about when to trigger the Azure functions in the CRON format (under the "schedule" field).



Preview of Update #1 succeeded in a few seconds
dev98 started preview ✓

```
    Type                                             Name                              Plan
+   pulumi:pulumi:Stack                              azurePull-async-resources         create
+   ├─ azure:appinsights:Insights                    davide-manca-appInsights          create
+   ├─ azure:appinsights:Insights                    davide-manca-appInsightsStopDB    create
+   ├─ azure-native:storage:StorageAccount           davide-manca-accountStopDB        create
+   ├─ azure-native:storage:BlobContainer            dmancastartdbzips                 create
+   ├─ azuread:index:Application                     davide-manca-application          create
+   ├─ azure-native:authorization:RoleAssignment     davide-manca-roleToApplication    create
+   ├─ azure-native:web:WebApp                       d-manca-stopDBServer              create
+   ├─ azure-native:web:WebApp                       d-manca-startDBServer             create
+   ├─ azure:lb:Rule                                 davide-manca-lbRule               create
+   ├─ azure-native:storage:BlobContainer            dmancastopdbzips                  create
+   ├─ azuread:index:ApplicationPassword             davide-manca-clientSecret         create
+   ├─ azure-native:storage:Blob                     dmancastopdbzip                   create
+   ├─ azure-native:storage:Blob                     dmancastartdbzip                  create
+   ├─ azure:monitoring:AutoscaleSetting             d-manca-autoscalingNodes          create
+   ├─ azure-native:storage:StorageAccount           davide-manca-accountStartDB       create
+   ├─ azure-native:web:AppServicePlan               davide-manca-planStartDB          create
+   ├─ azure-native:web:AppServicePlan               davide-manca-planStopDB           create
+   └─ azuread:index:ServicePrincipal                davide-manca-servicePrincipal     create
```

Fig 24: Pulumi resources preview for Azure provisioning

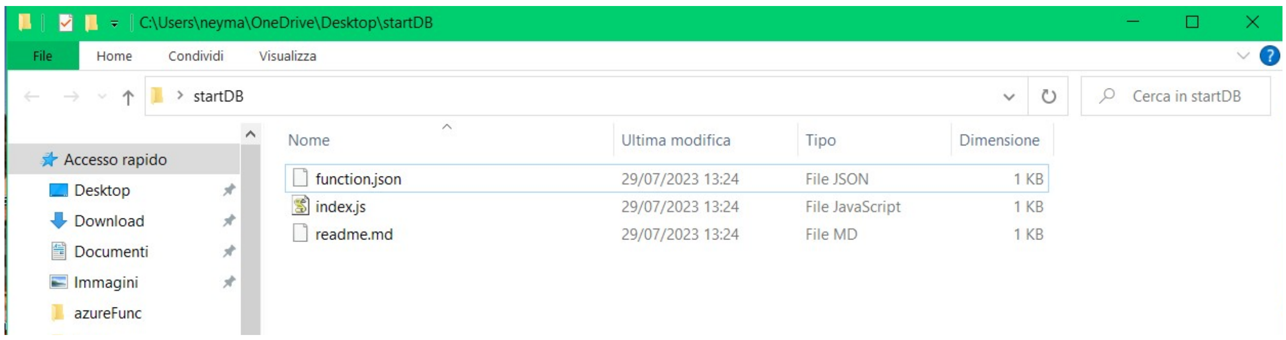


Update #1 succeeded in 18 minutes

dev98 updated ✓

```
   Type                                        Name                           Status
+  pulumi:pulumi:Stack                         azurePull-async-resources      created (4s)
+  ├─ azure-native:storage:StorageAccount      davide-manca-accountStopDB     created (40s)
+  ├─ azuread:index:ServicePrincipal           davide-manca-servicePrincipal  created (0.00s)
+  ├─ azure-native:authorization:RoleAssignment davide-manca-roleToApplication created (4s)
+  ├─ azuread:index:Application                davide-manca-application       created (4s)
+  ├─ azure-native:storage:BlobContainer       dmancastartdbzips              created (0.00s)
+  ├─ azure-native:web:WebApp                  d-manca-stopDBServer           created (36s)
+  ├─ azure-native:web:WebApp                  d-manca-startDBServer          created (26s)
+  ├─ azure:monitoring:AutoscaleSetting        d-manca-autoscalingNodes       created (4s)
+  ├─ azure-native:web:AppServicePlan          davide-manca-planStartDB       created (8s)
+  ├─ azure-native:web:AppServicePlan          davide-manca-planStopDB        created (12s)
+  ├─ azure-native:storage:StorageAccount      davide-manca-accountStartDB    created (28s)
+  ├─ azuread:index:ApplicationPassword        davide-manca-clientSecret      created (4s)
+  ├─ azure-native:storage:Blob                dmancastopdbzip                created (68s)
+  ├─ azure:lb:Rule                            davide-manca-lbRule            created (8s)
+  ├─ azure:appinsights:Insights               davide-manca-appInsightsStopDB created (8s)
+  ├─ azure:appinsights:Insights               davide-manca-appInsights       created (12s)
+  ├─ azure-native:storage:Blob                dmancastartdbzip               created (128s)
+  └─ azure-native:storage:BlobContainer       dmancastopdbzips               created (0.00s)
```

Fig 25: Deployment of Azure resources (pulumi up --yes)



C:\Users\neyma\OneDrive\Desktop\startDB

Nome	Ultima modifica	Tipo	Dimensione
function.json	29/07/2023 13:24	File JSON	1 KB
index.js	29/07/2023 13:24	File JavaScript	1 KB
readme.md	29/07/2023 13:24	File MD	1 KB

Fig 26: Content of StartDB folder related to the Azure Function to start the db at 7AM

