INTRO

text

1. CH
   1. GI
      1. P1
         1. Pa

[INTRO 1](#_Toc129269714)

[Chapter I. CH 1](#_Toc129269715)

[I. GI 1](#_Toc129269716)

[1. P1 1](#_Toc129269717)

[(a) Pa 1](#_Toc129269718)

[General topic 5](#_Toc129269719)

[Chapter II. Generalized context of the project 6](#_Toc129269720)

[I. Overview on the host organization 6](#_Toc129269721)

[II. Project description 6](#_Toc129269722)

[III. Analysis of existing processes 6](#_Toc129269723)

[IV. Constructive criticism on existing processes 6](#_Toc129269724)

[V. Problem statement 6](#_Toc129269725)

[VI. Ideation and proposed solution 6](#_Toc129269726)

[VII. Project management 7](#_Toc129269727)

[1. Comparative research 7](#_Toc129269728)

[2. Agile devSecOPS 7](#_Toc129269729)

[Chapter III. Requirement analysis 10](#_Toc129269730)

[I. Capturing requirements 10](#_Toc129269731)

[1. Identifying key actors 10](#_Toc129269732)

[2. Functional requirements 11](#_Toc129269733)

[3. Non-functional requirements 12](#_Toc129269734)

[II. Product backlog 13](#_Toc129269735)

[1. Backlog history 13](#_Toc129269736)

[2. Sprint planification(planning//to check) 14](#_Toc129269737)

[III. Modeling for cloud architecture and devSecOps 15](#_Toc129269738)

[1. Cloud related diagrams: 15](#_Toc129269739)

[(a) Class diagram of the provisioned cloud resources: 15](#_Toc129269740)

[2. CI/CD related diagrams 16](#_Toc129269741)

[(a) Sequence diagram of a sample deployment: 16](#_Toc129269742)

[(b) Sequence diagrams of the CI/CD workflow: 16](#_Toc129269743)

[3. Technological choices 18](#_Toc129269744)

[(a) Cloud provider choice: 18](#_Toc129269745)

[(b) Infrastructure as code tools 18](#_Toc129269746)

[(c) Containerization and orchestration techniques 20](#_Toc129269747)

[(d) Self-hosted PaaS services 21](#_Toc129269748)

[(e) Development tools 23](#_Toc129269749)

[(f) Development languages 23](#_Toc129269750)

[IV. Architectural specifications 25](#_Toc129269751)

[1. Physical architecture 25](#_Toc129269752)

[2. PaaS deployment architecture 26](#_Toc129269753)

[V. Workspace description 27](#_Toc129269754)

[CONCLUSION 27](#_Toc129269755)

[Chapter IV. 1st Sprint: Maintenance of existing resources 29](#_Toc129269756)

[I. Introduction 29](#_Toc129269757)

[II. Sprint backlog : 29](#_Toc129269758)

[III. Maintenance operations 29](#_Toc129269759)

[IV. Application overview and SCM structuring. 31](#_Toc129269760)

[V. Conclusion 33](#_Toc129269761)

[Chapter V. 2nd Sprint: Information gathering and cloud design 34](#_Toc129269762)

[I. Introduction 34](#_Toc129269763)

[II. Sprint backlog: 34](#_Toc129269764)

[III. Package diagram of cloud resources : 34](#_Toc129269765)

[IV. Comparative study on container orchestrators: 35](#_Toc129269766)

[V. Package diagram for the PaaS logical components: 36](#_Toc129269767)

[VI. Conclusion 36](#_Toc129269768)

[Chapter VI. 3rd Sprint: Resource provisioning 37](#_Toc129269769)

[I. Introduction 37](#_Toc129269770)

[II. Sprint backlog: 37](#_Toc129269771)

[III. UML design : class diagram for cloud infrastructure 37](#_Toc129269772)

[IV. “HCP config” diagram for cloud provisioning: 38](#_Toc129269773)

[V. Component diagram of provisioned resources: 40](#_Toc129269774)

[VI. Conclusion 41](#_Toc129269775)

[Chapter VII. 4th sprint: Preliminary infrastructure setup 42](#_Toc129269776)

[I. Introduction 42](#_Toc129269777)

[II. UML design: sequence diagram for infrastructure setup 42](#_Toc129269778)

[III. Playbook diagram for resource setup 44](#_Toc129269779)

[Chapter VIII. 5th Sprint: 47](#_Toc129269780)

[Chapter IX. 6th Sprint: 48](#_Toc129269781)

[Chapter X. 7th Sprint: 49](#_Toc129269782)

[Chapter XI. 8th Sprint: 50](#_Toc129269783)

[Chapter XII. 9th Sprint: 51](#_Toc129269784)

General topic

1. Generalized context of the project
   1. Overview on the host organization
   2. Project description
   3. Analysis of existing processes
   4. Constructive criticism on existing processes
   5. Problem statement
   6. Ideation and proposed solution
   7. Project management
      1. Comparative research

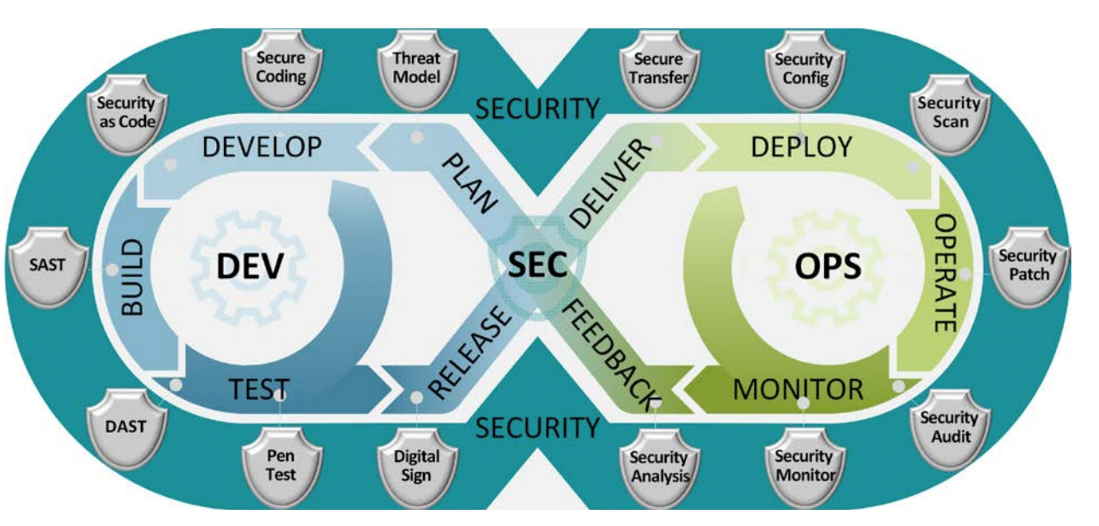
|  |  |  |  |
| --- | --- | --- | --- |
|  | Waterfall  (V model) | Agile | devSecOPS |
| Lifecycle | linear software development approach | Incremental and iterative sprint cycles. | Uses automation techniques to enable continuous deployment of change. |
| Automation level | Low | Varied | High |
| Delivery of value | Slow (Scale : months) | Rapid (daily/weekly) | Continuous |
| Responsiveness to business needs | Extremely limited | Responsive | Highly responsive |
| Collaboration | Low: teams operate in functional silos | Improved: short dev cycles | High: cross-functional teams are involved from project start. |
| Quality | Low: issues are only identified after tests | Improved: issues are identified after every sprint | High: Unit testing and code quality checks are performed during development. |
| Risk | Increases as project development progresses | Decreases as project development progresses | Decreases as project development progresses |

Based on the previous comparative study, it is apparent that the approach most suited to our use case is one that helps improve collaboration, reduce context-switching, introduce automation, and enable observability and monitoring.

* + 1. Agile devSecOPS

When evaluating devSecOPS as a practice and Agile as a delivery approach, it is important to note that they are not mutually exclusive. In spite of the evident differences between DevSecOPS and the Agile methodology, their overall goal of increasing speed and delivering quality software is similar in nature and together they produce great products and improve the software development.

The following figure showcases an overview of the process and the key DevSecOPS roles.



DevSecOPS Evangelist:

Usually the DevSecOPS team leader, he focuses on promoting the DevSecOPS advantages, identifying and quantifying companies’ benefits deriving from a higher agility.

Release Manager:

The product stability manager which is basically the product owner. He he cares about the product’s management and coordination.

Automation Architect:

Provides a complete automation role involving the DevSecOPS and Cloud solutions. He is an integration specialist that ensures the high availability of the pre-production and production systems.

Software Developer/Tester:

DevSecOPS developers are not responsible only for the transformation of new requirements into code, they also have to deal with testing, distribution and continuous monitoring processes.

Security Engineer

The DevSecOPS approach implements security by design.

1. Requirement analysis

In this chapter, our analysis is aimed at identifying the key actors in our design, the functional and non-functional needs our system is to provide for. We will then move on to an overview of the sprint bursts we have realized. Finally, we will provide a dissection of the ecosystem in the form of various UML diagrams.

* 1. Capturing requirements
     1. Identifying key actors

In this context, an actor is a user or any other system that interacts with the subject by exchanging signals and data.

Cloud architecture related actors:

In order to have a devSecOPS compliant approach, it needs to be built on top of a containerized, highly available cloud infrastructure:

* Cloud architect: He is responsible for designing and implementing cloud computing solutions.
* IaC tools: Although being a piece of software, it is necessary in order to automate provisioning and configuration of resources.
* Cloud provider: Usually a third-party entity that allows for elastically allocating resources.
* DevSecOPS team: A devSecOPS engineer is the consumer in this case, he uses the provisioned resources to build an ecosystem that is compliant with organizational needs.

CI/CD related actors:

The cloud resources hold the value of a tool that is then leveraged to assist the development process:

* DevSecOPS team: uses the provisioned resources and follows an agile devSecOPS approach to build an ecosystem that is compliant with organizational needs.
* Developer: A consumer of the devSecOPS ecosystem as well as the CI/CD workflow.
* Company client: He is the end user and provides the specifications on software development.
  + 1. Functional requirements

Formulating an understanding on functional requirements is a primordial phase in the implementation of the subject.