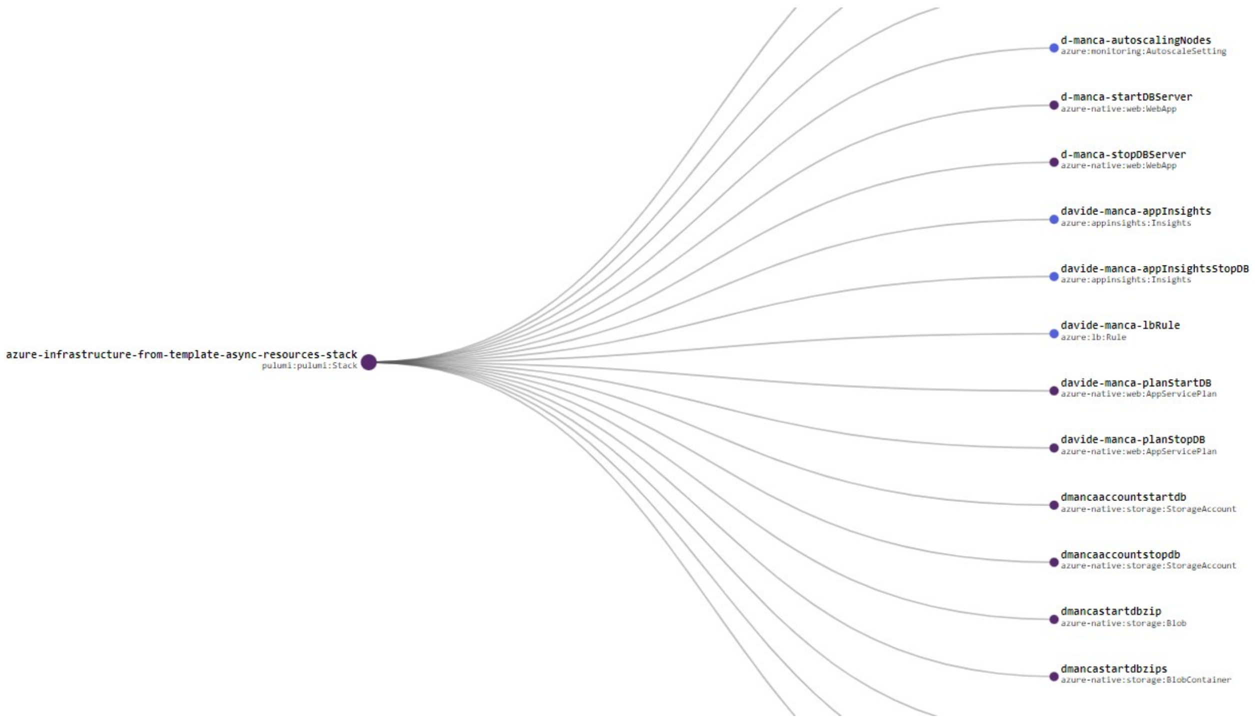
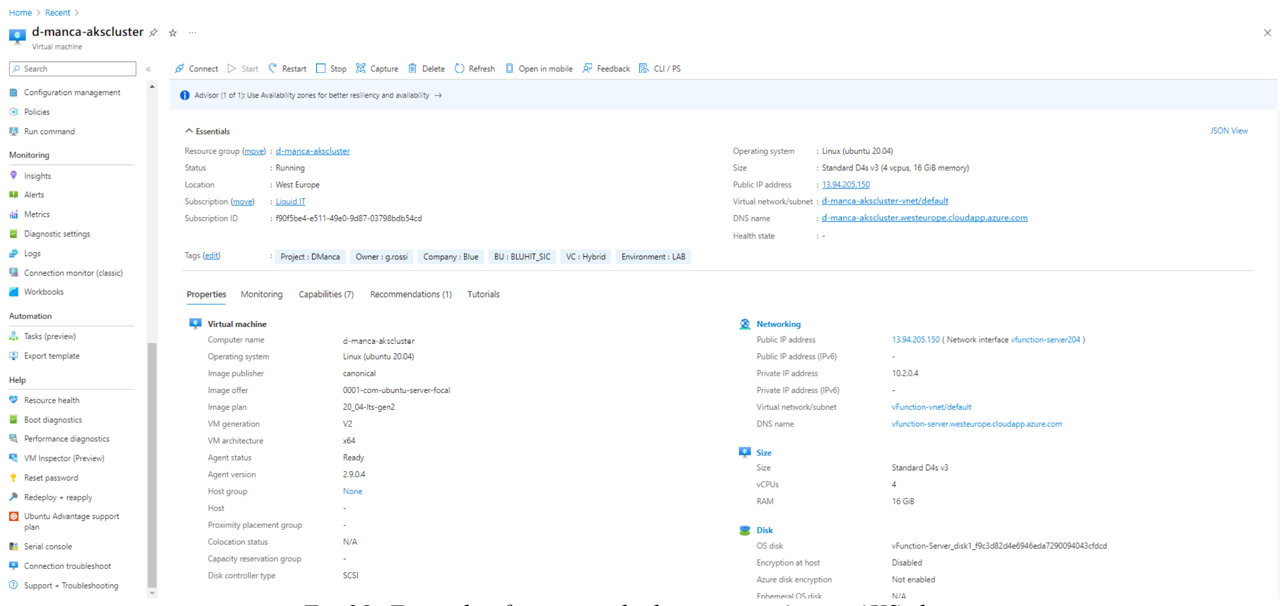

azure-infrastructure-from-template-async-resources-stack
pulumi:pulumi:Stack
d-manca-autoscalingNodes
azure:monitoring:AutoscaleSetting
d-manca-startDBServer
azure-native:web:WebApp
d-manca-stopDBServer
azure-native:web:WebApp
davide-manca-appInsights
azure:appinsights:Insights
davide-manca-appInsightsStopDB
azure:appinsights:Insights
davide-manca-lbRule
azure:lb:Rule
davide-manca-planStartDB
azure-native:web:AppServicePlan
davide-manca-planStopDB
azure-native:web:AppServicePlan
dmancaaccountstartdb
azure-native:storage:StorageAccount
dmancaaccountstopdb
azure-native:storage:StorageAccount
dmancastartdbzip
azure-native:storage:Blob
dmancastartdbzips
azure-native:storage:BlobContainer

Fig 27: Pulumi Cloud resource graph view for Azure



d-manca-akscluster
Virtual machine

Advisor (1 of 1): Use Availability zones for better resiliency and availability →

Essentials

Resource group (move)	: d-manca-akscluster
Status	: Running
Location	: West Europe
Subscription (move)	: Liquid IT
Subscription ID	: f90f5be4-e511-49e0-9d87-03798bdb54cd
Operating system	: Linux (ubuntu 20.04)
Size	: Standard D4s v3 (4 vcpus, 16 GiB memory)
Public IP address	: 13.94.205.150
Virtual network/subnet	: d-manca-akscluster-vnet/default
DNS name	: d-manca-akscluster.westeurope.cloudapp.azure.com
Health state	: -

Tags (edit) : Project : DManca | Owner : g.rossi | Company : Blue | BU : BLUHIT_SIC | VC : Hybrid | Environment : LAB

Virtual machine

Computer name	d-manca-akscluster
Operating system	Linux (ubuntu 20.04)
Image publisher	canonical
Image offer	0001-com-ubuntu-server-focal
Image plan	20_04-lts-gen2
VM generation	V2
VM architecture	x64
Agent status	Ready
Agent version	2.9.0.4
Host group	None
Host	-
Proximity placement group	-
Colocation status	N/A
Capacity reservation group	-
Disk controller type	SCSI

Networking

Public IP address	13.94.205.150 (Network interface vfunction-server204)
Public IP address (IPv6)	-
Private IP address	10.2.0.4
Private IP address (IPv6)	-
Virtual network/subnet	vFunction-vnet/default
DNS name	vfunction-server.westeurope.cloudapp.azure.com

Size

Size	Standard D4s v3
vCPUs	4
RAM	16 GiB

Disk

OS disk	vFunction-Server_disk1_f9c3d82d4e6946eda7290094043cfdcd
Encryption at host	Disabled
Azure disk encryption	Not enabled
Ephemeral OS disk	N/A

Fig 28: Example of resource deployment on Azure: AKS cluster

The figure 27 obviously shows just a part of the resources deployed on Azure but it is clear how all of them depends on the stack in which they have been created.

Each resource, in both the cloud providers, have been flanked with tags. These are descriptive labels used to understand, within the Liquid Reply context, which resources were created by me and how much did I use them in order to calculate the cost basing on the pay-per-use model.

As regards the deployment time execution it is not so fast because the amount of resources has been quite consistent so for both the cloud providers it requires a handful of minutes but for Azure it was 18 minutes, so worse than AWS due surely also to the presence of two stacks instead of one.

The entire Azure project has been pushed on the github repository **https://github.com/dev9815/pulumiAzureTemplateThesis/**.

5.7 Considerations and implementation differences

In addition to the previous consideration, there are some other observations to notice. Both the cloud providers present advantages and drawbacks, especially related under a Pulumi coding perspective. If Azure might be more comfortable than AWS to use because it manages and aggregates resources by itself handling the various dependencies between objects (such as AKS), it is not negligible to underline that sometimes it requires a lot of additional operations to make a resource working such as specifying more detailed parameters or even creating additional resources (not needed in Amazon).

AWS may require some additional explicit configuration, may not represent the resources efficiently in order to have a clear view such as Azure does but the management of resources and what they are meant to do is more clear and immediate. Moreover it does not require for some resources additional object creations, like for the Lambda function case.

Under a coding perspective, instead, the main difference is related to how Pulumi APIs are called from AWS and Azure. In order to provide a flexible as possible solution the created library functions receive parameters that are passed to the proper APIs so that the function can be used with whatever values. The main limitation affects resources deployment synchronization. Let's make an example to have a better comprehension.

When the kubernetes cluster is created, the next associated resource to it is the worker nodes autoscaling in order to manage the nodes scalability; so there is this kind of dependency where the cluster id has to be passed to the autoscaling resource API in order to have the reference to operate on the cluster itself. Although the Pulumi APIs for AWS provide a set of outputs, related to the parameters of the created resource, which can be passed to other calls that need them (e.g. autoscaler), some APIs for Azure do not support this mechanism. So the dependency of two objects is difficult to maintain because no set of useful outputs are provided to pass to another object in charge of managing the first one. This problem can be solved by directly passing the exact value of the needed parameter to the object that has the dependency, then the actual cluster ID to the autoscaler to use the same example, but it is not flexible at all because it would only work in that circumstance. For this reason I had to use the getters APIs to retrieve the values in a generic way, but here comes the limitation: get() Pulumi APIs are asynchronous so during the execution they are executed first and this clearly represent an issue because the autoscaler will operate on a cluster that has not been created yet since the cluster id would be retrieved by means of a get() as well.

The most effective solution would be to have needed parameters as outputs of each function so they can be used without performing any get() and having a clear and flexible solution, just like AWS does, but to overcome to that problem in Azure it has been necessary to create two separate stacks: one for "synchronous" resources (without dependencies or where dependency was allowed) and one for "asynchronous" resources. With this solution a stack at a time gets deployed so the asynchronous resources are deployed after the deployment of the synchronous stack.

Of course this mechanism slows a little bit the deployment process but it has been the proper solution to adopt. AWS, by providing a set of very useful parameters as output of each resource, does not present that problem and just a stack was needed. So the process has been quicker and more efficient.

Another difference to underline was the management of serverless functions: Lambda and Azure functions. In particular, the APIs for AWS allow you to incorporate within the API itself the code relating to the tasks to be performed, starting and stopping the database in a temporal recurrence, while for Azure the process is more complex because its API simply creates a Azure Functions instance containing a list of environment variables required to execute the function itself, but does not actually allow you to write the code within the API itself or explicitly. You need to create a Visual Studio Code project for your Azure Function in a separate folder that contains the function's specific code file, a JSON for time triggering, and a README file. In fact two separate folders, StartDB and StopDB, has been created.

In AWS, time scheduling is done via a specific API.

In light of all these considerations, the Pulumi experience with the AWS platform proved to be the most effective.

5.8 Disaster Recovery strategy

A cloud infrastructure model which is meant to be resilient and reliable has to consider the adoption of a disaster recovery plan. This is highly needed, especially in enterprise production, whenever failures, natural disasters, cyberattacks occurred because they are the cause of downtimes, in the best case, and of infrastructure loss in the worst. So disaster recovery provides for implementing a strategy which is able to retrieve the resources on the cloud in order to deploy them into another available region. The deployment should be as fast as possible and functioning because it must reduce the possible downtime and it must avoid disruption to the services for customers.

By an infrastructure as code perspective what it would be useful to have is a kind of cloud-independent mechanism or API which is able to retrieve all of the resources related to a cloud infrastructure, basing on some parameters in common such as tags, so that they could be deployed into another region, in a second moment. This operation obviously needs some basic characteristics like redundancy and high-availability.

By trying to use Pulumi for this purpose, this behavior has not been totally achievable because the tool presents some limitations in terms of API. Resources inside a stack are described with a specific label, known as URN (Uniform Resource Name) but unfortunately Pulumi does not provide an API able to retrieve resources by a common parameter such as tags; some API allows to get resources by common values but it makes the programmer forced to know also other specific parameters which a cloud engineer who use this project as a plug and play solution could even not know. Since the post-deployment backup cannot be performed, the proposed solution hence aims to leverage the importing property of Pulumi to import cloud resources into a different Pulumi stack and deploying them into another region.