* A cloud-based infrastructure capable of hosting the devSecOPS ecosystem in terms of compute resources, storage and networking.
* A continuous integration platform : The desired goal is to provide a CI workflow that channels the development effort in order to continuously ensure code quality.
* A CD workflow: provide an automated and continuous delivery and deployment process.
  + 1. Non-functional requirements
* High availability and resilience: Typically satisfied by distributed backend resources, orchestration, and load balancing of workloads.
* Performance and scalability: Usually dependent on the cloud provider as well as the used technologies.
* Security: An inbuilt quality that cloud and DevSecOPS offer by design.
* Observability: A highly achievable need due the pluggability of containerized environments.
* Usability: A somewhat hard to achieve requirement due to the rarity of a technologically adept workforce.
* Relatively low cost: the need to prioritize self-hosting and adopting opensource alternatives.
  1. Product backlog
     1. Backlog history

The following table describes the product backlog:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| EpicID | EPIC | StoryID | Story | Priority |
| 0 | Certified devSecOps training | 0.1 | Managing and running custom VMs. |  |
| 0.2 | Managing docker containers. |  |
| 0.3 | CI/CD pipelines. |  |
| 1 | Maintenance and cleanup. | 1.1 | Exploring existing infrastructure and resources. |  |
| 1.2 | Performing maintenance on enterprise assets. |  |
| 2 | Cloud design | 2.1 | Information gathering phase. |  |
| 3 | Resource provisioning | 3.1 | Provisioning resources using IaC playbooks. |  |
| 4 | Infrastructure setup. | 4.1 | Infrastructure setup using IaC playbooks. |  |
| 5 | Initial PaaS setup. | 5.1 | Setting up ingress controller and TLS certificate provisioner. |  |
| 5.2 | Setting distributed storage backend. |  |
| 5.3 | Setting up network load balancer. |  |
| 6 | Deployment of CI/CD platform | 6.1 | Setting up personalized CI/CD orchestrator. |  |
| 6.2 | Setting up quality gate (CI). |  |
| 6.3 | Setting up CD controller. |  |
| 7 | Preparation of automated CICD workflows. |  | Structuring SCM backend |  |
|  | Using GitOPS and devOPS tools to automate CI/CD pipelines. |  |
| 8 | Deployment of authentication/authorization backend |  | Self-managing distributed database storage backends (Redis, mongo DB, PostgreSQL). |  |
|  | Deployment and configuration of authentication and authorization services. |  |
|  | Configuring the forward auth middleware for secure access. |  |
| 9 | Implement a resilient disaster recovery strategy. |  | Provisioning cloud storage resources. |  |
|  | Preparing and applying backup strategy for application specific data. |  |
|  | Preparing and applying backup strategy for PaaS specific workloads. |  |

* + 1. Sprint planning

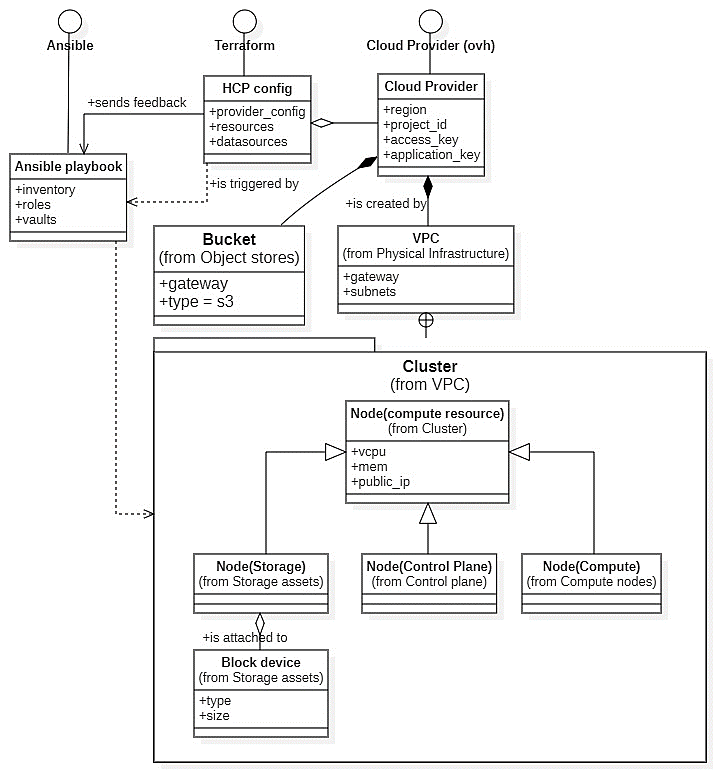
The project lasted seven months and a half starting on March 1st 2022. Overall, ten sprints have been followed with the typical duration of 3 weeks for each.

|  |  |  |
| --- | --- | --- |
| Sprint ID | Sprint Details | Duration |
| 0 | The first period was spent following a company sponsored devSecOps training. The following elements have been covered:  Creating vm images using packer, spinning up virtual machines using vagrant,  Using docker, docker compose plugin and docker container orchestration in swarm,  An overview on CI/CD pipelines using Jenkins. | 4 weeks |
| 1 | The following three weeks were dedicated to exploring the existing infrastructure. Performing some maintenance and planning for migration. | 3 weeks |
| 2 | Next, we have started the information gathering phase in which we have formed an initial overview of the desired goals we would like to reach. | 2 weeks |
| 3 | The use of IaC allowed us to provision the main cloud resources dedicated to hosting the PaaS infrastructure. | 2 weeks |
| 4 | An initial setup of the provisioned resources was then automated and performed. | 3 weeks |
| 5 | Putting in place the basic PaaS services, namely a distributed storage backend, a layer 2 load balancer, a cloud-native layer 4 ingress controller to route requests serving also as a reverse proxy and an application load balancer. | 3 weeks |
| 6 | Mounting the backing CI/CD services which are mostly personalized to company needs. | 3 weeks |
| 7 | Levering the CI/CD ecosystem to put in place pipelines to automate product testing, code quality checks, and delivery. | 4 weeks |
| 8 | Securing access to company assets using an Authentication/Authorization middleware. | 3 weeks |
| 9 | Putting in place a disaster recovery strategy that leverages distributed storage and S3 compliant object storage. | 3 weeks |

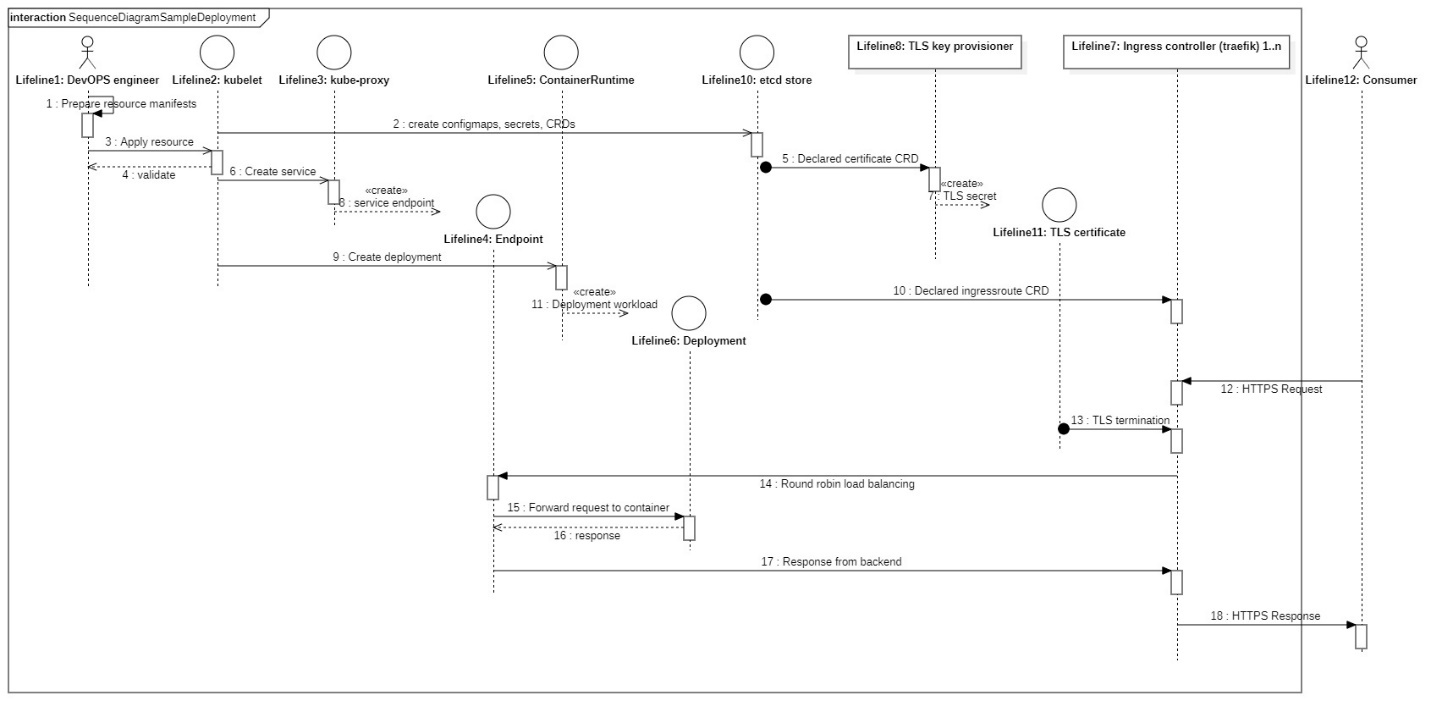
* 1. Modeling for cloud architecture and devSecOps

In order to facilitate collaboration and formulate an understanding of the system specifications and requirements, we have used the Unified Visual Language (UML).