The procedure, therefore, has been to save some useful information of the resources (URN, name, id) into a JSON file so that this could have been imported into another stack by means of the **pulumi import** command. In particular this JSON contains a list of items where each describes a specific resource URN, name and id. In the libraries implementation this has resulted in saving those three parameters during the resources deployment into a list named **resources[]**, the parameter indicated previously in each module library function. By calling **infrastructureToJson()** (of the *jsonForImport.ts* module) in the index.ts file, the resources contained into that list of resources are saved into a created JSON file named *resources.json* in terms of URN, name, id per each object.

The next step so regards the importing into another stack completely dedicated to the import process where the **pulumi import --file resources.json** is performed and the final result is the creation of an auto-generated typescript file containing the IaC code related to each resource described in the JSON file *resources.json* passed to the import command as a parameter.

Finally a bash script modifies this file to set the region in which the deployment is ready to be performed.

5.9 Template and automation scripts

This cloud infrastructure model provided to AWS and to Azure cloud platforms is addressed to organizations which decide to adopt it for deploying their proper applications; so in order to facilitate the processes execution and to create a plug-and-play solution, automation scripts and templates have been necessary.

Templates in Pulumi consists in creating a solution which takes a set of initial parameters for initializing the cloud infrastructure to build. In the created application there is just a command line interaction by the user perspective which consists in providing some data related to the infrastructure such as **the name of the project and its description**, **the region in which performing the deployment**, **the name of the virtual network and its address space**, **the name and the address space of the subnets.** These parameters resides in the *Pulumi.yaml* file and they are saved into variables already set and ready to be used inside the typescript code. Furthermore they can be also extended or reduced according to how much flexibility is requested by who is going to use this solution.

Templatizing a project means to leverage the potentialities of Pulumi concerning the templates concept (chapter 4.7) for making the project a plug-and-play application, so reusable in many different instances. Two templates have been provided: for AWS and for Azure.

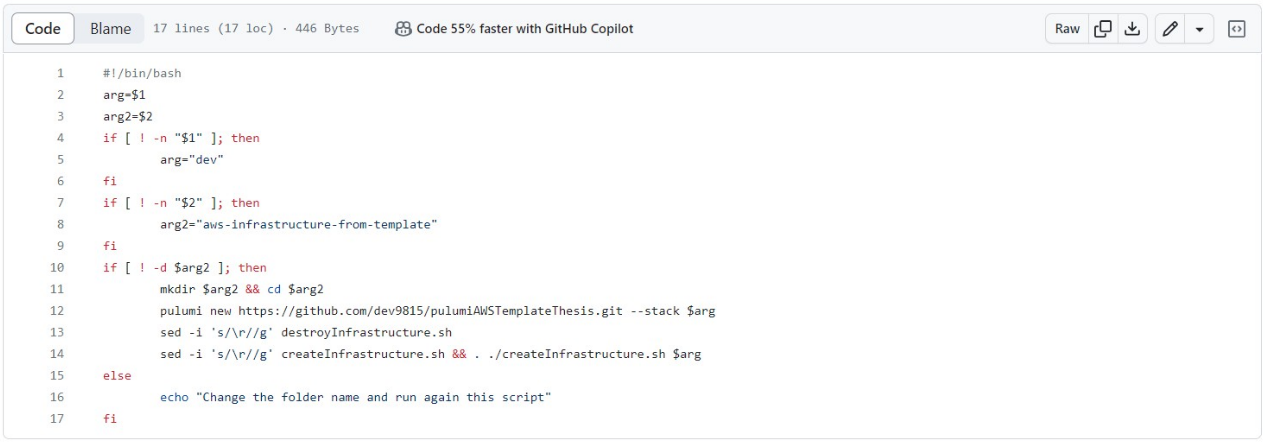
In addition to this aspect, automation scripts have been the other important instrument used to facilitate the processes. It is important to distinguish the structure of the project and automation scripts created. Since the templates have been published in two different github repositories each per cloud provider, the repository content is a project folder which includes a bash script, called *deployTemplate.sh*, and a folder (*aws-thesis* or *azure-thesis*) containing the cloud infrastructure to launch. Inside of this one, other three bash scripts have been involved: *createInfrastructure.sh*, *destroyInfrastructure.sh*, *importInfrastructure.sh*.

In order, *deployTemplate.sh* is the outer script to be called just once, responsible to launch the template and to perform all the cloud infrastructure creation operations, such as acquiring the template parameters, including the resources provisioning so it reproduce the plug-and-play behavior by just launching it. This script, internally, calls *createInfrastructure.sh* for the actual creation and deployment.

Successively *createInfrastructure.sh* is called by the outer script, as mentioned, or it can even be called separately; however this implies not to use *deployTemplate.sh* but to create the project locally by means of the **pulumi new https://github.com/dev9815/pulumi[$Cloud]TemplateThesis.git** (where [$Cloud] stands for "AWS" or "Azure") and only then calling the *createInfrastructure* script.

The script *destroyInfrastructure*, instead, is intended for destroying the cloud resources from the cloud platform so it has to be called for that purpose and it is in charge to remove the resources also from the Pulumi stacks.

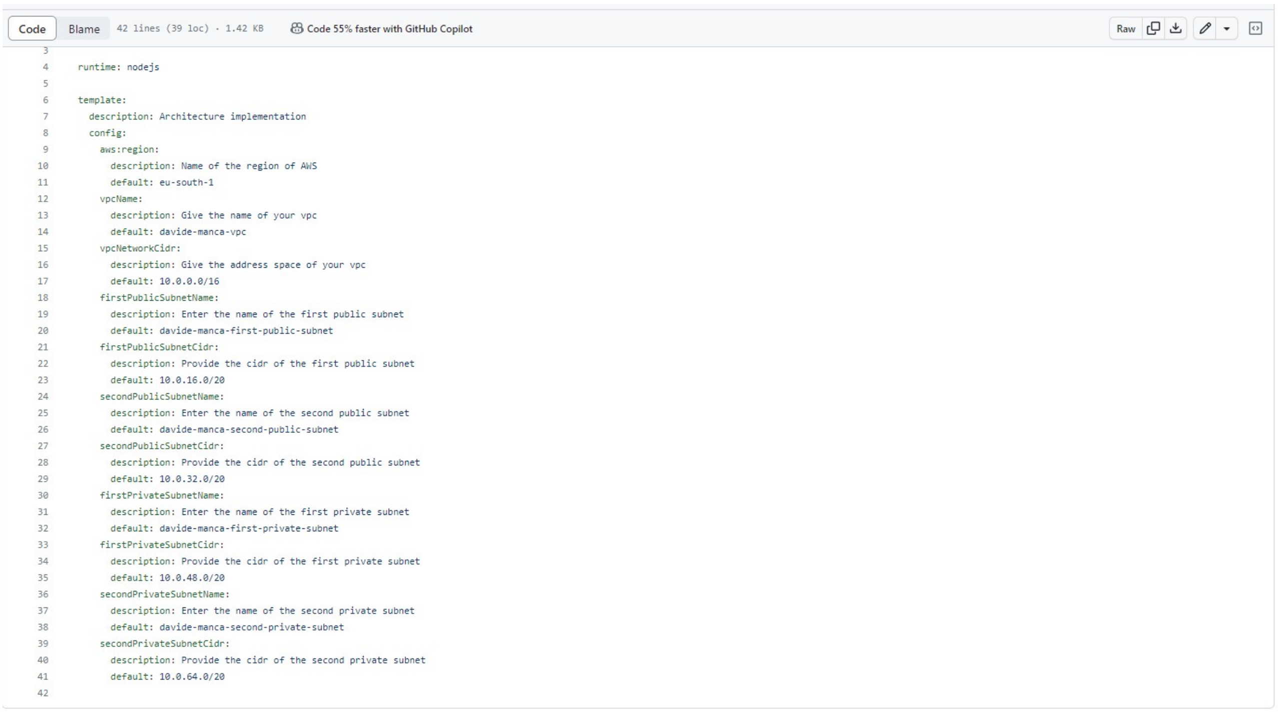
The script *importInfrastructure*, finally, is in charge to create a new Pulumi stack for importing the resources describe by the *resources.json* file created at the deployment phase.


Code
Blame
17 lines (17 loc) · 446 Bytes
Code 55% faster with GitHub Copilot
Raw

```bash
#!/bin/bash
arg=$1
arg2=$2
if [ ! -n "$1" ]; then
        arg="dev"
fi
if [ ! -n "$2" ]; then
        arg2="aws-infrastructure-from-template"
fi
if [ ! -d $arg2 ]; then
        mkdir $arg2 && cd $arg2
        pulumi new https://github.com/dev9815/pulumiAWSTemplateThesis.git --stack $arg
        sed -i 's/\r//g' destroyInfrastructure.sh
        sed -i 's/\r//g' createInfrastructure.sh && . ./createInfrastructure.sh $arg
else
        echo "Change the folder name and run again this script"
fi
```

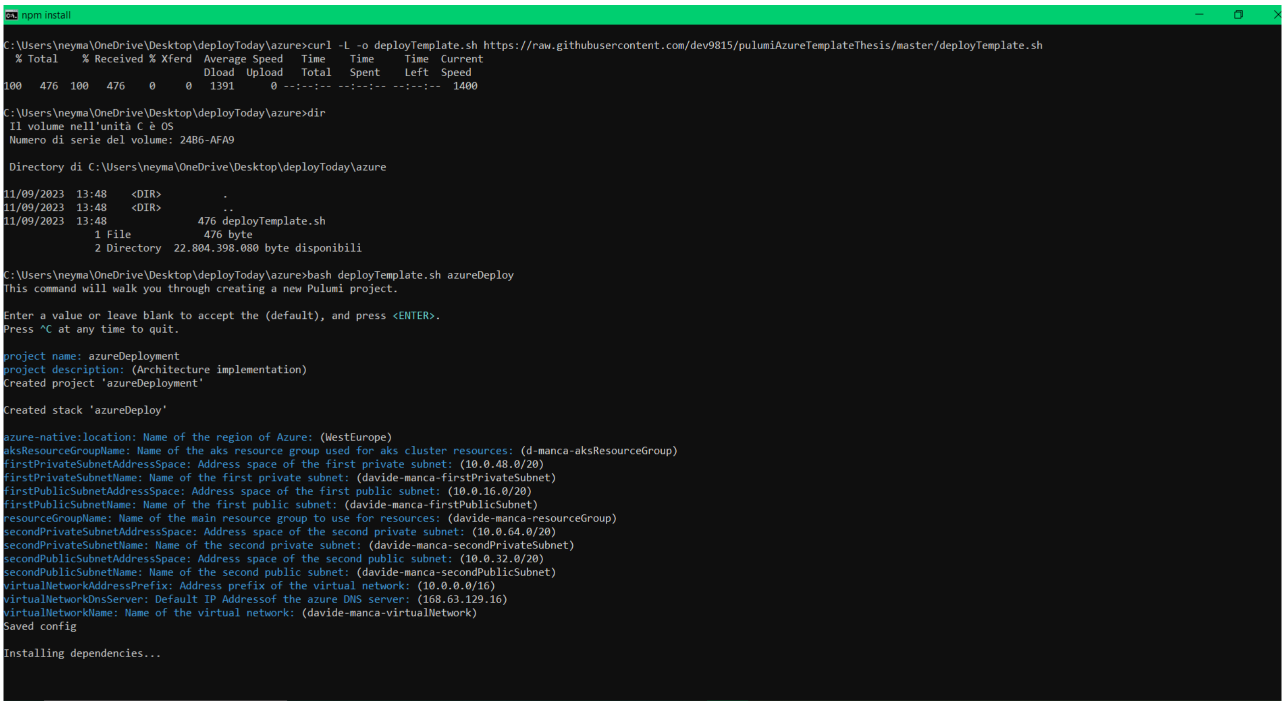
Fig 29: Example of a bash automation script: deployInfrastructure.sh

At the end of this script a new folder named *infrastructure* is created which contains the auto-generated infrastructure as code typescript file related to the imported resources and the last task performed is to change the location in order to be ready to do the deployment into another region.