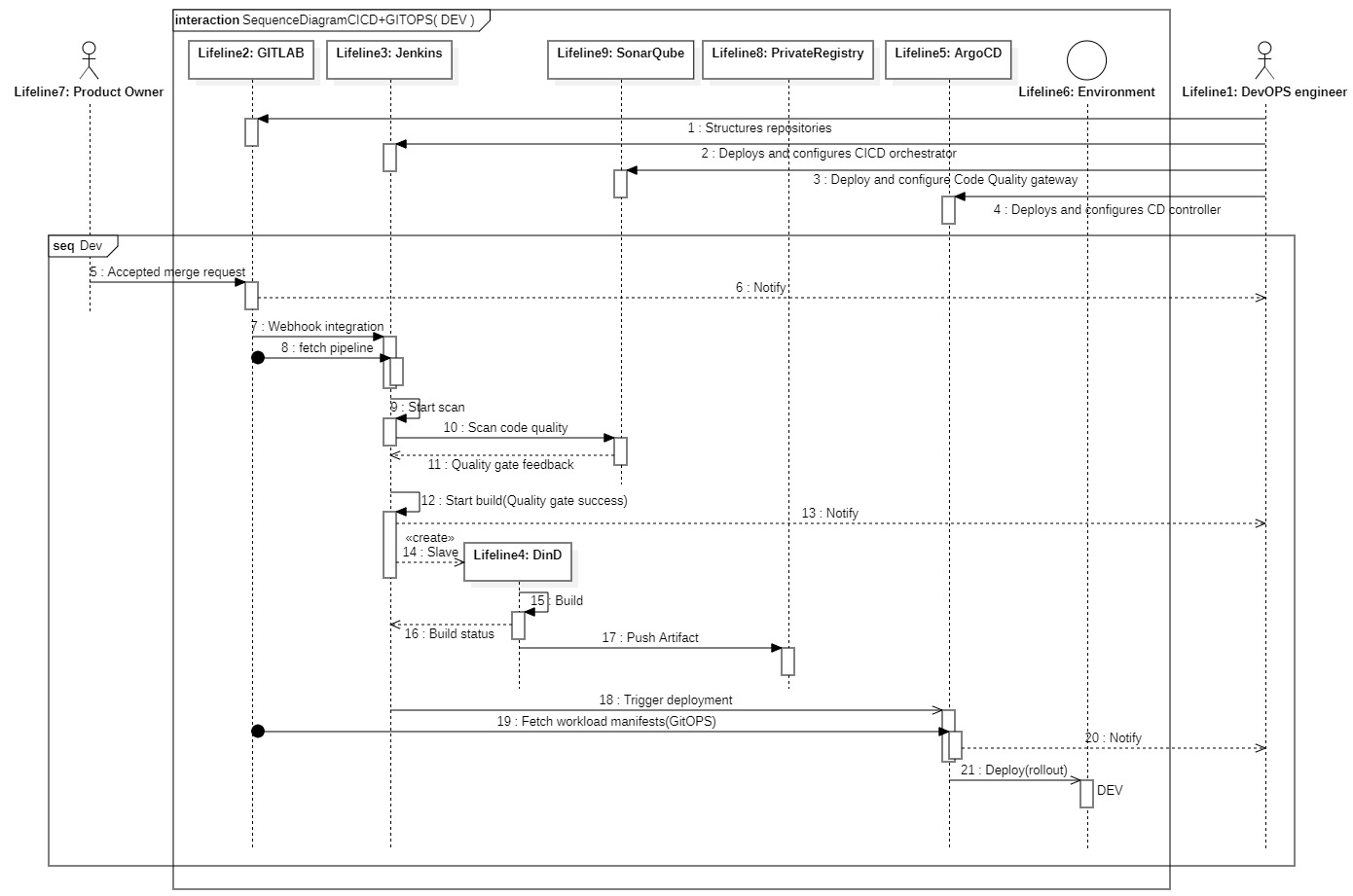
* + 1. Cloud related diagrams:
       1. Class diagram of the provisioned cloud resources:

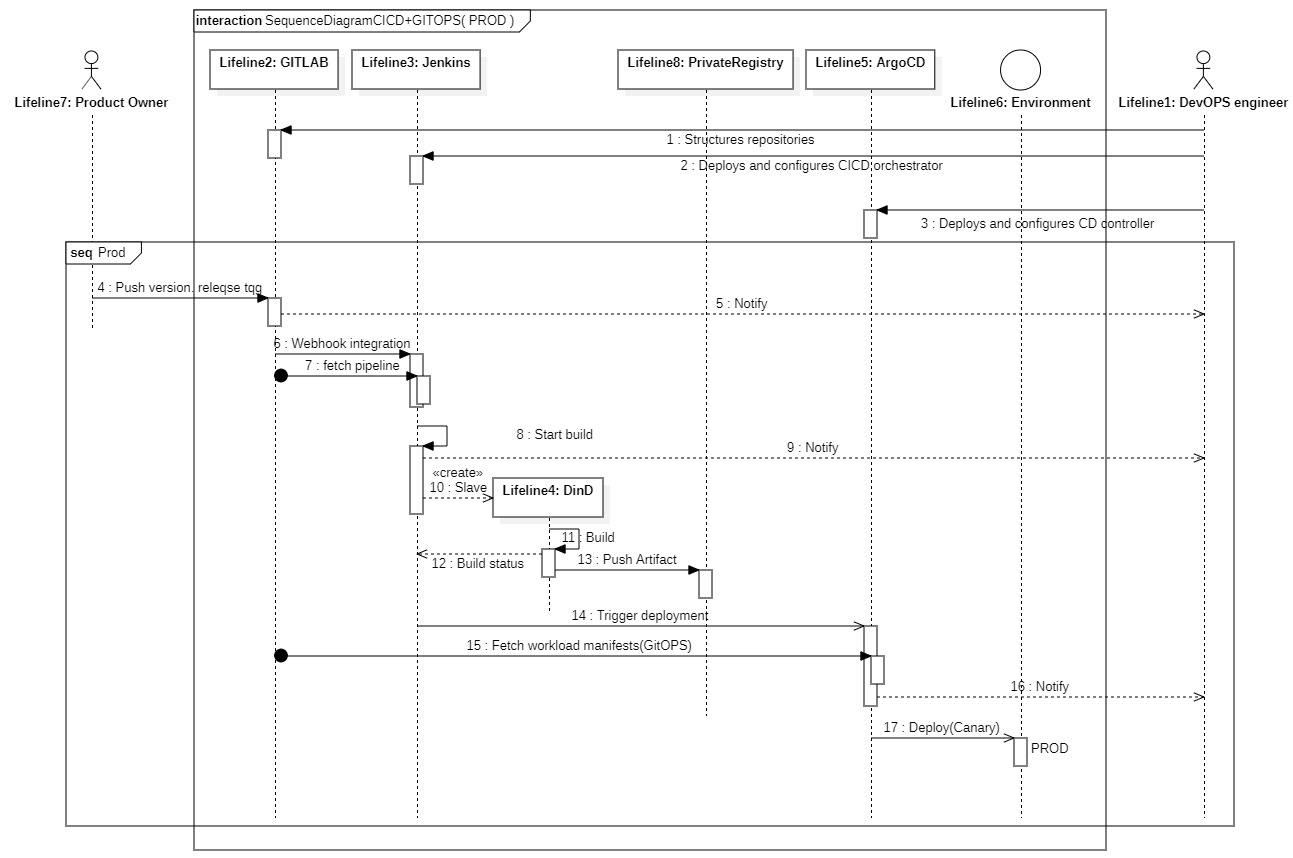


* + 1. CI/CD related diagrams
       1. Sequence diagram of a sample deployment:



* + - 1. Sequence diagrams of the CI/CD workflow:





* + 1. Technological choices

In this section, we layout the cloud provider choice,

* + - 1. Cloud provider choice:

OVH :

A decisive factor concerning the cloud provider choice is related to company’s desire to opt mainly for a provider that is based in France. OpenStack provides the foundation for the OVH Cloud infrastructure, including its virtual machines, object storage, and other cloud services. OVH also uses a range of other open-source technologies to support its cloud platform, such as Ceph for distributed storage.

Openstack:

OVH uses openstack as its backing cloud computing infrastructure. Thus, conversing with components as neutron, nova, glance, swift is a must.

* + - 1. Infrastructure as code tools

Terraform:

An industry standard for conversing with cloud providers in order to provision and configure resources.

Ansible:

We have opted for ansible because it’s faster than its alternatives namely chef, and easier to use.

* + - 1. Containerization and orchestration techniques

Container orchestrator: Kubernetes v1.21

An open source container orchestration tool that automates the deployment, scaling, and managing containerized applications.

Container runtime : containerd

A daemon for linux that manages the complete container lifecycle.

Container platform : docker-ce v20.10

The container management tool that uses containerd to manage container lifecycles and their underlying abstractions such as volumes and networking.

Container networking: Calico

A cloud-native plugin deployment in Kubernetes that uses the CNI API to provide a networking and security solution in the cluster.

* + - 1. Self-hosted PaaS services

Distributed storage backend: Ceph

A highly reliable, open source, distributed storage platform offering bloc , filesystem and object storage volumes to be leveraged by the orchestrator.

Ingress Controller: Traefik ingress controller

An open-source edge router serving as an ingress controller, a reverse proxy and a load balancer.

Layer 2 load balancer: MetalLB

This opensource load balancer provides support for bare metal Kubernetes clusters using standard protocols.

Private registry: harbor

An open-source private registry serving as a backend to sore artifacts in a secure manner.

Database storage (document): Redis, MongoDB

Database storage in document format is provided by a replicated mongoDB cluster. An in memory redis cluster allows for session storage.

Database storage (SQL): PostgreSQL

A PostgreSQL cluster in HA mode coupled with a PGpool middleware to distribute load and control replication.

Authentication middleware: Authelia

An open-source portal serving as an authentication and authorization server. It leverages the “forward auth” capability of the ingress controller to regulate access.

User management: OpenLDAP

An open-source implementation of the Lightweight Directory Access Protocol that is used to manage organizational user credentials and details.

SCM tool: Gitlab

An enterprise solution for source code management and versioning.

CI/CD orchestration: Jenkins

An open-source automation server which enables build, test, and deployment processes.

Quality gate: SonarQube

An open-source platform that allows for continuous inspection of code quality.

CD controller: ArgoCD, ArgoRollouts

An open-source declarative, GitOPS continuous delivery tool for Kubernetes.

Disaster recovery: Velero

An open-source tool to safely backup and restore, perform disaster recovery, and migrate Kubernetes cluster resources and persistent volumes.

Config formats: YAML, TOML, HCP config, JSON

* + - 1. Development tools

Development IDE: VS Code

A pluggable, lightweight opensource IDE.

SSH platform: Termius

A platform that offers port forwarding and secure file transfer over ssh.

* + - 1. Development languages

Scripting: python, shell, groovy

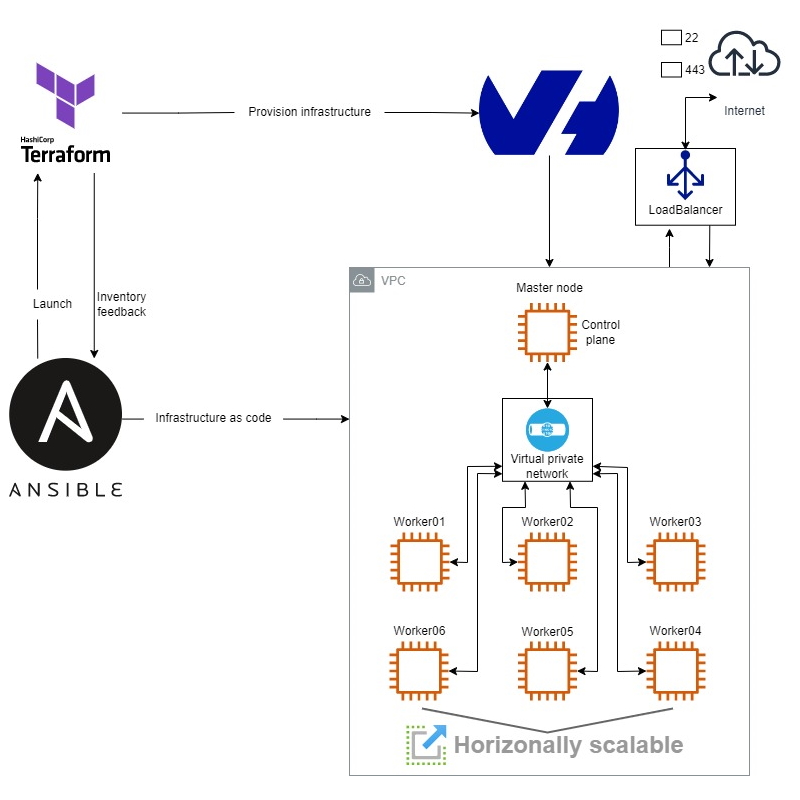
Shell was used to interface with the linux operating system. For conversing with APIs, an assortment of python scripts were developed to automate various tasks, namely, database backups. Jenkins pipelines and configurations were written in groovy.