Code
Blame
42 lines (39 loc) · 1.42 KB
Code 55% faster with GitHub Copilot
Raw

```yaml

runtime: nodejs

template:
  description: Architecture implementation
  config:
    aws:region:
      description: Name of the region of AWS
      default: eu-south-1
    vpcName:
      description: Give the name of your vpc
      default: davide-manca-vpc
    vpcNetworkCidr:
      description: Give the address space of your vpc
      default: 10.0.0.0/16
    firstPublicSubnetName:
      description: Enter the name of the first public subnet
      default: davide-manca-first-public-subnet
    firstPublicSubnetCidr:
      description: Provide the cidr of the first public subnet
      default: 10.0.16.0/20
    secondPublicSubnetName:
      description: Enter the name of the second public subnet
      default: davide-manca-second-public-subnet
    secondPublicSubnetCidr:
      description: Provide the cidr of the second public subnet
      default: 10.0.32.0/20
    firstPrivateSubnetName:
      description: Enter the name of the first private subnet
      default: davide-manca-first-private-subnet
    firstPrivateSubnetCidr:
      description: Provide the cidr of the first private subnet
      default: 10.0.48.0/20
    secondPrivateSubnetName:
      description: Enter the name of the second private subnet
      default: davide-manca-second-private-subnet
    secondPrivateSubnetCidr:
      description: Provide the cidr of the second private subnet
      default: 10.0.64.0/20

```

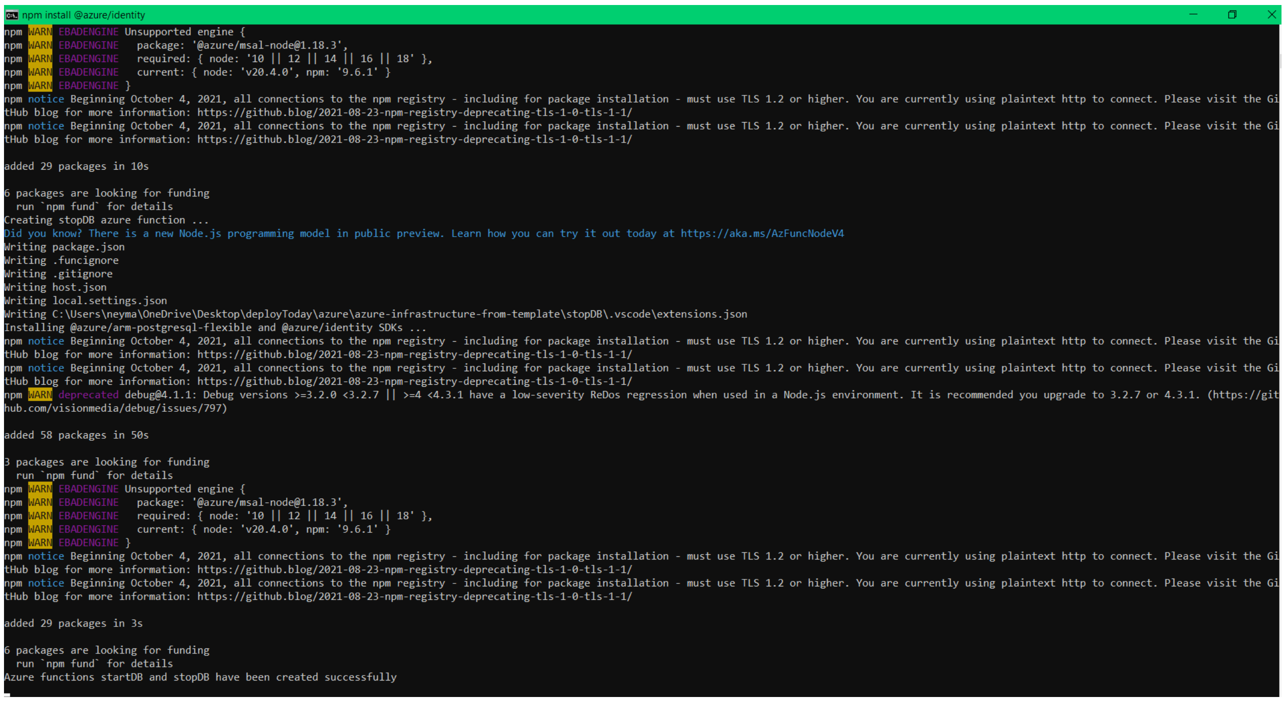
Fig 30: Pulumi.yaml file for template parameters



```
npm install

C:\Users\neyma\OneDrive\Desktop\deployToday\azure>curl -L -o deployTemplate.sh https://raw.githubusercontent.com/dev9815/pulumiAzureTemplateThesis/master/deployTemplate.sh
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   476  100   476    0     0   1391      0 --:--:-- --:--:-- --:--:--  1400

C:\Users\neyma\OneDrive\Desktop\deployToday\azure>dir
 Il volume nell'unità C è OS
 Numero di serie del volume: 24B6-AFA9

 Directory di C:\Users\neyma\OneDrive\Desktop\deployToday\azure

11/09/2023  13:48    <DIR>          .
11/09/2023  13:48    <DIR>          ..
11/09/2023  13:48               476 deployTemplate.sh
               1 File            476 byte
               2 Directory  22.804.398.080 byte disponibili

C:\Users\neyma\OneDrive\Desktop\deployToday\azure>bash deployTemplate.sh azureDeploy
This command will walk you through creating a new Pulumi project.

Enter a value or leave blank to accept the (default), and press <ENTER>.
Press ^C at any time to quit.

project name: azureDeployment
project description: (Architecture implementation)
Created project 'azureDeployment'

Created stack 'azureDeploy'

azure-native:location: Name of the region of Azure: (WestEurope)
aksResourceGroupName: Name of the aks resource group used for aks cluster resources: (d-manca-aksResourceGroup)
firstPrivateSubnetAddressSpace: Address space of the first private subnet: (10.0.48.0/20)
firstPrivateSubnetName: Name of the first private subnet: (davide-manca-firstPrivateSubnet)
firstPublicSubnetAddressSpace: Address space of the first public subnet: (10.0.16.0/20)
firstPublicSubnetName: Name of the first public subnet: (davide-manca-firstPublicSubnet)
resourceGroupName: Name of the main resource group to use for resources: (davide-manca-resourceGroup)
secondPrivateSubnetAddressSpace: Address space of the second private subnet: (10.0.64.0/20)
secondPrivateSubnetName: Name of the second private subnet: (davide-manca-secondPrivateSubnet)
secondPublicSubnetAddressSpace: Address space of the second public subnet: (10.0.32.0/20)
secondPublicSubnetName: Name of the second public subnet: (davide-manca-secondPublicSubnet)
virtualNetworkAddressPrefix: Address prefix of the virtual network: (10.0.0.0/16)
virtualNetworkDnsServer: Default IP Addressof the azure DNS server: (168.63.129.16)
virtualNetworkName: Name of the virtual network: (davide-manca-virtualNetwork)
Saved config

Installing dependencies...
```