Templating: YTT, jinja2

To template various configuration files that are dependent on the deployment environment, we used YTT for YAML/JSON. Jinja2 was the main templating tool for ansible playbooks.

* 1. Architectural specifications
     1. Physical architecture

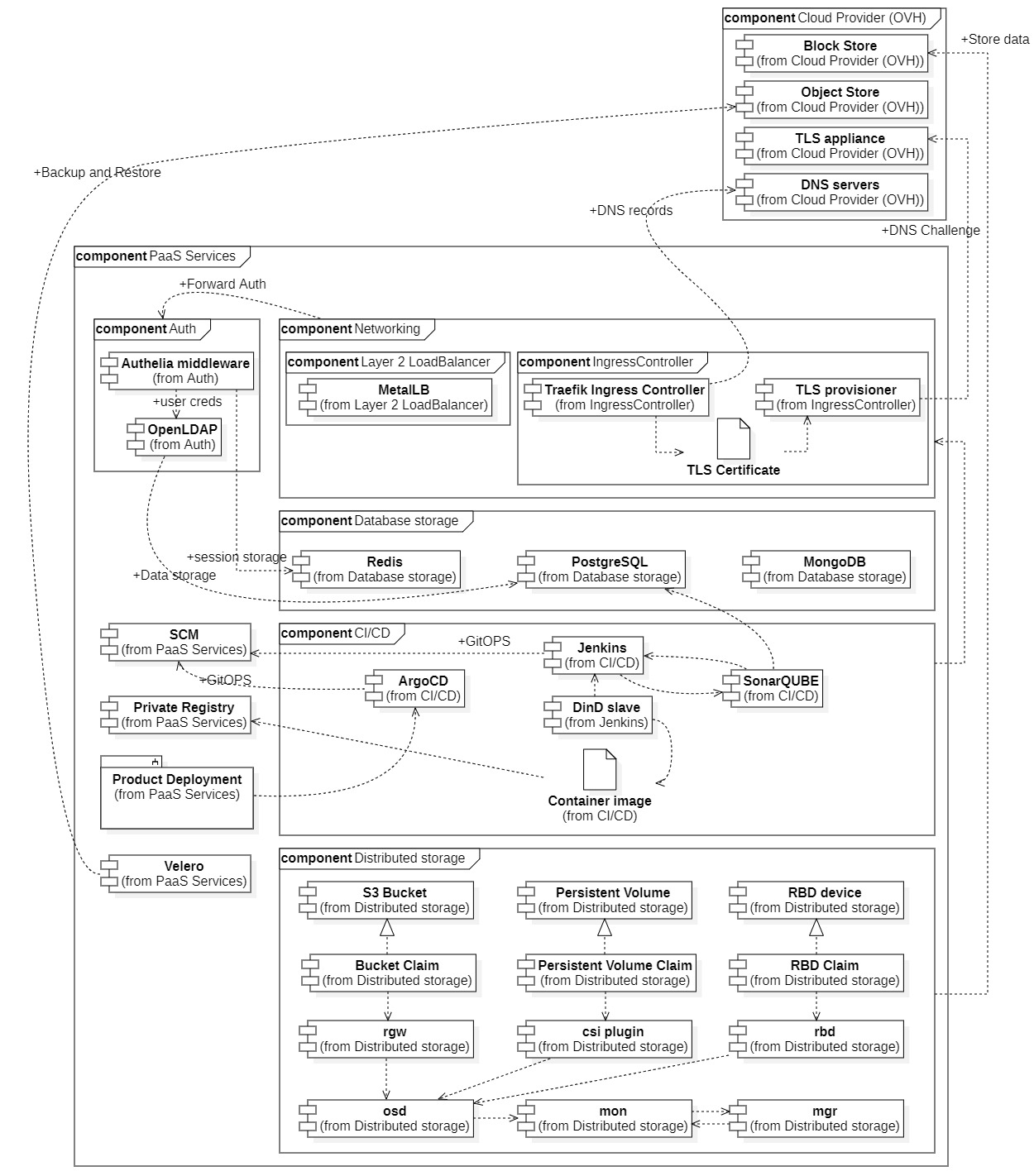
The following figure is an overview of the infrastructure resource setup:



In our use case we are using ansible to initiate the process, terraform to converse with the cloud provider and request descriptive data of the provisioned resources. This data is then fed back to ansible to continue with the infrastructure setup.

* + 1. PaaS deployment architecture

Next we layout the organization of the subsystems, software classes, and layers that make the complete logical system of our PaaS infrastructure:



The two main components of this logical infrastructure are:

* The cloud provider: with we are hosting our resources in.
* The platform as a service that we are aiming to put in place.

OVH, our cloud provider is built on top of openstack, some opensource technologies as well as other proprietary assets.

The PaaS we are aiming to put in place consists of various services that provide the organizational needs of the company in terms of :

* Storage
* Networking
* Security
* Automating deployments
* Self-hosting
  1. Workspace description

To implement the project, the company has provided us with the necessary hardware equipment, certified preliminary training as well as the desired cloud resources.

CONCLUSION

All the functional and non-functional requirements have been viewed in this chapter through the use of package, class and sequence diagrams. We have also formed an understanding on the underlying resources and services through the physical architecture and the PaaS deployment diagram.

1. 1st Sprint: Maintenance of existing resources
   1. Introduction

The first sprint was dedicated towards exploring the existing services and resources, performing some minor changes and putting in place a backup strategy.

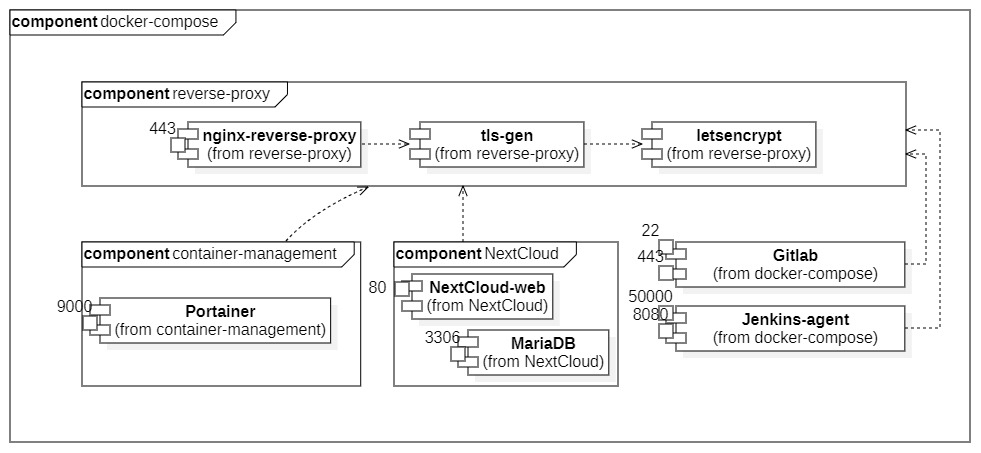
* 1. Sprint backlog :

|  |  |  |  |
| --- | --- | --- | --- |
| EpicID | Epic | StoryID | Story |
| 1 | Exploring assets. | 1.1 | SCM structuring: service components need to be split into different repos. |
| 1.2 | Keeping track of developed applications and their requirements. |
| 2 | Maintenance. | 2.1 | Rebuilding optimized containers for developed applications. |
| 2.2 | Backup of existing data on current infrastructure in S3 containers. |

* 1. Maintenance operations

In this sprint, we have assembled standalone self-hosted services into a single stack of upgraded container versions in order to provide better visibility of workloads. A container management platform was then added to this stack to improve on operability.

The following figure is the deployment diagram of the resulting docker-compose stack:



The stack includes the following services:

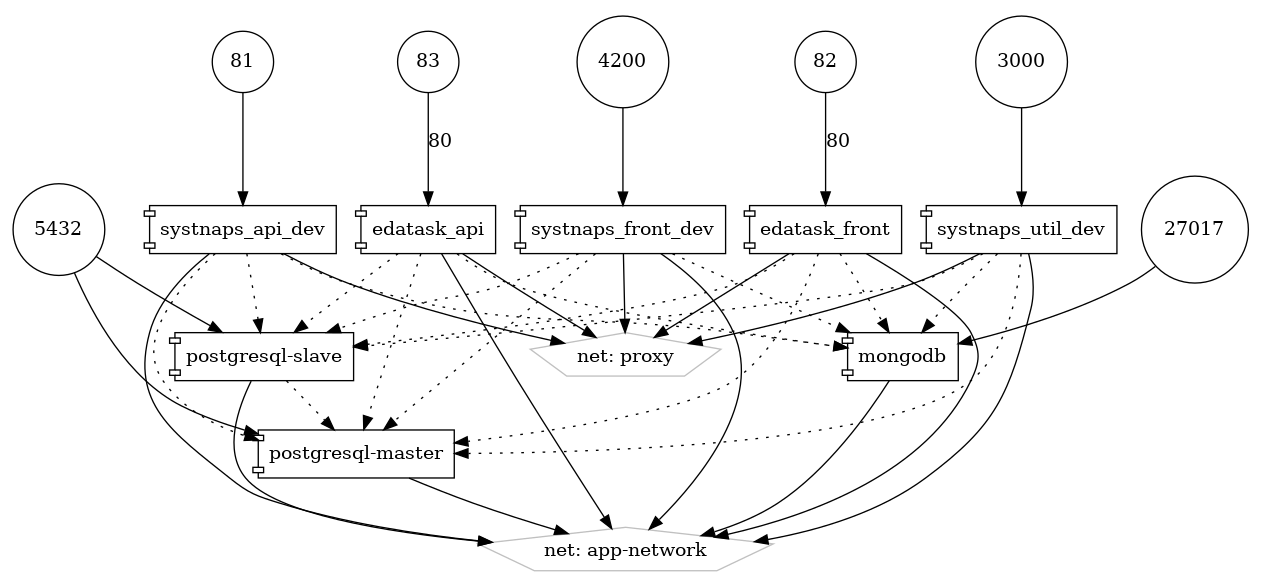
* Nginx reverse proxy: An HTTP server that acts as a reverse proxy for the other services in the stack. It is responsible for routing traffic to the correct service based on the URL and handling HTTPS encryption. It recognizes labels configured in the containers to route traffic and provide tls termination.
* TLS generator: A service that generates and renews TLS certificates using Let's Encrypt. It works with Nginx to provide HTTPS encryption for the other services in the stack.
* Let's Encrypt: A certificate authority that provides free TLS certificates.
* Portainer: A web-based management interface for Docker containers and services.
* Nextcloud: A cloud-based file sharing and collaboration platform.
* GitLab: A web-based Git repository manager, CI/CD pipeline tool, and issue tracker.
* Jenkins: An open source automation server that can be used to automate software build, test, and deployment processes.