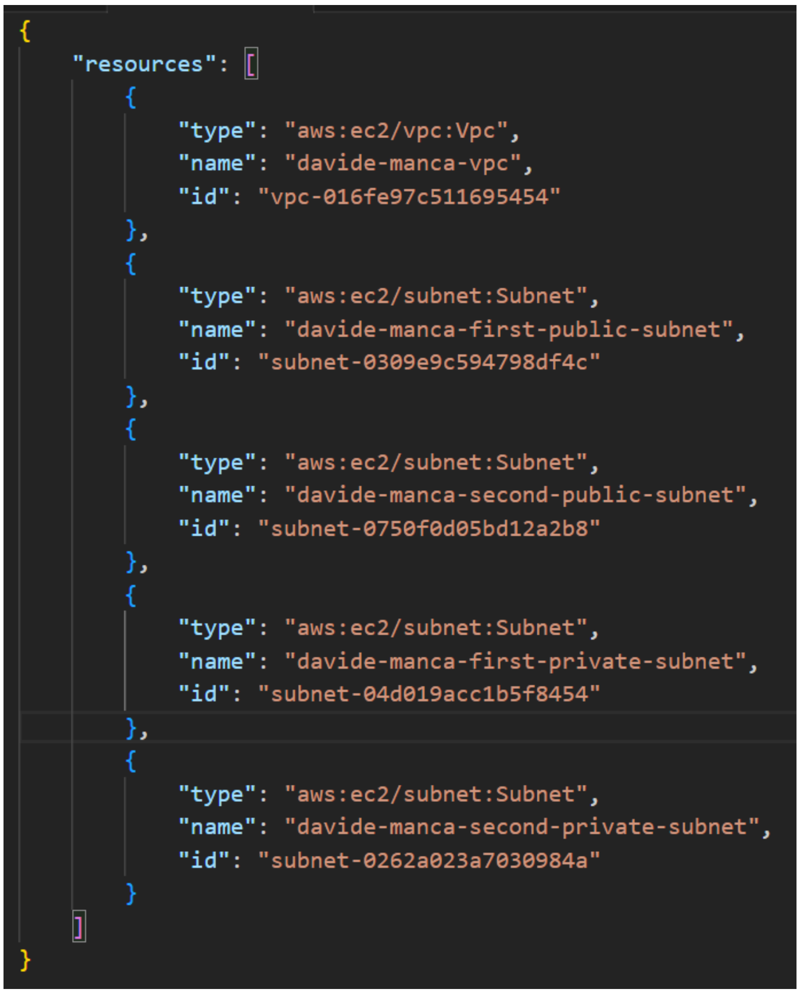
Fig 31: Azure Pulumi template in execution

The command to use for triggering the cloud infrastructure creation process is *curl -L -o deployTemplate.sh* https://raw.githubusercontent.com/dev9815/pulumiAzureTemplateThesis/master/deployTemplate.sh



```
npm install @azure/identity
npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: '@azure/msal-node@1.18.3',
npm WARN EBADENGINE   required: { node: '10 || 12 || 14 || 16 || 18' },
npm WARN EBADENGINE   current: { node: 'v20.4.0', npm: '9.6.1' }
npm WARN EBADENGINE }
npm notice Beginning October 4, 2021, all connections to the npm registry - including for package installation - must use TLS 1.2 or higher. You are currently using plaintext http to connect. Please visit the Gi
tHub blog for more information: https://github.blog/2021-08-23-npm-registry-deprecating-tls-1-0-tls-1-1/
npm notice Beginning October 4, 2021, all connections to the npm registry - including for package installation - must use TLS 1.2 or higher. You are currently using plaintext http to connect. Please visit the Gi
tHub blog for more information: https://github.blog/2021-08-23-npm-registry-deprecating-tls-1-0-tls-1-1/

added 29 packages in 10s

6 packages are looking for funding
  run `npm fund` for details
Creating stopDB azure function ...
Did you know? There is a new Node.js programming model in public preview. Learn how you can try it out today at https://aka.ms/AzFuncNodeV4
Writing package.json
Writing .funcignore
Writing .gitignore
Writing host.json
Writing local.settings.json
Writing C:\Users\neyma\OneDrive\Desktop\deployToday\azure\azure-infrastructure-from-template\stopDB\.vscode\extensions.json
Installing @azure/arm-postgresql-flexible and @azure/identity SDKs ...
npm notice Beginning October 4, 2021, all connections to the npm registry - including for package installation - must use TLS 1.2 or higher. You are currently using plaintext http to connect. Please visit the Gi
tHub blog for more information: https://github.blog/2021-08-23-npm-registry-deprecating-tls-1-0-tls-1-1/
npm notice Beginning October 4, 2021, all connections to the npm registry - including for package installation - must use TLS 1.2 or higher. You are currently using plaintext http to connect. Please visit the Gi
tHub blog for more information: https://github.blog/2021-08-23-npm-registry-deprecating-tls-1-0-tls-1-1/
npm WARN deprecated debug@4.1.1: Debug versions >=3.2.0 <3.2.7 || >=4 <4.3.1 have a low-severity ReDos regression when used in a Node.js environment. It is recommended you upgrade to 3.2.7 or 4.3.1. (https://git
hub.com/visionmedia/debug/issues/797)

added 58 packages in 50s

3 packages are looking for funding
  run `npm fund` for details
npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: '@azure/msal-node@1.18.3',
npm WARN EBADENGINE   required: { node: '10 || 12 || 14 || 16 || 18' },
npm WARN EBADENGINE   current: { node: 'v20.4.0', npm: '9.6.1' }
npm WARN EBADENGINE }
npm notice Beginning October 4, 2021, all connections to the npm registry - including for package installation - must use TLS 1.2 or higher. You are currently using plaintext http to connect. Please visit the Gi
tHub blog for more information: https://github.blog/2021-08-23-npm-registry-deprecating-tls-1-0-tls-1-1/
npm notice Beginning October 4, 2021, all connections to the npm registry - including for package installation - must use TLS 1.2 or higher. You are currently using plaintext http to connect. Please visit the Gi
tHub blog for more information: https://github.blog/2021-08-23-npm-registry-deprecating-tls-1-0-tls-1-1/

added 29 packages in 3s

6 packages are looking for funding
  run `npm fund` for details
Azure functions startDB and stopDB have been created successfully
```

Fig 32: Azure Pulumi template in execution: bash script involved in installing npm packages for Azure Functions



```json
{
    "resources": [
        {
            "type": "aws:ec2/vpc:Vpc",
            "name": "davide-manca-vpc",
            "id": "vpc-016fe97c511695454"
        },
        {
            "type": "aws:ec2/subnet:Subnet",
            "name": "davide-manca-first-public-subnet",
            "id": "subnet-0309e9c594798df4c"
        },
        {
            "type": "aws:ec2/subnet:Subnet",
            "name": "davide-manca-second-public-subnet",
            "id": "subnet-0750f0d05bd12a2b8"
        },
        {
            "type": "aws:ec2/subnet:Subnet",
            "name": "davide-manca-first-private-subnet",
            "id": "subnet-04d019acc1b5f8454"
        },
        {
            "type": "aws:ec2/subnet:Subnet",
            "name": "davide-manca-second-private-subnet",
            "id": "subnet-0262a023a7030984a"
        }
    ]
}
```

Fig 33: resources.json file containing every resource for the importing

6. Conclusion and future scenarios

With template and automation scripts the project has assumed the role of a plug and play application which can be adopted for raising up a resilient standard cloud infrastructure for deploying inside whatever kind of application in a multi-cloud context where AWS and Azure have been involved. Surely there are many aspects which can be improved in order to gain more efficiency and functionalities. For instance in the case of Pulumi APIs for Azure, it would be very powerful to return more values as outputs when a resource gets created so that other dependent resources can have those parameters specified internally instead of retrieving them by means of a getter function. This implies an intervention to Pulumi to have all of the resources in a synchronized deployment but if asynchronous behaviors have to be achieved it could be very recommended to structure every API asynchronously so that the environment could involve only asynchronous APIs. As possible additional future scenarios, it could be considered:

- the development of kubernetes applications to be deployed in this proposed cloud infrastructure model, by managing the app development always with Pulumi in terms of coding, and studying the performances, the efficiency and the relations between the deployed apps and the infrastructure underneath.
- to improve the limited disaster recovery strategy used by creating an external mechanism which allows to perform a backup of the cloud resources in a post-deployment phase and to deploy them into another region, or even to another cloud provider. By focusing on the solution implemented for this project it has to be completed with the actual resources deployment because the final result achieved so far is to get an auto-generated IaC code of each resource from the JSON file containing all of the resources. The missing part therefore is to upload those to another region on the cloud. It could be leveraged or created some automation bash script to achieve this behavior.
- to study how to improve the Pulumi APIs related to Azure functions in order to permit to declare the code of the function internally to the API, such as AWS does. In this way not only does the code results more clear, clean, and fast to program but also it would be avoided to create different external projects (folders) related to Azure functions improving so time execution performances and saving useless npm package installations.
- to develop the same cloud infrastructure model for both the cloud providers by using Terraform and comparing the deployment and performance differences with Pulumi.

7. Bibliography

- https://www.geeksforgeeks.org/cloud-computing-tutorial/?ref=lbp
- https://www.dataversity.net/how-the-cloud-has-evolved-over-the-past-10-years/#:~:text=Over%20the%20years%2C%20cloud%20computing,improved%20solutions%20for%20critical%20problems
- https://www.pulumi.com/
- https://blog.sparkfabrik.com/hubfs/Blog/Infrastructure-as-code-scheme.png
- https://d1.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf
- https://docs.aws.amazon.com/vpc/latest/userguide/how-it-works.html
- https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Internet_Gateway.html
- https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html
- https://aws.amazon.com/waf/?nc1=h_ls
- https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-groups.html
- https://aws.amazon.com/eks/?nc1=h_ls
- https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
- https://aws.amazon.com/eventbridge/faqs/#:~:text=Amazon%20EventBridge%20is%20a%20service,source%20on%20the%20EventBridge%20console
- https://aws.amazon.com/eventbridge/
- https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html
- https://docs.aws.amazon.com/vpc/latest/userguide/how-it-works.html
- https://aws.amazon.com/blogs/security/how-to-record-ssh-sessions-established-through-a-bastion-host/
- https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html
- https://aws.amazon.com/rds/
- https://aws.amazon.com/ecr/?nc1=h_ls
- https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal
- https://learn.microsoft.com/en-us/azure/virtual-network/network-overview
- https://learn.microsoft.com/en-us/azure/nat-gateway/nat-overview
- https://learn.microsoft.com/en-us/azure/bastion/bastion-overview
- https://learn.microsoft.com/en-us/azure/aks/intro-kubernetes

- **https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview**
- **https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview**
- **https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/overview**
- **https://learn.microsoft.com/en-us/azure/container-registry/container-registry-intro**
- **https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview?tabs=net**
- **https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction**
- **https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview**
- **https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview?pivots=programming-language-csharp**

Acknowledgments

I want to express my sincere gratitude to my supervisor Professor Guido Marchetto for his support, help, and guidance with thesis. From the beginning his availability has made possible the development of this project as well as the collaboration with Reply Liquid company. For this reason I am also grateful to the manager Dr. Danilo Abbaldo, who has offered me the opportunity to give my contribution in the development of this project by assigning me this thesis, and to the tutors Dr. Stefano Martinelli and Dr. Davide Sarais for taking their time to assist me when I needed development and personal help.

The period I have spent in the company for carrying out this project opened me up to the knowledge of many technical aspects of cloud computing but especially to meet a work team at Reply Liquid composed by very open-minded people who has inspired me with their positive mindset.

As a Politecnico di Torino student I have been very lucky and honored to be part of one of the greatest universities in the sectors of innovation, technology and engineering, as a student. Here I have had the possibility to learn very innovative technologies in the environment of cloud computing and networks by the most qualified and experienced professors. For this choice I am grateful and thankful to my brother Michael who drove and addressed me to this path against my initial doubts and concerns.

Despite I did not enjoy the student life in presence because of the enrollment at the pandemic period as an off-campus student, I have experienced anyway the Polito life in the following year by knowing students and people who has helped me in the university career and especially in the self growth. In particular I feel I was very lucky to have met people like Stefano (Strizzi to friends), Antonio, Ilaria, and Samuele. They were very important especially in my difficult moments, some of them treated me like a brother and I am very grateful to them. We shared many great and funny moments together which made me feel very good and better.

I have also to say thanks to my friend Mariano who has been light during my path. His help has been fundamental and the guitar recording session moments in Don Bosco Crocetta provided me that wellness, lightheartedness, and smiles I needed.

A particular consideration goes also to Bruk whose advices, point of views and ways of observing life have been crucial and significant to my personal improvement. I am very grateful to him above all to have contributed on leading me to choose the thesis experience in Reply Liquid.

In the end (only at the paragraph level but at first position in order of importance) I have to say thanks to my family for its never ended support. To my dad for his never ending love and continuous economically as well as temperamentally support; to my mum for her comfort in the difficult moments, her love and her daily humility; to my brother and my sister-in-law for never leaving me alone thanks to their motivation and support provided in useful advices but above all in the concrete help by hosting me in their home whenever I needed it.

I am extremely grateful to them all and I include in the end everyone I forgot to mention.

Second-half of my life starts off from here.