Each service is defined in a separate Docker container and is configured using environment variables, volumes, and ports. The Nginx and TLS generator services depend on each other, and the Nextcloud, GitLab, and Jenkins services depend on the TLS generator for HTTPS encryption.

This Docker Compose stack provides a scalable and easy-to-manage infrastructure for hosting multiple cloud-based services, all secured using HTTPS encryption and Let's Encrypt TLS certificates.

* 1. Application overview and SCM structuring.

Two main applications were in development during this project. We have recommended a better source code management strategy that resulted in separating services into different repos in order to optimize the container image creation process. The resulting deployment diagram Is as follows:



The stack includes the following services:

* Backend API: A Symfony API that serves as the backend for the application. It is responsible for processing requests and returning data to the frontend.
* Frontend: An Angular web application that serves as the frontend for the application. It is responsible for displaying data and interacting with the backend API.
* HA PostgreSQL: A relational database management system that stores data for the application. It is used to store structured data for the backend API.
* Replicated MongoDB: A NoSQL database management system that stores data for the application. It is used to store unstructured data for the frontend.
* Nginx: A web server that acts as a reverse proxy for the other services in the stack. It is responsible for routing traffic to the correct service based on the URL and handling HTTPS encryption.

The backend API, frontend, MongoDB and PostgreSQL are connected using the "app-network" network. All services are also connected to the "proxy" network where Nginx resides, enabling the reverse proxy detect labels and route traffic to the appropriate service based on the URL.

* 1. Conclusion

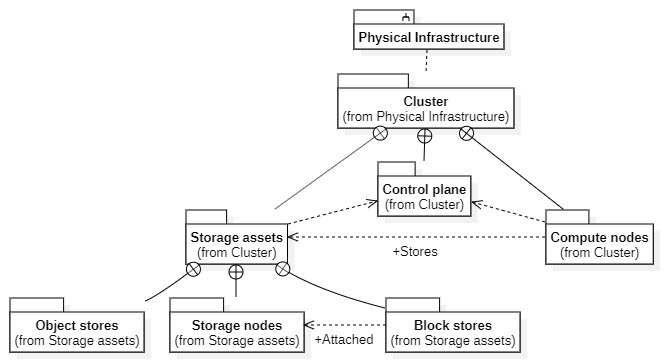
In this chapter we have successfully performed

1. 2nd Sprint: Information gathering and cloud design
   1. Introduction
   2. Sprint backlog:

|  |  |  |  |
| --- | --- | --- | --- |
| EpicID | Epic | StoryID | Story |
| 1 | Information gathering. | 1.1 | Research provider specific (OVH) constraints |
| 1.2 | Container orchestration choice. |
| 1.3 | Deciding on virtualized networking. |
| 1.4 | Deciding on application load balancer. |
| 1.5 | Deciding on network level load balancer. |
| 1.6 | Storage backend choice. |

* 1. Package diagram of cloud resources:

The cluster resources are basically cloud instances with different specifications tailored to their use cases. Three major groups of resources are distinguished in the following diagram: Control plane instances, Compute instances and storage assets which are object store buckets and compute instances to which raw block stores are attached.



* 1. Comparative study on container orchestrators:

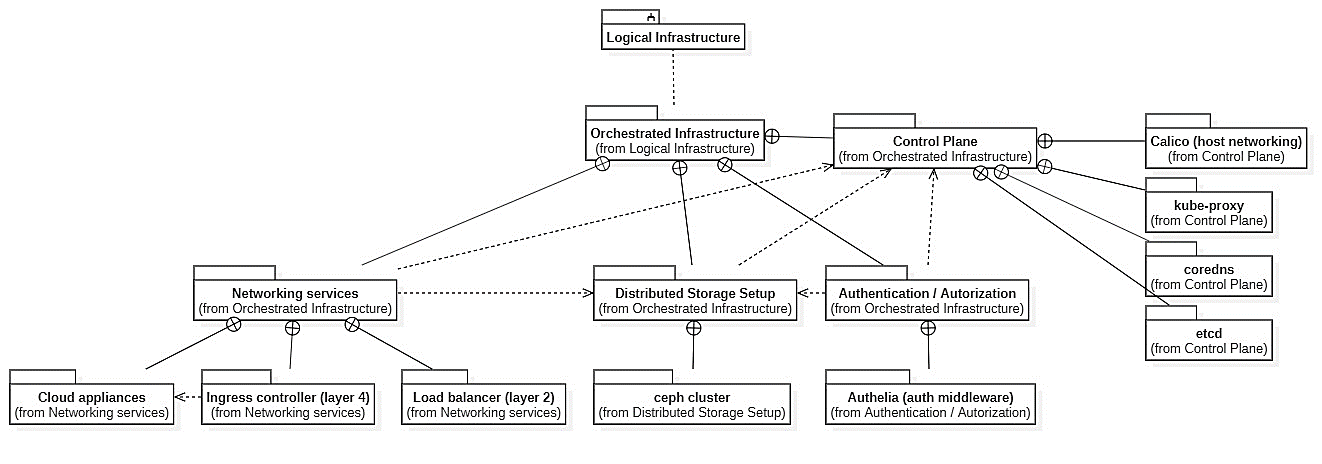
Container Orchestration Engines such as “Kubernetes”, “Apache Mesos” and “docker swarm” are platforms for managing containers and automating the deployment, scaling, and operations of containers across a cluster of nodes. This is achieved by pooling the discrete cloud resources into a single PaaS on which workloads can be deployed.

|  |  |  |  |
| --- | --- | --- | --- |
| Criteria | Kubernetes | Docker swarm | Apache Mesos |
| Ease of use | Medium | Easy | Complex |
| Cluster scalability | Medium to Large | Small to Medium | Very Large |
| Cluster installation | Complex | Easy | Medium |
| Container deployment | YAML based | Docker based | JSON based |
| Community Support | Large | Moderate | Small |
| Configuration | Declarative | Declarative | Imperative |

Kubernetes is our container orchestration tool of choice in this project. It is known for its highly advanced orchestration capabilities, built-in load balancing, rolling updates, and self-healing. It has a declarative configuration model and a highly extensible architecture. However, it does have a steeper learning curve compared to Docker Swarm.

Our goal is to tackle the task of establishing a fully sustainable PaaS by utilizing its flexibility via custom resource definitions.

* 1. Package diagram for the PaaS logical components:



This figure illustrates a package design of the main PaaS services. We mainly distinguish:

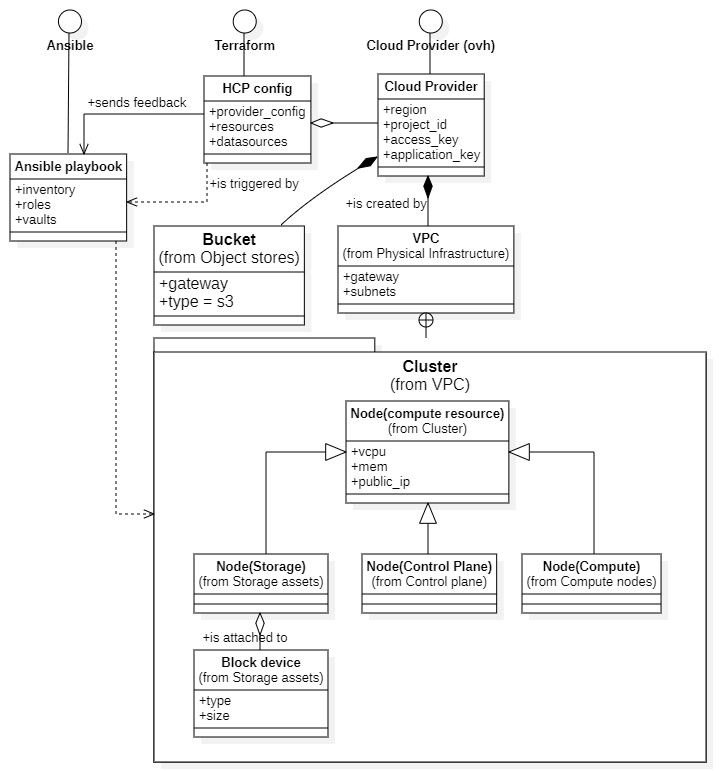
* A control plane: which manages assets in the cluster, namely, nodes, pods and other api resources.
* An assortment of networking services which allow for ingress control in both the network and application layers.
* An authentication and authorization service: which is aimed to control access to the cluster.
* A distributed, scalable, and replicated storage backend which is independent of the infrastructure in place to provide data redundancy and disaster recovery.
  1. Conclusion

1. 3rd Sprint: Resource provisioning
   1. Introduction
   2. Sprint backlog:

|  |  |  |  |
| --- | --- | --- | --- |
| EpicID | Epic | StoryID | Story |
| 1 | Provisioning resources using IaC playbooks and HCP config files. | 1.1 | Preparing provider specific(ovh) terraform config. |
| 1.2 | Preparing ansible playbook to initiate provisioning. |
| 1.3 | Setting up IaC host machine. |
| 1.4 | Cloud provider account and project creation. |

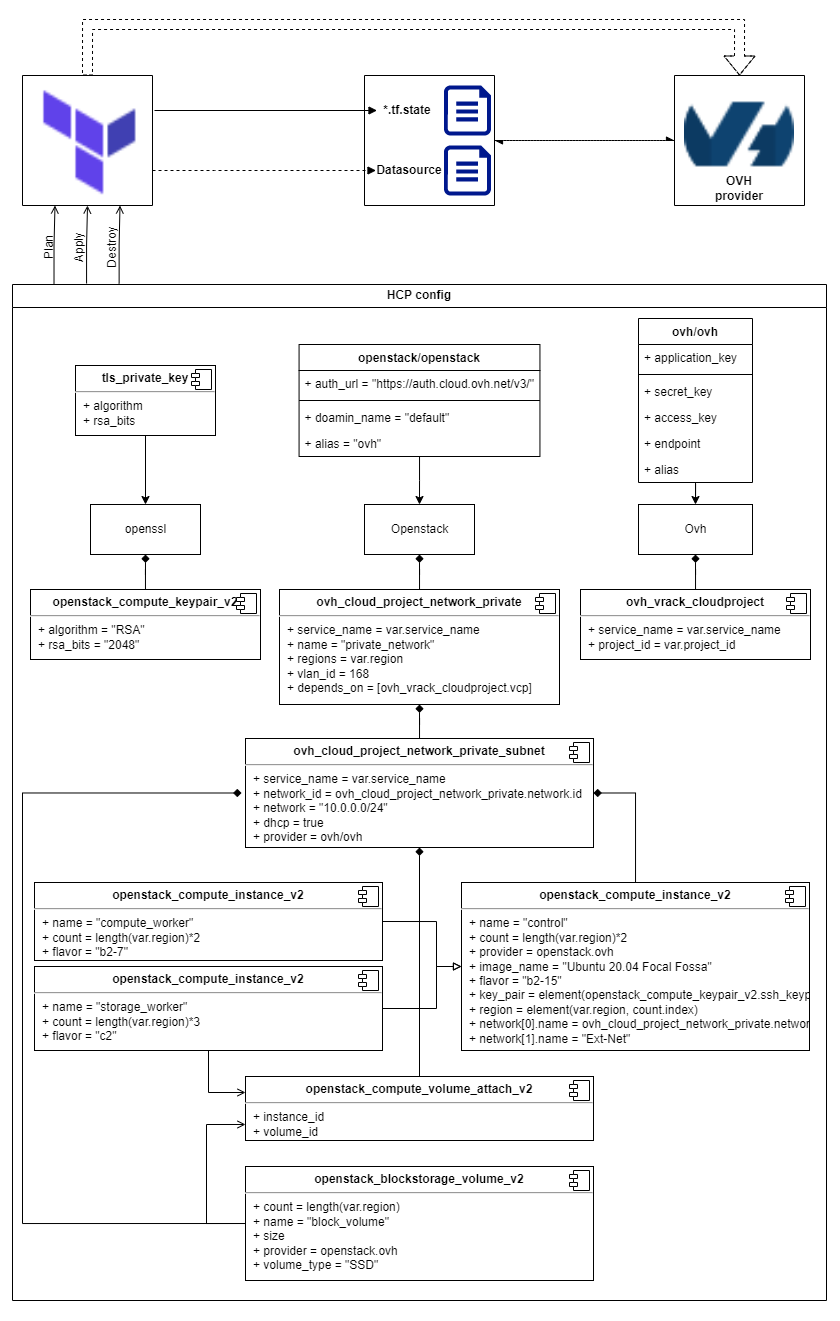
* 1. UML design: class diagram for cloud infrastructure

The following diagram showcases the interaction between the infrastructure as code tools and the cloud provider as well as the resulting infrastructure, its components, and the relation between them.



* 1. “HCP config” diagram for cloud provisioning:

The following figure illustrates the main resources declared in our terraform “HCP config”:



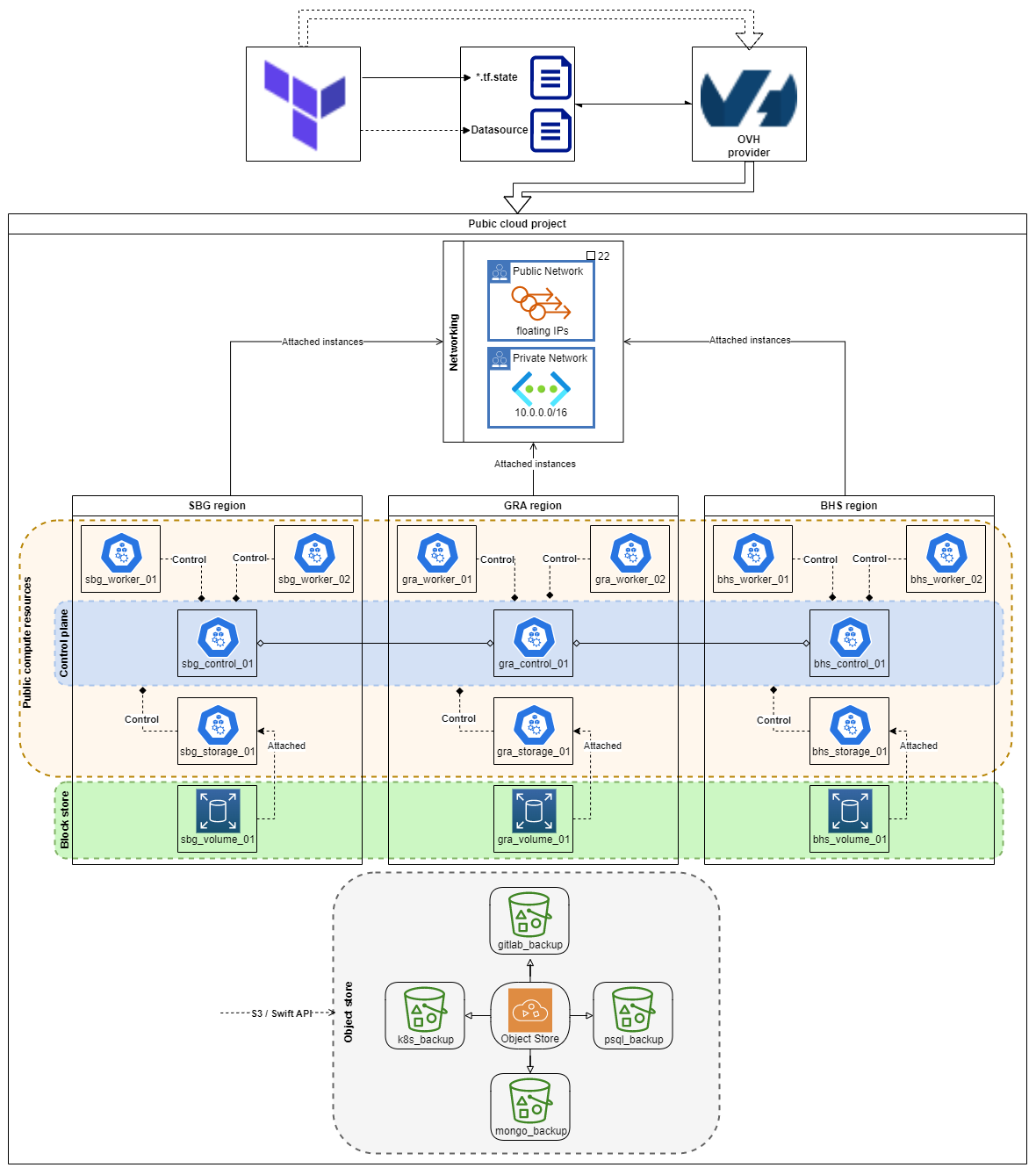
The diagram shows a Terraform configuration for deploying resources on OVH cloud provider. The configuration consists of four main components: provider, network, compute, and storage.

* The provider component specifies the OVH cloud provider and the authentication credentials to access the account.
* The network component defines the virtual private cloud (VPC) and its associated resources, such as subnets, security groups, and routers.
* The compute component defines the public cloud compute instances that will host our PaaS. The VMs are launched in the subnets defined in the network component, and their configurations are specified through variables.
* The storage component defines the object storage bucket that will store application and backup data, and the block storage volumes that will be attached to the VMs.

All the resources are managed by Terraform, which allows for easy provisioning, scaling, and management of the infrastructure.

Overall, this Terraform configuration with OVH cloud provider provides a scalable, secure, and highly available infrastructure for running applications.

* 1. Component diagram of provisioned resources:



The cluster resources were deployed to three regions. Namely, GRA, BHS, SBG. In each region, four instances were created. One of which will join the control plane, two will have the worker compute role and the last will have a second block device in raw format and will join the data storage backend. Various buckets from the object store will be provisioned and will serve as backup storage backends and high-speed data storage.

* 1. Conclusion

1. 4th sprint: Preliminary infrastructure setup
   1. Introduction
   2. Sprint backlog:

|  |  |  |  |
| --- | --- | --- | --- |
| EpicID | Epic | StoryID | Story |
| 1 | Setup and configuration of provisioned resources using ansible playbooks. | 1.1 | Writing resource-specific roles to setup the resources. |
| 1.2 | Configuring the written roles into playbook. |
| 1.3 | Starting the infrastructure setup process. |

* 1. UML design: activity diagram for infrastructure setup

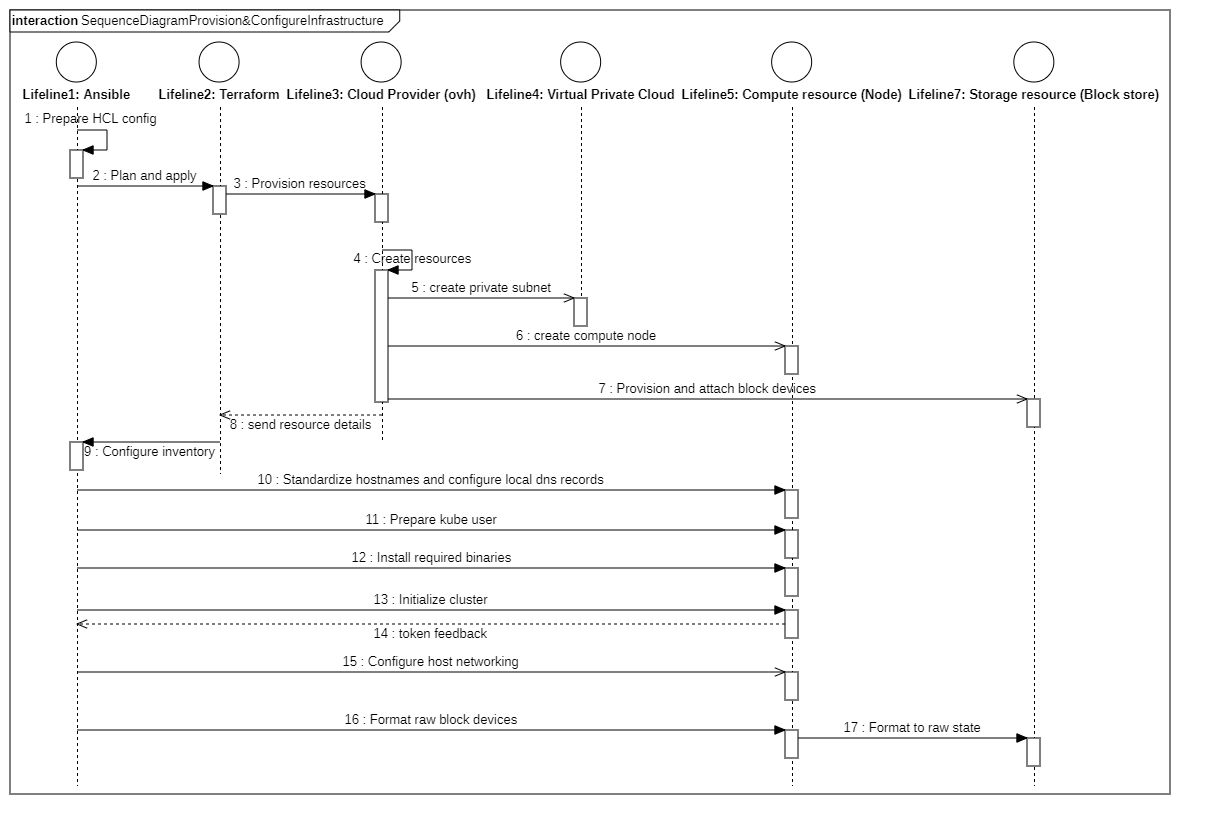
This activity diagram illustrates how Ansible and Terraform can work together to provision infrastructure resources and set up a PaaS environment.

Une image contenant texte

Description générée automatiquement

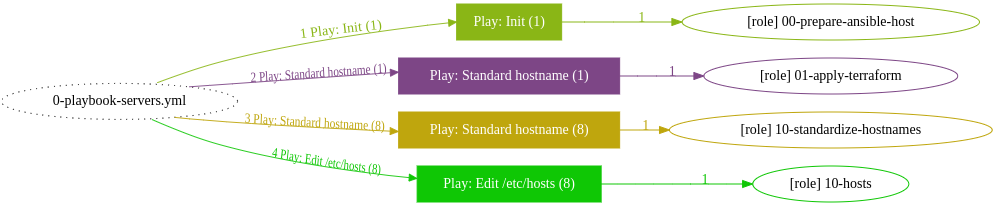
* 1. UML design: sequence diagram for infrastructure setup

The following is a sequence diagram that shows the flow of actions or events that occur during the process of setting up the infrastructure.



* The user initiates the infrastructure setup process by running starting an ansible playbook.
* The first ansible playbook sets up the host machine to run terraform. It invokes Terraform, which then communicates with the OVH API to provision the required resources.
* OVH API receives the request and creates the necessary resources such as virtual machines, block storage volumes, and network interfaces.
* Terraform receives the response from the OVH API and applies any necessary configurations to the provisioned resources.
* Ansible then takes over and configures the provisioned resources by installing the required packages, setting up the network configurations, and configuring the software applications.
* If everything is working as expected, a notification is sent via an office 360 webhook channel. The devSecOps team is notified that the infrastructure setup process is complete, and the system is ready for use.
  1. Playbook diagrams for resource setup
     1. Playbook 0: Setting up individual instances.

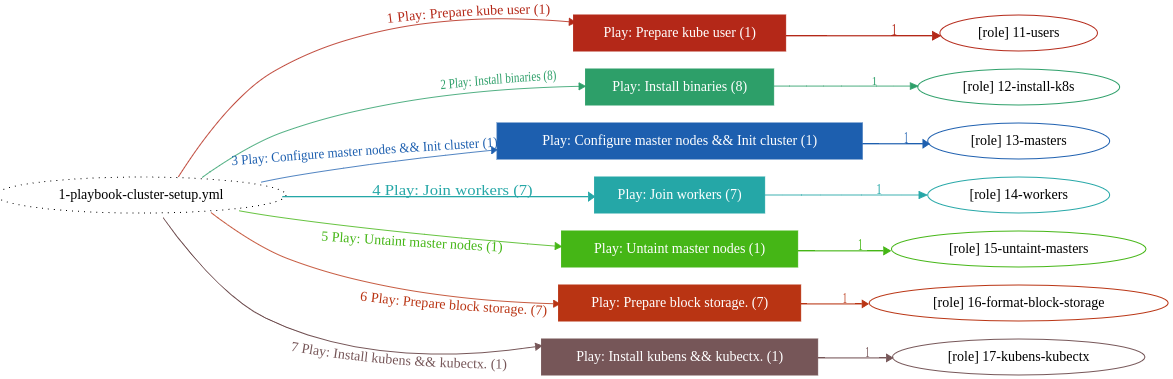
Next, we look at the playbook for setting up and standardizing the both the ansible host and the public cloud instances.



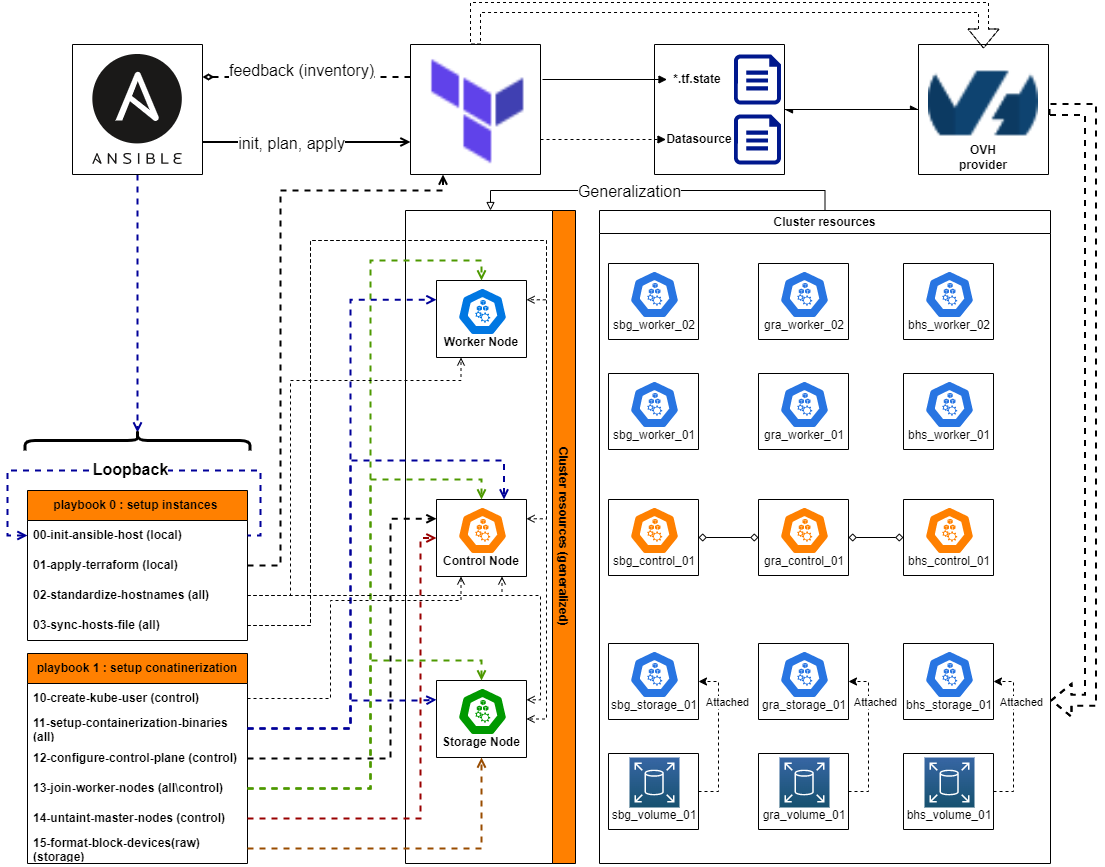
* + 1. Playbook 1: Orchestrated cluster setup.

For setting up the Kubernetes cluster on the provisioned infrastructure resources, the following step were automated using an ansible playbook:

* The playbook first sets up the necessary prerequisites on each node, such as disabling swap, disabling firewall, installing necessary packages, etc.
* Next, it initializes the first node from the control plane as a master and joins the others as secondary masters. This involves setting up the necessary configuration files, certificates, and keys for the Kubernetes control plane.
* After the master node is initialized, the playbook joins the worker nodes. This involves passing the appropriate configuration information, such as the API server address and token, to the worker nodes.
* Once all nodes are part of the cluster, the playbook sets up networking using the CNI plugin Calico. This involves deploying the necessary network configuration files and pods to allow communication between nodes.
* Finally, the playbook sets up any necessary add-ons or customizations to the cluster, such as untainting master nodes, formatting block volumes to raw, adding custom scripts to switch between namespaces and contexts easily.



A summary of the previous two diagrams is depicted in the following figure :



* 1. Conclusion

Overall, the playbook aims to automate the setup and configuration of the Kubernetes to prepare it to host the various PaaS services that will be seen next chapter, allowing for consistent and repeatable deployments of Kubernetes infrastructure.

1. 5th Sprint: Initial PaaS setup.
   1. Introduction

In this chapter, we will explore the process of setting up the Kubernetes-based PaaS. Our PaaS will include Traefik, a powerful ingress controller that provides load balancing and routing capabilities, MetalLB, a load balancer that simplifies network configuration, and Ceph, a distributed storage system that provides scalability and high availability.

We will use Ansible to automate the deployment of our PaaS environment.

* 1. Sprint backlog:

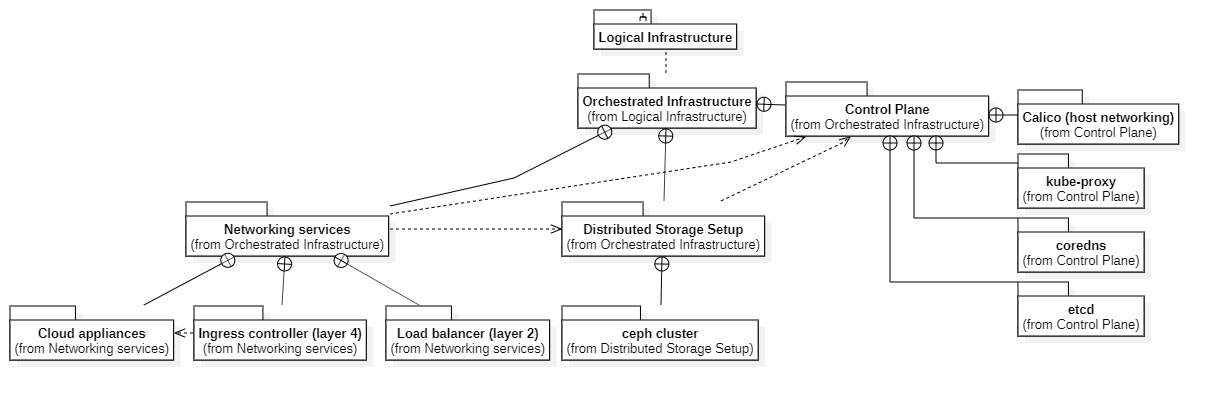
|  |  |  |  |
| --- | --- | --- | --- |
| EpicID | Epic | StoryID | Story |
| 1 | PaaS setup:  Setting the service groups for networking and storage. | 1.1 | Setting up the ingress controller (traefik). |
| 1.2 | Setting up the network level load balancer (metalLB). |
| 1.3 | Setting up the load balancer. |

* 1. UML design: package diagram for the initial PaaS setup

Here, the PaaS is the main package and contains all the other packages.

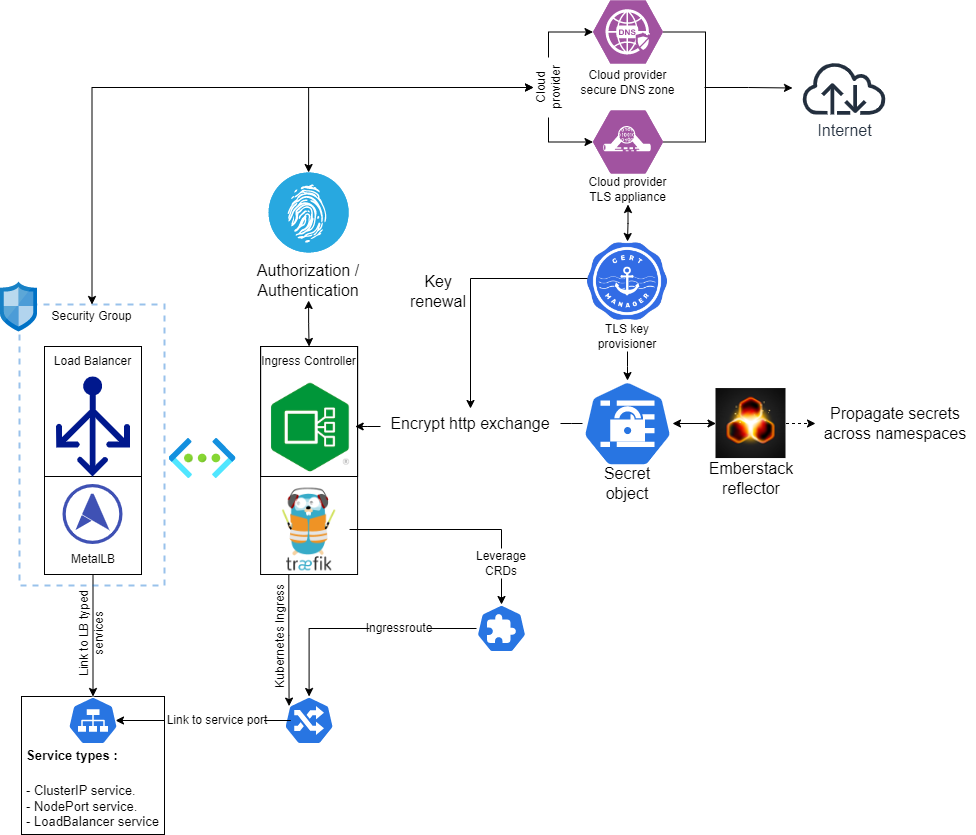
Within the PaaS package, there are three sub-packages: the Kubernetes control plane, Traefik/MetalLB, and Ceph. The control plane package contains all of the components necessary for running Kubernetes (e.g., etcd, kube-proxy, etc.), while the Traefik/MetalLB package contains the Traefik and MetalLB components used for ingress and load balancing, respectively.

Finally, the Ceph package contains all of the necessary components for running distributed storage using Ceph.



* 1. Networking services
     1. Overview on the networking services :

As illustrated in the following figure, Traefik, Cert-manager, and MetalLB work together to provide a secure and scalable way to route traffic inside Kubernetes using Services and IngressRoutes.



Taking into consideration the following setup :

1. Application workloads are exposed through services of different types:

* ClusterIP: access from within the cluster access.
* LoadBalancer: access from outside the cluster using the external ip allocated through the loadbalancer.
* NodePort: access from outside the cluster using the ip of the node on which the application container is running.

1. Cert-manager is configured as a cluster issuer and uses a “DNS challenge” to provision TLS certificates specific to every subdomain or domain.
2. Cert-manager listens to every declared “certificate” resource in order to process it and generate a valid “TLS certificate” that is then stored in a secret.
3. Cert-manager is configured to renew the generated certificates periodically.
4. Emberstack reflector is used to propagate secrets and configmaps across namespaces.

The flow is as follows:

1. Traffic enters from the Internet and is directed either to MetalLB for network level load balancing, or to the Traefik ingress controller for application level load balancing.
2. Traefik Proxy receives the traffic and routes it to the appropriate Kubernetes Service that is linked to a previously configured ingressroute, based on the domain name in the request.
3. The Service forwards the traffic to the appropriate Kubernetes Pod.
4. IngressRoutes are defined for workload, and Traefik uses these IngressRoutes to direct traffic to the appropriate Service.
5. Traefik is configured to use the TLS certificates generated by Cert-manager to terminate SSL/TLS traffic.
6. Traffic is routed to the correct Kubernetes Pods based on the domain name in the request.
7. The Kubernetes Pods process the requests and respond to the Traefik Proxy.
8. Traefik handles TLS termination using the secrets previously generated and sends the response back to the client through HTTPS.
   * 1. Secret provisioning and propagation:
        1. TLS secret provisioning:

In order to provide secure HTTPS communications between clients and the deployed workloads, valid TLS certificates need to be provisioned from our provider tls appliances. Our tool of choice is cert-manager.

Cert-manager provides us with a powerful and flexible way to manage TLS certificates in Kubernetes, enabling automatic provisioning and renewal of certificates from various certificate authorities, namely: “OVH Cloud Provider” and “Cloudflare”.

By automating certificate management, cert-manager helps improve the security and reliability of our Kubernetes services, while reducing the operational burden of manual certificate provisioning and management.

Leveraging the capabilities of cert-manager is summarized by the following steps:

1. Having setup the stock cert-manager deployment, we first create the “ClusterIssuer” Kubernetes resource. Here we showcase a sample manifest:



The cluster issuer manifest contains the credentials cert-manager uses to authenticate with our domain registrar (OVH). It needs these credentials to perform a DNS-01 challenge mechanism to validate domain ownership and issue TLS certificates.

1. Next, for every domain or subdomain a “Certificate” resource is created. This certificate resource is then processed by cert-manager and results in a secret containing a TLS certificate. The following is a certificate declaration for wildcard subdomain:



* + - 1. Secret propagation

Emberstack Reflector, an open-source tool that provides a simple and efficient way to propagate Kubernetes secrets across multiple namespaces and clusters. We are using it to automatically replicate a secret from a source namespace to one or more targets, ensuring that all applications that require access to the secret can access it easily.

Having deployed this tool, adding the following annotation to any secret or configmap will replicate it to every namespace:

* + 1. Layer 7 load balancing: application level

Traefik, our ingress controller of choice, functions both as a reverse proxy as well as a load balancer for the deployed workload which are then accessed through secure HTTPS.

In the following section we provide, in detail, how traefik is plugged into the cluster and how it is leveraged:

* Deploy traefik as a “DaemonSet”. Deploying traefik as a daemonset allows us to spawn instances of it in every node and thus take full advantage of the internet bandwidth since the workload is distributed across all cluster instances. The following Kubernetes resources are created in order to setup it up:

1. “traefik\_rbac.yml”: defines Roles and RoleBindings to grant users or groups specific permissions for accessing Traefik's API and resources.
2. “traefik\_crds.yml”: allow us to extend the Kubernetes API and define custom resources that enable more advanced features and configuration options such as “IngressRoutes” for services, “TLSStores”, “Middlewares”, etc.
3. “traefik\_service\_account.yml”:
4. “traefik\_pvc.yml”:
5. “traefik\_daemonset.yml”:
6. “traefik\_ingressclass.yml”:
7. “traefik\_tls\_store.yml”:
   * 1. Layer 4 load balancing: network level

In this level, load balancing is mostly related to the network, IP addresses, network address translation ( NAT ), and packets.

* + - 1. Using the built in Kubernetes capability : NodePort services

The NodePort Service exposes a deployment or a set of pods by mapping a specific port on each node in the cluster to the port of the service.

This allows external clients to access the service by connecting to any of the nodes in the cluster.

Here's how NodePort Services work:

1. A NodePort Service is defined as a Kubernetes object with the “type: NodePort” field in the YAML manifest.
2. When a NodePort Service is created, Kubernetes assigns a high port number (in the range 30000-32767) to the service.
3. Kubernetes then maps the assigned port to the port of the service, which can be any valid port number.
4. The NodePort Service is then exposed on all nodes in the cluster, allowing external clients to access the service by connecting to any of the nodes on the assigned port.
5. Traffic received on the assigned port is forwarded to the service's target port, which is the port on which the service is listening.
6. Kubernetes ensures that the traffic is load balanced across all pods that are part of the service.
   * + 1. Using metalLB for load balancing : LoadBalancer services

MetalLB is used to provide network load balancing for services running in the cluster. Here's how MetalLB works in our Kubernetes cluster:

* MetalLB is deployed in the cluster using the following manifests:

1. “metalLB\_configmap.yml”: a configmap resource containing the ip address pool that metalLB is allowed to use to allocate to loadbalancer-type services. As a sample, here is the code snippet for this configmap, the rest will be added in the index section of this document :



1. “metalLB\_rbac.yml”: a method of providing fine-grained access control in Kubernetes. RBAC is a security mechanism that restricts access to resources and operations based on the roles assigned to users or service accounts in the cluster.
2. “metalLB\_service\_account.yml”: an identity that is used by a pod or a set of pods to access the Kubernetes API server and other resources in the cluster.
3. “metalLB\_controller.yml”: a deployment that watches for changes to load balancer services in the cluster and dynamically assigns IP addresses to those services as needed.
4. “metalLB\_speaker.yml”: a daemonset that spawns a pod in every node in the cluster it is responsible for announcing the allocated IP addresses for load balancer services.
5. “metalLB\_pod\_security\_policy.yml”: It allows the definition of a set of security policies that pods must comply with to be scheduled and run on nodes in the cluster. In the context of MetalLB, this PSP is used to enforce security measures for the MetalLB components, including the MetalLB controller and the MetalLB speaker.

* Exposing a workload using the deployed metalLB loadbalancer is as follows:

1. After creating the application deployment, we create a Kubernetes Service that exposes this workload using the “type” field of the Service to “LoadBalancer”.
2. Once the Service is created, MetalLB will allocate an IP address from the configured pool of addresses and assign it to the Service.
3. We then use this IP address to access your workload from outside the cluster.

* This is a sample service of the “LoadBalancer” type:

Une image contenant texte, capture d’écran, télévision, écran

Description générée automatiquement

1. 6th Sprint:
2. 7th Sprint:
3. 8th Sprint:
4. 9th Sprint: