Ref. Ares(2022)4186122 - 06/06/2022

Tailored IoT & BigData Sandboxes and Testbeds for Smart, Autonomous and Personalized Services in the European Finance and Insurance Services Ecosystem

D6.6 – Tools and Techniques for Tailored Sandboxes and Management of Datasets - III

Revision Number	3.0
Task Reference	T6.2 - T6.3
Lead Beneficiary	HPE
Responsible	Alessandro Mamelli
Partners	Participating partners in Task according to DOA
Deliverable Type	Report (R)
Dissemination Level	Public (PU)
Due Date	2022-05-31
Delivered Date	2022-06-06
Internal Reviewers	RRD - UPRC
Quality Assurance	CCA
Acceptance	WP Leader Accepted and Coordinator Accepted
EC Project Officer	Beatrice Plazzotta
Programme	HORIZON 2020 - ICT-11-2018
	This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no 856632

© INFINITECH Consortium

Contributing Partners

Partner Acronym	Role [1]	Author(s) [2]
HPE	Lead Beneficiary	Alessandro Mamelli – Domenico Costantino – Andrea Toro
LXS	Beneficiary	Pavlos Kranas
INNOV	Beneficiary	George Fatouros
UPRC	Beneficiary	Georgios Makridis
ENG	Beneficiary	Domenico Messina
RRD	Internal Peer Review	Stephanie Jansen - Kosterink
UPRC	Internal Peer Review	Dimitrios Kotios
CCA	Quality Assurance	Paul Lefrere

Revision History

Version	Date	Partner(s)	Description
0.1	2022-03-20	HPE	ToC Version
0.2	2022-04-20	HPE	Updated ToC Version with requested contributions
0.3	2022-04-29	HPE and contrib. partners	1st round of contributions
0.4	2022-05-13	HPE and contrib. partners	2nd round of contributions
0.5	2022-05-20	HPE and contrib. partners	3rd round of contributions
1.0	2022-05-27	HPE	First Version for Internal Peer Review
2.0-2.1	2022-06-02	HPE - RRD – UPRC - CCA	Internal Peer Review and version for Quality Assurance - Review by the Quality Manager
3.0	2022-06-06	HPE	Version for Submission

Executive Summary

Within INFINITECH Work Package 6 “Tailored Sandboxes and Testbeds for Experimentation and Validation”, this document describes the results of Tasks T6.2 “Mechanisms and Tools for Tailored Sandboxes Provision and Configuration” and T6.3 “Integrated Management of Testbeds' Datasets” achieved during the third phase of the project and provides the third version (D6.6) of this deliverable (out of the three foreseen in the whole of the work plan for WP6, i.e. the final version).

With respect to the general WP6 objectives, the deliverable mainly focuses on two of them, i.e.:

i. To provide tools and techniques for creating tailored sandboxes based on the selection of proper INFINITECH data assets, technological & regulatory building blocks.
ii. To provide a mechanism for integrated management of testbeds’ datasets, based on a continuous integration approach.

The achieved results have provided key contributions for the fulfilment of the 2nd major WP6 milestone (MS14 – Second Version of ALL Sandboxes & Testbeds Available – achieved in M24 of the project) and also of the 3rd major WP6 milestone (MS17 – Final Version of ALL Sandboxes & Testbeds Available – foreseen for M33 of the project), and provides the third release of the proposed tools and techniques.

This release includes additional and enhanced features with respect to the previous one: within them, it is worth mentioning the refined Blueprint guidelines for the “INFINITECH Way” deployments of project pilots and technologies and their concrete implementation as three of the official INFINITECH pilots.

The document is the accompanying textual specification of the other major deliverable result: the third and final release of the proposed tools and techniques integrated and deployed into the INFINITECH blueprint reference testbed setup, designed and implemented upon two of the target INFINITECH infrastructures.

The document and the developed INFINITECH blueprint reference testbed setup constitutes the overall deliverable output.

As consistently done since the beginning of WP6 activities, the work has been carried out in close cooperation and coordination with the other INFINITECH WP6 tasks and work packages 2-3-4-5 tasks and partners, taking into account and integrating the delivered results and concepts (e.g. the INFINITECH Reference Architecture proposed by WP2) in a coherent and uniform manner. Moreover, it has taken into account the feedback from the 2nd round of the INFINITECH work package dedicated to the Large Pilots’ Operations and Stakeholders’ Evaluation of the proposed Financial and Insurance Services (WP7).

By the project’s end, the overall progress of such WP6 tasks will be one of the major drivers of the upcoming 3rd round of the WP7 Pilots’ Operations and Stakeholders’ Evaluation.

Index

1 Introduction 10
1.1 Objective of the Deliverable 10
1.2 Insights from other Tasks and Deliverables........ 11
1.3 Structure 11
2 Relation to the general INFINITECH Reference Architecture 13
2.1 INFINITECH Reference Architecture 13
3 Tools and techniques for Testbeds and Sandboxes 16
3.1 Containers benefits 16
3.2 Microservices approach 17
3.3 Kubernetes containers orchestration........ 18
3.3.1 Kubernetes architecture........ 18
3.4 INFINITECH Testbeds 20
3.5 INFINITECH Sandboxes 21
4 Tools and techniques for Management of Datasets 23
4.1 Data Sources and Data Access........ 23
4.1.1 Static data ingestion 24
4.1.2 Dynamic data ingestion 24
4.1.3 IoT Streaming........ 24
4.1.4 Direct access to on-premise data sources........ 25
4.1.5 Blockchain data access 25
4.1.6 Third Party data access........ 25
4.2 Datasets management from the Blueprint reference environment perspective 26
5 Overview of the INFINITECH blueprint reference testbed 29
5.1 Development view........ 29
5.2 Deployment view........ 32
5.2.1 Creation of the EKS INFINITECH cluster (Blueprint) 32
5.2.2 Namespace, Network and Quote policies 40
5.2.3 EKS for Kubeflow environment........ 42
5.2.4 How to recreate the blueprint testbed for a specific INFINITECH Pilot 47
6 Blueprint guidelines for the “INFINITECH way” deployments of project pilots and technologies........ 49
6.1 Guidelines overview 49
6.2 Building an INFINITECH component 53
6.3 Building an image using pre-compiled libraries/binaries........ 59
6.3.1 Building an image using pre-compiled libraries/binaries........ 59
6.3.2 Building an image using a package container........ 60
6.3.3 Building a pilot-specific component 64
6.4 Deploy a pilot use case in a Sandbox........ 69

6.4.1 Preparing the deployment using Kubernetes ... 75
6.4.2 Deploying using Kubernetes ... 81
6.5 MLOps Integration ... 83
6.5.1 Introduction to Kubeflow Pipelines ... 84
6.5.2 How to build a Kubeflow Pipeline ... 86
6.5.3 End to End Blueprint Training components ... 88
6.5.4 The serving component ... 102
6.5.5 Running the pipeline ... 104
7 Implementation of the Blueprint guidelines to INFINITECH blueprint pilots ... 112
7.1 Blueprint environment for Pilot #5b: Business Financial Management tools delivering Smart Business Advice ... 112
7.1.1 Pilot Objectives ... 112
7.1.2 Pilot Workflow ... 112
7.1.3 Blueprint reference testbed implementation ... 114
7.2 Blueprint environment for Pilot #2: Real-time risk assessment in Investment Banking ... 118
7.2.1 Pilot Objectives ... 118
7.2.2 Pilot Workflow ... 118
7.2.3 Blueprint reference testbed implementation ... 119
7.3 Blueprint environment for Pilot #10: Platform for Real-time cybersecurity analytics on Financial Transactions' BigData ... 121
7.3.1 Pilot Objectives ... 121
7.3.2 Pilot Workflow ... 122
7.3.3 Blueprint reference testbed implementation ... 123
8 Conclusions ... 127
9 Appendix A: Literature ... 129

List of Figures

Figure 1 – Task dependencies in WP6 ... 11
Figure 2 – INFINITECH Reference Architecture logical view ... 14
Figure 3 – VM vs Container ... 16
Figure 4 – Container kernel properties ... 17
Figure 5 – Monolithic vs Microservice ... 17
Figure 6 – Kubernetes Architecture ... 19
Figure 7 – Kubernetes POD ... 20
Figure 8 – Testbed ... 20
Figure 9 – Testbeds and Pilots ... 21
Figure 10 – Sandboxes in a dedicated Testbed ... 21
Figure 11 – Sandboxes in a shared Testbed ... 22
Figure 12 - CI/CD workflow ... 31
Figure 13 – EKS Control Plane deployment [27] ... 32
Figure 14 – INFINITECH blueprint reference testbed ... 42

Figure 15 – Blueprint environment recreation ways ... 48
Figure 16 – Gitlab login ... 50
Figure 17 – List of projects visible to a specific user ... 50
Figure 18 – The INFINITECH groups ... 52
Figure 19 – The Pilot groups ... 52
Figure 20 – Projects under the Blueprint group ... 53
Figure 21 – Gitlab create a new project ... 54
Figure 22 – Gitlab setting new project details ... 54
Figure 23 – Gitlab add gitlabinfinitech user to access the new project ... 54
Figure 24 – Git clone an existing project ... 55
Figure 25 – Defining the Jenkins file ... 56
Figure 26 – Checking the webhooks of the project ... 57
Figure 27 – Manually trigger a CI pipeline ... 57
Figure 28 – Checking the status of a CI pipeline ... 57
Figure 29 – Specific pipeline details ... 58
Figure 30 – Checking the logs of a CI pipeline ... 58
Figure 31 – Infinistore project files ... 59
Figure 32 – Maven project files ... 61
Figure 33 – Project ID ... 62
Figure 34 – mvn command output ... 63
Figure 35 – Gitlab project files ... 63
Figure 36 – Overall architecture of the Pilot-Ref solution ... 65
Figure 37 – Gitlab list of available groups ... 66
Figure 38 – Creating the pilot-specific backend project ... 66
Figure 39 – Gitlab Project users and roles ... 67
Figure 40 – Our pilot-ref/backend project ... 67
Figure 41 – Setting Jenkins for our pilot-ref/backend project ... 68
Figure 42 – Checking the CI pipeline for our pilot project ... 69
Figure 43 – Checking the logs of the CI pipeline of our pilot project ... 69
Figure 44 – Pilot-ref Gitlab project ... 76
Figure 45 – Infinistore StatefulSet manifest ... 77
Figure 46 – Infinistore Service manifest ... 78
Figure 47 – Backend container spec ... 78
Figure 48 – lx-kafka-configmap Configmap manifest ... 79
Figure 49 – lx-kafka StatefulSet manifest ... 79
Figure 50 – Lx-kafka properties file ... 80
Figure 51 – postscript.sh script ... 81
Figure 52 – Jenkins deployment stages ... 82
Figure 53 – Jenkins build log ... 82
Figure 54 - Pipeline Graph Representation ... 84
Figure 55 - Docker container component and function based component ... 85
Figure 56 - Components Input/Output behind the scenes ... 85
Figure 57 - Simple Pipeline Example ... 86
Figure 58 - Blueprint Pipeline Graph ... 89
Figure 59 - Pipeline python file ... 90
Figure 60 - SeldonDeployment pipeline code ... 104
Figure 61 - Kubeflow Dashboard main page ... 108
Figure 62 - Kubeflow Dashboard Upload a pipeline ... 109
Figure 63 - kubeflow choose a pipeline yaml file ... 109
Figure 64 - Uploaded pipeline ... 110
Figure 65 - Choose an experiment to run a pipeline ... 110
Figure 66 - Start a pipeline ... 111
Figure 67 - Pipeline running view ... 111

Figure 68 - Pilot #5b workflow 114
Figure 69 - Prototype_v3 components diagram 116
Figure 70 - Pilot #5b blueprint reference architecture 117
Figure 71 - Pilot #2 workflow 119
Figure 72 - Pilot #2 blueprint reference architecture 121
Figure 73 - Pilot #10 workflow 122
Figure 74 - Pilot 10 blueprint reference architecture 123
Figure 75 - Pilot #10 speed layer 125
Figure 76 - Pilot #10 batch supervised layer 125
Figure 77 - Pilot #10 batch unsupervised layer 126

List of Tables

Table 1 - Abbreviations 8
Table 2 - Prototype_v3 vs Blueprint_v3 components 115
Table 3 - Prototype vs Blueprint components 119
Table 4 - Prototype vs Blueprint components 123
Table 5 - Mapping of DoA/Tasks Objectives with Deliverable achievements 127
Table 6 - Mapping of DoA/Tasks KPIs with Deliverable achievements 128

Abbreviations/Acronyms

Table 1 - Abbreviations

API	Application Programming Interface
AI	Artificial Intelligence
AWS	Amazon Web Services
AWS EBS	Elastic Block Store
AWS EKS	Elastic Kubernetes Service
AWS ELB	Elastic Load Balancer
AWS KMS	Key Management Service
BOC	Bank Of Cyprus
CA	Certificate Authority
CICD	Continuous Integration Continuous Development
CNI	Container Network Interface
DevOps	Development and Operations
DNS	Dynamic Name Resolution
ERP	Enterprise resource planning
HCL	HashiCorp Configuration Language
HTAP	Hybrid Transactional and Analytical Processing
IAC	Infrastructure As Code
IAM	Identity and Access Management
IoT	Internet Of Things
K8s	Kubernetes
LDAP	Lightweight Directory Access Protocol
ML	Machine Learning
MLOps	Machine Learning and IT Operations
MVN	MaVeN
PoC	Proof of Concept
PV	Persistent Volume
PVC	Persistent Volume Claim
REST	REpresentational State Transfer
SME	Small/Medium Enterprise
SSH	Secure Shell Protocol

TGZ	TarGZip
UI	User Interface
URL	Uniform Resource Locator
YAML	YAML Ain't Markup Language
VPC	Virtual Private Cloud

1 Introduction

INFINITECH is developing and validating BigData, Internet of Things (IoT) and Artificial Intelligence (AI) technologies for the finance and insurance sectors. In this direction, the project is advancing the state of the art in several technological development areas such as infrastructures for integrated, incremental and real-time analytics, semantic interoperability, as well as decentralized information sharing between stakeholders of the sector. Furthermore, the project designs and implements a range of pilots and use cases that aim at validating these technologies in real-life scenarios of the sectors.

Within INFINITECH, WP6 – Tailored Sandboxes and Testbeds for Experimentation and Validation – aims to achieve the following objectives:

i. To analyse the existing testbeds and specify their enhancements and upgrades.
ii. To provide tools and techniques for creating tailored sandboxes based on the selection of proper INFINITECH data assets, technological & regulatory building blocks.
iii. To provide a mechanism for integrated management of testbeds' datasets, based on a continuous integration approach.
iv. To establish the 10+2 testbeds for experimentation and validation, including all relevant sandboxes.
v. To ensure continuous technical support for all testbeds, while establishing processes for certification/standardization of digital finance/insurance solutions.

1.1 Objective of the Deliverable

This document describes the final results of INFINITECH WP6 Tasks T6.2 "Mechanisms and Tools for Tailored Sandboxes Provision and Configuration" and T6.3 "Integrated Management of Testbeds' Datasets" and provides the third version (D6.6) of the deliverable (out of the three foreseen in the whole of the work plan for WP6, i.e. the final version).

With respect to the general WP6 objectives mentioned before, this deliverable mainly focuses on objectives ii. and iii.

The results that have been achieved during the work have provided key contributions for the fulfilment of the 2nd major WP6 milestone (MS14 – Second Version of ALL Sandboxes & Testbeds Available – achieved in M24 of the project) and also of the 3rd major WP6 milestone (MS17 – Final Version of ALL Sandboxes & Testbeds Available – foreseen for M33 of the project) and provides the third release of the proposed tools and techniques.

The document is (on purpose) self-contained, i.e. there's no need to read the previous versions (D6.4 and D6.5) to get a full understanding of the updated contents and achieved results. In particular, this release includes additional and enhanced features with respect to the previous ones, which can be summarized as:

- General updates and enhancements of the proposed tools and techniques (in Chapter 4);
- General updates and enhancements of the INFINITECH blueprint reference testbed (in Chapter 5)
- General updates and enhancements of the Blueprint guidelines for the "INFINITECH way" deployments of project pilots and technologies (in Chapter 6)
- Concrete implementation of the "INFINITECH way" blueprint guidelines described in the previous Chapter in the context of three (they were two in the previous release) of the official INFINITECH pilots, the so-called "INFINITECH blueprint pilots" (in Chapter 7)

The document is the accompanying textual specification of the other major result of the deliverable: the third and final release of the proposed tools and techniques, which have also been integrated and deployed into the INFINITECH blueprint reference testbed package, designed and implemented using two of the target INFINITECH infrastructures.

The document and the implemented INFINITECH blueprint reference testbed setup constitute the overall deliverable output.

1.2 Insights from other Tasks and Deliverables

The following picture (Figure 1) depicts the interconnections between all the tasks in Work Package 6:

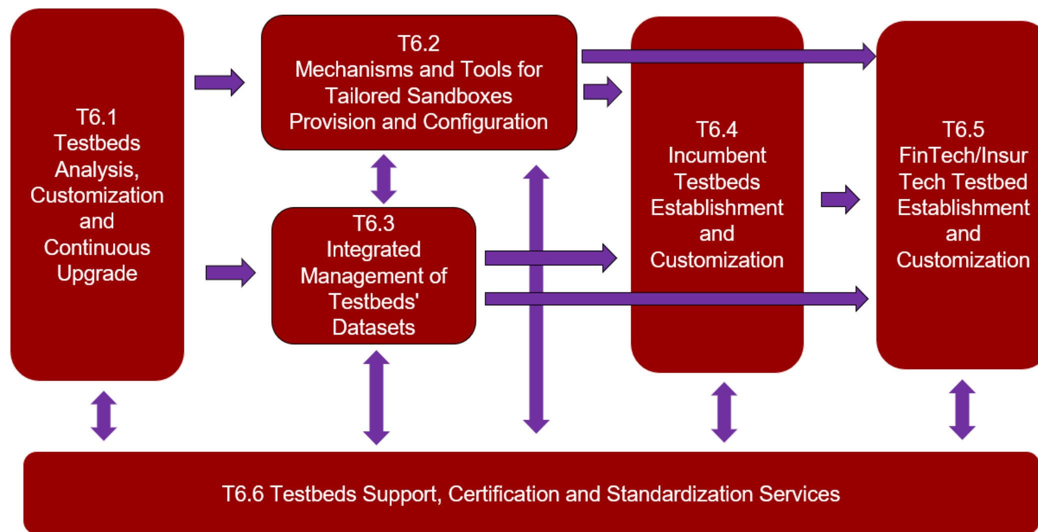


Figure 1 – Task dependencies in WP6

An important input within WP6 for these tasks are the outcomes of Task 6.1 and its third available deliverable "D6.3 – Testbeds Status and Upgrades – III", which mainly contains an updated analysis of Testbeds infrastructure (hardware & software) that each Pilot will use for Pilot development and the relevant Sandboxes that will be hosted.

Moreover, the outcomes of T6.2 and T6.3 will continue to feed the work of T6.4 and T6.5 for the deployment and customization in the target field infrastructure of the proposed tools and techniques for testbeds, sandboxes and management of testbeds' datasets and finally, will also continue to feed the work of T6.6 that specifies and implement processes for certifying and standardizing digital finance/insurance solutions in the project's tailored sandboxes and testbeds.

Furthermore, looking to INFINITECH outside the WP6 border (and as consistently done since the beginning of WP6 activities) the work done in T6.2 and T6.3 embodies a strict and continuous collaboration and alignment with the INFINITECH work packages 2-3-4-5 tasks and partners, towards the integration of the delivered outcomes (e.g. the INFINITECH Reference Architecture proposed by WP2 or datasets management concepts by WP3) in a consistent way. Moreover, it has taken into account the feedback coming from the 2nd round of the INFINITECH work package dedicated to the Large Pilots' Operations and Stakeholders' Evaluation of the proposed Financial and Insurance Services (WP7).

In the end, the overall progress of such WP6 tasks will be one of the major drivers of the upcoming 3rd round of the WP7 Pilots' Operations and Stakeholders' Evaluation.

1.3 Structure

The deliverable consists of the following chapters:

- Chapter 2 "Relation to the general INFINITECH Reference Architecture" provides a summary of the INFINITECH Reference Architecture approach, and how the work done in the D6.6 deliverable relates to it in a coherent way
- Chapter 3 "Tools and techniques for Testbeds and Sandboxes" describes the approaches and technologies leveraged to implement the Testbeds and Sandboxes concepts within the INFINITECH project
- Chapter 4 "Tools and techniques for Management of Datasets " describes the tools and techniques that leveraged for the management of datasets within the INFINITECH project

- Chapter 5 ”Overview of the INFINITECH blueprint reference testbed” describes the intermediate design and implementation of the INFINITECH blueprint reference testbed, through the actual realization (with a full compliance) of the INFINITECH Reference Architecture Development and Deployment views.
- Chapter 6 “Blueprint guidelines for the “INFINITECH way” deployments of project pilots and technologies describes the guidelines for the partners about how to organize their artifacts in the project’s code repository, and make use of the CI/CD pipelines to i) make their solutions available to other partners and ii) automate the deployment of a pilot.
- Chapter 7 “Implementation of the Blueprint guidelines to INFINITECH blueprint pilots” provides the description of the actual and concrete implementation of the “INFINITECH way” Blueprint guidelines described in the previous chapter to three of the official INFINITECH pilots.
- Chapter 8 ”Conclusions” summarizes the results of the work done in the deliverable and the next steps foreseen for the related tasks.
- Chapter 9 “Appendix A: Literature” provides details of all the cited work.

2 Relation to the general INFINITECH Reference Architecture

This chapter illustrates a summary of the updated INFINITECH Reference Architecture approach, described in depth in deliverable "INFINITECH D2.15 – Reference Architecture – III", and how the work done in the D6.6 deliverable relates to it in a coherent way.

2.1 INFINITECH Reference Architecture

The INFINITECH Reference Architecture (IRA) leverages the "4+1" architectural view model as the methodology to cover all the aspects of the different problems the Project and the Consortium want to address.

The "4+1" architectural view model [3] is a methodology to design an architecture for a software platform, having the main capacity of describing it from 5 concurrent "views".

These views represent the different stakeholders who could deal with the platform and the architecture, from the management, development and user perspectives. The four main views which facilitate the definition and design of the architecture are the logical, process, development and physical ones, while the "+1" view is represented by the use cases or scenarios, thus making this model an abstraction of the developed solution/platform and the basis for the development.

Below, the meaning of the different views is explained:

- **Logical view:** it represents the range of functionalities or services that the system provides to the end users, and can be shown as block diagrams.
- **Process view:** it represents the system processes and data flows and how the different processes and building blocks communicate between each other, with details of the runtime behaviour of the system.
- **Development view (or Implementation view):** it illustrates the software management aspects of the system, from the programmer's point of view.
- **Physical view (or Deployment view):** it describes the lower levels of the architecture, dealing with the physical infrastructures where the software components are deployed and run, and the physical connections between them. Deployment diagrams can represent it.
- **Scenarios:** these represent the architecture of the system explained from the end user's point of view too, with the description of use cases. The use cases or "scenarios" aim at describing some possible functioning situations of the system and interactions between different components. The validation and assessment of functionalities are usually performed using this view.

The IRA presented in Figure 2 defines *layers* as a way to logically group components.

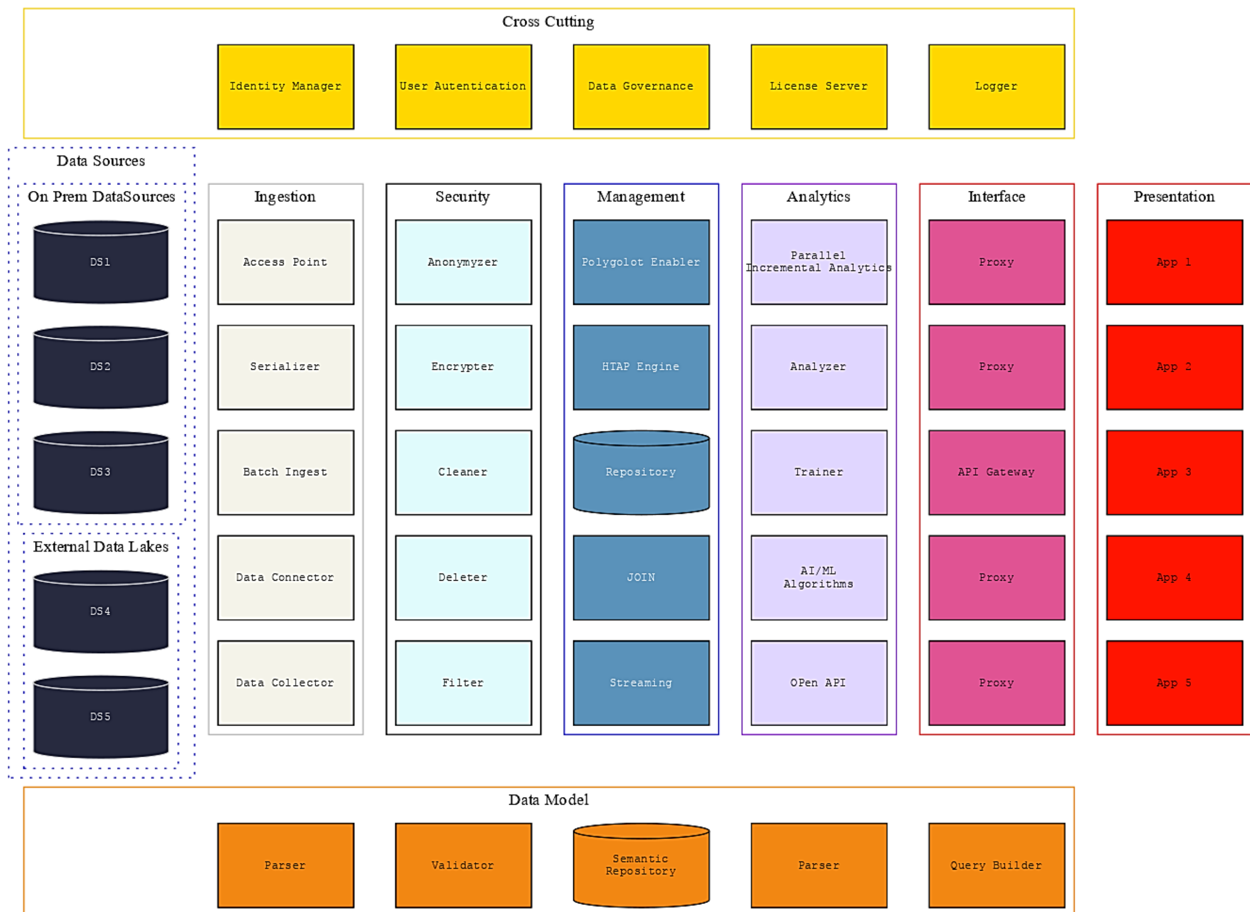


Figure 2 – INFINITECH Reference Architecture logical view

The identified layers are:

- **Data Sources:** at the infrastructure level, there are the different sources of data (database management systems, data lakes holding non-structural data, etc.).
- **Ingestion:** a layer of data management usually associated with data import, semantic annotation and filtering from data sources.
- **Security:** a layer for management of the clearance of data for security, anonymization, cleaning of data before any further storage or processing.
- **Management:** a layer responsible for the data management aspects, including the persistent storage in the central repository and the data processing enabling advanced functionalities such as Hybrid Transactional and Analytical Processing (HTAP), polyglot capabilities, etc.
- **Analytics:** a layer for the AI/ML/DL components.
- **Interface:** a layer for the definition data to be produced for clients (e.g., user interfaces, etc.).
- **Cross Cutting:** a layer with service components that provide functionalities orthogonal to the data flows (e.g., Authentication, Authorization, Accounting, etc.).
- **Data Model:** a cross-cutting layer for modelling and semantics of data in the data flow.
- **Presentation/Visualization:** a layer usually associated with the presentation applications (desktop, mobile apps, dashboards and the like).

It should be noted that the IRA does not impose any pipelined or sequential composition of nodes. However, it is recommended to consider each different layer and the relative components to solve specific problems of the use case.

In the end, the major D6.6 document purpose and its relation to IRA with a full alignment to its building blocks, is to describe in particular how the IRA Development and Physical views have been tackled and

designed by the Consortium in terms of the concrete specification and realization of the fundamental and target INFINITECH concepts of Testbeds, Sandboxes and Datasets management, as well as related tools and techniques for their effective setup and deployment in the INFINITECH pilots and validation scenarios.

3 Tools and techniques for Testbeds and Sandboxes

This chapter describes the tools and techniques that will be leveraged to implement the testbeds and sandboxes concepts within the INFINITECH project, considering that the IRA is designed leveraging a paradigm based on a **microservices** [4] architecture implementation, with services interacting among them through REST APIs.

Key pillars for the implementation and deployment of a microservices based architecture are the **containers** technology [5] and its leading open-source containers orchestration solution, i.e. **Kubernetes** [6]. Therefore, in order to understand such methodological and technological choices for the realization of the IRA, some key concepts related to containers, microservices and Kubernetes, and how their appearance in the IT environments has deeply changed the software development approaches, are presented.

Finally, the details of how the concrete usage of such technologies benefits and enables the definition of the INFINITECH testbeds and sandboxes concepts are also provided.

3.1 Containers benefits

In the last few years there has been a strong transformation in conceptualising benefits of containers, similar to what happened in the early 2000s with the advent of virtualization, due to containers' spread which led to rethinking both how to manage the infrastructure and how to design and build the applications.

The introduction and the wide spread of containers concept and technologies have made it possible to improve the computational management of infrastructures, thanks to the possibility of removing the overhead generated by the use of the hypervisor (integrated software that allows users to virtualize the HW resources of a server and make them available among several applications) and through the usage of the functionalities already available within the Linux OS kernel, as in Figure 3.

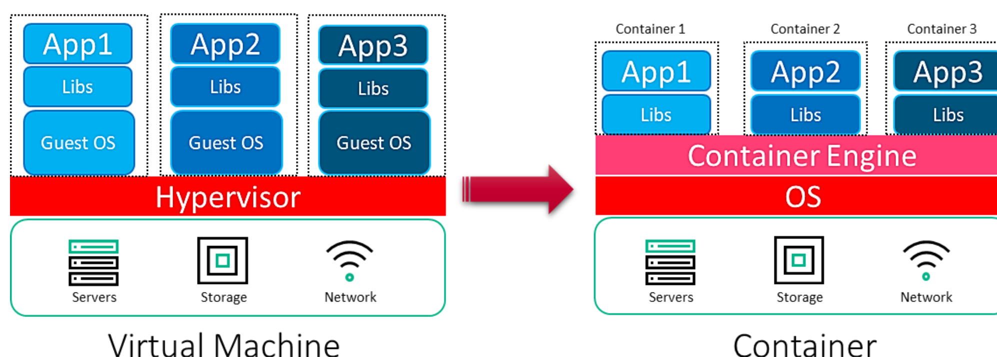


Figure 3 – VM vs Container

The container engine, to date the most-used one is Docker [7], takes advantage of Linux-Control groups and Linux-Namespaces to run containers like VMs isolated from each other:

- **Linux-Control groups**: Allow each container to get its fair share of memory, CPU, disk I/O and Network stack. At the same time a single container cannot bring the system down by exhausting one of those resources.
- **Linux-Namespaces:** Provide the possibility to isolate the processes running into a container from processes running in another container or in the host system.

In the end, a container is a package that contains code, system libraries, dependencies and software tools.

An example is given in Figure 4.

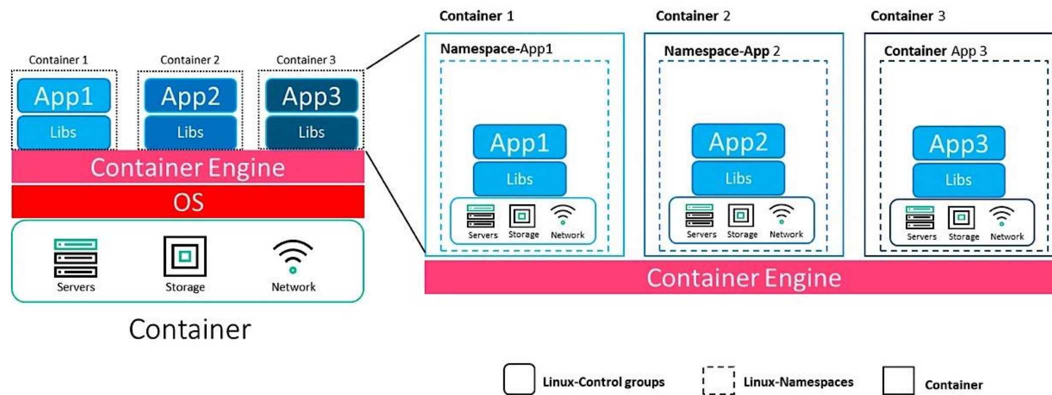


Figure 4 – Container kernel properties

The main advantages to using containers to respect VMs can be summarized in the following macro points:

- **Size**: A container is small.
- **Overhead**: No full OS is required.
- **Speed**: Boot time is faster.
- **Scaling**: Real time provisioning.

At the same time, to take advantage of containers it is necessary to rethink and redesign the currently monolithic applications as microservices-based applications.

3.2 Microservices approach

The spread of containers led, as already mentioned, to the necessity to change the approach of the developers in the creation of an application, moving from design of monolithic applications, where the various components (generally UI, Business logic and Data-layer) were strongly coupled among them, to microservices applications where the various components (i.e. microservices) are decoupled from each other (see Figure 5)

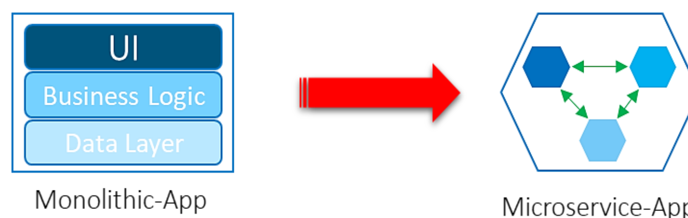


Figure 5 – Monolithic vs Microservice

This methodological change has both advantages and disadvantages. The advantages can be summarized as:

- **Simple to develop**: Each microservice is independent and small.
- **Simple to upgrade**: Since each microservice is independent, it's possible to upgrade each component independently.
- **Simple interaction**: Each microservice communicates with each other through well-defined and standard interfaces (API).
- **Simple to scale**: Each microservice can be scaled independently.
- **Scale on demand**: It is possible to run multiple microservices behind a load balancer to scale an on-demand request.

The disadvantages of the methodological change can be summarized as:

- **Complexity**: Splitting an application into multiple independent microservices increases the complexity of the deployment process.
- **Monitoring**: Monitoring each microservice requires having many metrics and logs to manage it.
- **Performance**: All communications occur on the network so these are slower than memory communications.
- **Debugging**: a Monolithic application is much easier to debug and test due to the fact it is composed of single indivisible units.

Our approach for a rapid iterative development of a microservice-based software infrastructure builds upon a widely-used methodology like **DevOps**, as will be described in section 5.1.

3.3 Kubernetes containers orchestration

The arrival on the market of containers and related microservices-based applications, on the one hand enabled applications that quickly scale according to the requirements and that could be easily updated, on the other hand meant that software previously managed as a single indivisible piece was split into several dozen microservices (containers), making it more difficult to manage them.

In this context, the necessity to develop a tool that was able to manage the life-cycle of the microservices (deployment, scaling, and management) arose: such a tool was developed by Google with the name of "Project Seven of Nine "and released as open-source software in 2014. Today that tool is widely known as **Kubernetes**.

Kubernetes provides:

- **Service discovery and load balancing**
  Kubernetes can expose a container using the DNS name or using its own IP address. If traffic to a container is high, Kubernetes is able to load-balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration**
  Kubernetes allows to automatically mount a storage system of different types, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks**
  You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new containers.
- **Automatic bin packing**
  You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing**
  Kubernetes restarts containers that fail, replaces containers, kills containers that do not respond to your user-defined health check, and does not advertise them to clients until they are ready to serve.
- **Secret and configuration management**
  Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

3.3.1 Kubernetes architecture

A Kubernetes (aka **K8s**) cluster is made up of two macro blocks, the first one called Control Plane (Master) and the second one called Data Plane (Worker).

The Control Plane constitutes the brain of the cluster and internally it is made up of the following components:

- **Kube-APIserver** is the component that exposes cluster API and truly is the main component, since Kubernetes has been designed and built to base all the operations on the use of the API.
- **Etcd** is a key value database that maintains all information relating to the status of the cluster.
- **Kube-scheduler** schedules which nodes of the Data Plane run the containers (in Kubernetes named POD, the smallest deployable units of computing that can be created and managed in Kubernetes) based on the resources required, cluster status and affinity and anti-affinity rules.
- **Kube-controllermanager** consists of a set of control processes which:
    - Check if cluster nodes are active.
    - Check if number and status of running POD is aligned with the required one, in case some POD became unavailable for example.
    - Control and create tokens to access on the K8s resource.
    - Populates the Endpoints object (that is, joins Services & PODs).

The Data Plane is the part where the workload is carried out, i.e. where the PODs are put into execution and it is characterized by the following components:

- **Kube-proxy** is a proxy that allows communication to the PODs from within and outside the cluster.
- **Kubelet** checks that PODs are running.
- **Container runtime (engine)** is software responsible for running containers; Kubernetes can use any CRI-O implementation like: Docker, CRI-O and containerd.

In Figure 6, the Kubernetes Cluster components are depicted:

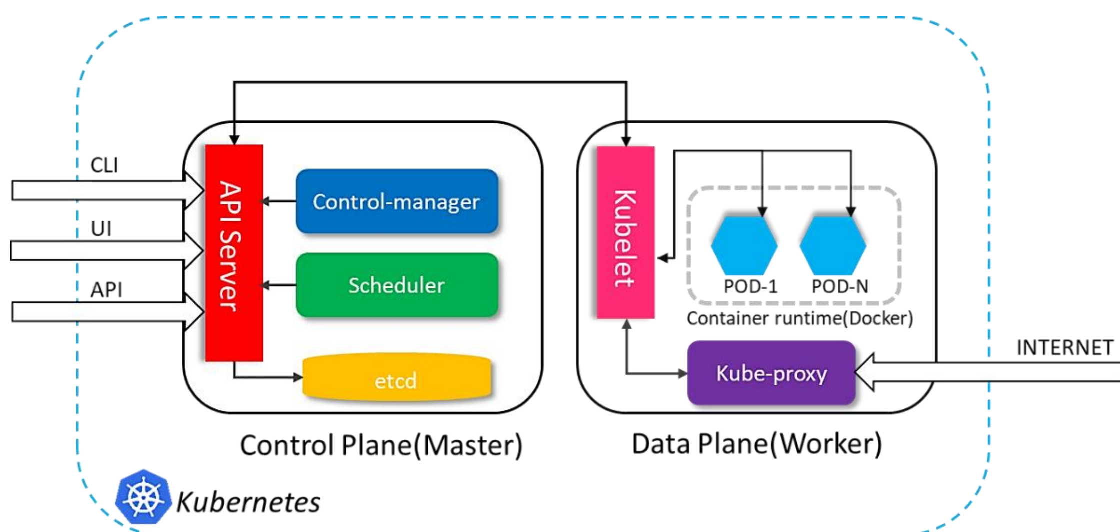


Figure 6 – Kubernetes Architecture

For an outline of the entire Kubernetes solution, see the documentation on kubernetes.io.

However, for the purpose of this chapter it is sufficient to now outline two main concepts available in Kubernetes that we will use later to implement the Sandbox concept.

1. **Namespaces**: They are a logical grouping of a set of Kubernetes objects to whom it's possible to apply some policies, in particular:
    - **Quote** sets the limits on how many HW resources can be consumed by all objects.
    - **Network** defines if the namespace can be accessed or can get access to other Namespaces, in other words if the Namespace is isolated or accessible.

Different policies can be given to different namespaces.

2. **POD** (Figure 7): This is the simplest unit in the Kubernetes object. A POD encapsulates one container, but in some cases (when the application is complex) a POD can encapsulate more than one container. Each POD has its own storage resources, a unique network IP, access port and options related to how the container/s should run.

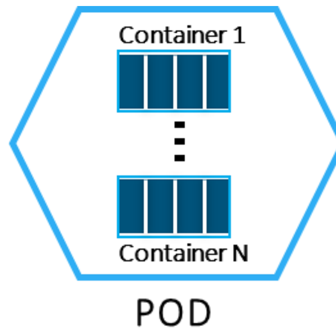


Figure 7 – Kubernetes POD

3.4 INFINITECH Testbeds

At the time of writing and in alignment with the content reported in the last major current outcome of WP6 task T6.1 (see section 1.2), i.e. the deliverable "D6.3 – Testbeds Status and Upgrades – III", INFINITECH makes available a number of testbeds for experimentation, testing and validation of BigData, AI and IoT solutions, including:

- 10 testbeds that are established in the data centres of incumbent financial organizations (on-premise testbeds).
- 1 testbed that has been provisioned and established in order to support the experimentation of the FinTech and InsurTech pilots and enterprises of the consortium, hosted on the partner NOVA's Data Center.
- 1 testbed that will be provisioned and established in order to support the experimentation of the INFINITECH blueprint reference testbed (see Section 5 4.2 of the deliverable), hosted on the AWS (Amazon Web Services) [8] public provider.

Accordingly, the current (at the time of writing, subject to further possible evolutions along the project lifecycle, which should not occur since we're now in the last phase of the project), INFINITECH project plan is to deliver 15 pilots: 10 out of 15 will be carried out on dedicated on-premise Data Centres, while the remaining 5 out of 15 will be carried out on the NOVA's Data Centre, a shared INFINITECH Data Centre. In addition, a blueprint reference testbed will also be provided, built upon the requirements of one of the INFINITECH pilots, as stated above. The set of hardware resources like storage, compute and network will be considered a **testbed**, as shown in Figure 8.

Figure 8 – Testbed

It is not relevant where these resources are deployed: they can be inside a private Data Centre or in any cloud provider. Therefore, the 15 pilots that have been foreseen will be executed in 10+1 testbeds, in addition to the blueprint reference testbed, as shown in Figure 9.

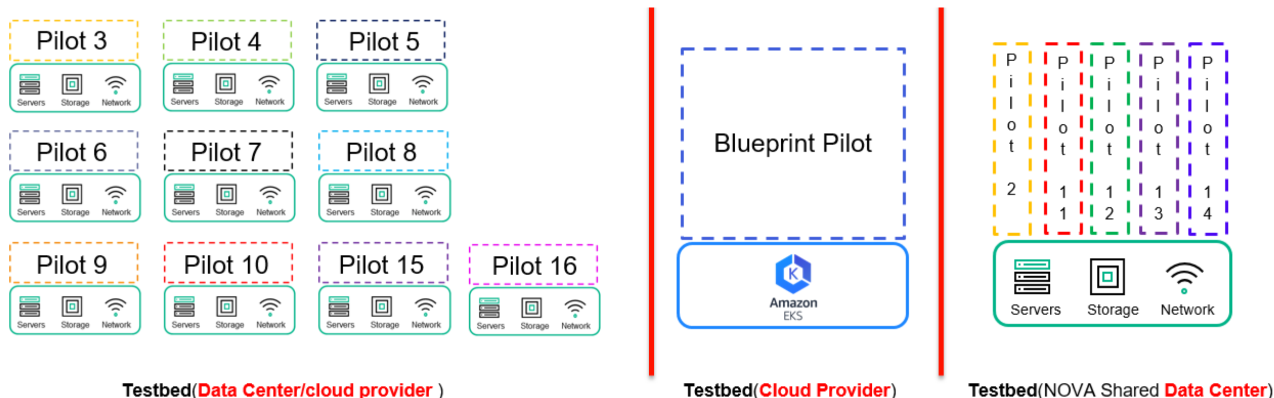


Figure 9 – Testbeds and Pilots

3.5 INFINITECH Sandboxes

Each INFINITECH pilot has one or more Use Cases (realized by one or more pilot Apps, each one realized by one or more INFINITECH microservices): in our vision, each Use Case will be a **Sandbox** provisioned by the leverage of Kubernetes Namespaces.

In fact, as we already said in section 3.3.1, the Kubernetes Namespace feature makes it possible to logically isolate the objects (mainly PODs) inside it from other Namespaces. Therefore, each **dedicated Testbed** will only have one Kubernetes cluster with as many Namespaces as the number of Use Cases to be implemented for a single pilot (see Figure 10). In the other case, the **shared Testbed** will have one Kubernetes cluster for each pilot it has to host and manage (see Figure 11).

In the end, in this general context, each pilot App will be realized as a Kubernetes POD.

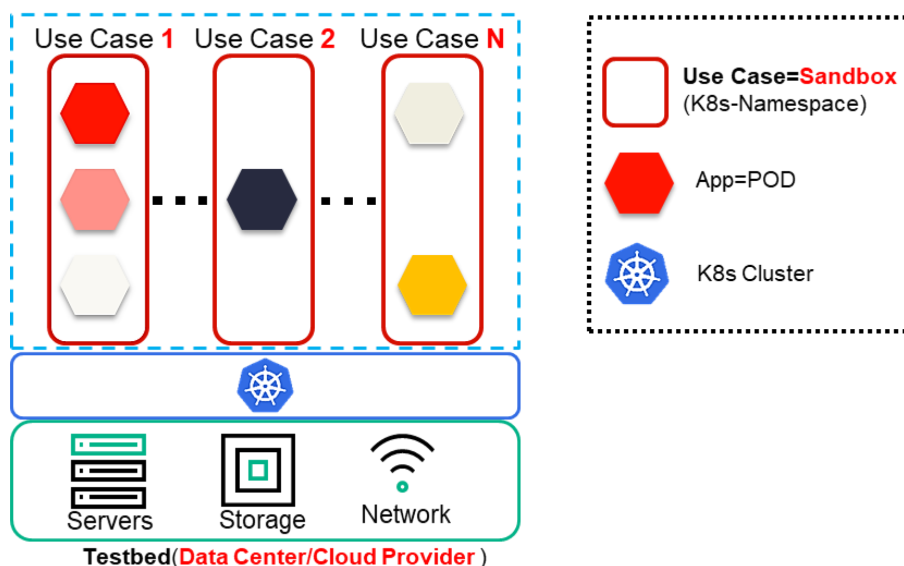


Figure 10 – Sandboxes in a dedicated Testbed

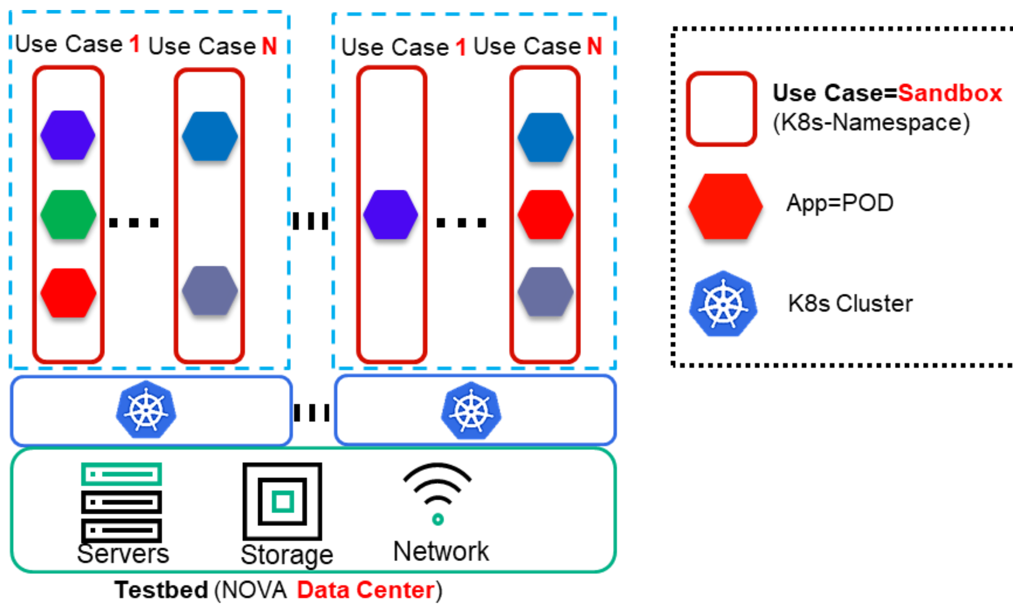


Figure 11 – Sandboxes in a shared Testbed

4 Tools and techniques for Management of Datasets

This chapter describes the tools and techniques that will be leveraged for the management of datasets within the INFINITECH project and how they are linked and mapped with the concepts and techniques related to testbeds and sandboxes (see Section 3) and also with respect to the blueprint reference testbed environment (see Section 5).

4.1 Data Sources and Data Access

The INFINITECH platform aims to host and serve the needs of a variety of applications and tools used by the finance and insurance sectors, which have diverse needs and requirements for data access. We envisage that different types of data sources will be supported such as:

- structural, semi-structural or completely unstructured data; or
- static data, often called data at-rest or streaming data, while the source might be stored on-premise, in a third-party organization or should be imported inside the deployed sandbox. For instance, a common solution might need to use historical data in order to perform a post processing and apply ML/DL analytics for risk assessment.

However, modern enterprises are not only interested in extracting knowledge and identifying risks or opportunities based on historical and often obsolete data, but they also need to extract this type of information from operational data, which introduces a significant challenge regarding data management from the tailored sandbox perspective. To further complicate the needs of the platform, other requirements coming from the finance and insurance sectors are the identification of potential fraudulent financial transactions as they occur, or pre-processing of data from IoT devices as it arrives. Data analysts often tend to use different datasets that can be retrieved from third party organizations or other external sources such as social media to enrich their algorithms with deeper knowledge in order to build a more accurate profile of their customers, either referring to individuals or to enterprises. Finally, additional considerations must be taken into account when dealing with the volume and privacy of the data of a finance organization, where moving data from the source to a sandbox might not be feasible either due to regulatory constraints, or simply due to the overall volume of the datasets.

In order to address all these requirements, the INFINITECH RA has been designed in order to take into account the different types of data sources. With respect to the implementation to support the functionality needed for data management and processing, details have been given in the corresponding deliverables of WP3, WP4 and WP5.

Looking at the techniques and tools that will allow the tailored sandboxes to enable the data management of these diverse environments, the focus is on data access. In other words, how the different components that will be deployed inside the sandbox can be integrated in a way that allows them to make use of all the aforementioned data sources.

We can identify 6 different focuses for data access:

- Static data ingestion
- Dynamic data ingestion
- IoT streaming
- Direct access to on-premise data sources
- Blockchain data access
- Third party data access

In the following subsections, we will describe the details of each of those different means of data access and will analyse the requirements and technical considerations that need to be tackled by the infrastructure, when orchestrating the deployment and maintenance of the integrated solution. Moreover, in the following

we will give more insights on how the INFINITECH platform overcomes the barriers and deals with the challenge of securely allowing those diverse means for data access, thus enabling the automated deployment of the integrated solution.

4.1.1 Static data ingestion

This is the most typical case for data access that is foreseen to be a requirement for the majority of the cases that will need to be deployed into an INFINITECH sandbox. The data analyst has their own data stored into a database management system or other form of persistent storage, and needs to extract a specific dataset and ingest it into the central data repository deployed inside the sandbox so that they can use the tools provided by the platform to perform the analysis. They will have to extract this information in a static file in a specific format (i.e. csv) and trigger the data ingestion process of INFINITECH. We call this data access mode static data ingestion, as the file will be created once, and will be used in each deployment. This access mode also covers the needs for a data analyst to make use of an existing synthetic dataset that has been provided and made available to the platform, via the INFINITECH marketplace. The generated file must be stored into a persistent storage volume and the latter must be visible from the sandbox so that the tools deployed inside can have access to it. This mainly refers to the data ingestion component that will take care of the data migration process, possibly applying a pre-processing algorithm to clean, harmonize and anonymize the dataset and finally store the raw data into the data repository. It will also support cases where direct access to the static file might be needed by the analytical tools on the analytics layer of the IRA, where there is no need for additional processing by the lower logical levels of the architecture (i.e. a spark job needs to grab everything from a csv file, where no additional processing like a filter or join operation can be pushed down to the processing layers).

4.1.2 Dynamic data ingestion

In this scenario, data ingestion does not take place once in the initialization of the sandbox, rather the data might be migrated periodically to the data repository. We call this dynamic data ingestion as the dataset inside the sandbox will be updated with new data after a given period of time. This will cover cases where the requirement is to have the sandbox deployed, integrated and synchronised with the data sources of the organisation. For instance, the integrated solution might need to get updated with the current snapshot of the data, so that the data analyst can benefit from a daily picture of the transactions of any given customer. Rather than having a human intervene to manually extract the data needed into a static file that will be loaded into the system, the data provider can implement specific APIs so that the data migration can be done in a fully automated manner every time (i.e. end of the business day). The data ingestion component of INFINITECH will need to open a connection to those APIs in order to retrieve the data according to the given specified protocol or configuration. The APIs can vary, from REST implementations that imply HTTP connections, to database-specific ones (JDBC, ODBC, etc.) that will imply TPC connections, or even SFTP connections to a file server. This introduces the requirement for the sandbox to allow the INFINITECH component responsible for this job to access endpoints outside the sandbox using different communication protocols. Regarding the need for data access, only the data ingestion component will need to have access to those endpoints, in order to migrate the data into the central data repository. All other components in the different logical layers of the RA will retrieve the data via the data repository.

4.1.3 IoT Streaming

As we already mentioned, it becomes essential for modern applications in the finance and insurance sector to perform real-time analysis in order to respond to possible risks, identify opportunities or even detect fraudulent transactions as they occur. This type differentiates from the others as it does not rely on data at-rest, but on streaming. As a result, content event processing over a data stream is becoming popular. Data transmitted over a stream is usually small and contains information from either a sensor deployed in a vehicle or in the soil, information coming from a finance transaction, logging information produced when a user is

navigating the web or even a tweet or post on social media. We consider all these types of data as IoT, and we categorize this type for data access as IoT Streaming. The unified query processing framework of INFINITECH will require access to data streams, as it provides the means to deploy continuous queries that can perform live processing over the content of the streams, possibly making use of the other layers that allow the combination of operators targeting live data with static data at-rest. The unified query processing framework contains a streaming engine that consumes the stream from the source. The requirement for the infrastructure in this case is to allow the components that are deployed inside the sandbox to be accessible from outside of the sandbox. To concretize the scenario, the sources of the data streams must connect to the query processing framework of the platform by establishing static TPC connections to the latter. All other components in the different logical layers of the RA will retrieve data and information from this layer, so there is no need for others to be accessible from external data sources.

4.1.4 Direct access to on-premise data sources

This covers scenarios where it is not enough for the integrated solution to have access only to a snapshot of the dataset that has been periodically loaded into the sandbox, but it requires access to the overall dataset. It will also cover scenarios where data cannot be migrated into the sandbox due to their volume or due to other regulatory constraints. In the case of 'big data' management, it might be better to push down the data processing to the source and grab only the results that will feed the tools and components in the analytical layer, rather than moving all the data inside the sandbox to be loaded to the central repository beforehand. We call this mode for data access as direct access to on-premise sources as the sandbox needs to access directly the data source and send the processing there, and it will not migrate any data inside. The polyglot component of INFINITECH is responsible for this functionality, which lies in the processing layer of the RA, even if it pushes down to processing to the source. However, when the processing takes place, from a logical point of view, the polyglot takes care of this and all other layers in the analytical layers will make use of the latter. The requirements for the sandbox are similar with the direct data ingestion mode, but it is the polyglot component now that needs to open connections to external endpoints from the sandbox, using different communication protocols, varying from HTTP, to TPC and SFTP.

4.1.5 Blockchain data access

As the name of this access mode implies, it covers scenarios where secure access to data stored in the Blockchain is required. We are interested here in the uses of Blockchain as a means of persistent storage of data, and not as a means of verifying the consensus for a given transaction. As it has been described in the corresponding deliverables of the tasks related with the Blockchain technology in WP4, the access is being granted via a specified API. Therefore, from a functional view, the components that need to access Blockchain data will have to open a connection to the provided APIs and retrieve data. The Blockchain data needs no pre-processing to the source nor can it benefit from the processing capabilities of the INFINITECH data management components. Therefore, it is the components that are lying in the analytical layer of the RA that need to be granted access to the Blockchain storage via the appropriate designated endpoint. However, from the perspective of the infrastructure that will orchestrate the automated deployment and maintenance of the integrated solution, the requirements are similar to the dynamic data ingestion: to allow components inside the sandbox to establish connections with endpoints that are located externally from the sandbox.

4.1.6 Third Party data access

In this access mode the components and analytical tools provided by the INFINITECH platform need to retrieve data and information that has been provided by external sources. This scenario is different from the previous ones, and especially to the direct access to on-premise sources, as the third party does not grant access to its entire dataset, but rather provides specific APIs that will allow others to extract only specific information. In other words, it can allow to submit an operational query (i.e. how popular is the given trend in the articles of UK during the last week) and retrieve its result. The third-party organization has access to

other datasets, and has already performed an initial pre-processing that generates the results of the types of queries (also called "consults") that its API allows it to perform. Therefore, there is no need for the data management components that are suited in the processing layers of the RA to perform any further analysis and this information can be directly used by the analytical tools. However, from the perspective of the infrastructure that manages the sandboxes, the requirement is the same as the Blockchain: to allow the components deployed in the sandbox to open connections to APIs that are deployed outside.

4.2 Datasets management from the Blueprint reference environment perspective

Even if we have several and diverse types for data access, the requirements for the datasets and data access management tools and the techniques from the blueprint reference environment perspective (see Section 5), or rather from the testbeds and sandboxes perspective, can be described and addressed by grouping the methodologies presented in the previous paragraphs into the following macro categories:

1. **Static data ingestion**: The blueprint must be able to store static data into a persistent volume that needs to be accessible by the sandboxes. This requirement is fulfilled using the Kubernetes PV (Persistent Volume) available inside the blueprint and in particular by using the PVC (Persistent Volume Claim) that allows splitting the PV according to the sandboxes' needs.
2. **Dynamic data ingestion - Blockchain data access - Third party data access**: The blueprint must be able to allow the components of different layers of the logical architecture view to open connections to external endpoints. Accordingly, the components deployed inside the sandboxes need to communicate with the external endpoints. This communication is permitted by default, unless otherwise stated, through the API Gateway which forwards HTTP / HTTPS requests to and from the sandboxes.
3. **IoT Streaming - Direct access to on-premise data sources**: The blueprint must be able to allow components that have been already deployed to connect and maintain open connections with components that have been deployed inside a sandbox. Accordingly, the components that have been deployed inside a sandbox have to maintain an open connection on TPC protocol. In this case the connections are not managed by the API Gateway, but using the Kubernetes NodePort. The NodePort functionality exposes the sandbox service on each worker's IP at a static port in the range between 30000 and 32767. Each worker proxies such port (the same port number on every Node) into the sandbox service. In order to clarify this concept, let's suppose that the sandbox service works on the port 390: this port is then remapped on the port 30390 on each worker node, and in this way the sandbox can then be reached by external components which can reach the nodes and maintains an open connection on the port 30390. In the blueprint environment for security reasons, the Kubernetes worker nodes are deployed in a private network, so in order to expose the NodePort to the outside world a Bastion [9] host has been implemented. The Bastion is based on HA-Proxy [10], a free reverse proxy software, and it is the only system allowed to connect with the Kubernetes nodes that exposes a public IP as a bridge from the nodes and internet.
   So in the above example, in order to connect with the NodePort 30390 on the worker nodes, it's mandatory to connect on Bastion IP public on the same port and then it will forward, balancing the requests, on all worker nodes.

Let's now see how we provide dataset management from the blueprint reference perspective into practice. We will demonstrate how we solve these concepts by using the project's pilot#2 ("Real-time risk assessment in Investment Banking"). This pilot requires the use of an historical data set to be migrated into the Infinistore component deployment deployed in the sandbox and to additionally ingest with data in real-time coming from an external data stream. The latter is being achieved using the INFINITECH image of Apache Kafka queue, which has been extended with direct connectors to the Infinistore and will be part of the tailored sandbox for this pilot.

Loading the historical data into the Infinistore for new deployment of the sandbox fits into the *static data ingestion* category. Therefore, we will make use of a PVC and we will upload the data there once. Then, we will mount the PVC with the POD of the Infinistore in each deployment, and during the initialization phase, we will migrate the data into the central repository. The following code snippet depicts the definition of our PVC.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: infinistore-datasets-pvc
  namespace: pilot2
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 10Gi
```

Here, we request a 10GB storage where the historical dataset will be uploaded. Then, the following code snippet depicts how to mount this storage element to the POD of the Infinistore:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: infinistore
  namespace: pilot2
  labels:
    app: infinistore
spec:
  template:
    spec:
      containers:
          volumeMounts:
            - name: infinistore-datasets-storage
              mountPath: /datasets
      volumes:
        - name: infinistore-datasets-storage
          persistentVolumeClaim:
            claimName: infinistore-datasets-pvc
```

Firstly, we define our *statefulset* for the INFINSTORE which will make use of the *infinistore-datasets-pvc* that we previously define and contains the already uploaded historical dataset. Then, we mount the persistent storage to the folder of the POD where the Infinistore is expecting to find such information.

Regarding the need for data ingestion of streaming data into the deployed sandbox, this fits into the category of *IoT Streaming - Direct access to on-premise data sources*. As a matter of fact, we will rely on the *NodePort* services of EKS along with the Bastion proxy. As the configuration of the proxy has been made by the system administrator of the infrastructure, it is not the focus of this document. For our scenario, we need to allow

and maintain TCP connections to ports 9092 (for the data stream to connect to the Kafka worker instance) and 8081 (for the data stream to interact with the embedded *Avro Registry*, in order to serialize the data items appropriately). The following code snippet depicts how we can define our *NodePort* to let us do this:

```
apiVersion: v1
kind: Service
metadata:
  name: lx-kafka-np
  namespace: pilot2
spec:
  type: NodePort
  selector:
    app: lx-kafka
  ports:
    - name: "9092"
      protocol: TCP
      port: 9092
      targetPort: 9092
      nodePort: 30040
    - name: "8081"
      protocol: TCP
      port: 8081
      targetPort: 8081
      nodePort: 30041
```

Here, we map the POD's port of 9092 to 30040 and its 8081 to 30041. The Bastio proxy will direct connections to the external IP in those ports to the same ports of the worker that hosts the POD, which will be then mapped to the original 9092 and 8081 ones, as described previously. We also notice that both 30040 and 30041 are inside the range between 30000 and 32767. Our final step would be to also set up the corresponding environment variables of the Kafka container accordingly, via its *configmap*. The following code snippet depicts this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: lx-kafka-configmap
  namespace: pilot2
data:
  advertised.url: 18.197.20.179
  advertised.port: "30040"
```

This information is needed by the Kafka runtime execution environment in order to send to its clients (in our case, the data streaming) and advertise where they should try to connect. As our image is deployment-agnostic and can be portable in different testbeds, we need to provide this information in our configuration file. Here, we can see the number of the advertised port that the client needs to connect and will be later mapped to the 9092 port that the Kafka worker is listening to inside the sandbox. Last but not least, we also need to define the external IP of the testbed that is publicly accessible. The Bastion proxy will take care of directing the connections to this IP to the specific POD.

5 Overview of the INFINITECH blueprint reference testbed

This chapter describes the final design and implementation of the INFINITECH blueprint reference testbed, through the actual realization (with a full compliance) of the INFINITECH RA Development and Deployment views, in terms of the concrete specification and realization of the fundamental and target INFINITECH concepts of Testbeds, Sandboxes and Datasets management, and related tools and techniques for their effective setup and deployment in the INFINITECH pilots and validation scenarios. In other words, we explain how the concepts described in chapter 3 have been realized using the target INFINITECH infrastructure environments.

The main changes compared to the initial version of this deliverable refer to the methodologies used to recreate the blueprint environment on the same cloud provider AWS and in the bare-metal environment, as described in section 5.2.4.

In particular, the AWS `ekctl` tool used to create the blueprint environment on AWS has been replaced by the Terraform tool [11]. Terraform has been selected because it is one of the best tools for IaC (Infrastructure as Code) that allows recreating an infrastructure everywhere, and always in a predictable and safe way. It is an open-source software with a very large community and it is infrastructure-agnostic, so it has been chosen in order to avoid any lock-in with AWS functionalities.

At the same time the previous choice to use `kubespray` tool to create K8s cluster in a bare-metal environment has been replaced with Rancher tool [12], because Rancher has a lot more features than `kubespray`.

Rancher is an open-source software that facilitates the creation and administration of Kubernetes clusters; among the different features offered, Rancher has the ability to:

- Create K8s clusters in any environment both public and private cloud
- Include an application catalog
- Provide a centralized infrastructure monitoring system at a multi-cluster level
- Control accesses via RBAC access control system, a feature based on RBAC (Role-Based Access Control), that enable the integration with various Identity services like LDAP, OpenID and Active Directory.

All features described above can be managed from a single UI (User Interface) or via REST-API so that it can be called up by any automation tool.

5.1 Development view

In the context of WP6 and this deliverable, with respect to the Development view, it has been decided to implement DevOps (Development and Operations) processes. DevOps represents a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach. DevOps emphasizes people (and culture), and seeks to improve collaboration between operations and development teams. DevOps implementations utilize technology (especially automation tools) that can leverage an increasingly programmable and dynamic infrastructure from a life cycle perspective [13].

The practical implementation of DevOps goes through the CI/CD processes, which are delivered through the combined practices of Continuous Integration (CI) and Continuous Delivery (CD).

In particular:

- **Continuous Integration** is a practice where development teams frequently commit (many times per day) application code changes to a shared repository. These changes automatically trigger new builds that are then validated by automated testing (as in Development Testing, DevTest [14]), to ensure that they do not break any functionality.
- **Continuous Delivery** is an extension of the CI process. It's the automation of the release process so that new code is deployed to target environments, typically to test environments, in a repeatable and automated fashion.

In order to fulfil the INFINITECH project goals, the CI/CD processes have been created in the context of the blueprint reference testbed environment. Moreover, to build a system working consistently as a whole, the developers writing the individual components of the INFINITECH platform need an integrated environment where they can test their components working together with the other services. To support this process, we implemented a Continuous Integration environment based on EKS (Elastic Kubernetes Service) [15], a managed Kubernetes service on the AWS public cloud. More details on EKS will be provided in Section 5.2. Kubernetes is an ideal choice for a Continuous Integration environment, since it allows easy updates of deployments when new application images are built, with manifests containing deployment configurations versioned like Git [16] alongside the application source code. Furthermore, it is easy to spin up new test environments from scratch, which enables future scenarios including automated end-to-end integration testing. Build agents are also created on demand and removed when done, providing efficient resource utilization and clean environments to ensure build reproducibility.

On the target cluster, a namespace named *devops* has been created for hosting the DevOps tools, which are:

- **Gitlab** [17]: This is a Git repository manager that lets each developer teams collaborate on INFINITECH's source code.
- **Jenkins** [18]: This is the de-facto standard open-source automation server for orchestrating CI/CD workflows.
- **Sonatype Nexus** [19]: This is a popular artifact repository that also works as a Docker registry, as required in our case.
- **OpenLDAP** [20]: This is used as the single user directory for all tools, centralizing authentication and simplifying management of developer accounts.
- **Helm** [21]: This is a package manager that streamlines installing and managing Kubernetes applications.

Figure 12 shows how CI/CD works for a specific partner (e.g. Partner "A"). When a developer pushes new component code, Gitlab invokes a webhook on Jenkins, which starts any job affected by the code changes. The job builds the component, runs unit tests and, if everything has worked in a proper way, builds an updated Docker image and pushes it to Nexus. The following step is deploying the updated component in the specific partner namespace; in fact, we will have as many namespaces as partners, to maintain the correct isolation between all INFINITECH partners. In order to deploy as required, the component Helm manager will be used. At the end of the process, Jenkins sends a notification to a dedicated CI/CD channel on the INFINITECH Slack [22] project, so that developers are informed that a new build occurred and whether it was successful or not. In case of errors, developers will have to inspect the build logs, find the problem and correct it. In case of success, developers will go ahead and test that the new version works correctly in the test environment.

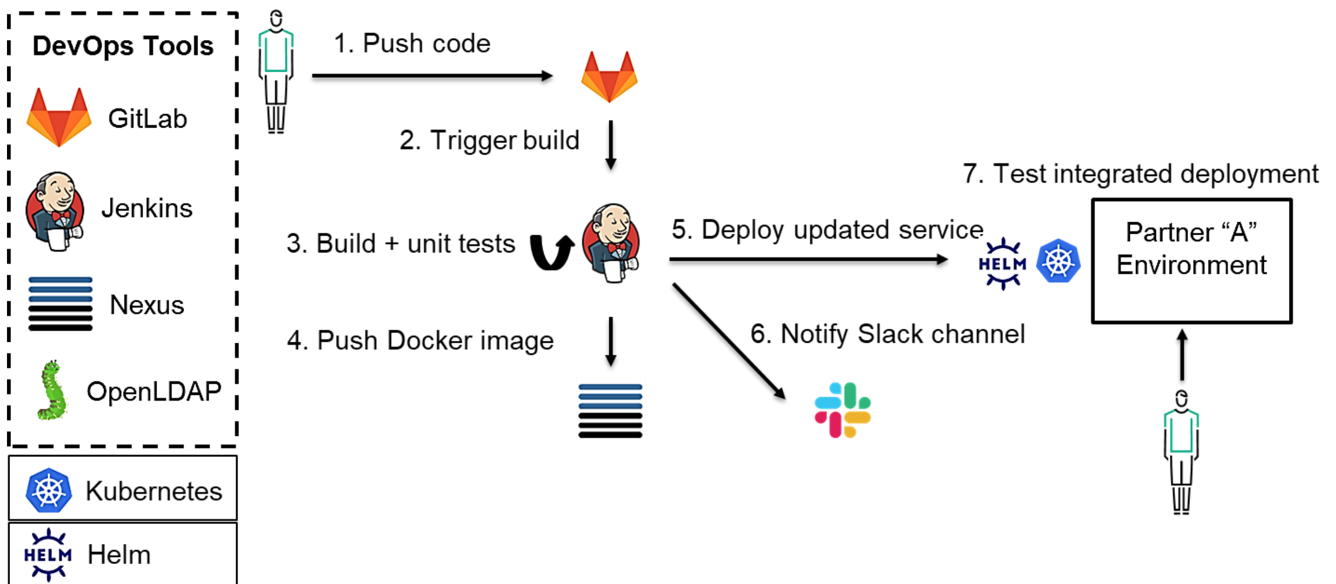


Figure 12 - CI/CD workflow

Moreover, in the following iteration of WP6 T6.2 and T6.3 tasks and activities (e.g. in the future deliverable D6.6), we would plan to enhance the process by adopting a DevSecOps [23] approach and including the related tools like **Sonarqube** [24]in the CI/CD pipeline.

DevSecOps aims to include security in the software development life cycle from the beginning, following the same principles of DevOps. Security is then considered throughout the process and not just as an afterthought at the end of it, so that different kinds of security checks are executed continuously and automatically, giving developers quick feedback if the latest changes introduced a vulnerability that must be corrected.

DevSecOps of course requires that security experts work side by side with developers and operations to make sure that security requirements are addressed and best practices followed, in addition to validating product design and architecture.

Moreover, in this context, in order to facilitate the Machine Learning (ML) development and testing (that is a key pillar technological topic in the INFINITECH project), in the current and second iteration of WP6, T6.2 and T6.3 tasks and activities we have also introduced the **MLOps** [25] (a combination of Machine Learning and IT Operations) methodology and processes focused on:

- Facilitating communication and collaboration between teams.
- Improving model tracking, versioning, monitoring and management.
- Standardizing the machine learning process to prepare for increasing regulation and policy.

This add-on enhanced methodology has enabled us to automate the porting of the machine learning algorithms, as much as possible, in production environments.

Putting this into practice is often very complicated for IT managers, because ML processes are often based on heterogeneous environments. Therefore, the first step towards MLOps requires the standardization of these environments as much as possible, and in this respect the Kubernetes technology and containers provide the abstraction, scalability, portability, and reproducibility required to run the same piece of software in all these environments. As a second step, it is necessary to make standard the workflows used for the construction and building of the ML models.

In this sense, we have selected the **Kubeflow** [26] platform for integration in our blueprint reference testbed, in order to provide an infrastructure to build models and capable of enabling the portability of these models and workflows.

All the details about the MLOps integration and how it has enabled us to enhance the INFINITECH way for the deployments of project pilots and technologies are reported in the following section 6.5.

5.2 Deployment view

In the context of WP6 and this deliverable, with respect to the Deployment view, it has been decided to create the INFINITECH blueprint reference testbed on the AWS (Amazon Web Services) public provider. In particular, each such blueprint is hosted on the Amazon EKS (Elastic Kubernetes Service). Amazon EKS is a managed service that allows the use of a Kubernetes environment without the need to install, operate, and maintain Kubernetes control plane or nodes.

The EKS control plane (Figure 13) is composed of at least two API server nodes and three etcd nodes that run across three Availability Zones within a Region. In case of issues, Amazon EKS automatically detects and replaces unhealthy control plane instances, restarting them across the Availability Zones within the Region as needed. From the security perspective all information stored on etcd nodes and associated Amazon EBS (Elastic Block Store) volumes is encrypted using AWS KMS (Key Management Service).

Regarding the data plane, we started using 2 nodes, but we have also defined the Auto Scaling rules in order to scale the Kubernetes cluster according to the real usage of the platform.

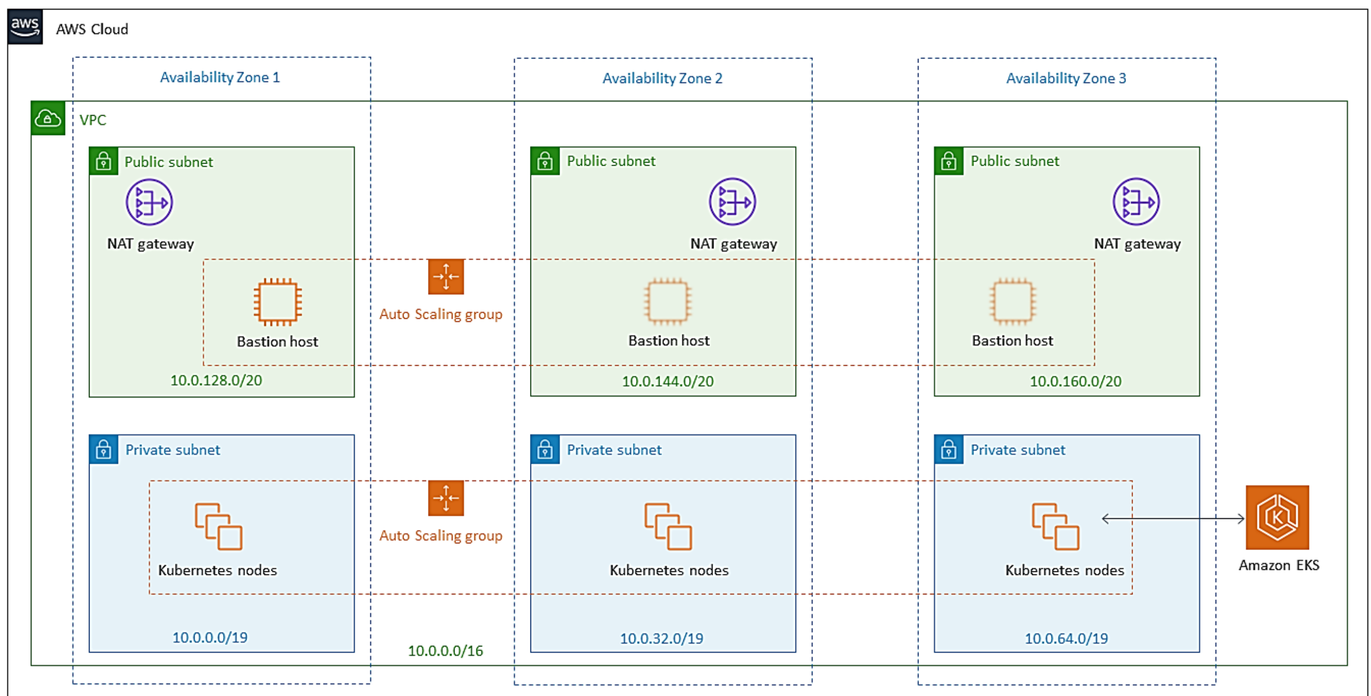


Figure 13 – EKS Control Plane deployment [27]

5.2.1 Creation of the EKS INFINITECH cluster (Blueprint)

The creation of EKS INFINITECH cluster involves four principal steps:

1. Create a specific IAM (Identity and Access Management) Role able to create a provisioning EKS cluster.
2. Create the VPC (Virtual Private Cloud).
3. Create the EKS Control plane.
4. Create the Worker node.

To perform all steps, we leveraged on the AWS console and where possible also the CloudFormation templates that allowed us to simplify the installation steps. For the creation of the IAM role the following Amazon CloudFormation template has been used:

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'Amazon EKS Cluster Role'


Resources:

  eksClusterRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
        - Effect: Allow
          Principal:
            Service:
            - eks.amazonaws.com
          Action:
          - sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/AmazonEKSClusterPolicy

Outputs:

  RoleArn:
    Description: The role that Amazon EKS will use to create AWS resources
for Kubernetes clusters
    Value: !GetAtt eksClusterRole.Arn
    Export:
      Name: !Sub "${AWS::StackName}-RoleArn"
```

To create the VPC for EKS we used the following template:

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'Amazon EKS Sample VPC - Private and Public subnets'

Parameters:

  VpcBlock:
    Type: String
    Default: 192.168.0.0/16
    Description: The CIDR range for the VPC. This should be a valid private
(RFC 1918) CIDR range.

  PublicSubnet01Block:
    Type: String
```

```
    Default: 192.168.0.0/18
    Description: CidrBlock for public subnet 01 within the VPC

  PublicSubnet02Block:
    Type: String
    Default: 192.168.64.0/18
    Description: CidrBlock for public subnet 02 within the VPC

  PrivateSubnet01Block:
    Type: String
    Default: 192.168.128.0/18
    Description: CidrBlock for private subnet 01 within the VPC

  PrivateSubnet02Block:
    Type: String
    Default: 192.168.192.0/18
    Description: CidrBlock for private subnet 02 within the VPC

Metadata:
  AWS::CloudFormation::Interface:
    ParameterGroups:
      -
        Label:
          default: "Worker Network Configuration"
        Parameters:
          - VpcBlock
          - PublicSubnet01Block
          - PublicSubnet02Block
          - PrivateSubnet01Block
          - PrivateSubnet02Block

Resources:
  VPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock:  !Ref VpcBlock
      EnableDnsSupport: true
      EnableDnsHostnames: true
      Tags:
      - Key: Name
        Value: !Sub '${AWS::StackName}-VPC'

  InternetGateway:
    Type: "AWS::EC2::InternetGateway"

  VPCGatewayAttachment:
    Type: "AWS::EC2::VPCGatewayAttachment"
    Properties:
      InternetGatewayId: !Ref InternetGateway
```

```
      VpcId: !Ref VPC

  PublicRouteTable:
    Type: AWS::EC2::RouteTable
    Properties:
      VpcId: !Ref VPC
      Tags:
      - Key: Name
        Value: Public Subnets
      - Key: Network
        Value: Public

  PrivateRouteTable01:
    Type: AWS::EC2::RouteTable
    Properties:
      VpcId: !Ref VPC
      Tags:
      - Key: Name
        Value: Private Subnet AZ1
      - Key: Network
        Value: Private01

  PrivateRouteTable02:
    Type: AWS::EC2::RouteTable
    Properties:
      VpcId: !Ref VPC
      Tags:
      - Key: Name
        Value: Private Subnet AZ2
      - Key: Network
        Value: Private02

  PublicRoute:
    DependsOn: VPCGatewayAttachment
    Type: AWS::EC2::Route
    Properties:
      RouteTableId: !Ref PublicRouteTable
      DestinationCidrBlock: 0.0.0.0/0
      GatewayId: !Ref InternetGateway

  PrivateRoute01:
    DependsOn:
    - VPCGatewayAttachment
    - NatGateway01
    Type: AWS::EC2::Route
    Properties:
      RouteTableId: !Ref PrivateRouteTable01
      DestinationCidrBlock: 0.0.0.0/0
      NatGatewayId: !Ref NatGateway01
```

```
PrivateRoute02:
  DependsOn:
  - VPCGatewayAttachment
  - NatGateway02
  Type: AWS::EC2::Route
  Properties:
    RouteTableId: !Ref PrivateRouteTable02
    DestinationCidrBlock: 0.0.0.0/0
    NatGatewayId: !Ref NatGateway02

NatGateway01:
  DependsOn:
  - NatGatewayEIP1
  - PublicSubnet01
  - VPCGatewayAttachment
  Type: AWS::EC2::NatGateway
  Properties:
    AllocationId: !GetAtt 'NatGatewayEIP1.AllocationId'
    SubnetId: !Ref PublicSubnet01
    Tags:
    - Key: Name
      Value: !Sub '${AWS::StackName}-NatGatewayAZ1'

NatGateway02:
  DependsOn:
  - NatGatewayEIP2
  - PublicSubnet02
  - VPCGatewayAttachment
  Type: AWS::EC2::NatGateway
  Properties:
    AllocationId: !GetAtt 'NatGatewayEIP2.AllocationId'
    SubnetId: !Ref PublicSubnet02
    Tags:
    - Key: Name
      Value: !Sub '${AWS::StackName}-NatGatewayAZ2'

NatGatewayEIP1:
  DependsOn:
  - VPCGatewayAttachment
  Type: 'AWS::EC2::EIP'
  Properties:
    Domain: vpc

NatGatewayEIP2:
  DependsOn:
  - VPCGatewayAttachment
  Type: 'AWS::EC2::EIP'
  Properties:
```

```
      Domain: vpc

  PublicSubnet01:
    Type: AWS::EC2::Subnet
    Metadata:
      Comment: Subnet 01
    Properties:
      MapPublicIpOnLaunch: true
      AvailabilityZone:
        Fn::Select:
        - '0'
        - Fn::GetAZs:
            Ref: AWS::Region
      CidrBlock:
        Ref: PublicSubnet01Block
      VpcId:
        Ref: VPC
      Tags:
      - Key: Name
        Value: !Sub "${AWS::StackName}-PublicSubnet01"
      - Key: kubernetes.io/role/elb
        Value: 1

  PublicSubnet02:
    Type: AWS::EC2::Subnet
    Metadata:
      Comment: Subnet 02
    Properties:
      MapPublicIpOnLaunch: true
      AvailabilityZone:
        Fn::Select:
        - '1'
        - Fn::GetAZs:
            Ref: AWS::Region
      CidrBlock:
        Ref: PublicSubnet02Block
      VpcId:
        Ref: VPC
      Tags:
      - Key: Name
        Value: !Sub "${AWS::StackName}-PublicSubnet02"
      - Key: kubernetes.io/role/elb
        Value: 1

  PrivateSubnet01:
    Type: AWS::EC2::Subnet
    Metadata:
      Comment: Subnet 03
    Properties:
```

```
      AvailabilityZone:
        Fn::Select:
        - '0'
        - Fn::GetAZs:
            Ref: AWS::Region
      CidrBlock:
        Ref: PrivateSubnet01Block
      VpcId:
        Ref: VPC
      Tags:
      - Key: Name
        Value: !Sub "${AWS::StackName}-PrivateSubnet01"
      - Key: kubernetes.io/role/internal-elb
        Value: 1

  PrivateSubnet02:
    Type: AWS::EC2::Subnet
    Metadata:
      Comment: Private Subnet 02
    Properties:
      AvailabilityZone:
        Fn::Select:
        - '1'
        - Fn::GetAZs:
            Ref: AWS::Region
      CidrBlock:
        Ref: PrivateSubnet02Block
      VpcId:
        Ref: VPC
      Tags:
      - Key: Name
        Value: !Sub "${AWS::StackName}-PrivateSubnet02"
      - Key: kubernetes.io/role/internal-elb
        Value: 1

  PublicSubnet01RouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
      SubnetId: !Ref PublicSubnet01
      RouteTableId: !Ref PublicRouteTable

  PublicSubnet02RouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
      SubnetId: !Ref PublicSubnet02
      RouteTableId: !Ref PublicRouteTable

  PrivateSubnet01RouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
```

```
    Properties:
      SubnetId: !Ref PrivateSubnet01
      RouteTableId: !Ref PrivateRouteTable01

  PrivateSubnet02RouteTableAssociation:
    Type: AWS::EC2::SubnetRouteTableAssociation
    Properties:
      SubnetId: !Ref PrivateSubnet02
      RouteTableId: !Ref PrivateRouteTable02

  ControlPlaneSecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Cluster communication with worker nodes
      VpcId: !Ref VPC

Outputs:

  SubnetIds:
    Description: Subnets IDs in the VPC
    Value: !Join [ ",", [ !Ref PublicSubnet01, !Ref PublicSubnet02, !Ref
PrivateSubnet01, !Ref PrivateSubnet02 ] ]

  SecurityGroups: eith
    Description: Security group for the cluster control plane communication
with worker nodes
    Value: !Join [ ",", [ !Ref ControlPlaneSecurityGroup ] ]

  VpcId:
    Description: The VPC Id
    Value: !Ref VPC
```

The previous template created the VPC with two public and two private subnets. One public and one private subnets are deployed to the same Availability Zone. The second public and private subnets are deployed to a second Availability Zone in the same Region. We have chosen this solution because it is that recommended by AWS for a critical environment.

After that the VPC is available; it's possible to proceed with the EKS Control plane creation that can be done directly from the AWS console at https://console.aws.amazon.com/eks/home#/clusters through the input of the following parameters:

- Name: **INFINITECH-BP**
- Kubernetes version: **1.17**
- Public and private: **true**
- role: **eksClusterRole** (create with in the previous step)
- Encryption: false (Activate if is necessary)
- Subnet: **all**
- Security groupsInfo: **ControlPlane**
- Access point Public: **True**

When the EKS Control plane is available (the creation takes about 15 minutes), it is possible to create the worker nodes.

In order to add the nodes, it is mandatory to create a specific IAM role named *WorkerRole* with the following permissions:

- AmazonEKSWorkerNodePolicy.
- AmazonEKS_CNI_Policy.
- AmazonEC2ContainerRegistryReadOnly.

Moreover, to facilitate the role management we will add the tag value *INFINITECHBPNode*.

At this time, it is possible to create from the same AWS console the worker nodes by clicking on the Compute tab and by entering these values:

- Name: **T3xlarge_16GB4-4vCPU**
- Node IAM role name: **WorkerRole**
- AMI type: **Amazon Linux 2 (AL2_x86_64)**
- Instance Type: **t3a.xlarge(4vCPU/16GB)**
- Disk size: **50 GB**
- Minimum size: **2**
- Maximum size: **3**
- Desired size: **2**
- Subnets: **private-sub01; private-sub02**
- Allow remote access to nodes: **true**
- Allow remote access from: **all**

At this stage, the Kubernetes cluster is ready to use.

5.2.2 Namespace, Network and Quote policies

As previously described in section 3.5, we have decided to implement the INFINITECH Sandbox concept leveraging the Kubernetes namespace feature. Therefore, the first step is to create a dedicated Namespace for each of the INFINITECH target pilots that can be leveraged by each of the owning partners to develop and test their own pilot applications.

To create the namespace we have defined a Kubernetes YAML template like the following:

```
kind: Namespace
apiVersion: v1
metadata:
  name: Pilot1
  labels:
    name: Pilot1
```

The first namespace that has to be created has been named *devops*, aimed to host all the DevOps tools.

Some of these tools (like Jenkins and Gitlab) need to be exposed on the Internet for convenient access during the development and testing phases. All the endpoints will be HTTPS-based, with free certificates generated by Let's Encrypt [28], so we do not need to set up a Certification Authority or buy certificates from commercial CAs.

However, we do need public DNS entries on the project domain, mapped to a public IP that exposes our services outside of the Kubernetes cluster.

In order to get the public IP, we will use the AWS ELB (Elastic Load Balancer). This is provisioned automatically when creating a Kubernetes Service with type LoadBalancer. In our case, this happens while configuring the

NGINX Ingress Controller. The NGINX Ingress Controller that will be deployed will have a dedicated namespace named *ingress-nginx.*

To implement the network policies to isolate each namespace from each other, and eventually also to manage the connection between different PODs within the same namespace (i.e. to guarantee the security requirements), we will implement the Cilium as CNI (Container Network Interface) on EKS.

In order to do that, we will remove the AWS CNI and install the Cilium following these steps:

```
kubectl -n kube-system delete daemonset aws-node
helm repo add cilium https://helm.cilium.io/

helm install cilium cilium/cilium --version 1.7.9 \
  --namespace kube-system \
  --set global.eni=true \
  --set global.egressMasqueradeInterfaces=eth0 \
  --set global.tunnel=disabled \
  --set global.nodeinit.enabled=true
```

After the previous step, it is possible to isolate the namespace implementing the following rule (the rules are written in the YAML format):

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "isolate-pilot1"
  namespace: pilot1
spec:
  endpointSelector:
    matchLabels:
      {}
  ingress:
  - fromEndpoints:
    - matchLabels:
        {}
```

Moreover, we will apply the resource quote on memory and CPU that each namespace can consume with this YAML template:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-namespace
spec:
  hard:
    requests.cpu: "800m"
    requests.memory: 2Gi
    limits.cpu: "1"
    limits.memory: 3Gi
```

After the execution of the previous steps, the Deployment view has been completed, as shown in Figure 14.

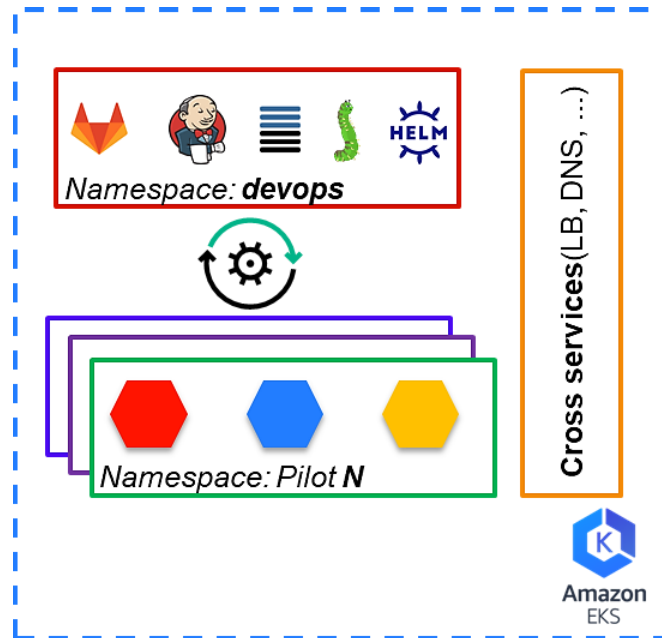


Figure 14 – INFINITECH blueprint reference testbed

5.2.3 EKS for Kubeflow environment

The platform selected for the implementation of ML processes, as described in section 5.1, is Kubeflow. Kubeflow is a complex framework that requires well-defined infrastructural characteristics for its functioning, for this reason it was decided to use a new dedicated Kubernetes cluster (EKS) for its implementation. For the infrastructural implementation has been used the reference IaC tool for the Infinitech project i.e. Terraform.

The scripts used to create this infrastructure have been organized as follows:

- `versions.tf`
- `outputs.tf`
- `variables.tfvars`
- `vpc.tf`
- `security-groups.tf`
- `eks-cluster.tf`
- `kubernetes.tf`

The `version.tf` script contains all the versions of the Terraform components that we used to create the infrastructure; like the AWS Terraform provider. The details are shown below

```
required_providers {
  aws = {
    source  = "hashicorp/aws"
    version = ">= 3.20.0"
  }

  random = {
    source  = "hashicorp/random"
    version = "3.1.0"
  }

  local = {
```

```
      source  = "hashicorp/local"
      version = "2.1.0"
    }

    null = {
      source  = "hashicorp/null"
      version = "3.1.0"
    }

    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = ">= 2.0.1"
    }
  }

  required_version = ">= 0.14"
}
```

The `output.tf` script is used to print some information related to some components created by Terraform, information such as the cluster endpoint, the kubectl configuration file, and so on. This approach is also useful because this information could be used as input for other Terraform scripts. The script is below:

```
output "cluster_id" {
  description = "EKS cluster ID."
  value       = module.eks.cluster_id
}

output "cluster_endpoint" {
  description = "Endpoint for EKS control plane."
  value       = module.eks.cluster_endpoint
}

output "cluster_security_group_id" {
  description = "Security group ids attached to the cluster control plane."
  value       = module.eks.cluster_security_group_id
}

output "kubectl_config" {
  description = "kubectl config as generated by the module."
  value       = module.eks.kubeconfig
}

output "config_map_aws_auth" {
  description = "A kubernetes configuration to authenticate to this EKS cluster."
  value       = module.eks.config_map_aws_auth
}

output "cluster_name" {
  description = "Kubernetes Cluster Name"
  value       = local.cluster_name
```

```
}
```

The `variables.tfvars` file contains all the variables that have been defined for use within the Terraform scripts, below the detail

```
cluster_name = "EKS-KUBEFLOW"
vpc_name = "KUBEFLOW"
```

The `vpc.tf` script contains the instructions for the VPC we want to create, so in our case we want to have 3 Availability Zones (AZ), with 3 public networks, one for each AZ, 3 private networks with only one NAT GW

```
variable "cluster_name" {
  description = "EKS Cluster name"
}
variable "vpc_name" {
  description = "VPC name"
}
provider "aws" {
 region = var.region
}
data "aws_availability_zones" "available" {}

locals {
  cluster_name = var.cluster_name
}
resource "aws_eip" "nat" {
  count = 1

  vpc = true
}
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.2.0"
  name                 = var.vpc_name
  cidr                 = "10.0.0.0/16"
  azs                  = data.aws_availability_zones.available.names
  private_subnets      = ["10.0.1.0/24", "10.0.2.0/24"]
  public_subnets       = ["10.0.4.0/24", "10.0.5.0/24"]
  one_nat_gateway_per_az = false
  enable_nat_gateway   = true
  single_nat_gateway   = true
  enable_dns_hostnames = true
  enable_dns_support   = true
  reuse_nat_ips       = true                   # <= Skip creation of EIPs
for the NAT Gateways
  external_nat_ip_ids = "${aws_eip.nat.*.id}"   # <= IPs specified here as
input to the module

  tags = {
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
```

```
  }

  public_subnet_tags = {
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/elb"                      = "1"
  }

  private_subnet_tags = {
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/internal-elb"             = "1"
  }
}
```

The `security-groups.tf` script describes the type of inbound and outbound connections allowed through the worker nodes that make up the EKS cluster. In that case we have only allowed the SSH connection, as you can see in detail below

```
resource "aws_security_group" "worker_group_mgmt_one" {
  name_prefix = "worker_group_mgmt_one"
  vpc_id      = module.vpc.vpc_id

  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"

    cidr_blocks = [
      "10.0.0.0/8",
    ]
  }
}

resource "aws_security_group" "worker_group_mgmt_two" {
  name_prefix = "worker_group_mgmt_two"
  vpc_id      = module.vpc.vpc_id

  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"

    cidr_blocks = [
      "192.168.0.0/16",
    ]
  }
}

resource "aws_security_group" "all_worker_mgmt" {
  name_prefix = "all_worker_management"
  vpc_id      = module.vpc.vpc_id
```

```
  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"

    cidr_blocks = [
      "10.0.0.0/8",
      "172.16.0.0/12",
      "192.168.0.0/16",
    ]
  }
}
```

The `eks-cluster.tf` script is the core of this solution because it contains the information related to the EKS cluster we want to create, then the EKS version, the number of workers and what type of EC2 we want for the creation of the workers, the detail is shown below

```
module "eks" {
  source          = "terraform-aws-modules/eks/aws"
  version         = "17.24.0"
  cluster_name    = local.cluster_name
  cluster_version = "1.19"
  subnets         = module.vpc.private_subnets

  vpc_id = module.vpc.vpc_id

   tags = {
    Environment = var.cluster_name
  }
  workers_group_defaults = {
    root_volume_type = "gp2"
  }

  worker_groups = [
    {
      name                          = "worker-group"
      instance_type                 = "t3a.xlarge"
      additional_security_group_ids                                     =
[aws_security_group.worker_group_mgmt_one.id]
      asg_desired_capacity          = 2
    }
  ]
}

data "aws_eks_cluster" "cluster" {
  name = module.eks.cluster_id
}

data "aws_eks_cluster_auth" "cluster" {
```

```
  name = module.eks.cluster_id
}
```

The `kubernetes.tf` script creates, after Terraform is finished, a file named “kubeconfig_<CLUSTER-NAME>” that could be used with kubectl client to interact with EKS cluster

```
provider "kubernetes" {
  host                   = data.aws_eks_cluster.cluster.endpoint
  token                  = data.aws_eks_cluster_auth.cluster.token
  cluster_ca_certificate                                          =
base64decode(data.aws_eks_cluster.cluster.certificate_authority.0.data)
}
```

These files have been placed in a directory on a PC suitably configured to interact with the AWS account. The command was then executed:

```
terraform plan -var-file="variables.tfvars" -out first
```

Which performs two steps; the first makes a formal check on the Terraform scripts to verify that they do not contain errors, the second produces an output file with all the resources that must be created to satisfy the requests indicated in the scripts. With the output of this command in this case called "first", the creation of the infrastructure was performed with the following command:

```
terraform apply first
```

The creation took about 30 minutes to complete.

On the same machine from which the terraform commands were executed, the official Kubeflow code was downloaded using the command:

```
git clone https://github.com/kubeflow/manifests.git
```

Then to install Kubeflow we followed the official documentation visible at https://github.com/kubeflow/manifests#installation”, and this command has been used:

```
while ! kustomize build example | kubectl apply -f -; do echo "Retrying to apply
resources"; sleep 10; done
```

Deployment took about 10 minutes, after which the Kubeflow was ready for use.

5.2.4 How to recreate the blueprint testbed for a specific INFINITECH Pilot

One of the most powerful capabilities and features that are enabled by the technological choice on software tools and techniques described in the previous sections, is that the blueprint reference testbed created on AWS can be recreated from scratch, with respect to the Deployment view perspective, by each of the partners for their own pilots in two possible ways (see Figure 15):

I. On the same AWS cloud provider
II. In a bare metal environment, leveraging their on-premise private data centre infrastructure or the shared NOVA’s Data Centre.

The two ways are a little bit different from each other, because:

I. In the first case, the recreation of a blueprint environment can be done in a fully automated way using the Terraform tool. Terraform uses HCL (HashiCorp Configuration Language) to describe the infrastructure that it is going to implement on the public/private cloud or on a bare-metal

environment, starting from this language it generates a plan file in which are described all steps that are required to reach the final desired state and it follows this plan to provision the infrastructure.

Its strength is linked to the advantages deriving from the use of the IaC (Infrastructure as Code) approach so:

- **Rapidity**: Using a code to create infrastructure is quicker than creating it through the graphical interface because it is not necessary to go through different pages to create the specific resources, instead, all infrastructure is described with one file.
- **Reliability**: Using a code instead of manually creating reduces the likelihood of error
- **Scalability**: IaC makes it possible to describe the infrastructure only once and replicate it as many times as you want.

II. In the second case, it is possible to recreate the cluster in a partially automated way, because in this case a mandatory prerequisite is manually preparing the entire infrastructure like storages, servers and network that will accommodate the INFINITECH components, after that when it will be ready it will be possible to deploy the blueprint environment using Rancher tools.
Regarding NOVA testbed, the physical infrastructure has been virtualized using VMware's vSphere hypervisor.
Three VMs have been created on this infrastructure on which the Rancher software has been installed. Using this software made it possible to recreate a copy of the blueprint environment for pilot 2, pilot 11, pilot 12, pilot 13 and pilot 14, in an easy and fast way through the user interface.
In addition, the centralized infrastructure monitoring as well as the possibility of implementing RBAC functionality has greatly facilitated the administration of the platform.

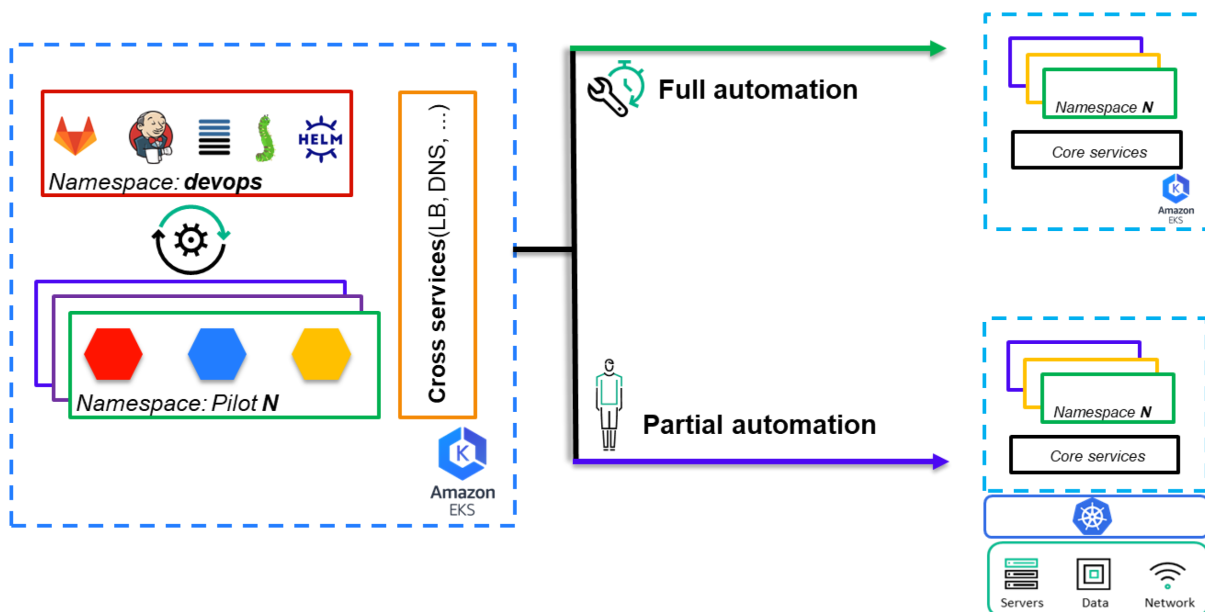


Figure 15 – Blueprint environment recreation ways

Nevertheless, one of the major objectives of having provided a blueprint reference testbed is exactly the powerful concept explained above: a potentially easy and straightforward replication of it for all the INFINITECH target pilots' environments.

6 Blueprint guidelines for the “INFINITECH way” deployments of project pilots and technologies

This chapter provides guidelines for the partners of the “INFINITECH way” on how to organize their artifacts in the project’s code repository, and make use of the CI/CD pipelines to i) make their solutions available to other partners and ii) automate the deployment of a pilot.

It makes use of an imagined pilot solution that has been implemented to validate the process and can be used as a reference for other pilots to help them build their solutions. This reference pilot makes use of two INFINITECH building blocks, the Infinistore and the lx-kafka, and a micro-service that has been implemented specifically for the needs of that pilot. In this document, we will see how we organize the code and artifacts: platform building blocks, which are artifacts provided by INFINITECH, must go under the corresponding groups in order to be available for all pilot solutions, while pilot-specific solutions must go under separate groups. We provide guidelines on how to create new projects under specific groups, set up the Dockerfile that instructs the CI process on how to build the image, and how to define the CI process itself, by making use of a Jenkins file. Finally, at the last step, we provide guidelines on how to make use of the CD process to define the artifacts that the solution consists of and how to automate the deployments.

The provided information should be used as the “INFINITECH Way” guidelines for all partners of the INFINITECH consortium, whether they are technology providers who implement one or more building blocks of the INFINITECH platform itself; a data analyst who implement AI algorithms and tools; or developers and integrators who work on pilot-specific applications.

With reference to the previous version of the deliverable, in this one the guidelines (in addition to a general refinement and further validation) have been enhanced, as already introduced in section 5.1, with the introduction of the MLOps methodology and processes that has enabled the project to facilitate the Machine Learning (ML) and Deep Learning (DL) development and testing (which are key pillar technological topics in INFINITECH). Such integration has been performed through a joint effort between WP6 tasks T6.2 and T6.3 and WP5 tasks T5.4 and T5.5, and in the end has brought to the integration of the INFINITECH End To End framework for the definition and deployment of ML/DL microservices developed in WP5 (see deliverable D5.9) with the “INFINITECH way” guidelines developed in WP6. All the details about the MLOps integration are reported in the following section 6.5.

6.1 Guidelines overview

This section provides some basic information on how to have access to the code repository and how the artifacts are organized per category. The INFINITECH code repository uses the Gitlab, which can be accessed by the private Infinitech Gitlab url [29] (see Figure 16), deployed in AWS. Each authorized user account (previously properly created for a member of an INFINITECH partner) can click at the aforementioned link, and provide the credentials in terms of a username and password. The LDAP option has to be selected and then users must add their credentials, as the following figure shows.

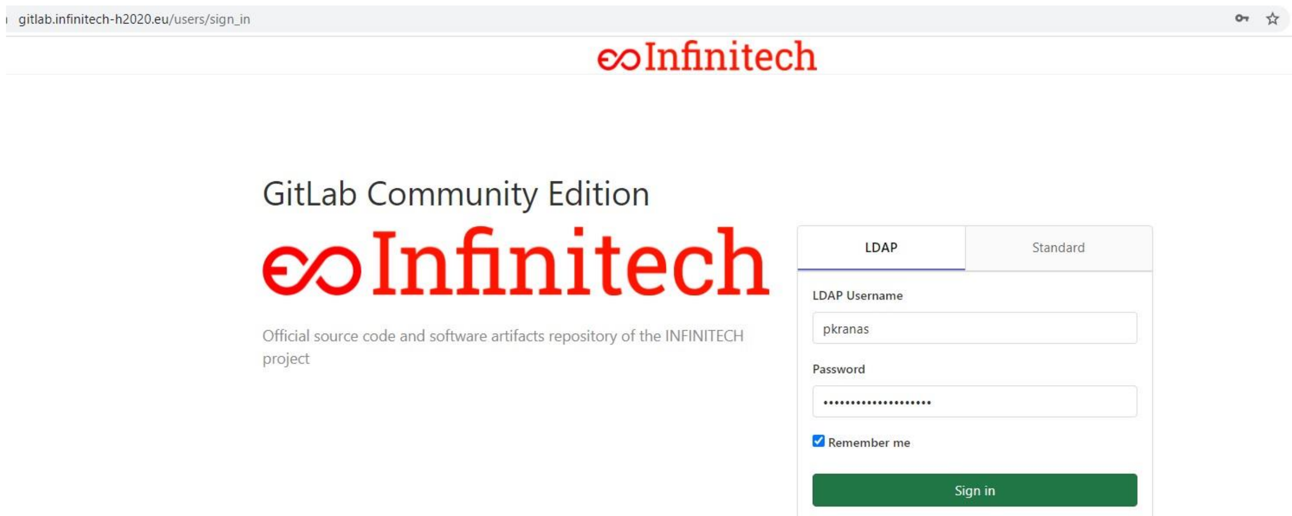


Figure 16 – Gitlab login

After a user login, the available projects belonging to this specific user are shown (see Figure 17). In the following example the user *pkranas* is allowed to see the following projects:

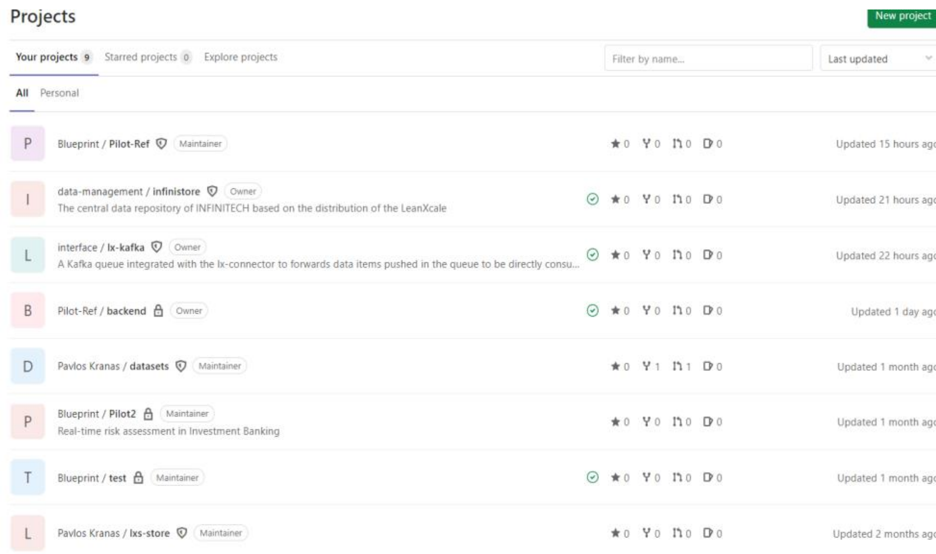


Figure 17 – List of projects visible to a specific user

In Gitlab various groups has been created where the user should place their projects. First, each user has their own group to experiment and upload a code for internal experimentation, testing purposes or whatever reason. Apart from that, there are 2+1 categories of groups: groups that include projects that implement a building block of the INFINITECH platform and are of general purpose, and pilot-specific groups that shall contain projects that implement a functionality/building block that is specific to the pilot. Last but not least, there is the *Blueprint* group, which contains the projects that build the integrated solution of each pilot.

Groups that contain INFINITECH's building blocks, have been categorized based on the Reference Architecture of INFINITECH itself. These must be generic and must be re-used by different pilots or other potential customers in the future. As a result, these types of groups must not contain any pilot-specific information. All components in this category of groups must be *confidential.* This means that they must only be available to all partners of the consortium. The groups are as follow:

- Data-management: This group contains the component building blocks related with all data management functionalities of the platform. They include the data ingestion building block, the data repository of the platform (for example *Infinistore)* etc.
- Analytics: This group is designed to contain all projects that implement an AI algorithm and are related to the work that has been carried out under the scope of T5.4. Analytics groups must be available to all partners of the consortium. The artifacts produced by these projects can be reused by one or many pilot solutions.
- Blockchain: This group contains all projects that implement a building block that is related with the Blockchain technology.
- Datasemantics: This group contains all projects that are part of the *Semantic Interoperability Framework* of INFINITECH.
- Securityprivacy: Here lives all building blocks that implements functionalities related with the security and privacy aspects of the platform. It includes anonymization algorithms/tools, data regulatory constraints solutions etc.
- Presentation: This group contains projects that provide general purpose UI functionalities including some user interfaces and visualization graphs that can be reused by pilots to show the results of analytical tools. It is important to highlight that this group does not include front-end implementations that are specific to the pilots, rather than implementations that are generic and can be reused by each pilot.
- Interface: This group includes projects which implements functionalities which allows the connectivity of tools/services external to the platform of INFINITECH itself. The gateway, Kafka/RabbitMQ queues provided by INFINITECH also belong to this group.

Apart from the INFINITECH category of groups, there is also the PILOT category which contain Pilot1, Pilot2, Pilot15 groups, each one of those is related to the specific pilot of the project. For instance, let's suppose to work on group PilotX, where X is the number of the INFINITECH Project Pilot, here's where all projects that are specific to a PilotX will be placed. AI algorithms that are private and must be only used by the PilotX, any backend specific functionality implemented as a service, front-end tools that are also specific for the PilotX solution and so on can be placed here. These projects cannot be shared among other pilots, as they implement a specific functionality and they are not for general purpose. In this guideline, an imaginary pilot, called Pilot-Ref, will be used to demonstrate the use of the CI/CD pipelines. This pilot can be used as a reference for all others. For the pilots of INFINITECH, groups called Pilot1, Pilot2 etc. for pilot1, pilot2 etc. are expected as per our naming convention.

Finally, there is a group *Blueprint,* which has a list of projects. Each pilot has its corresponding project. This includes the manifest files that define which artifacts consist of a pilot solution and how they are interconnected. They use confidentially-available projects from the general-purpose groups and the pilot's specific projects. For the imaginary Pilot-Ref, there is a Pilot-Ref project under this group to be used as an example in this guideline.

The following sample diagram (Figure 18) shows the organization under the groups which consist within the INFINITECH platform. The orange block shows the category, the grey blocks the group and the green the project itself, as they appear in Gitlab.

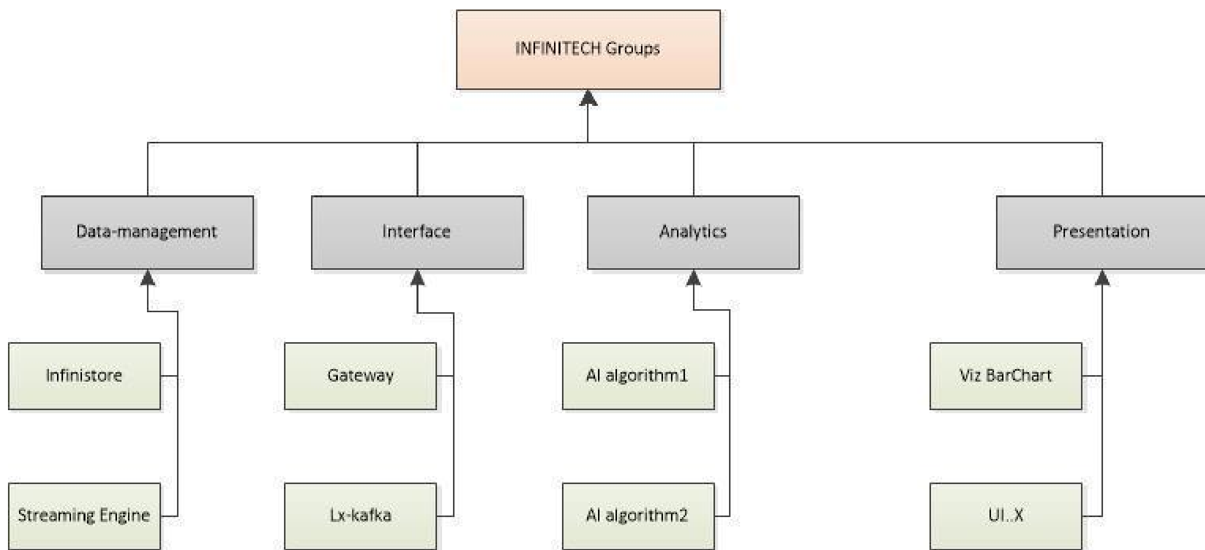


Figure 18 – The INFINITECH groups

The following diagram (Figure 19) shows the organization under the pilot groups:

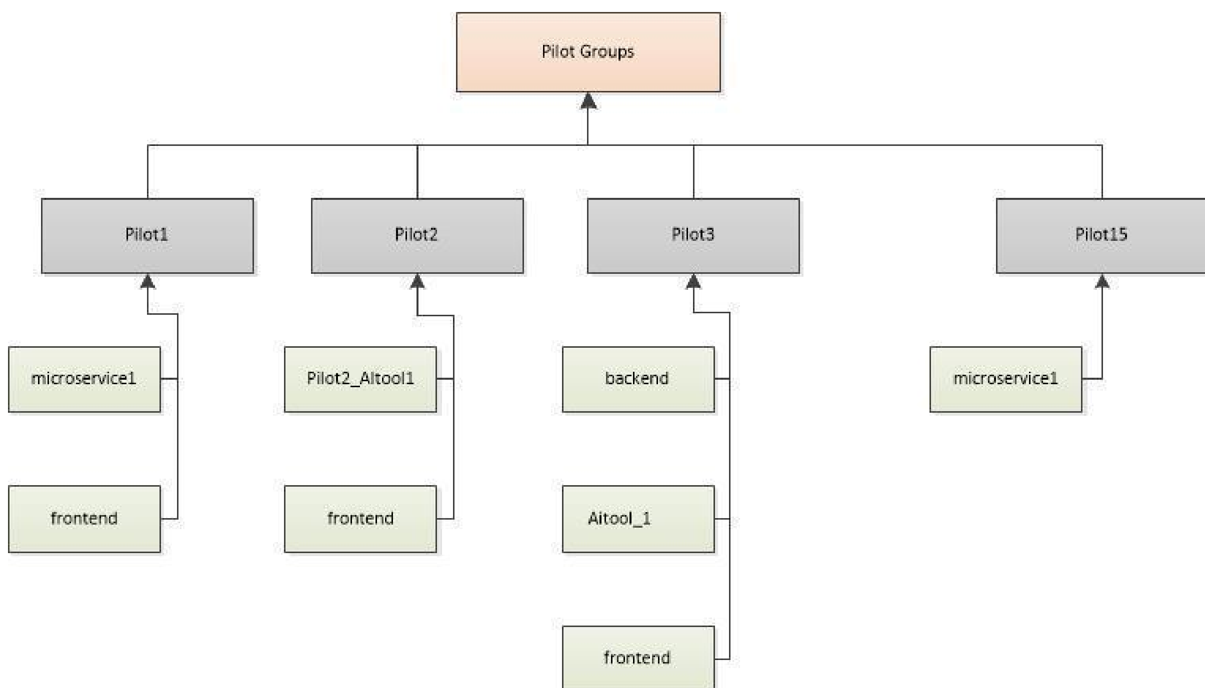


Figure 19 – The Pilot groups

It is important to underline that projects under different pilots may have the same name. However, as they belong to different groups (i.e. pilot1/frontend and pilot2/frontend), this is allowed. Moreover, if there are microservices or AI tools, pilot-specific must be placed under pilot group. Therefore, the pilot3/aitool_1 has been developed only for the purposes of Pilot3 and its visibility is private. This means that this cannot be reused by other pilots nor can it be discovered by the marketplace and be further exploitable by other parties.

Finally, the next figure (Figure 20) shows the projects under the group *Blueprint*.

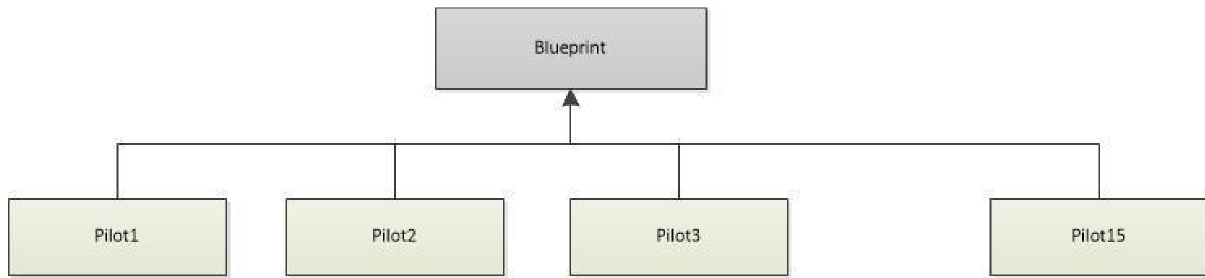


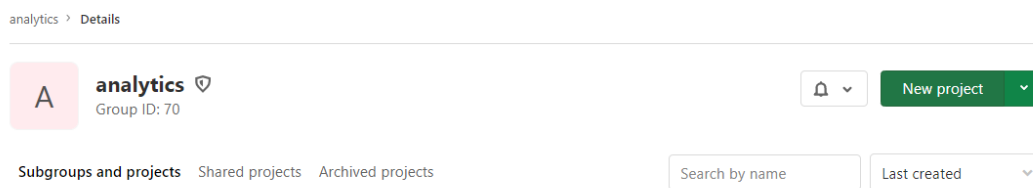
Figure 20 – Projects under the Blueprint group

The difference between the pilots in the last two figures (Figure 19 and Figure 20) is that Pilot1, Pilot2,…Pilot15 of Figure 19 are *groups* that contains other projects, while Pilot1, Pilot2,…Pilot15 of Figure 20 are projects themselves, which are located under the blueprint/PilotX where X is the specific pilot number. In the latter case, these projects include all information regarding the deployment of the integrated solution of a specific Pilot, while the former is the group that contains projects (building blocks and AI tools) for this specific pilot. More details will be given in the following sections.

The following sections will provide information on how to deploy and make use of different types of software artifacts and how to deploy a pilot solution under the AWS infrastructure using the CI/CD pipelines.

6.2 Building an INFINITECH component

This section provides information on how to deploy a component that is either a platform component or an AI tool that can be reused by more than one pilot scenario. It is important to underline that all these artifacts must be containerized and enable the build of a docker image. The section describes in detail the procedure to deploy the Infinistore component of INFINITECH, which is part of the data-management group. In this specific case, the user must refer to the Gitlab Group data-management [30], while in case of an AI tool the group analytics must be used [31]. To create a new project under the analytics group, the user can use the New Project button (see Figure 21).



A group is a collection of several projects.

If you organize your projects under a group, it works like a folder.

You can manage your group member's permissions and access to each project in the group.

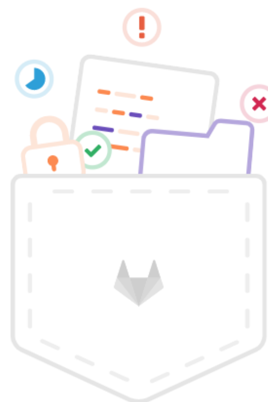


Figure 21 – Gitlab create a new project

When creating a new project, it is necessary to provide some information, filling in the appropriate fields. In the example in Figure 22, the group name is *data-management* and the project name is *Infinistore*. It is important to make sure that the visibility level is set to internal, in case it is necessary to allow others to edit. After filling in this basic information and clicking on create project button, an empty project will be created with the given name.

Figure 22 – Gitlab setting new project details

Once the project is created, members allowed access to it will be added. It is very important to ensure that the user gitlabinfinitech can have access as a maintainer of the project, to allow the CI pipelines to be executed as depicted in the following picture.

Figure 23 – Gitlab add gitlabinfinitech user to access the new project

The user can now clone the project in his/her own dev local machine and upload any code required. The shortest way to clone a project is using the command git clone followed by the url suggested by Gitlab itself, as depicted in the following figure:

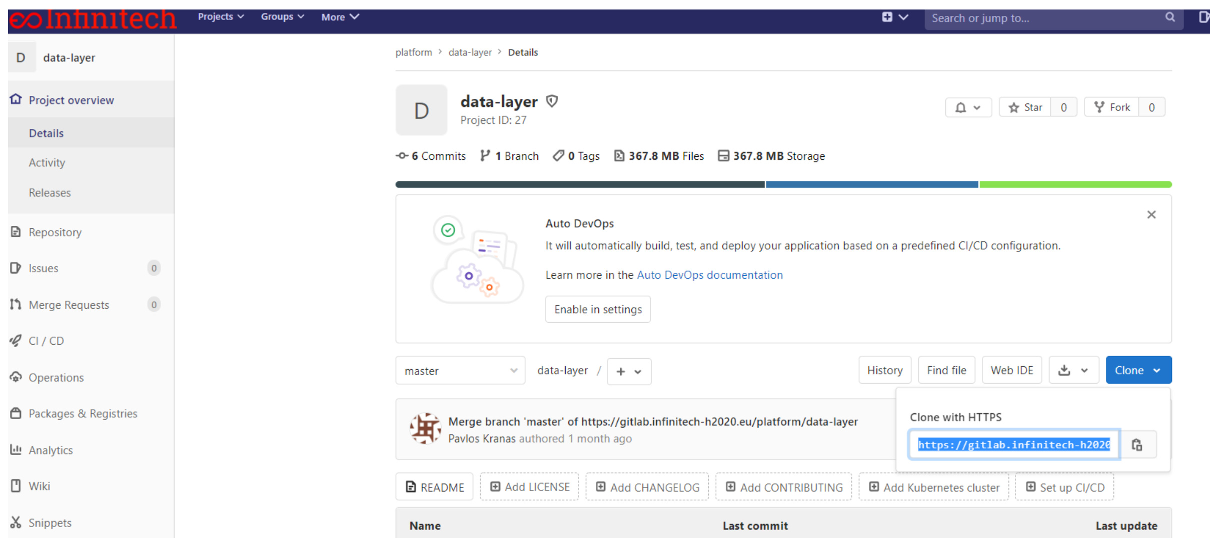


Figure 24 – Git clone an existing project

Cloning the project allows the user to work in his/her local dev environment and to pull all the project source code needed to create the binaries that will be further used to create the docker image. There might be several possibilities here: either to upload the source code which must be compiled during the building process or to provide pre-compiled binaries. In the latter case, the binaries must be reachable during the creation of the Docker image in Gitlab. The suggested manner is to either build a base image locally that will be pushed to the project's docker Harbour, or make use of the INFINITECH package repository, so that the binaries can be accessible. More information regarding those two methods will be provided at section 6.2. Finally, in case a project belongs to a pilot group and is private to the organization, this can be kept on the premise of the organization.

In either of these options, a Dockerfile must be provided, it defines the instructions for the CI pipeline to build the image, along with the Jenkinsfile that provides instructions for the CI pipeline itself. The Dockerfile is unique to each artifact, as it provides artifact-specific instructions to build the image. An example for the data-layer component can be found here [32]. Regarding the Jenkinsfile, there is a template provided by HPE that should be used by the providers of the components. Users can simply copy the Jenkinsfile already available in one of the other projects (i.e. the *Infinistore*) and apply some minor but very important modifications. The following figure points out the most important Jenkinsfile information that could be to be customized for a specific project.

```
node(label) {
    def cloneRepo = checkout scm
    // docker image name and version
    def image = "infinistore:latest"
    def gitCommit = cloneRepo.GIT_COMMIT
    def gitBranch = cloneRepo.GIT_BRANCH
    def project = "data-management"
    def dockerInfinitech = "harbor.infinitech-h2020.eu"
    def registryInfinitech = "https://harbor.infinitech-h2020.eu"
        try {
updateGitlabCommitStatus name: 'build', state: 'running'
stage('Clone repository') {
        checkout scm
    }
stage('Build docker') {
        try{
          notifySlack('STARTED')
          container('docker-cmds') {
         sh """
         docker build -t ${dockerInfinitech}/${project}/${image} .
         """
         updateGitlabCommitStatus name: 'build', state: 'pending'
            }
      }
        catch (exc) {
        println "Failed to build - ${currentBuild.fullDisplayName}"
        notifySlack('UNSTABLE')
        updateGitlabCommitStatus name: 'build', state: 'failed'
        throw(exc)
         }
}
```

Figure 25 – Defining the Jenkins file

In the red circle of Figure 25 there is the name of the image that will be created by the CI pipeline. It is important to ensure that the name of the project is the same as the group that this artifact belongs to (i.e. in the previous picture this is called *data-management*) while the name of the image should be the same as of the current project (i.e. in the previous picture this is called *Infinistore*).

Once these are defined, the user must communicate with the HPE team using the slack channel to set up the corresponding webhooks that will be triggered each time a push event occurs to the Gitlab project to create a new image and store it under the INFINITECH's docker registry (Harbor), so that it can be available for all pilots. Once the webhook is created, it is possible to check it by clicking in the corresponding tab, as Figure 26 indicates.

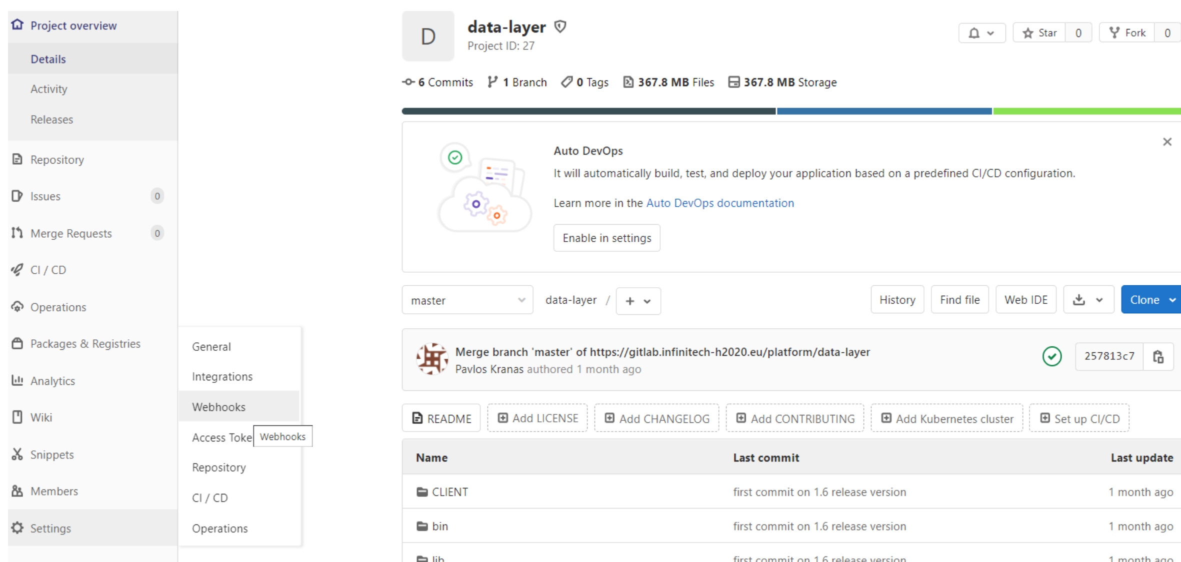


Figure 26 – Checking the webhooks of the project

Once clicked, it is possible to check that the webhook has been configured correctly; by default, the webhook is triggered when a push event takes place on the master branch. In case the developer makes use of other branches that are being merged for the master, the webhook can be triggered manually, as Figure 27 indicates.

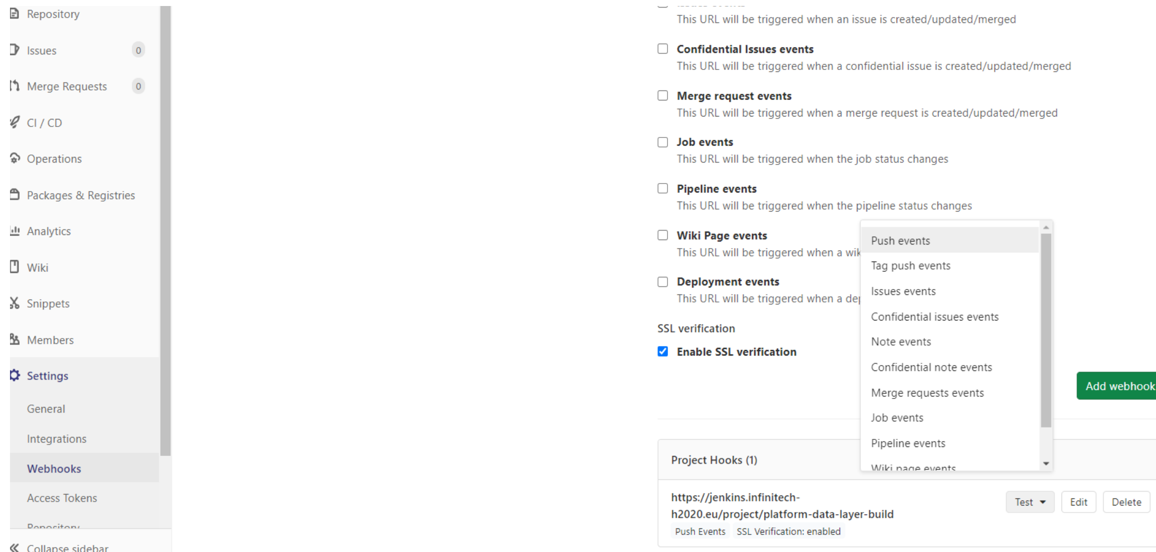


Figure 27 – Manually trigger a CI pipeline

It is useful to check the status and the logs of a pipeline, by going to the specific tab, as shown in Figure 28.

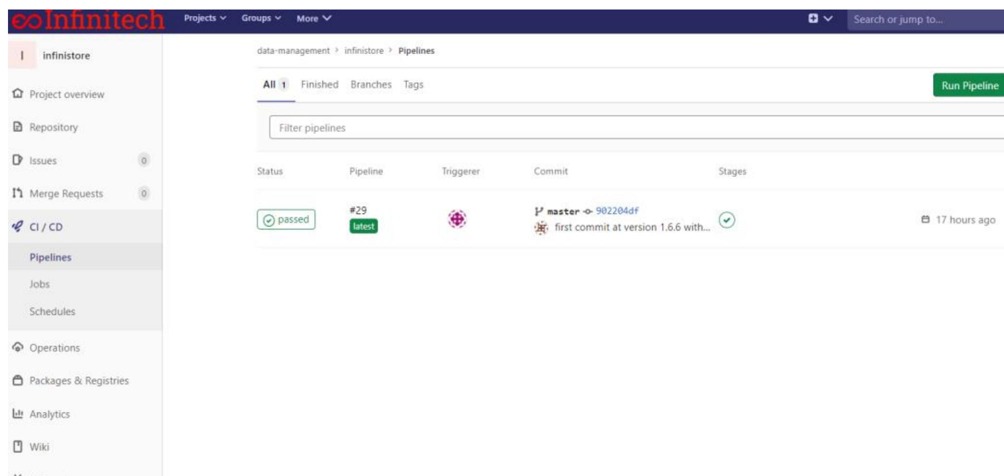


Figure 28 – Checking the status of a CI pipeline

Figure 28 shows that the latest pipeline was passed, whose trigger was the gitlabinfinitech user. This is the reason why it is very important to grant access to this user to your project. Clicking the corresponding #29 (the latest) pipeline, it is possible to see the different steps in the pipeline itself. In this Jenkinsfile only the build process is defined. Details are shown in Figure 29.

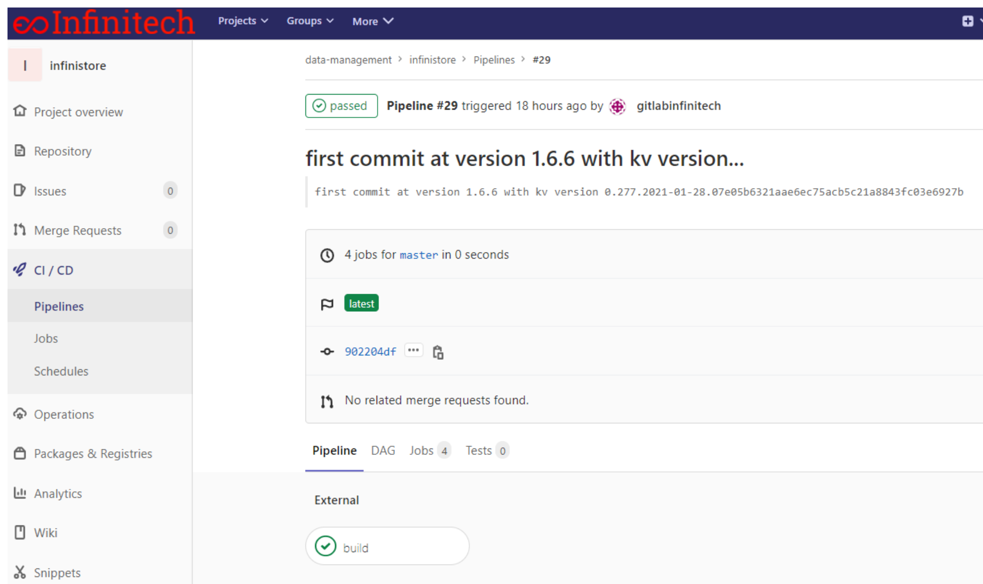


Figure 29 – Specific pipeline details

To have more details, it is also possible to see the pipeline inside Jenkins simply by clicking on the build: Gitlab will redirect the user to the appropriate Jenkins pipeline. After providing the user's corresponding credentials to Jenkins (which are the same as for the Gitlab), it is also possible to click on the console output and check the logs to identify reasons for possible failures or other statistics needed (Figure 30).

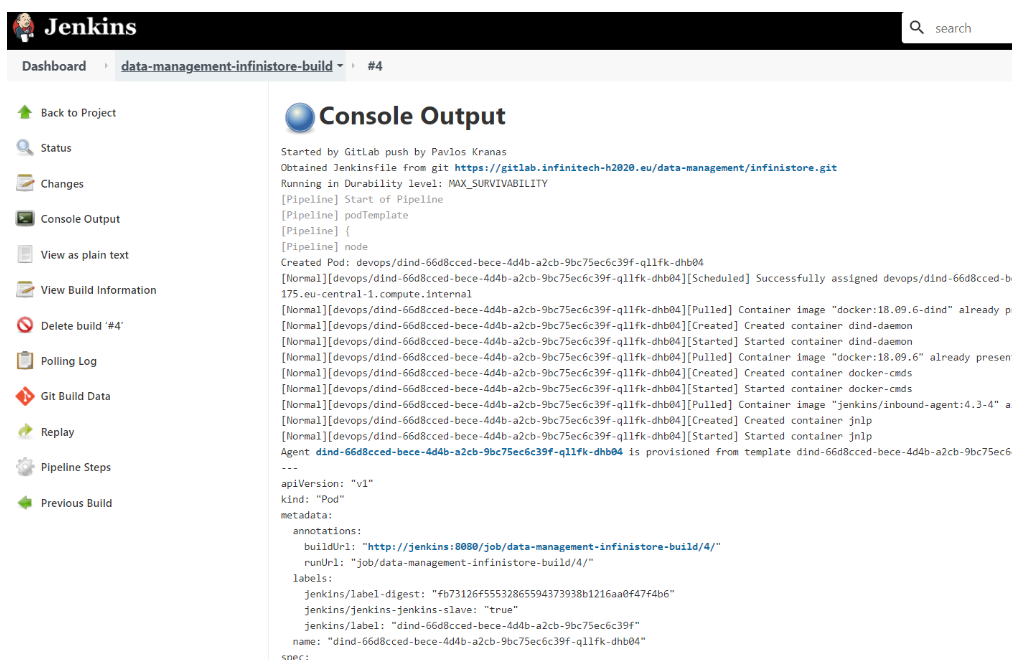


Figure 30 – Checking the logs of a CI pipeline

If the user has managed to successfully build the image using the CI pipeline, the software artifacts have successfully built and added in the project's docker registry. Now it is available and can be used by pilot developers who are building their solutions based on the INFINITECH building blocks. It is important to highlight at this point that all artifacts hosted by these INFINITECH groups must be of general purpose and not bind to a specific pilot. As they must be used by different solutions, there might be a need for a specific instantiation of the component, once it has been deployed. For instance, the Infinistore might need to be instantiated with a predefined data schema. That means, there might be the need to automatically create the data tables and structures that will be used for storing the data, once the component is up and running.

In that sense, the platform and analytic providers can allow the use of configuration files that will be taken into account when the component is starting. More details on those scenarios can be found in the section related with the continuous deployment (CD) and the corresponding pipelines.

6.3 Building an image using pre-compiled libraries/binaries

An image can be built upon the source code. The CI pipeline compiles the code and builds the image using the compiled binaries. However, in many cases, the source code is private and cannot be uploaded to the internal repository of the project due to IPR rights. Or in other cases, the developer does not own the whole source code and makes use of open source libraries that are available to the community. In both cases, the image is built based on pre-compiled libraries or binaries in general. As it is not good practice to upload such files to Git, where only source code and script files should be contained, there are two alternatives for such cases: either to use an already built image as a base, or to make use of the INFINITECH package repository. The following subsections give more details on these.

6.3.1 Building an image using pre-compiled libraries/binaries

The easiest way to handle scenarios where there is a need to build an image using binaries, is to firstly create a base image. In this section the *Infinistore* project will be used as an example. This project consists of the files shown in Figure 31.

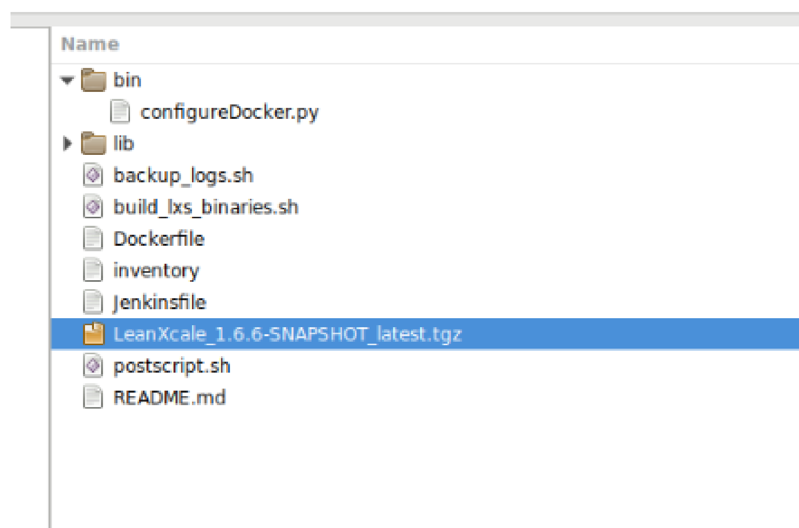


Figure 31 – Infinistore project files

The tgz file is a binary which contains the datastore of INFINITECH. This cannot be uploaded to Git. Instead, the user must build a base image locally. To do so, it is necessary to firstly login to the docker harbor, build the image locally giving a corresponding name, and then push the image to the harbor:

```
docker login harbor.infinitech-h2020.eu
docker build -t harbor.infinitech-h2020.eu/data-management/infinistore:base .
docker push harbor.infinitech-h2020.eu/data-management/infinistore:basevail
```

In this example the group is *data-management,* the image is called *Infinistore* and it is tagged with the *base* attribute, to distinguish it from the *latest* that will be created by the pipeline. Having pushed the base image to the Infinitech Harbor, now it can be retrievable from Gitlab. It is necessary however to create the project under the corresponding group and setup the CI pipelines there using Jenkins and a Dockerfile to build the

image. This is a necessity, otherwise the component developed cannot be retrievable by others, nor by the marketplace. The dockerfile inside the *Infinistore* project now can be as simple as the following:

```
FROM infinistore:base
MAINTAINER Pavlos Kranas <pavlos@leanxcale.com>
WORKDIR ${BASEDIR}
CMD cd ${BASEDIR} \
  && ./start.sh
```

This will make use of the pre-built base image, and only defines the command to execute when the container is created. This approach has the benefit that it is simple enough and straightforward. It comes, however, with the drawback that the image must be maintained by the user locally, and the user cannot benefit from a versioning repository such as the Gitlab. In fact, in this way the developer is versioning the whole application container on the Harbour repository and the document changes done in the current release can be documented using a release file. Moreover, developers can test in their own environment the container before deploying the blueprint solution. The next approach, explained in section 6.3.2, removes this drawback, even if it requires some configuration steps beforehand.

6.3.2 Building an image using a package container

A package container has been deployed in Gitlab, which allows us to store libraries and binary files. It is a maven [33] artifactory. Usually, maven artifactories are used to store Java libraries that can be shared among different projects. The developer configures maven projects defining the corresponding dependencies, and the maven software management tool downloads the corresponding jar files from various artifactories. The developer, after compiling the code, can deploy the produced java archive into his own artifactory and make it available to other projects. This is a well-known procedure/practice for java developers, so no more details will be given on this specific process. However, the maven artifactory can be used to upload whichever type of files. We make use of this, to upload the tgz file used in the previous scenario, taking as example the *Infinistore* project.

In order to make use of the maven artifactory, a *personal access token* [34] is required. This is an alphanumeric that grants access to the artifactory. It is assumed that the user is familiar with, and has already installed, maven in his local machine. Once the token has been created, it is necessary to configure Maven to make use of it by editing the settings.xml file. This is located in different places according to the OS, but the most common are ~/.m2 and /usr/share/maven/conf. To proceed, the following lines should be added to the *server* section of this file:

```
<server>
  <id>gitlab-maven-infinitech</id>
  <configuration>
    <httpHeaders>
      <property>
        <name>Private-Token</name>
        <value>XXXXXXXX</value>
      </property>
    </httpHeaders>
  </configuration>
</server>
<server>
  <id>gitlab-maven-infinitech-dist</id>
  <configuration>
    <httpHeaders>
      <property>
        <name>Private-Token</name>
        <value>XXXXXXXX</value>
      </property>
```

```
    </httpHeaders>
  </configuration>
</server>
```

Then, a new maven project should be created to upload the project binaries to the artifactory. The binaries should be placed under this project, so it should look like shown in Figure 32.

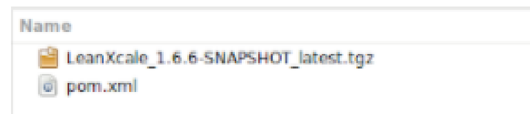


Figure 32 – Maven project files

The pom.xml in Figure 32 must be edited to define the repository where files will be stored, it is possible to make use of a maven plugin to do this job automatically. The pom should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>eu.infinitech</groupId>
    <artifactId>infinistore-bins</artifactId>
    <version>1.6.6-SNAPSHOT</version>
    <packaging>pom</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <repositories>
        <repository>
          <id>gitlab-maven-infinitech</id>
          <url>https://gitlab.infinitech-h2020.eu/api/v4/packages/maven</url>
        </repository>
    </repositories>

    <distributionManagement>
        <repository>
          <id>gitlab-maven-infinitech-dist</id>
          <url>https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven</url>
        </repository>
        <snapshotRepository>
          <id>gitlab-maven-infinitech-dist</id>
          <url>https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven</url>
        </snapshotRepository>
    </distributionManagement>

    <build>
        <plugins>
            <!--this plugin deploys binary files to the artifactory-->
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>build-helper-maven-plugin</artifactId>
                <version>1.10</version>
                <executions>
                  <execution>
                    <id>attach-artifacts</id>
                    <phase>package</phase>
                    <goals>
                      <goal>attach-artifact</goal>
```

```
                    </goals>
                    <configuration>
                      <artifacts>
                        <artifact>
                          <file>LeanXcale_1.6.6-SNAPSHOT_latest.tgz</file>
                          <type>tgz</type>
                        </artifact>
                      </artifacts>
                    </configuration>
                  </execution>
                </executions>
              </plugin>
        </plugins>
    </build>
</project>
```

It is important to point out that the id of the repository in this file must match one of the ones previously declared in the file settings.xml. When it tries to connect there, it will make use of the private token created in a previous step. Another thing that is important to notice is the url of the distribution management section: **https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven**. Here, the number 33 corresponds to the id of the project that uses this binary. In this case, it is 33 as depicted in the Figure 33. Moreover, the build-helper-maven-plugin was used, and the name of the file to upload and its type were defined. In case there are several binary files, it is possible to define several *artifacts* under the *artifacts* element.

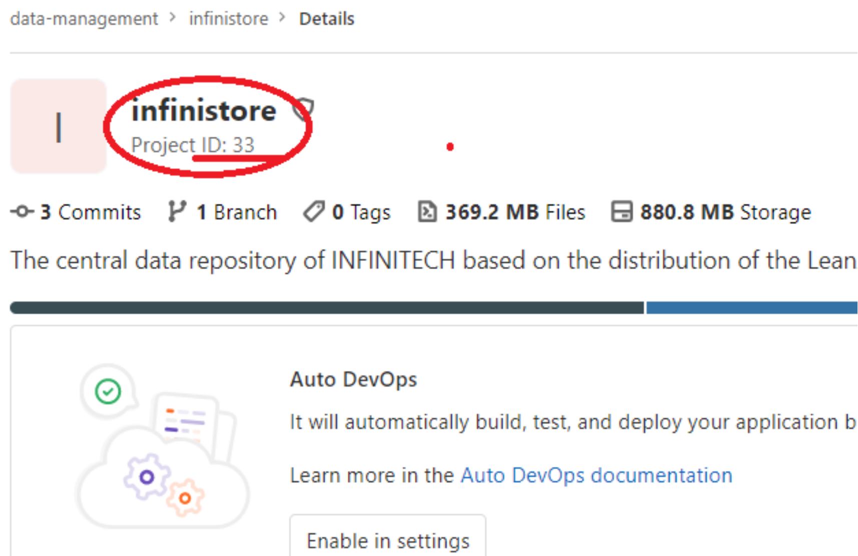


Figure 33 – Project ID

Once the file pom.xml has been edited, it is possible to pull the binaries to the artifactory. The command to realize this action is “mvn clean deploy”. The result should look as shown in Figure 34.

```
pavlos@LX-pavlos:~/leanxcale/Projects/INFINITECH/dev/data-management/infinistore-bins$ mvn clean deploy
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building infinistore-bins 1.6.6-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ infinistore-bins ---
[INFO]
[INFO] --- build-helper-maven-plugin:1.10:attach-artifact (attach-artifacts) @ infinistore-bins ---
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ infinistore-bins ---
[INFO] Installing /home/pavlos/leanxcale/Projects/INFINITECH/dev/data-management/infinistore-bins/pom.xml to /home/pavlos/.m2/repository/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/infinist
HOT.pom
[INFO] Installing /home/pavlos/leanxcale/Projects/INFINITECH/dev/data-management/infinistore-bins/LeanXcale_1.6.6-SNAPSHOT_latest.tgz to /home/pavlos/.m2/repository/eu/infinitech/infinistore-
infinistore-bins-1.6.6-SNAPSHOT.tgz
[INFO]
[INFO] --- maven-deploy-plugin:2.7:deploy (default-deploy) @ infinistore-bins ---
Downloading: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/maven-metadata.xml
Downloaded: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/maven-metadata.xml (955 B at 906 B/s)
Uploading: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/infinistore-bins-1.6.6-20210211.135323-3.pom
Uploaded: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/infinistore-bins-1.6.6-20210211.135323-3.pom (2.2 kB at 2.6 kB/s)
Downloading: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/maven-metadata.xml
Downloaded: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/maven-metadata.xml (289 B at 1.2 kB/s)
Uploading: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/maven-metadata.xml
Uploaded: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/maven-metadata.xml (955 B at 1.1 kB/s)
Uploading: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/maven-metadata.xml
Uploaded: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/maven-metadata.xml (289 B at 357 B/s)
Uploading: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/infinistore-bins-1.6.6-20210211.135323-3.tgz
Uploaded: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/infinistore-bins-1.6.6-20210211.135323-3.tgz (268 MB at 278 kB/s)
Uploading: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/maven-metadata.xml
Uploaded: https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-bins/1.6.6-SNAPSHOT/maven-metadata.xml (955 B at 1.3 kB/s)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 16:09 min
[INFO] Finished at: 2021-02-11T15:09:30+01:00
[INFO] Final Memory: 11M/176M
[INFO] ------------------------------------------------------------------------
```

Figure 34 – mvn command output

Pay attention to the url that the tgz file has uploaded; it will be used soon. Now that the binary is available through the artifactory, it is possible to copy it from there, instead of from the Gitlab. Now the Gitlab contains the files depicted in Figure 35, where there is no binary included.

master | infinistore / + | History | Find file | Web IDE | Clone

change binaries to tgz
Pavlos Kranas authored 3 hours ago — 9d9fc09d

README | Add LICENSE | Add CHANGELOG | Add CONTRIBUTING | Add Kubernetes cluster | Set up CI/CD

Name	Last commit	Last update
bin	first commit at version 1.6.6 with kv version 0.277.2021-01-28.07e05b63...	1 week ago
Dockerfile	change binaries to tgz	3 hours ago
Jenkinsfile	first commit at version 1.6.6 with kv version 0.277.2021-01-28.07e05b63...	1 week ago
README.md	first commit at version 1.6.6 with kv version 0.277.2021-01-28.07e05b63...	1 week ago
backup_logs.sh	first commit at version 1.6.6 with kv version 0.277.2021-01-28.07e05b63...	1 week ago
build_lxs_binaries.sh	first commit at version 1.6.6 with kv version 0.277.2021-01-28.07e05b63...	1 week ago
inventory	first commit at version 1.6.6 with kv version 0.277.2021-01-28.07e05b63...	1 week ago
postscript.sh	first commit at version 1.6.6 with kv version 0.277.2021-01-28.07e05b63...	1 week ago

Figure 35 – Gitlab project files

The Dockerfile inside the *infinitech* project must copy these files and make use of the url shown in the result of the maven execution to copy the binaries from the artifactory. A code snippet of the Dockerfile looks like the following:

```
FROM ubuntu:18.04

MAINTAINER  Pavlos Kranas <pavlos@leanxale.com>

#Add prerequisitites (i.e python, java, libraries etc)
RUN apt-get update && apt-get -y install software-properties-common \
 && add-apt-repository -y ppa:webupd8team/java \
```

```
 && add-apt-repository -y ppa:openjdk-r/ppa \
 && apt-get update \
 && apt-get -y install screen iputils-ping netcat vim vsftpd net-tools python3 python3-dev \
 python3-pip libssl-dev silversearcher-ag bc libc6-dbg gdb openssh-server lsof psmisc \
binutils  virtualenv telnet apt-transport-https openjdk-8-jdk iproute2 curl gcc python-dev git \
 && mkdir /var/run/sshd \
 && rm -rf /var/lib/apt/lists/* \
 && rm -rf /usr/share/doc && rm -rf /usr/share/man \
 && apt-get clean \
 && pip3 install pyjolokia flask pyyaml  psutil

#add environment variables for LXS
ENV BASEDIR="/lx" \

#copy the binaries
COPY inventory /tmp/inventory

RUN cd ${BASEDIR} \
 && curl --header 'Private-Token:XXXX' -L -O \
 && https://gitlab.infinitech-h2020.eu/api/v4/projects/33/packages/maven/eu/infinitech/infinistore-
bins/1.6.6-SNAPSHOT/infinistore-bins-1.6.6-20210211.095013-2.tgz
.
.
.
```

The dockerfile above instructs docker on how to build the image: it starts from the base Ubuntu 18.04 image and then it installs the various prerequisites needed in the runtime. Afterwards, it starts copying files from the git and it makes use of the *curl* command to grab the binaries from the artifactory, using the url it has got from maven. It is worth noticing that the token created in the beginning is used by the pipeline to grant access to the artifactory.

This second approach gives the user the advantage of using a versioning system such as Gitlab to maintain the code. In this way all scripts and files related to the offering are put there, except for the precompiled libraries/binaries. On the other hand, scripts related to the configuration of a component or the way it is started will most likely change frequently during the development and integration process. Having these files in Gitlab gives the benefit of versioning them. Therefore, even if this process seems more complicated than providing a base image, the whole configuration takes place once at the very beginning, and then everything can be put in place in the package container using the maven tool.

6.3.3 Building a pilot-specific component

This section describes the process on how to deploy components that have been implemented specifically for a pilot solution. It follows the same approach as section 6.2 with the difference that the project must be kept on Gitlab, in a separate group.

To demonstrate the process, an imaginary pilot called Pilot-Red will be created, it consists of two INFINITECH building blocks, the *Infinistore* and the *lx-kafka*, and a micro-service that implements a backend functionality specific for this pilot. The architecture of this solution is shown in Figure 36.

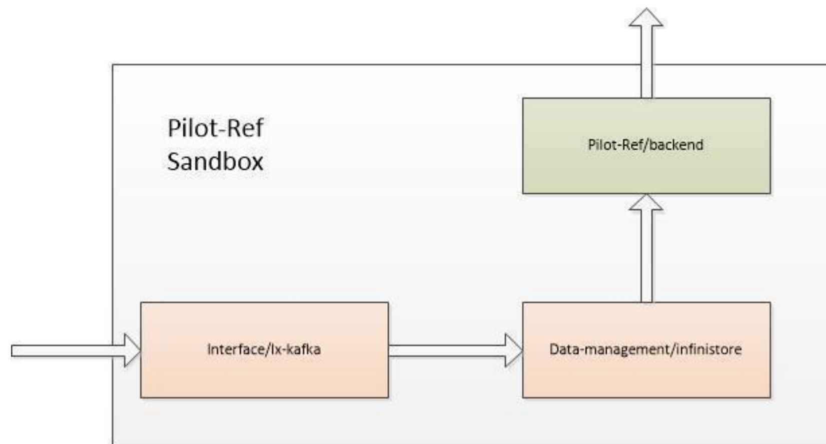


Figure 36 – Overall architecture of the Pilot-Ref solution

In the diagram of Figure 36, the arrows show the data flow. In this imaginary pilot, data come in the form of a data stream from an external resource and is being pushed to a kafka queue. From there, the data are being directly ingested into the INFINITECH datastore. Finally, a microservice exposes a REST API for the data consumer to retrieve data from the datastore. The *Infinistore* provided by INFINITECH, which is available under the *data-management* group, and the kafka queue provided also by INFINITECH, which is available under the *interface* group, will be used to build the imaginary pilot Pilot-Ref. The microservice is pilot-specific, and it is available under the *pilot-ref/backend* project (under the Pilot-Ref group) which means that it has been implemented specifically for the needs of this imaginary pilot. As a result, the deployed sandbox for Pilot-Ref will consist of those 3 artifacts. From the diagram, there is clearly a necessity to allow access from an external source to the deployed sandbox "to *put"* (so that the data stream can be pushed inside the kafka queue), and also to allow access from an external source "to *get"* (so that the REST API can be invoked from a mobile application). This section will cover all the details of how to solve all these requirements and how to deploy Pilot-Ref.

As already mentioned, before the *Infinistore* is already available under the *data-management* group and can be used by various pilots. Regarding the micro-service of the backend, it exposes a list of REST endpoints that perform some basic operations over the data management layer: they read data from tables and return the data in JSON format. The backend functionality has been implemented in Java and makes use of an embedded Tomcat servlet container, which will deploy the REST web services, along with the swagger that documents the exposed functionality.

In this group, just the pilot-specific *backend* should be provided*,* as all other components are already available by the INFINITECH itself. A Dockerfile will provide instructions on how to build the image of the *backend* and a Jenkinsfile created for the *all* projects will be used as template, changing the name of the target image.

Following the guidelines presented in the previous sections, a new group called *Pilot-Ref* will be created, it will host all projects that are specific to this pilot. It is possible to create a new group by visiting this Gitlab explore group url [35] (see Figure 37).

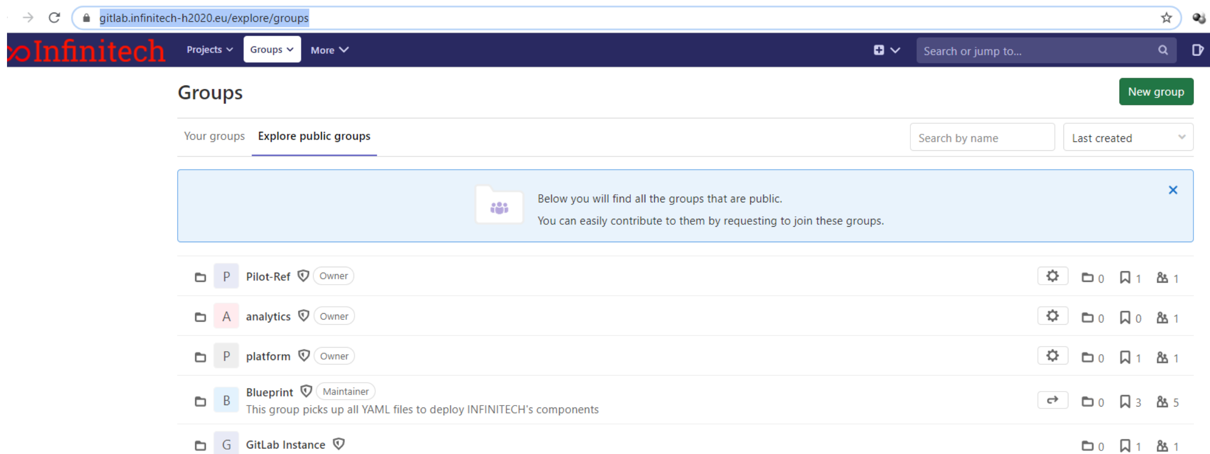


Figure 37 – Gitlab list of available groups

After naming the project as Pilot-Ref it is important to change the visibility to ***internal***, in this way it is possible to allow access to some partners from the consortium. The newly created group can be accessed by clicking the following url [36]. For the real pilots of the project, the name should be Pilot1, Pilot2 etc., while the url should be changed accordingly. Now that a specific group for the imaginary Pilot-Ref has been created, it is time to add the pilot-specific projects under this group. For each of the components, a new git project must be created. The imaginary pilot Pilot-Ref has only one component called *backend*. The process to follow is the same as for the other projects. The only thing that changes is that the projects should be placed under the pilot-ref group, this is shown in Figure 38.

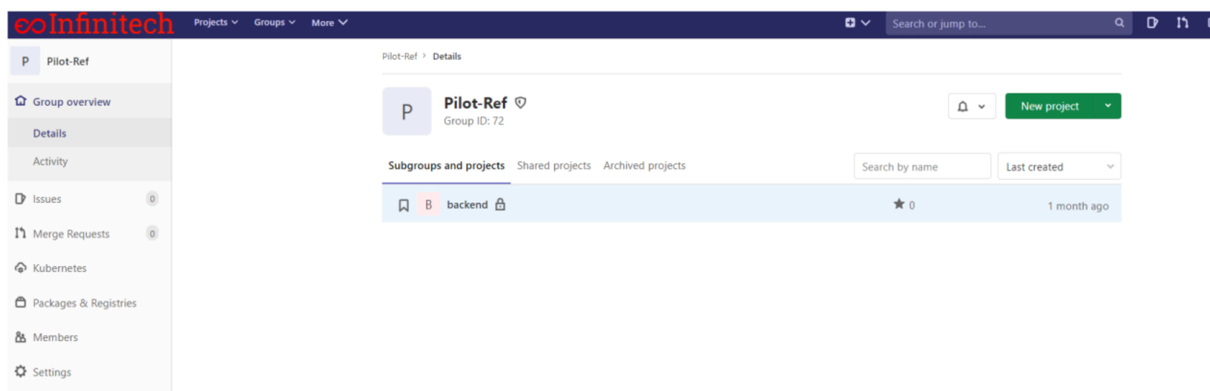


Figure 38 – Creating the pilot-specific backend project

Clicking on this project, it is possible to see its details. Once again, it is very important to add the gitlabinfinitech user in the members of this group with the role maintainer, as depicted in Figure 39. This will allow the execution of the CI pipelines.

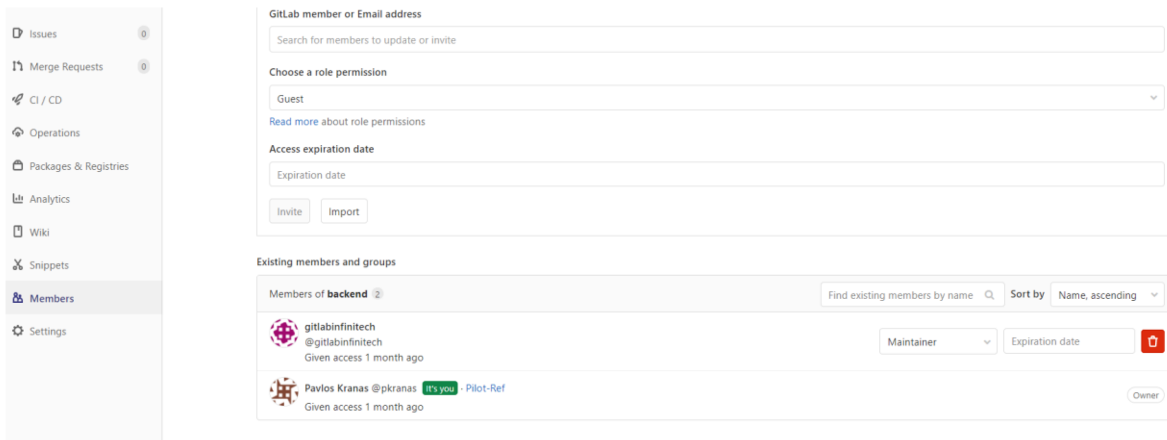


Figure 39 – Gitlab Project users and roles

In Figure 40 it is assumed that a git clone of the empty project has been already performed and all the files have been already pushed under the master branch: the figure shows how the project looks like now.

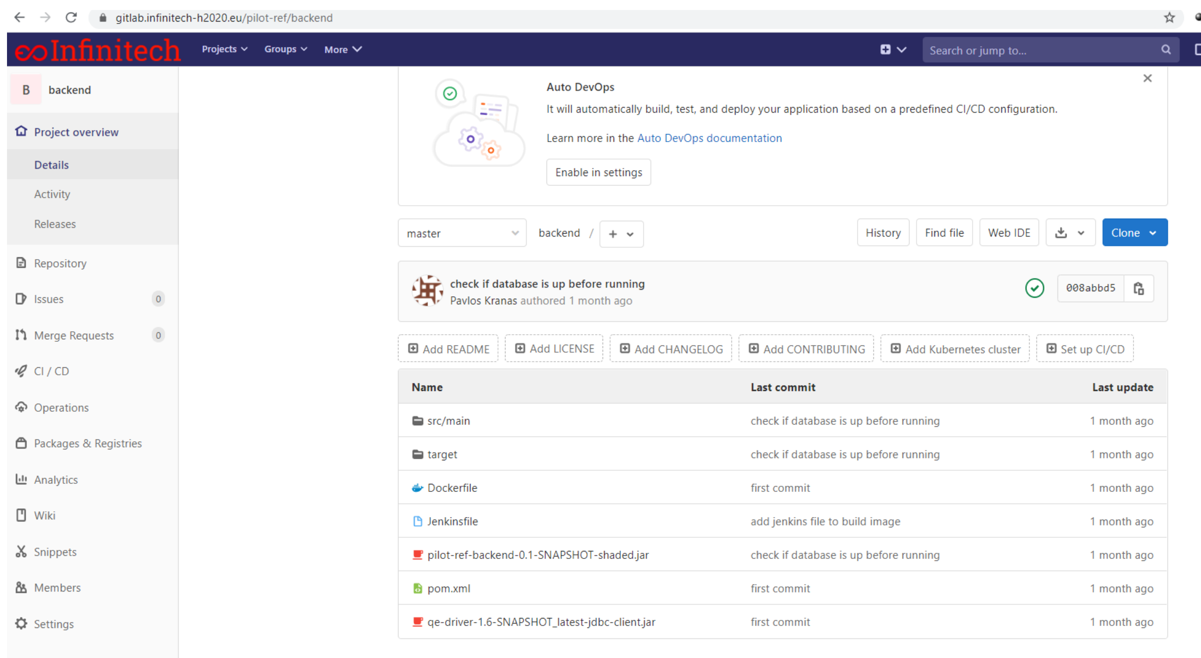


Figure 40 – Our pilot-ref/backend project

The Dockerfile's instructions are as follows:

```
FROM ubuntu:18.04

MAINTAINER  Pavlos Kranas <pavlos@leanxale.com>

ENV BACKEND_HOME="/lx"
ENV BACKEND_DIST="/lx_dist"

RUN apt-get update && apt-get -y install software-properties-common \
 && add-apt-repository -y ppa:webupd8team/java \
 && add-apt-repository -y ppa:openjdk-r/ppa \
 && apt-get update \
 && apt-get -y install screen iputils-ping netcat vim vsftpd net-tools \
 openssh-server virtualenv telnet apt-transport-https openjdk-8-jdk curl  \
 && groupadd -r appuser \
 && useradd -d ${BACKEND_HOME} -r -s /bin/false -g appuser appuser \
```

```
 && mkdir ${BACKEND_HOME} \
 && mkdir ${BACKEND_DIST} \
 && mkdir /tmp/src/ \
 && mkdir /tmp/src/main/ \
 && mkdir /tmp/src/main/webapp

RUN curl --header 'Private-Token:XXXX' -L -O \
 https://gitlab.infinitech-h2020.eu/api/v4/projects/54/packages/maven/eu/infinitech/infinistore-bins/0.1-SNAPSHOT/pilot-ref-backend-0.1-SNAPSHOT-shaded.jar
RUN curl --header 'Private-Token:XXXX' -L -O \
 https://gitlab.infinitech-h2020.eu/api/v4/projects/54/packages/maven/eu/infinitech/infinistore-bins/0.1-SNAPSHOT/webapp.tgz
RUN tar vfzx webapp.tgz


RUN chown -R appuser:appuser ${BACKEND_HOME}

EXPOSE 22 23 54735

WORKDIR /tmp

CMD cd /tmp \
  && java -jar pilot-ref-backend-0.1-SNAPSHOT-shaded.jar
```

Similarly, instead of creating the Jenkinsfile from scratch, it is convenient to use a Jenkinsfile defined in other projects and modify just the corresponding names as shown in Figure 41.

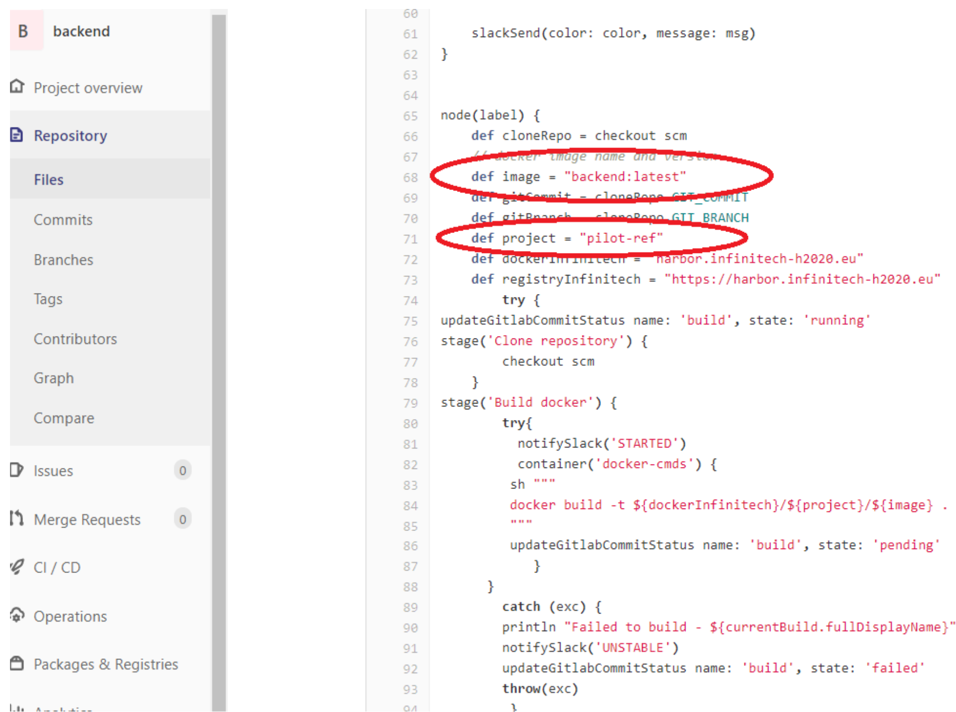


Figure 41 – Setting Jenkins for our pilot-ref/backend project

As shown in Figure 41, the name of the image is *backend* which is under the *pilot-ref* project. By setting both the name and the project, it is impossible to have name conflicts between components that share the same name (i.e. backend) but belong to different pilot solutions, which means in different groups as well.

Once the user has defined everything, it is necessary to communicate, via slack channel, to the HPE team that is responsible for setting up and configuring the *webhooks* that will be triggered to automatically build the image and make it available when something is being pushed to the master branch. The procedure is the

same as for the artifacts that belong to the *INFINITECH* groups. Figure 42 shows a pipeline that has been passed.

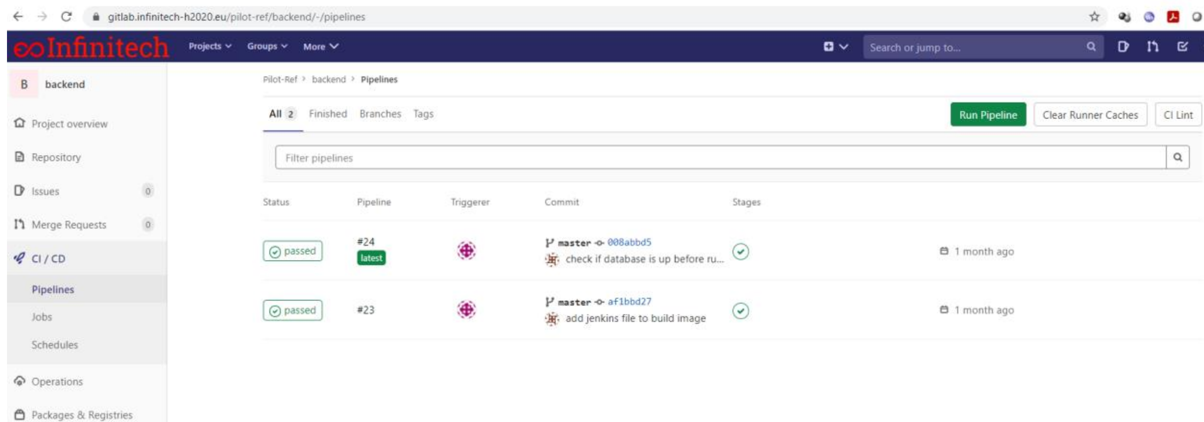


Figure 42 – Checking the CI pipeline for our pilot project

Clicking on the #24 pipeline, it is possible to see the logs of its execution, after being redirected to the Jenkins page (see Figure 43).

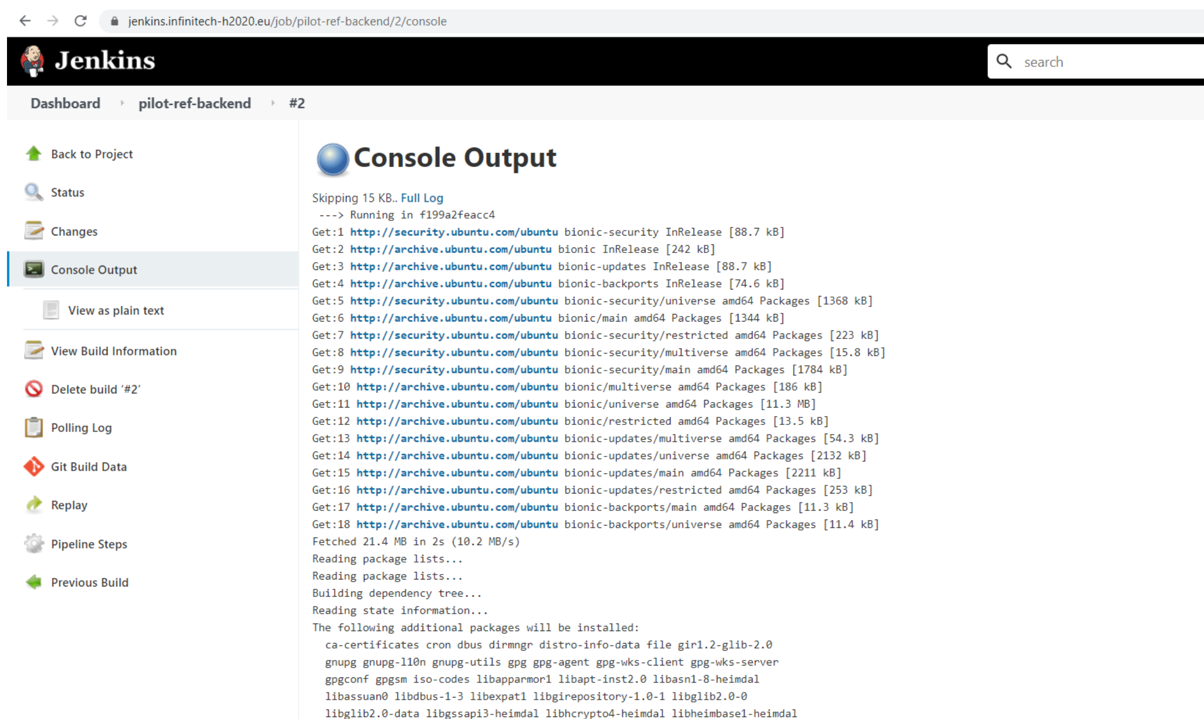


Figure 43 – Checking the logs of the CI pipeline of our pilot project

Once the pipeline has been passed, the artifact is pushed into the project's docker registry and can be used in the deployment. Now that all pieces are in place, it is time to proceed to the following section that will demonstrate how to deploy the imaginary pilot solution using the CD tools and the artifacts already provided.

6.4 Deploy a pilot use case in a Sandbox

This section introduces the continuous deployment process and will provide the guidelines on how to define the manifest files that describe the deployment of a pilot solution.

As mentioned in the previous sections 3.4 and 3.5, the overall pilot integrated solution might consist of several use cases that need to be deployed in the testbed inside a sandbox. Compliant with the INFINITECH terminology, a sandbox reflects whatever is inside a Kubernetes namespace and refers to a specific use case. As a fact, an integrated pilot solution might consist of several use cases, and therefore several sandboxes.

In the following, the imaginary pilot Pilot-Ref used in the previous sections, (which corresponds to a single use case) will be deployed inside a sandbox. This sandbox will consist of the *Infinistore* and the *lx-kafka* queue, which are core INFINITECH building blocks, available for all pilots under the corresponding INFINITECH groups, and a *backend* micro-service, that has been implemented specifically for the imaginary pilot, which provides pilot-specific backend functionalities. Another pilot solution might be more complicated, making use of the *data-layer* of INFINITECH, its *semantic interoperability framework,* functionalities that enable the use of *Blockchain,* the *data ingestion* artifact and a set of *data regulatory tools,* all provided by INFINITECH, whose components will be stored under the *INFINITECH* group. Moreover, the pilot might make use of 2-3 AI tools implemented in the scope of INFINITECH and provided under the *analytics* group, while also relying on one key AI implementation that is specific to this pilot, 2 pilot-specific micro-services and a front-end UI tool. The AI tool implemented specifically for the pilot, along with the 2 micro-services and the UI code must be placed under the PilotX group.

In order to define the CD pipelines, the *blueprint* group should be used [37]. If the user has not been granted access to this group, it is sufficient to ask the HPE team using the *cicd* resources slack channel. Once the access is granted, the first thing to do is to create the corresponding project for the Pilot-Ref deployment. The process for creating this project is the same as in the previous sections (click on the *new project* button and create a new project). The user needs to ensure that it will grant access to the *gitlabinfinitech* user that will allow the CD pipelines to be executed. Once the project has been created, the user needs now to define the manifest files for the deployment.

It is important to highlight here that the pilot projects under the *blueprint* group must not contain any code or other building blocks. The artifacts for all building blocks must be defined in the corresponding *INFINITECH* and *pilot* groups, where the images can be built. The *blueprint* group is the place where the manifest files is placed, along with any configuration files that must be injected into the running containers to instantiate the deployment of each of the building blocks according to the needs of the pilot. Therefore, the deployment directives must be defined and loaded on such build projects by using Kubernetes manifest files. If the developers/integrators are unable to write a Kubernetes manifest, they can still create a temporal docker-compose file and share it with the HPE team, so that they can be aware of the architecture of the overall solution and produce the relevant Kubernetes files on the behalf of the pilots. These descriptors must contain detail about:

- Constraints on resource: cpu and memory required and limits
- Storage: needs for persistent volumes
- Network requirements: connection with outside networks, load balancer, other specific

An additional Readme document must be also completed by the developer, and it will contain a general description of the use case it deploys and other information that must be taken into account during the deployment.

For the imaginary pilot, the docker-compose could be the following:

```
version: '3.1'

services:
  infinistore:
    image: harbor.infinitech-h2020.eu/data-management/infinistore:latest
```

```
    container_name: infinistore
    restart: unless-stopped
    mem_limit: 8g
    mem_reservation: 4g
    cpus: 2
    ports:
      - 2181:2181
      - 1529:1529
      - 9876:9876
      - 9992:9992
      - 14400:14400  backend:
    image: harbor.infinitech-h2020.eu/pilot-ref/backend:latest
    container_name: backend
    restart: unless-stopped
    mem_limit: 512m
    mem_reservation: 128m
    cpus: 0.5
    ports:
      - 54735:54735
    depends_on:
      - infinistore
    links:
      - infinistore
    environment:
      - DATASTORE_HOST=infinistore
      - USEIP=yes
  lx-kafka:
    image: harbor.infinitech-h2020.eu/interface/lx-kafka:latest
    container_name: lx-kafka
    restart: unless-stopped
    mem_limit: 2g
    mem_reservation: 1g
    cpus: 1
    ports:
      - 8081:8081
      - 9092:9092
    depends_on:
      - infinistore
    links:
      - infinistore
```

In the above code snippet three *services* are defined: the *Infinistore and lx-kafka,* along with the *backend* that makes use of the *Infinistore*. The *image* attribute defines the image name that each of the *services* will make use of. It must match one of the available images in the docker Harbour, as defined in the Jenkinsfile in the CI pipelines. It is also necessary to provide the resource requirements in the Readme file, so that they can be taken into account and included in the generated Kubernetes manifests.

It must be highlighted at this point that there might be a need for some pilots not to put their pilot-specific images in the common Harbour used in the project due to IPR rights or other constraints. This is feasible also in this scenario, the image source of the corresponding service will point to the docker registry of the pilot, which is private to it. This means that the *backend* component will point to an internal registry. As a result, the pilot is deployed in the common infrastructure, the deployment orchestrator will grab the images for *Infinistore* and *lx-kafka,* but it will fail to get the image for the *backend*, as it has no access. However, if the deployment is triggered in an infrastructure installed in the premise of the pilot, then the deployment orchestrator will grab the *Infinistore* and *lx-kafka* ones, and it will also be granted access to the pilot-specific ones. As a result, the process will succeed and the deployment can only take place inside the organization. It is worth noting that for the solutions that cannot be deployed in the common infrastructure, there is no need to provide a Jenkins file for the automated deployment, as it will always fail. The system administrators of

the organization can git clone the relevant pilot blueprint, and manually deploy the solution on their premises.

Taking into account that both *Infinistore* and *lx-kafka* are components of general purpose, they must be instantiated accordingly during the deployment process. Moreover, each component must reach each other over the network during the runtime, after the deployment of the sandbox. As this happens dynamically, it is not possible to know beforehand network addresses etc. So the imaginary pilot has the following requirements:

1. The *backend* must connect to the *Infinistore* using a *connection url,* but it cannot know in advance the domain name of the container that hosts the datastore.
2. The *lx-kafka* must connect to the *Infinistore* in a similar way, which means that it cannot know its domain beforehand.
3. The *lx-kafka* is of general purpose, therefore it cannot know in advance the name of the schema of the incoming data tuples. This information must be provided during the deployment phase and let the queue instantiate itself accordingly.
4. There is no such need for the *Infinistore,* as the schema (in this scenario) will be automatically created by the corresponding connector of the *lx-kafka.*
5. As the schema will be created based on the configuration file that is specific for the pilot, the *backend* (which has been also implemented for the specific needs of the pilot) is already aware of it.
6. A mobile device must consume the REST services exposed by the *backend,* but it cannot know the internal IP of the container.
7. The same goes for the stream of data that needs to push data items into the queue.
8. When connecting to a Kafka queue, the latter returns to the client the list of its *brokers* so that the client can connect to those brokers. To do so, Kafka must be configured to *advertise* the IP of its brokers. However, it cannot know the external IP of the sandbox beforehand.
9. An external *service* in Kube's terminology must be defined that allows access from external sources. Kafka must be configured with that domain name.

The following paragraphs show how it is possible to meet all these requirements and configure the solutions to be portable and deployed in whichever is the target infrastructure, using Kubernetes.

For 1, it is suggested to use an environment variable for the domain name of the *Infinistore.* For instance, in the docker-compose file, this is defined in *DATASTORE_HOST=infinistore* where *Infinistore* is the service that hosts the datastore. The value of this variable would have been assigned automatically by the deployment orchestrator during the deployment time, in case docker-compose is used. To be compliant with Kubernetes, this environment variable needs to be replaced by the name of the *service kind* of Kubernetes, that enables the network connectivity, and the *Infinistore* domain name must be replaced by the name of this *service kind.*

For 2 similarly, *lx-kafka* uses a configuration file to instantiate the instance accordingly. A template of this file can be shown in the following code snippet.

```
name=local-lx-sink
connector.class=com.leanxcale.connector.kafka.LXSinkConnector
tasks.max=1
topics=REF_TOPIC

connection.url=REF_CONNECTION_URL
connection.user=APP
```

```
connection.password=APP
connection.database=REF_CONNECTION_DATABASE
auto.create=true
batch.size=500
connection.check.timeout=20

key.converter=io.confluent.connect.avro.AvroConverter
key.converter.schema.registry.url=http://localhost:8081
value.converter=io.confluent.connect.avro.AvroConverter
value.converter.schema.registry.url=http://localhost:8081

sink.connection.mode=kivi
sink.transactional=false

table.name.format=${topic}
pk.mode=record_key
pk.fields=REF_PK_FIELDS
fields.whitelist=REF_FIELDS_WHITELIST
fields.whitelist=trip_duration_secs,start_time,stop_time,start_station_id,start_station_name, \
start_station_latitude,start_station_longitude,end_station_id,end_station_name,
end_station_latitude, \
end_station_longitude,bike_id,user_type,user_birth_year,user_gender
```

Also, the *connection.url* property makes use of the *Infinistore* value that need to be replaced with the actual value of the container of *Infinistore* that will be resolved during deployment time. In such scenarios, *configmaps* provided by the Kubernetes are used to define the runtime parameters. An example can be shown at the following code snippet:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: lx-kafka-config-map
  namespace: pilot-ref
data:
  advertised.url: {name_of_service}
  topics: GTP
  connection.url: kivi:lxis://{name_of_service}:9876
  connection.database: PILOTREF
  pk.fields: id
  fields.whitelist: trip_duration_secs,start_time,stop_time,start_station_id,start_station_name, \
start_station_latitude,start_station_longitude,end_station_id,end_station_name,
end_station_latitude, \
end_station_longitude,bike_id,user_type,user_birth_year,user_gender
```

During the deployment, the values of the parameters defined in this *configmap* will be available as environment variables to the newly-created pod/container. These can be obtained by the software component. In case the latter relies on such properties files, this information must be provided to the properties files, before the software process starts. The following script can do this trick.

```
sed -i "s/IPOFTHEMACHINE/$(hostname -i)/g" ${KAFKA_HOME}/etc/kafka/server.properties
sed -i "s/IPOFTHEMACHINE/$(hostname -i)/g" ${KAFKA_HOME}/etc/kafka/connect-standalone.properties

sed -i "s/REF_TOPIC/${topics}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-sink.properties
sed -i "s/REF_CONNECTION_URL/${connection_url}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-sink.properties
sed -i "s/REF_CONNECTION_DATABASE/${connection_database}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-
sink.properties
sed -i "s/REF_PK_FIELDS/${pk_fields}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-sink.properties
sed -i "s/REF_FIELDS_WHITELIST/${fields_whitelist}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-
sink.properties
sed -i "s/IP_EXTERNAL_OF_THE_MACHINE/${advertised_url}/g" ${KAFKA_HOME}/etc/kafka/server.properties
```

```
echo 'will wait a bit for the datastore to start'
sleep 20
echo 'will start zookeeper'
nohup ${KAFKA_HOME}/bin/zookeeper-server-start ${KAFKA_HOME}/etc/kafka/zookeeper.properties >
nohup_zk.out &
sleep 2
echo 'will start kafka server'
nohup ${KAFKA_HOME}/bin/kafka-server-start ${KAFKA_HOME}/etc/kafka/server.properties >
nohup_kafka_server.out &
sleep 5
echo 'will start schema registry'
nohup ${KAFKA_HOME}/bin/schema-registry-start ${KAFKA_HOME}/etc/schema-registry/schema-
registry.properties > nohup_schema_registry_server.out &
sleep 3
echo 'will start lxs connector'
nohup ${KAFKA_HOME}/bin/connect-standalone ${KAFKA_HOME}/etc/kafka/connect-standalone.properties
${KAFKA_HOME}/etc/kafka/connect-lx-sink.properties > nohup_connect_lx.out &
sleep infinity
```

This script is defined to be executed via the CMD docker instruction when the container is created. This container has 4 different java processes that run inside and the processes are configured via properties file. Before starting the processes, the *sed* batch command is used to override the properties file, placing the values of the environment variables passed via the *configmap*. After replacing the *placeholders* (i.e. the **REF_TOPIC** in this case) with these values (i.e. GTP), the java processes can be safely started in the background.

For 3, this configuration file will be used again for the key *table.name*, *topics, pk.mode, pk.fields* etc that define the schema, the database name and the name of the queue for the connector to consume the data. This configuration file is placed inside the container of *lx-kafka* under the {BASEDIR}/etc/kafka directory, so that it can be available when the containers start and the corresponding processes start. To fill these parameters, a new *configmap* is defined for this component and a script will be executed when the container is created.

For 6, it is an opportunity to define a *service/route* that will be bound to the container of the *backend* that exposes the REST services. This must map the external port to the internal port that the *backend* is listening to.

Finally, for 7, 8 and 9 it is necessary to define an additional *service/route.* The kafka queue is configured via a server.properties file. This file is placed in the *blueprint/Pilot-Ref* project, and a code snippet is as follows:

```
broker.id=0

listeners=PLAINTEXT://IPOFTHEMACHINE:9092

advertised.listeners=PLAINTEXT://IP_EXTERNAL_OF_THE_MACHINE:9092

num.network.threads=3

num.io.threads=8

socket.send.buffer.bytes=102400
```

From this code snippet, it is clear that two variables need to be resolved during deployment time and cannot be known in advance, the *IPOFTHEMACHINE* that will be assigned to the *listeners* and the *IP_EXTERNAL_OF_THE_MACHINE* that will be assigned to the *advertised.listeners.* It is possible to know the former when the container is deployed and it can only be set during the starting process. However, it is not possible to know the latter. Therefore, one way to solve this problem is to define a *service/route* that will

expose access to external sources, map its port to the internal port 9092 that is advertised internally in the container of Kafka, and then the *IP_EXTERNAL_OF_THE_MACHINE* will be replaced with the name of this *service/route,* accordingly with the name declared in the configmap shown above. Thankfully, it is possible to do that before the actual deployment. Finally, this configuration file under the {BASEDIR}/etc/kafka directory now has the *service/route* that must be advertised, and it can be available when the container is started and the corresponding processes are starting.

Section 6.4.1 describes how to deploy the pilot reference solution using Kubernetes.

6.4.1 Preparing the deployment using Kubernetes

After defining the components and their requirements for the overall solution that will be deployed within the testbed, the Kubernetes manifest files that will guide the orchestration of the deployment must be provided. In the imaginary pilot, these are the following:

a. StafulSets:
    i. Infinistore
    ii. Lx-kafka
b. DeploymentConfigs:
    i. Backend
c. Services:
    i. Infinistore
    ii. Lx-kafka
    iii. Backend
d. ConfigMaps:
    i. Lx-kafka

Infinistore and lx-kafka StatefulSet maintain a state and they additionally need to preserve their internal IPs in cases that a POD is recreating, so K8s StatefulSet resources are needed. The backend is a stateless microservice, so a DeploymentConfig can fits its purposes. Three *services* are defined to allow the internal network communication between the components and a ConfigMap that holds the parameters needed to configure lx-kafka. The project now looks like Figure 44. In the rest of this section a detailed description of each file is provided.

P
Pilot-Ref
Project ID: 29

28 Commits 1 Branch 0 Tags 379 KB Files 379 KB Storage

Star 0 Fork 0

Auto DevOps
It will automatically build, test, and deploy your application based on a predefined CI/CD configuration.
Learn more in the Auto DevOps documentation
Enable in settings

master pilot-ref / + History Find file Web IDE Clone

increase kafka minimum resources
Pavlos Kranas authored 24 seconds ago 05f1ac5d

Add README Add LICENSE Add CHANGELOG Add CONTRIBUTING Add Kubernetes cluster Set up CI/CD

Name	Last commit	Last update
Jenkinsfile_deploy	Update Jenkinsfile_deploy	18 hours ago
backend-deployment.yaml	pilot2 changed to pilot-refW	3 days ago
backend-service.yaml	pilot2 changed to pilot-refW	3 days ago
docker-compose.yml	configure pilot-ref to use kafka queue	2 weeks ago
infinistore-service.yaml	pilot2 changed to pilot-refW	3 days ago
infinistore-statefulset.yaml	pilot2 changed to pilot-refW	3 days ago
lx-kafka-configmap.yaml	change configmap	31 minutes ago
lx-kafka-service.yaml	pilot2 changed to pilot-refW	3 days ago
lx-kafka-statefulset.yaml	increase kafka minimum resources	24 seconds ago

Figure 44 – Pilot-ref Gitlab project

Firstly, all these *kinds* need to be under a Kubernetes Namespace. A general rule is to call the namespace with the same name as the pilot *namespace:pilot-ref,* so this sandbox will be called that way. For other pilots, it should be *namespace:pilot1, namespace:pilot2* etc. In a more general way, as a namespace defines a sandbox and a pilot might have several use cases that need to be deployed in several sandboxes, a deeper categorization might be needed. In that case, it is suggested to use several deploy projects with related yaml files and related Jenkins deploy pipelines, each one related to each of the use cases of the pilot.

The first component analyzed is *Infinistore.* Here's is how the definition of its stateful set looks like shown in Figure 45.

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: infinistore
  namespace: pilot-ref
  labels:
    app: infinistore
spec:
  serviceName: infinistore-service
  replicas: 1
  selector:
    matchLabels:
      app: infinistore
  updateStrategy:
    type: RollingUpdate
  podManagementPolicy: OrderedReady
  template:
    metadata:
      labels:
        app: infinistore
    spec:
      containers:
        - image: harbor.infinitech-h2020.eu/data-management/infinistore:latest
          name: infinistore
          ports:
            - containerPort: 2181
            - containerPort: 1529
            - containerPort: 9876
            - containerPort: 9992
            - containerPort: 14400
          startupProbe:
            exec:
              command:
                - /bin/sh
                - -c
                - python3 /lx/LX-BIN/scripts/lxManageNode.py check QE
            timeoutSeconds: 5
            failureThreshold: 30
            periodSeconds: 10
          resources:
            limits:
              cpu: 4000m
              memory: 8Gi
            requests:
              cpu: 2000m
              memory: 4Gi
```

Figure 45 – Infinistore StatefulSet manifest

The arrows in Figure 45 point out the most important elements in this file. Firstly, line 2 defines a *StatefulSet*, whose name is defined in line 4 as *Infinistore*, under the namespace *pilot-ref* in line 5. Line 23 specifies the address of the image in the Harbor registry, as defined in the CI process. Line 25 lists the ports exposed by the container. Line 31 defines a *startupProbe* which checks periodically if the container is working. As Kubernetes cannot know the status of the processes inside the container, this command returns with the status. This is internal to the *Infinistore* and each component needs to provide its own command for checking the internal status. This helps Kubernetes to understand if the processes inside the POD/container are down, which will cause it to restart. A very important element is the definition of the resource in line 40. As these are defined under the *blueprint/pilot-ref* project, it means that these are the resources required for the *Infinistore* for the deployment of that pilot. Other pilots might need a more intensive data processing, thus more resources.

After defining the *StatefulSet* for *Infinistore,* it is necessary to define its *Service*, which is the element that will allow this pod/container to be reachable by network, once deployed. Its definition is as shown in Figure 46.

infinistore-service.yaml 467 Bytes

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: infinistore-service
5    namespace: pilot-ref
6    labels:
7      app: infinistore
8  spec:
9    ports:
10     - name: "2181"
11       port: 2181
12       targetPort: 2181
13     - name: "1529"
14       port: 1529
15       targetPort: 1529
16     - name: "9876"
17       port: 9876
18       targetPort: 9876
19     - name: "9992"
20       port: 9992
21       targetPort: 9992
22     - name: "14400"
23       port: 14400
24       targetPort: 14400
25   selector:
26     app: infinistore
```

Figure 46 – Infinistore Service manifest

Again, line 5 in Figure 46 provides the same namespace, while line 9 provides the list of ports that can be reachable via this service. Finally, in line 26 there is the component that this service is related to. In this case, this is the *Infinistore*, which is the name defined in the corresponding *StatefulSet* (in its line 4) for the given namespace. As the IP cannot be known before deployment, other pods/containers that need to connect to the *Infinistore,* need to use this domain name. This will be used by both the *backend* and the *lx-kafka* components.

In the following line the *backend* is described (see Figure 47). As this is a stateless component, a *DeployConfig is used*.

```
17      spec:
18        containers:
19          - env:
20              - name: DATASTORE_HOST
21                value: infinistore-service
22            image: harbor.infinitech-h2020.eu/pilot-ref/backend:latest
23            name: backend
24            ports:
25              - containerPort: 54735
26            resources:
27              limits:
28                cpu: 200m
29                memory: 128Mi
30              requests:
31                cpu: 100m
32                memory: 128Mi
```

Figure 47 – Backend container spec

Here, additionally with the declaration of where to download the image for this container, the ports and the resources, it is also important to define an environment variable called DATASTORE_HOST, whose value is *infinistore-service.* This is the name of the *Service* created for the *Infinistore.* That way, when the backend will try to connect to the datastore, it will get the domain name of the latter through this environment variable, which it will be able to reach through the network. The corresponding *Service* for the backend is similar to the one we just have just shown, so it is omitted.

The next component is *lx-kafka.* The*Service* definition has no differences from the others, so just the definition of its *StatefulSet* is provided. For that component, some parameters are needed during run-time, and it makes use of a *ConfigMap*, that holds these values. It can be defined as done in Figure 48.

```
lx-kafka-configmap.yaml 497 Bytes
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: lx-kafka-configmap
5    namespace: pilot-ref
6  data:
7    advertised.url: lx-kafka-service
8    topics: GTP
9    connection.url: infinistore-service
10   connection.database: PILOTREF
11   pk.fields: id
12   fields.whitelist: trip_duration_secs,start_time,stop_time,start_station_id,start_station_name, start_station_latitude,start_station_longitude,end_sta
```

Figure 48 – lx-kafka-configmap Configmap manifest

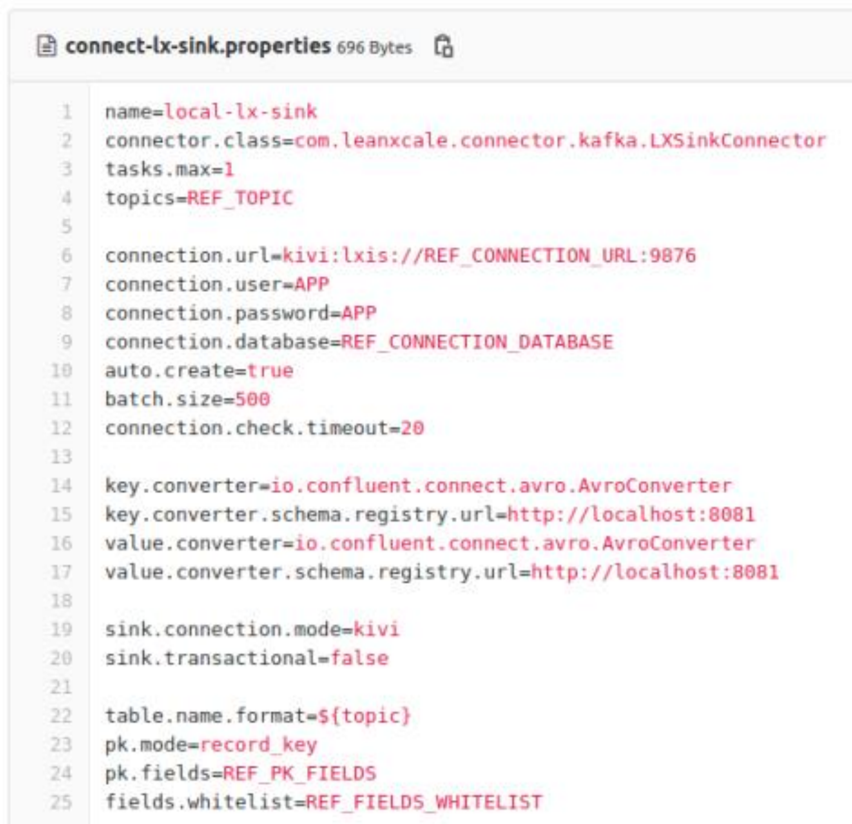
In line 6 of Figure 48 there is a list of 6 parameters, in a key/value pair. The Statefulset consumes these values referencing the configmap entries as done in Figure 49.

```
21      spec:
22        containers:
23          - image: harbor.infinitech-h2020.eu/interface/lx-kafka:latest
24            name: lx-kafka
25            ports:
26              - containerPort: 8081
27              - containerPort: 9092
28            resources:
29              limits:
30                cpu: 2000m
31                memory: 2Gi
32              requests:
33                cpu: 1000m
34                memory: 1Gi
35            env:
36              - name: advertised_url
37                valueFrom:
38                  configMapKeyRef:
39                    name: lx-kafka-configmap
40                    key: advertised.url
41              - name: topics
42                valueFrom:
43                  configMapKeyRef:
44                    name: lx-kafka-configmap
45                    key: topics
46              - name: connection_url
47                valueFrom:
48                  configMapKeyRef:
49                    name: lx-kafka-configmap
50                    key: connection.url
51              - name: connection_database
52                valueFrom:
53                  configMapKeyRef:
54                    name: lx-kafka-configmap
55                    key: connection.database
```

Figure 49 – lx-kafka StatefulSet manifest

Line 23-34 of Figure 49 define the name of the container, the url to grab the docker image, the ports and resource requirements as always. From line 35 are defined environment variables retrieved from *ConfigMaps.* It is worth noting that the name of the *ConfigMap* defined in line 4 of Figure 48 is used in the

StatefulSet manifest to retrieve the values: for the variable *connection_url,* it will search in the *ConfigMap* named lx-kafka-configmap and it will grab the corresponding key called *connection.url*. In the *ConfigMap*, this is defined in line 9 (Figure 48). Once the container/POD is up, it is possible to print ${connection_url} from inside the container and it will respond with *infinistore-service.* This means that this is an effective way to pass this parameter during run-time, so that the *lx-kafka* can connect to the Infinistore using this *Service.* However, the configuration of this component is still not finished: *Lx-kafka* needs to read some properties file to instantiate itself. So, it is necessary to pass this information there and overwrite that file. The properties file is located under *interface/lx-kafka* project as depicted in Figure 50.



connect-lx-sink.properties 696 Bytes

```
1  name=local-lx-sink
2  connector.class=com.leanxcale.connector.kafka.LXSinkConnector
3  tasks.max=1
4  topics=REF_TOPIC
5
6  connection.url=kivi:lxis://REF_CONNECTION_URL:9876
7  connection.user=APP
8  connection.password=APP
9  connection.database=REF_CONNECTION_DATABASE
10 auto.create=true
11 batch.size=500
12 connection.check.timeout=20
13
14 key.converter=io.confluent.connect.avro.AvroConverter
15 key.converter.schema.registry.url=http://localhost:8081
16 value.converter=io.confluent.connect.avro.AvroConverter
17 value.converter.schema.registry.url=http://localhost:8081
18
19 sink.connection.mode=kivi
20 sink.transactional=false
21
22 table.name.format=${topic}
23 pk.mode=record_key
24 pk.fields=REF_PK_FIELDS
25 fields.whitelist=REF_FIELDS_WHITELIST
```

Figure 50 – Lx-kafka properties file

This properties file has various placeholders where the values defined in the *ConfigMap* should be set*.* For the *connection_url* parameter discussed above, its definition in the properties file is in line 6 (Figure 50). Here, it is necessary to replace the REF_CONNECTION_URL with the valid domain name, which in this case is the *infinistore-service.* The idea is to replace these values when the container/POD is created before starting the corresponding processes inside the container/POD. For this purpose, a *postscript.sh* script has been provided and is being executed when the container/POD is started. This is depicted in Figure 51.

postscript.sh 1.52 KB   Edit   Web IDE   Re

```
sed -i "s/IPOFTHEMACHINE/$(hostname -i)/g" ${KAFKA_HOME}/etc/kafka/server.properties
sed -i "s/IPOFTHEMACHINE/$(hostname -i)/g" ${KAFKA_HOME}/etc/kafka/connect-standalone.properties


sed -i "s/REF_TOPIC/${topics}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-sink.properties
sed -i "s/REF_CONNECTION_URL/${connection_url}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-sink.properties
sed -i "s/REF_CONNECTION_DATABASE/${connection_database}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-sink.properties
sed -i "s/REF_PK_FIELDS/${pk_fields}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-sink.properties
sed -i "s/REF_FIELDS_WHITELIST/${fields_whitelist}/g" ${KAFKA_HOME}/etc/kafka/connect-lx-sink.properties
sed -i "s/IP_EXTERNAL_OF_THE_MACHINE/${advertised_url}/g" ${KAFKA_HOME}/etc/kafka/server.properties



echo 'will wait a bit for the datastore to start'
sleep 20
echo 'will start zookeeper'
nohup ${KAFKA_HOME}/bin/zookeeper-server-start ${KAFKA_HOME}/etc/kafka/zookeeper.properties > nohup_zk.out &
sleep 2
echo 'will start kafka server'
nohup ${KAFKA_HOME}/bin/kafka-server-start ${KAFKA_HOME}/etc/kafka/server.properties > nohup_kafka_server.out &
sleep 5
echo 'will start schema registry'
nohup ${KAFKA_HOME}/bin/schema-registry-start ${KAFKA_HOME}/etc/schema-registry/schema-registry.properties > nohup_schem
sleep 3
echo 'will start lxs connector'
nohup ${KAFKA_HOME}/bin/connect-standalone ${KAFKA_HOME}/etc/kafka/connect-standalone.properties ${KAFKA_HOME}/etc/kafka,
sleep infinity

```

Figure 51 – postscript.sh script

In lines 1-2 (Figure 51), 2 parameters in the properties file relying on the *hostname* of the container/pod are replaced. Then, in lines 5-10, the value taken from the ConfigMap replace the placeholder of a list of parameters. For example, looking at line 6, The REF_CONNETION_URL placeholder defined in the properties file, will be replaced by the value of the *connection_url* environment variable. This is defined in the *StatefulSet* of this component, where this parameter will take the value defined in the referenced *ConfigMap,* which is *infinistore-service.*

The Lx-kafka service and backend service are exposed outside the sandbox using the method described in section 4.2 point 3.

6.4.2 Deploying using Kubernetes

Having defined the Kubernetes manifest files, it is time to start the deployment pipeline that will create the containers/PODs of the integration solution of the pilot, into the sandbox. In contrast to the CI pipelines, the CD pipelines are not triggered automatically. The reason is that the AWS resources are not enough to afford the deployment of all 15 pilot solutions. Due to this, the deployment will be started manually, for validation purposes.

Following the same procedure as in the CI part, a Jenkins file needs to be provided. This is different from the one used in the CI, and the pilot integrators need to communicate with the HPE team to help them with this definition. The procedure to manually trigger the process follows.

The pilot integrator needs to go to the CI/CD part under *blueprint* project, and to click in one of the available pipelines, that will redirect in the Jenkins file. This procedure has been documented in Chapter 2. In the Jenkins main page of the deployment, a list of *stage* deployments is shown. To start the deployment, it is sufficient to click on the *build now* button, as depicted in Figure 52.

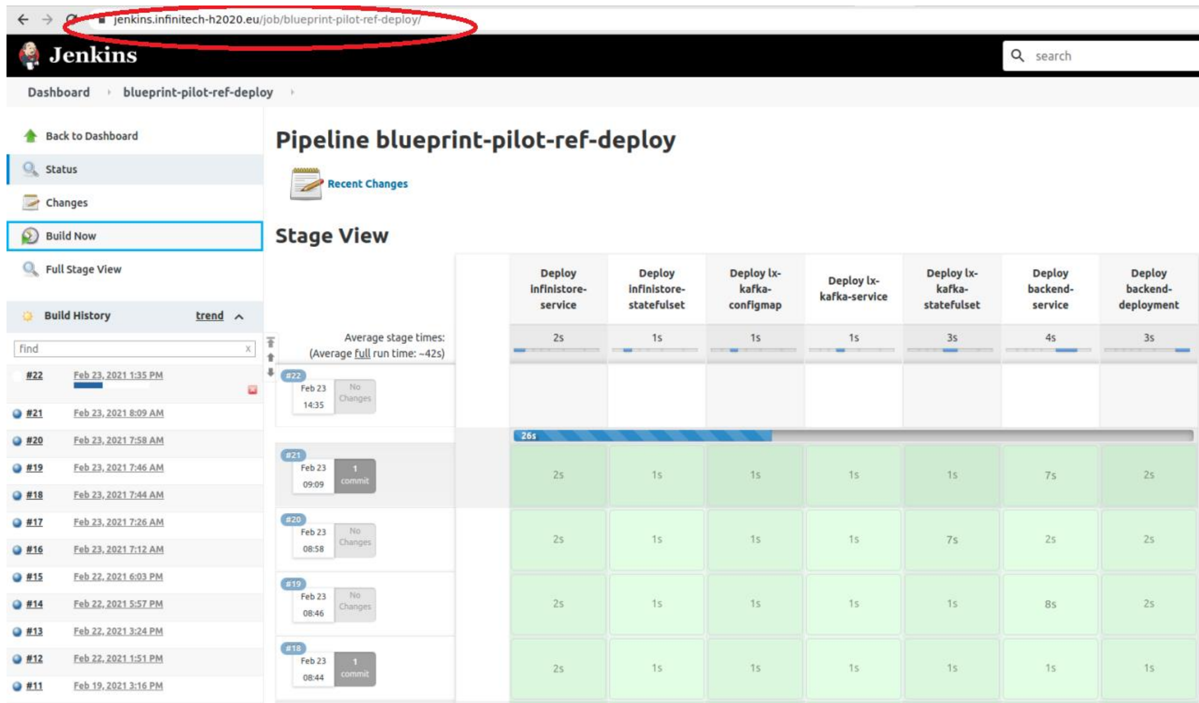


Figure 52 – Jenkins deployment stages

Clicking in one specific build, and then in the *Console Output*, makes it possible to see the logs from the building process (see Figure 53).

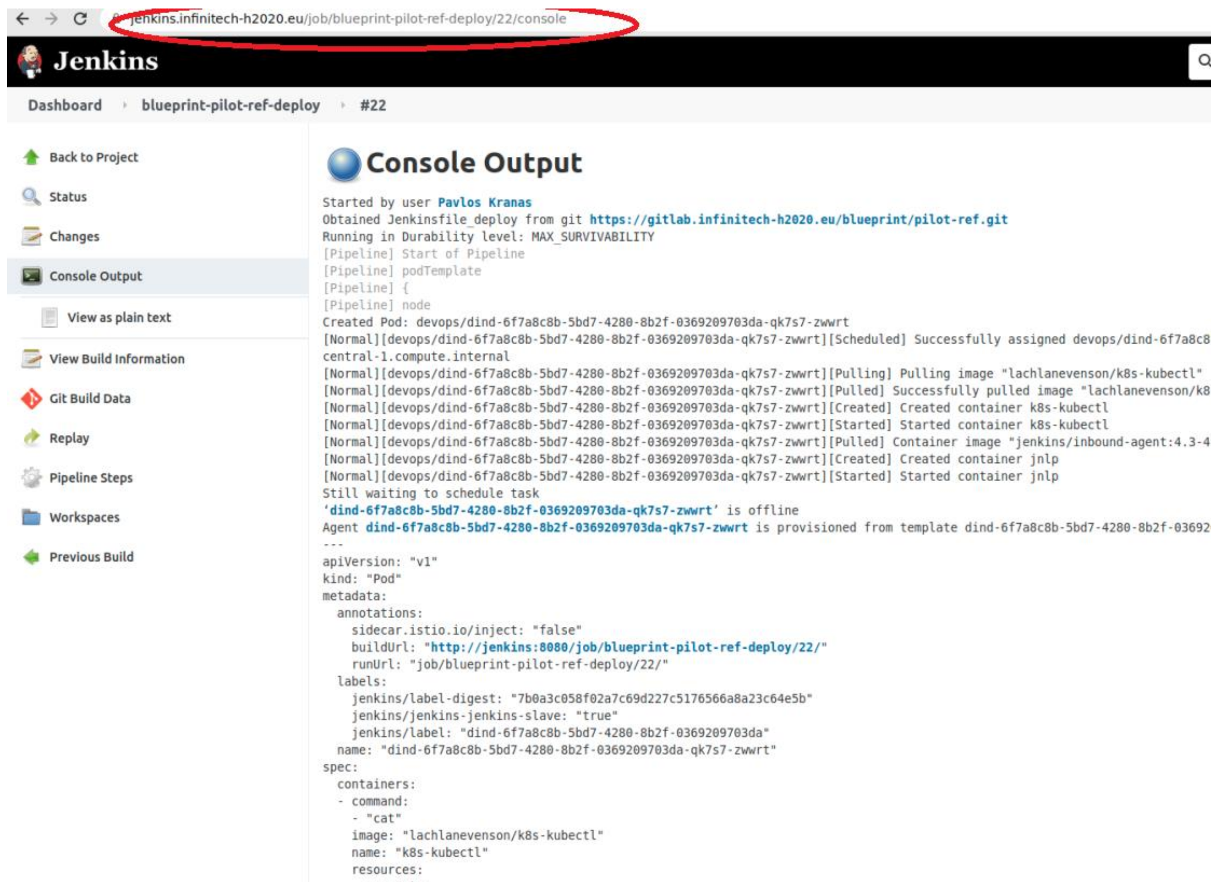


Figure 53 – Jenkins build log

If the build process is successful, Kubernetes resources have been deployed correctly and they could be managed using the *kubectl* [38] command line client. A *config* file must be provided to allow the authentication required to connect. The HPE colleagues will circulate such files that usually need to be placed under the ~/.kube. Some examples of how to use the kubectl command follow:

```
./kubectl -n pilot-ref get all
```

will print all available resources under the pilot-ref namespace, as follows

```
NAME                              READY     STATUS    RESTARTS    AGE
pod/backend-6687d79cd6-gk2sm      1/1       Running   0           5h35m
pod/infinistore-0                 1/1       Running   0           5h36m
pod/lx-kafka-0                    1/1       Running   0           5h36m

NAME                             TYPE        CLUSTER-IP       EXTERNAL-IP    PORT(S)
AGE
service/backend-service          ClusterIP   10.100.38.213    <none>         54735/TCP
5h36m
service/infinistore-service      ClusterIP   10.100.220.84    <none>
2181/TCP,1529/TCP,9876/TCP,9992/TCP,14400/TCP   5h36m
service/lx-kafka-service         ClusterIP   10.100.217.189   <none>         8081/TCP,9092/TCP
5h36m

NAME                        READY     UP-TO-DATE    AVAILABLE    AGE
deployment.apps/backend     1/1       1             1            5h35m

NAME                                   DESIRED    CURRENT    READY     AGE
replicaset.apps/backend-6687d79cd6     1          1          1         5h35m

NAME                              READY     AGE
statefulset.apps/infinistore      1/1       5h36m
statefulset.apps/lx-kafka         1/1       5h36
```

The snippet above shows that three PODs have been deployed. It is possible to see the logs of lx-kafka for instance, by issuing the following:

```
./kubectl -n pilot-ref logs lx-kafka-0
```

The following command connects to the lx-kafka POD via ssh in order to manually perform some actions.

```
./kubectl -n pilot-ref exec -it lx-kafka-0 bash
```

Finally, a resource can be completely removed by issuing the following

```
./kubectl -n pilot-ref delete statefulset.apps/lx-kafka
```

Additionally, it might be also useful to define a Jenkins pipeline that can be executed to delete a specific sandbox, removing all its resources. This will be further investigated with each pilot, to identify how often the sandbox will be created and how complicated it is to remove it manually.

6.5 MLOps Integration

This section describes in details the work done on the integration of the MLOps methodology and processes that has enhanced the “INFINITECH way” to enable the project to facilitate the Machine Learning (ML) and Deep Learning (DL) development and testing, as previously introduced in section 5.1 and at the beginning of chapter 6.

In order to achieve our goal, we have selected the **Kubeflow** platform for the MLOps integration in our blueprint reference testbed, in order to provide an infrastructure to build models and capable of enabling the portability of these models and workflows.

In essence, ML workflows are defined as Kubeflow **pipelines** and a pipeline consists of the following steps:

1. Data preparation.
2. Training.
3. Testing.
4. Serving.

Each step is a container and the output of each step becomes the input of the following one. Once compiled, the pipeline is then portable across different environments.

In the following, we describe in detail how a Kubeflow pipeline actually works.

6.5.1 Introduction to Kubeflow Pipelines

A Pipeline is a Kubeflow extension that provides a way to create a machine learning (ML) workflow. Each workflow is composed by tasks called components (or steps). They could be represented in a graph flow like the following:

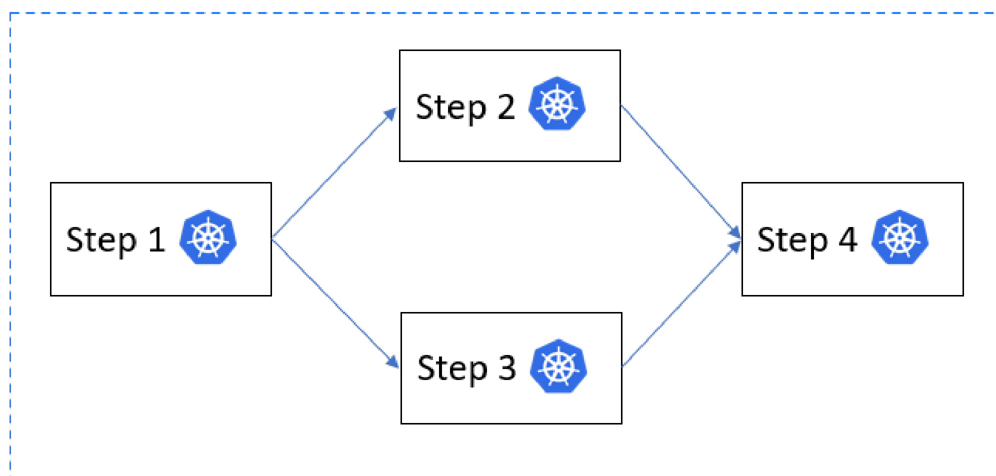


Figure 54 - Pipeline Graph Representation

Each step runs in its own Kubernetes Pod and the output of a step could be the input of another step in a way that will be clarified later in this paragraph.

To create a pipeline, it is necessary to:

- Create the ML code which realize a single step logic.
- Define in which container Kubeflow will run this particular ML code.
- Declare how the steps are linked together and which are the input and output of each one.

Kubeflow provides a *pipeline SDK* called KFP [39] that is a set of Python packages used to realize all the task described above. An important part of this SDK is the DLS [40] package that is used to describe how the pipeline steps interact with each other.

If a component is particularly simple that could described within a single Python function, is possible to simplify the component description and link the function directly as a *pipeline step* with the KFP pipeline function operator (*function_container_op()).*

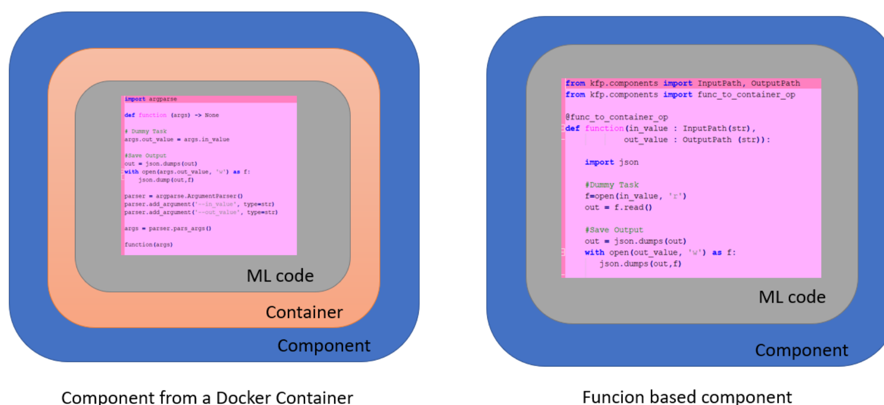


Figure 55 - Docker container component and function based component

An important aspect of the Kubeflow pipeline is how each component can exchange data with another component: this is realized behind the scene by the Kubeflow engine storing the output of a component in a particular location of a MinIO Bucket, so this output can be retrieved by the next component addressing it in the MinIO Bucket. The Kubeflow MinIO is installed by default and used by Kubeflow transparently.

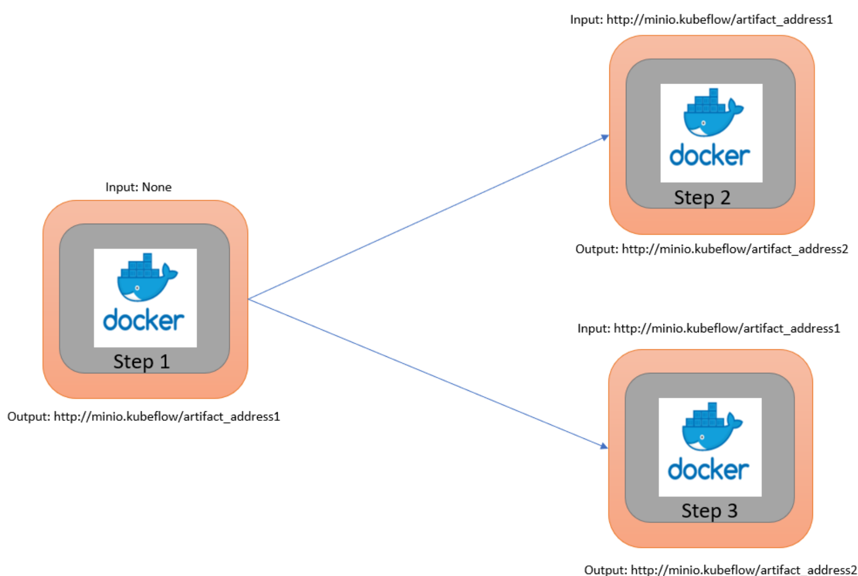


Figure 56 - Components Input/Output behind the scenes

The following section describes the End-to-end Framework integration "Neural Collaborative Filtering model" realized as blueprint for the "INFINITECH way" integration of MLOPS realized in collaboration with WP5 tasks T5.4 and T5.5 [41].

The goal of this work is to realize a Kubeflow pipeline that:

- Retrieves the dataset (train, validate and test data)
- Trains the model
- Shows metric results

 © INFINITECH Consortium

- Serves the Model

6.5.2 How to build a Kubeflow Pipeline

To illustrate how to build a pipeline consider the following simple pipeline graph example:

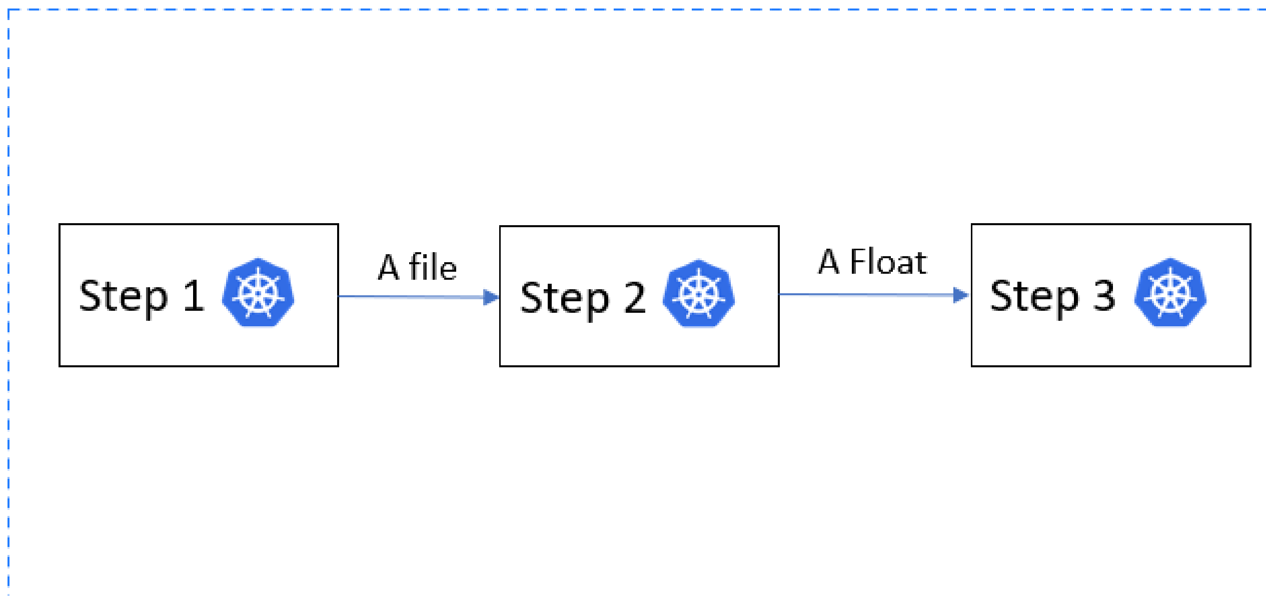


Figure 57 - Simple Pipeline Example

In this example Step1 and Step2 will be defined as "docker container steps", while Step3 will be defined as "function-based step." (See Figure 55 for reference).

For Step1 and Step2 it is necessary to define:

1. A base docker image on a Docker Registry (or a Dockerfile to build the image and push it in a registry)
2. The ML code to run that should be stored inside the Docker image.
3. The Input and Output of the specific Step.

Referring to the picture below Step1 has no input but has one file as output to be passed to Step2. The Step2 has an output (Float) that will be passed to Step3.

At Kubeflow level each pipeline component is described by a yaml file. In order to make it clear consider Step1 and Step2 depicted in the figure above, their yaml file follows:

Step1 file description

```
name: Step1
description: Step2 component declaration

outputs:
- {name: MyData1, type: LocalPath, description: 'Path where MyData1 data will be stored.'}

implementation:
  container:
    image: <docker-registry-address>/step1:latest
    command: [
      python, ML_code_for_step1.py,

      --data,
      {outputPath: MyData1},
```

```
    ]
```

Step2 file description

```
name: Step2
description: Step2 component declaration

inputs:
- {name: myData1, type: LocalPath, description: 'Path where MyData1 is stored.'}
outputs:
- {name: MyFloat, type: Float, description: 'MyFloat description.'}

implementation:
  container:
    image: <docker-registry-address>/step2:latest
    command: [
      python, ML_code_for_step2.py,

      --data,
      {inputPath: MyData1},

      --accuracy,
      {outputPath: MyFloat},

    ]
```

The step description file is just the skeleton of the component and assumes that there is a Docker image available in a Registry with the Python code inside.

Finally, the Pipeline itself is described with another yaml file that describes the mutual relationship between the components: it has to be noted that Step3 is inserted as function component directly in the Pipeline file description.

Pipeline python file

```
import yaml
from yaml.loader

import kfp
from kfp import dsl
from kfp.components import func_to_container_op

@func_to_container_op
def step3(MyFloat) -> None:
    …
       ML Python code
    ….

@dsl.pipeline(name='Pipeline for Infinitech Blueprint, description='implement the movie project ML')
def first_pipeline():

    # Loads the yaml manifest for each component
    Step1 = kfp.components.load_component_from_file(step1.yaml')
    Step2 = kfp.components.load_component_from_file(step2.yaml')

    # Run Step1, Step2, Step3
    Step1 = add_env(step1(), [<args>])
    step2 = add_env(step2(step1.output), [<args>])
    step3(step2.outputs[<args>])

if __name__ == '__main__':
    kfp.compiler.Compiler().compile(first_pipeline, compiled-pipeline.yaml')
```

It is worthwhile to note that the components are first declared (with kfp.components.load_component_from_file or with *@func_to_container_op*) and then executed.

The last two lines of the Pipeline description

```
if __name__ == '__main__':
    kfp.compiler.Compiler().compile(first_pipeline, compiled-pipeline.yaml')
```

compile the pipeline by combining all of the steps and produce a final yaml file understandable by Kubeflow (*compiled-pipeline.yaml*). To upload the pipeline it is possible to use the GUI (refer to paragraph 6.5.5 fo the details) or to use the kfp function *client.create_run_from_pipeline_func* to run it directly from python code.

It is important to mention that also in the case of a *float* type the parameter is passed to the next component storing it on MinIO as a file, and this is done behind the scene by Kubeflow: the user doesn't have to worry about where the information is stored on MinIO.

6.5.3 End to End Blueprint Training components

The "Neural Collaborative Filtering model" [41] chosen for the Blueprint integration has been realized using MLFlow [42], which is an open-source platform that manages the ML lifecycle. MLFlow doesn't provide a tool to automatize the procedures, and for such reason we have used Kubeflow.

In other words, a dedicated Kubernetes cluster for MLflow and Kubeflow has been installed, allowing us to take advantage of the best of both tools. The Kubeflow pipelines allow it to automate and grant pipelines. Furthermore, they can be versioned and produced by leveraging the Kubeflow interface. Modular components can be reused or replaced in other pipelines. On the other hand, the MLflow model registry allows users to keep the model versions and these are securely stored on an AWS S3 bucket, while the associated metadata, such as parameters and metrics, are stored into a MySQL database (PostrgresSQL).

In the end, the solution involves two MinIO servers, one for MLFlow and one for Kubeflow.

The serving part has been realized using the Seldon Core Serving Framework that is supported by Kubeflow since it is capable of serving a saved model directly in the MLFlow format. Details about the serving component are provided the next paragraph.

The following graph depicts the blueprint End to End Machine Learning Pipeline:

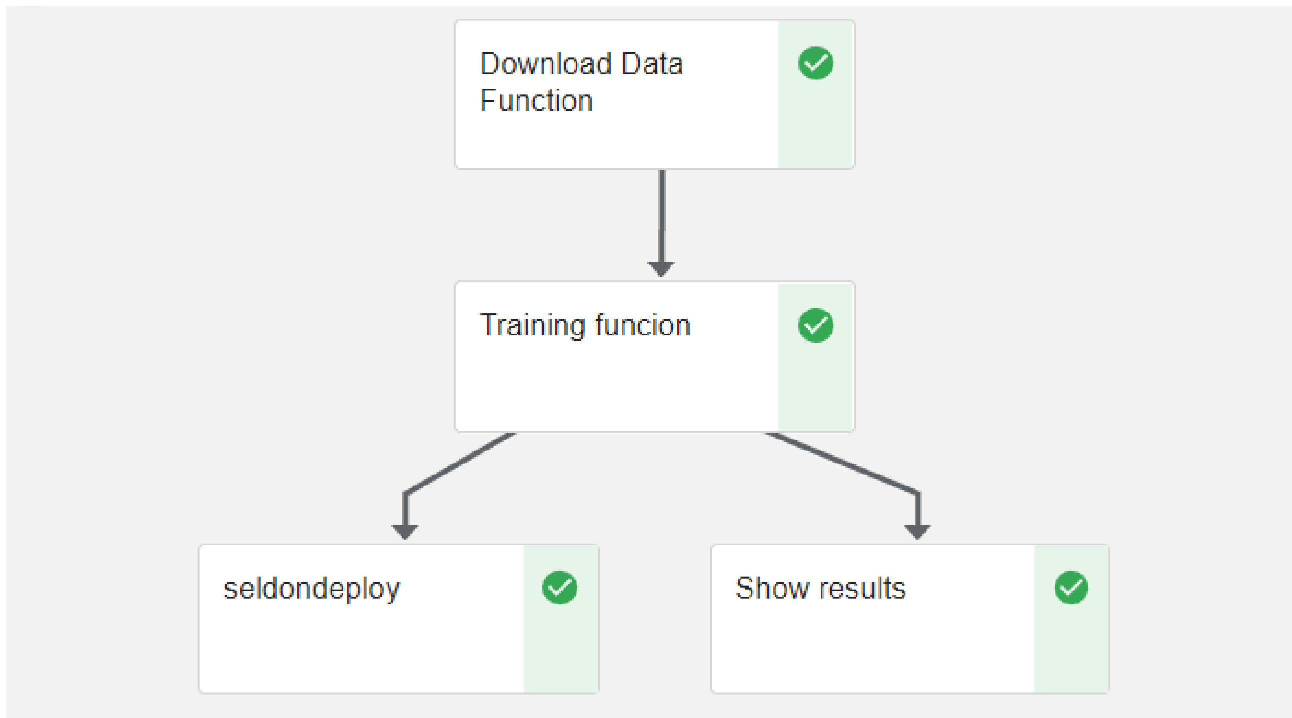


Figure 58 - Blueprint Pipeline Graph

This graph is described by the following python code snippet:

```
import yaml
from yaml.loader import SafeLoader

import kfp
from kubernetes.client.models import V1EnvVar
from kfp import dsl
from kfp.components import func_to_container_op

buck = V1EnvVar(name='MINIO_BUCKET', value='mlflow')
acc_key = V1EnvVar(name='AWS_ACCESS_KEY_ID', value='minio')
sec_key = V1EnvVar(name='AWS_SECRET_ACCESS_KEY', value='domenico1984')
url = V1EnvVar(name='MLFLOW_S3_ENDPOINT_URL', value='http://s3.mlflow.svc.cluster.local:9000')
db = V1EnvVar(name='MLFLOW_TRACKING_URI', value='postgresql+psycopg2://mlflow:mlflow@postgresql.mlflow.svc.cluster.local:5432/mlflow-db')


def add_env(cont, envs):
    for env in envs:
        cont = cont.add_env_variable(env)
    return cont


@func_to_container_op
def show_results(key: str,
                 bucket: str) -> None:
    print(f"Key: {key}")
    print(f"Bucket: {bucket}")


@dsl.pipeline(name='Pipeline for Matteo Modeld', description='implement the movie project ML')
def first_pipeline():

    # Loads the yaml manifest for each component
    download = kfp.components.load_component_from_file('download_data/download_data_pipeline_component.yaml')
    training = kfp.components.load_component_from_file('training/training_pipeline_component.yaml')

    # Run download_data task and training task
    download_task = add_env(download(), [acc_key, sec_key, url, buck])
    training_task = add_env(training(download_task.output), [acc_key, sec_key, url, db])
    show_results(training_task.outputs["ModelKey"], training_task.outputs["ModelBucket"])

    # Open the file and load the file
    with open("seldon.yaml") as f:
        seldon_config = yaml.load(f, Loader=SafeLoader)

    deploy_step = dsl.ResourceOp(
        name="seldondeploy",
        k8s_resource=seldon_config)

    deploy_step.after(training_task)


if __name__ == '__main__':
    kfp.compiler.Compiler().compile(first_pipeline, 'test-pipeline.yaml')

```

Figure 59 - Pipeline python file

To understand the above code, it is useful to start with a general description and then proceed by analysing each component:

- **Pipeline python file general description**

With the exception of the *seldondeploy* component (lines 41-49) each step depicted in **Error! Reference source not found.** Is first declared or loaded (lines 33-34) and then executed (lines 37-39), as we have seen in the example in 6.5.2.

The Download component and the Training component (lines 33, 34) are declared using the *kfp.components.load_component_from_file* as the Step1 and Step2 in the example in 6.5.2

The *show_results* component (lined 22-26) is declared using the @func_*to_container_op* as the Step3 in 6.5.2

After the three components are declared they can be executed (lines 37-39).

The *seldondeploy* (lines 41-49) component is a bit different and will be explained in the next paragraph 6.5.4.

- **Download Data Function**

<u>*Purpose of this component*:</u>

Gathering the data (training, validation, and test) from an OneDrive folder and save it to the MLFlow MinIO bucket.

Download Data component yaml file:

```
name: Download Data Function
description: Download data from OneDrive and upload on 91unction MINIO

outputs:
- {name: DataBucket, type: String, description: 'Name of Minio data bucket'}

implementation:
  container:
    image: harbor.infinitech-h2020.eu/test/download-data:v6
    command: [
      python, /pipeline/download_data.py,
      --data-bucket,
      {outputPath: DataBucket}
    ]
```

Dockerfile

```
FROM python:3.7.5
WORKDIR /pipeline
COPY requirements-download-data.txt /pipeline
RUN pip install -r requirements-download-data.txt
COPY download_data.py /pipeline
```

Download_data.py code

```
import asyncio
import os
import zipfile
from base64 import b64encode
from pathlib import Path

import aiofiles
import aiohttp
import boto3
import nest_asyncio
from requests import Session

nest_asyncio.apply()

DATA_URL = "https://1drv.ms/u/s!AjMahLyQeZqugU-siALoN5y9eaCq?e=jsgoOB"
DATA_PATH = "data/"
DATA_FILE = str(Path(DATA_PATH, "global_n_neg_100.zip"))

S3_CRED = {
    "endpoint_url": os.environ.get("MLFLOW_S3_ENDPOINT_URL"),
    "aws_access_key_id": os.environ.get("AWS_ACCESS_KEY_ID"),
    "aws_secret_access_key": os.environ.get("AWS_SECRET_ACCESS_KEY")
}
S3_BUCKET = os.environ.get("MINIO_BUCKET")


class OneDrive:
    """"Download shared file/folder to localhost with persisted structure.
    Download shared file/folder from OneDrive without authentication.
    Params:
    `str:url`: url to the shared one drive folder or file
    `str:path`: local filesystem path
```

```
    methods:
    `download() -> None`: fire async download of all files found in URL
    “””

    def __init__(self, url=None, path=None):
        “””Init OneDrive Class.”””
        If not (url and path):
            raise ValueError(“URL to shared resource or path to download is missing.”)

        self.url = url
        self.path = path
        self.prefix = “https://api.onedrive.com/v1.0/shares/”
        self.suffix = “/root?expand=children”
        self.session = Session()
        self.session.headers.update(
            {
                “User-Agent”: “Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36”
                “ (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36”
            }
        )

    def _token(self, url):
        “””Decode share link.”””
        Result = “u!” + b64encode(url.encode()).decode()
        result = result.rstrip(“=”)
        result = result.replace(“/”, “_”)
        result = result.replace(“+”, “-“)
        return result

    def _traverse_url(self, url, name=””):
        “””Traverse the folder tree and store leaf urls with filename.
        Get all the shared files given the url.
        Args:
            url: the shared link of files/folder
        “””
        r = self.session.get(f”{self.prefix}{self._token(url)}{self.suffix}”).json()
        name = name + os.sep + r[“name”]

        # shared file
        if not r[“children”]:
            file = {}
            file[“name”] = name.lstrip(os.sep)
            file[“url”] = r[“@content.downloadUrl”]
            self.to_download.append(file)
            print(f”Found {file[‘name’]}”)

        # shared folder
        for child in r[“children”]:
            if “folder” in child:
                self._traverse_url(child[“webUrl”], name)

            if “file” in child:
                file = {}
                file[“name”] = (name + os.sep + child[“name”]).lstrip(os.sep)
                file[“url”] = child[“@content.downloadUrl”]
                self.to_download.append(file)
                print(f”Found {file[‘name’]}”)

    async def _download_file(self, file, session):
        async with session.get(file[“url”], timeout=None) as r:
            filename = os.path.join(self.path, file[“name”])
            os.makedirs(os.path.dirname(filename), exist_ok=True)
            async with aiofiles.open(filename, “wb”) as f:
                async for chunk in r.content.iter_chunked(1024 * 16):
```

```
                    if chunk:
                        await f.write(chunk)

        self.downloaded += 1
        progress = int(self.downloaded / len(self.to_download) * 100)
        print(
            f"Download progress: {self.downloaded}/{len(self.to_download)}, {progress}%"
        )

    async def _downloader(self):
        async with aiohttp.ClientSession() as session:
            await asyncio.wait(
                [self._download_file(file, session) for file in self.to_download]
            )

    def download(self):
        """Download files from OneDrive.
        Download files from OneDrive with the given share link.
        """
        print("Traversing public folder\n")
        self.to_download = []
        self.downloaded = 0
        self._traverse_url(self.url)

        print("\nStarting async download\n")
        asyncio.get_event_loop().run_until_complete(self._downloader())


def un_zip(file_name, target_dir):
    """
    Unzip zip files.
    """
    zip_file = zipfile.ZipFile(file_name)
    for names in zip_file.namelist():
        print(f"unzip file {names}")
        zip_file.extract(names, target_dir)
    zip_file.close()


def upload_to_s3(path, bucket):
    """
    Upload file to s3.
    """
    s3 = boto3.client("s3", **S3_CRED)
    for root, _, files in os.walk(path):
        for file in files:
            print(f"upload {file} to s3")
            s3.upload_file(os.path.join(root, file),
                           bucket,
                           os.path.join(root, file))


def main(args):
    folder = OneDrive(DATA_URL, DATA_PATH)
    folder.download()
    un_zip(DATA_FILE, DATA_PATH)
    os.remove(DATA_FILE)
    upload_to_s3(DATA_PATH, S3_BUCKET)

    import json
    with open(args.data_bucket, "w") as f:
        json.dump(S3_BUCKET, f)
```

```
if __name__==”__main__”:

    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument(“—data-bucket”, type=str)
    args = parser.parse_args()

    Path(args.data_bucket).parent.mkdir(parents=True, exist_ok=True)

    main(args)
```

- **Training Function**

Purpose of this component:

Training and validation of the Neural Collaborative Filtering model and its storage in the MLFlow format to the Mlflow Minio storage.

Training component yaml file

```
name: Training 94unction
description: Train the model

inputs:
- {name: DataBucket, type: String, description: ‘Name of Minio data bucket’}
outputs:
- {name: ModelKey, type: String, description: ‘Key of model in Minio bucket’}
- {name: ModelBucket, type: String, description: ‘Name of Minio model bucket’}

implementation:
  container:
    image: harbor.infinitech-h2020.eu/test/training:v5
    command: [
      python, /work/main_my_INFINITECH_pod.py,

      --data-bucket,
      {inputPath: DataBucket},

      --model-key,
      {outputPath: ModelKey},

      --model-bucket,
      {outputPath: ModelBucket}
    ]
```

Dockerfile

```
FROM  python:3.7.5
RUN mkdir -p /work
WORKDIR /work
ENV LC_ALL=C.UTF-8
ENV LANG=C.UTF-8

COPY ./requirements_all.txt .
ADD main_my_INFINITECH_pod.py /work
ADD metrics.py /work
ADD data.py /work

RUN apt-get update && apt-get install -y \
    build-essential \
    python3-dev \
    libpq-dev
```

```
RUN pip install -U pip && \
    pip install -r requirements_all.txt
ADD main_my_INFINITECH_pod.py /work
```

metrics.py code & data.py

`metrics.py` module contains a collection of functions used to evaluate the model performances, while `data.py` module includes a series of functions to manipulate the dataset for the training process.

The source code of the two previous libraries is very long so it will be omitted, while the main part of the ML code follows:

main_my_INFINITECH_pod.py

```
import json
import os
import shutil
import sys
from pathlib import Path
import warnings

import boto3
import mlflow
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from mlflow.utils.environment import _mlflow_conda_env
from pytorch_lightning import LightningDataModule, Trainer
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
from pytorch_lightning.callbacks.early_stopping import EarlyStopping
from pytorch_lightning.core.lightning import LightningModule
from torch.utils.data.dataloader import DataLoader

try:
    import data as d
    import metrics as m
except ImportError:
    from . import data as d, metrics as m


sys.path.append(".")
sys.path.append("..")
warnings.filterwarnings("ignore")


# DATASET from https://1drv.ms/u/s!AjMahLyQeZqugU-siALoN5y9eaCq?e=jsgoOB

PATH_BASE = str(Path(__file__).parent.absolute())

# Data params
PATH_DATA = PATH_BASE + "/data/"
DEFAULT_USER_COL = "col_user"
DEFAULT_ITEM_COL = "col_item"
DEFAULT_RATING_COL = "col_rating"


S3_CRED = {
    "endpoint_url": os.environ.get("MLFLOW_S3_ENDPOINT_URL"),
    "aws_access_key_id": os.environ.get("AWS_ACCESS_KEY_ID"),
    "aws_secret_access_key": os.environ.get("AWS_SECRET_ACCESS_KEY")
}
```

```
NAME_EXPERIMENT = "DemoMLFlow"
DIRPATH_CKPT = PATH_BASE + "model/"

# Other Params
LOG_KW = {
    "on_step": True,
    "on_epoch": True,
    "prog_bar": True,
    "logger": True}

CONFIG = {
    "n_users": 943,
    "n_items": 1678,
    "emb_dim": 32,
    "lr": 1e-3,
    "bs": 5120,
    "dropout": 0.4,
    "optimizer": "adam",
    "epochs": 20,
    "n_layers": 3,
    "metrics": ["precision", "recall", "map"],
    "k": [5,10,20]
}


class NeuMF(LightningModule):
    """NeuMF Class."""

    Def __init__(self, config):
        """Initialize NeuMF Class."""
        Super().__init__()
        self.config = config
        self.n_users = config["n_users"]
        self.n_items = config["n_items"]
        self.emb_dim = config["emb_dim"]
        self.n_layers = config["n_layers"]
        self.dropout = config["dropout"]
        self.latent_dim_mlp = self.emb_dim * (2 ** (self.n_layers)) // 2
        self.latent_dim_gmf = self.emb_dim

        self.embedding_user_mlp = torch.nn.Embedding(
            num_embeddings=self.n_users, embedding_dim=self.latent_dim_mlp
        )
        self.embedding_item_mlp = torch.nn.Embedding(
            num_embeddings=self.n_items, embedding_dim=self.latent_dim_mlp
        )
        self.embedding_user_mf = torch.nn.Embedding(
            num_embeddings=self.n_users, embedding_dim=self.latent_dim_gmf
        )
        self.embedding_item_mf = torch.nn.Embedding(
            num_embeddings=self.n_items, embedding_dim=self.latent_dim_gmf
        )

        MLP_modules = []
        for I in range(self.n_layers):
            input_size = self.emb_dim * (2 ** (self.n_layers–- i))
            MLP_modules.append(nn.Dropout(p=self.dropout))
            MLP_modules.append(nn.Linear(input_size, input_size // 2))
            MLP_modules.append(nn.ReLU())
        self.fc_layers = nn.Sequential(*MLP_modules)
        self.affine_output = torch.nn.Linear(
            in_features=self.emb_dim * 2, out_features=1
```

```
        )
        self.logistic = torch.nn.Sigmoid()

        self.loss = torch.nn.BCELoss()

    def forward(self, data):
       “””"Train the model”””"

        try:
            user_indices, item_indices = data
        except ValueError:
            print(data)
            data = data.transpose(1, -2)
            user_indices, item_indices = data
            user_indices = user_indices.long()
            item_indices = item_indices.long()
            print(”"user_indices = {user_indices”")
            print(”"item_indices = {item_indices”")

        user_embedding_mlp = self.embedding_user_mlp(user_indices)
        item_embedding_mlp = self.embedding_item_mlp(item_indices)
        user_embedding_mf = self.embedding_user_mf(user_indices)
        item_embedding_mf = self.embedding_item_mf(item_indices)

        mlp_vector = torch.cat(
            [user_embedding_mlp, item_embedding_mlp], dim=-1
        )  # the concat latent vector
        mf_vector = torch.mul(user_embedding_mf, item_embedding_mf)

        for idx, _ in enumerate(range(len(self.fc_layers))):
            mlp_vector = self.fc_layers[idx](mlp_vector)
            mlp_vector = torch.nn.ReLU()(mlp_vector)

        vector = torch.cat([mlp_vector, mf_vector], dim=-1)
        logits = self.affine_output(vector)
        rating = self.logistic(logits)

        return rating

    def predict(self, data):

        if isinstance(data, pd.DataFrame):
            user_indices = torch.LongTensor(data[DEFAULT_USER_COL])\
                                    .to“"cp”").long()
            item_indices = torch.LongTensor(data[DEFAULT_ITEM_COL])\
                                    .to“"cp”").long()
            data = user_indices, item_indices
        elif isinstance(data, (list or tuple)):
            user_indices = torch.LongTensor(data[0]).to“"cp”").long()
            item_indices = torch.LongTensor(data[1]).to“"cp”").long()
            data = user_indices, item_indices

        self.eval()
        with torch.no_grad():
            return self.forward(data)

    def training_step(self, train_batch, batch_idx):
        users, items, ratings = train_batch
        scores = self((users, items))
        loss = self.loss(scores.view(-1), ratings)
        self.log“"train_los”", loss, **LOG_KW)
        return loss

    def validation_step(self, val_batch, batch_idx):
```

```
        users, items, ratings = val_batch
        scores = self((users, items))
        loss = self.loss(scores.view(-1), ratings.float())
        self.log“"val_los”", loss, **LOG_KW)
        #return loss

    def test_step(self, test_batch, batch_idx):
        pass

    def configure_optimizers(self):
        if self.config“"optimize”"] ==“"sg”":
            self.optimizer = torch.optim.SGD(
                self.parameters(), lr=self.config“"l”"],
            )
        elif self.config“"optimize”"] ==“"ada”":
            self.optimizer = torch.optim.Adam(
                self.parameters(), lr=self.config“"l”"],
            )
        elif self.config“"optimize”"] ==“"rmspro”":
            self.optimizer = torch.optim.RMSprop(
                self.parameters(), lr=self.config“"l”"],
            )
        self.scheduler = {
           “"schedule”": torch.optim.lr_scheduler.ReduceLROnPlateau(
                self.optimizer, mode”"mi”", factor=0.7, patience=5, min_lr=1e-6, verbose=True,
            ),
           “"monito”":“"train_los”",
        }
        return [self.optimizer], [self.scheduler]


# Data
class DataMod(LightningDataModule):

    def __init__(self, bs, threads, bucket):
        super().__init__()
        self.bs = bs
        self.threads = threads
        self.bucket = bucket

    def prepare_data(self, *args):
        # DataFrame

        s3 = boto3.client‘'s’', **S3_CRED)


        paginator = s3.get_paginator‘'list_objects_v’')
        pages = paginator.paginate(Bucket=self.bucket)
        for page in pages:
            if‘'Content’' in page:
                for obj in page‘'Content’']:
                    try:
                        Path(PATH_DATA + obj‘'Ke’']).parent.mkdir(parents=True, exist_ok=True)
                        s3.download_file(self.bucket,
                                         obj‘'Ke’'],
                                         PATH_DATA + obj‘'Ke’'])
                        print(PATH_DATA + obj‘'Ke’'])
                    except:
                        raise

        path = str(Path(PATH_DATA,“"data/global_n_neg_100”"))

        split_dataset = d.load_split_data(path, n_test=1)
        self.dataset = d.BaseData(split_dataset)
```

```
    def train_dataloader(self):
        dataloader = self.dataset.instance_bce_loader(num_negative=2,
                                                      batch_size=self.bs,
                                                      device”"cp”")
        return dataloader

    def val_dataloader(self):
        valid = self.dataset.valid[0]
        user = valid[DEFAULT_USER_COL]
        item = valid[DEFAULT_ITEM_COL]
        rtng = valid[DEFAULT_RATING_COL]
        user = torch.LongTensor(user).to“"cp”")
        item = torch.LongTensor(item).to“"cp”")
        rtng = torch.LongTensor(rtng).to“"cp”")
        dataloader = d.UserItemRatingDataset(user, item, rtng)
        return DataLoader(dataloader,
                          batch_size=self.bs*10,
                          num_workers=self.threads)

    def test_dataloader(self):
        pass


# Conda env
def conda():
    import cloudpickle
    import torchvision
    import pytorch_lightning

    return _mlflow_conda_env(
            path=None,
            additional_conda_channels=[
               “"pytorc”",
            ],
            additional_conda_deps=[
               “"pytorch={”".format(torch.__version__),
               “"torchvision={”".format(torchvision.__version__),
               “"pytorch-lightning={”".format(pytorch_lightning.__version__),
            ],
            additional_pip_deps=[
               “"cloudpickle=={”".format(cloudpickle.__version__),
               „"pandas=={“".format(pd.__version__),
               „"numpy=={“".format(np.__version__),
            ])


def copy_model_to_deploy(last_run):

    s3_client = boto3.client‘'s’', **S3_CRED)
    resp = s3_client.list_objects_v2(Bucket’'mlflo’')
    keys = []
    for obj in resp‘'Content’']:
        keys.append(obj‘'Ke’'])

    s3_resource = boto3.resource‘'s’', **S3_CRED)

    pref = ”"DemoMLFlow/{last_run”"
    keys = [key for key in keys if key.startswith(pref)]

    for key in keys:
        copy_source = {
           ‘'Bucke’':‘'mlflo’',
           ‘'Ke’': key
```

```
        }
        new_key = key.replace(pref,“"LatestMode”")
        print(new_key)

        dest = s3_resource.Bucket‘'mlflo’')
        dest.copy(copy_source, new_key)


def main(args=None):

    # Read bucket S3
    with open(args.data_bucket,“"”") as f:
        bucket = json.load(f)

    artifact_loc = ”"s3://{bucket}/DemoMLFlow”"


    # Load data
    data = DataMod(CONFIG“"b”"], os.cpu_count(), bucket)


    # Set early stopping callback
    early_stop_callback = EarlyStopping(monitor”"train_los”",
                                    min_delta=0.00,
                                    patience=20,
                                    verbose=True,
                                    mode”"mi”")

    # Save as checkpoint the model with minimum train loss
    checkpoint_callback = ModelCheckpoint(monitor”"train_los”",
                                      dirpath=DIRPATH_CKPT,
                                      filename=”"mode”",
                                      save_top_k=1,
                                      mode”"mi”")
    global lr_monit
    lr_monit = LearningRateMonitor()

    # Initialize model
    model = NeuMF(CONFIG)



    # Set MLFlow logger
    mlflow.pytorch.autolog(log_models=False)
    try:
        mlflow.create_experiment(NAME_EXPERIMENT,
                             artifact_location=artifact_loc)
    except:
        pass
    mlflow.set_experiment(NAME_EXPERIMENT)


    # Log hyperparameters
    mlflow.log_params(CONFIG)

    # Initialize trainer
    trainer = Trainer(max_epochs=CONFIG“"epoch”"],
                      check_val_every_n_epoch=5,
                      callbacks=[lr_monit,
                                 checkpoint_callback,
                                 early_stop_callback])
```

```
    # Start training
    print“"Starting train”")
    trainer.fit(model, data)


    # Test
    test = data.dataset.test[0]
    test_predictions = model.predict(test).detach().numpy().flatten()

    metrics = m.evaluate(test, test_predictions, CONFIG“"metric”"], CONFIG“"”"])
    mlflow.log_metrics(metrics)

    # Retrieve best model and log it
    best_model_path = checkpoint_callback.best_model_path
    best_model = NeuMF.load_from_checkpoint(best_model_path, config=CONFIG)
    mlflow.pytorch.log_model(best_model,
                             artifact_path”"mode”",
                             conda_env=conda(),
                             registered_model_name”"DemoMode”")

    # Cleanup checkpoints
    try:
        shutil.rmtree(DIRPATH_CKPT)
        print“"Cleaned checkpoin”")
    except FileNotFoundError:
        pass

    # Return model path key on minio
    exp_id = mlflow.get_experiment_by_name(NAME_EXPERIMENT).experiment_id
    last_run = mlflow.list_run_infos(exp_id)[0].run_id

    key = ”"DemoMLFlow/{last_run}/artifacts/model/data/model.pt”"

    with open(args.model_key,“"”") as f:
        json.dump(key, f)

    with open(args.model_bucket,“"”") as f:
        json.dump“"mlflo”", f)

    copy_model_to_deploy(last_run)


if __name__=”"__main_”":

    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument“—-data-bucke”", type=str)
    parser.add_argument“—-model-ke”", type=str)
    parser.add_argument“—-model-bucke”", type=str)

    args = parser.parse_args()

    Path(args.model_key).parent.mkdir(parents=True, exist_ok=True)
    Path(args.model_bucket).parent.mkdir(parents=True, exist_ok=True)

    main(args)
```

- **Show Result** [43]

Purpose of this component:

This component logs the address of the saved model in the MLFlow MinIO bucket.

Example:

Key:“"DemoMLFlow/d404a442be3b4df69766cfe3485bf539/artifacts/model/data/model.pth"

Bucket: "mlflow"

As already mentioned, for this specific component there is no docker image neither a yaml file because it is added directly in the pipeline yaml file using *function_container_op()* operator (see Figure 59, lines 22-26).

6.5.4 The serving component

To understand the serving component, it is worth discussing the Kubeflow serving model at Kubernetes level.

The Serving model provided by Kubeflow supports a framework called *kserve* [43] and a multi-framework model serving based on *KFServing* or on *Seldon Core* [44] .

Seldon core is able to directly understand the model saved in MLFlow format without the need to transform it into other format, and this is the main reason of its choice for serving the Neural Collaborative Filtering model object of this blueprint integration.

Seldon Core is a Kubeflow extension that define a new resource (CRD) called *SeldonDeployment.*

The seldondeploy is described by the following Kubernetes manifest:

seldon.yaml

```
apiVersion: machinelearning.seldon.io/v1alpha2
kind: SeldonDeployment
metadata:
  name: mlflow-model-matteo
  namespace: kubeflow-user-example-com
spec:
  name: mymodel
  predictors:
  - componentSpecs:
    - spec:
        # We are setting high failureThreshold as installing conda dependencies
        # can take long time and we want to avoid k8s killing the container prematurely
        containers:
        - name: seldon-serve
          livenessProbe:
            initialDelaySeconds: 80
            failureThreshold: 200
            periodSeconds: 5
            successThreshold: 1
            httpGet:
              path: /health/ping
              port: http
              scheme: HTTP
          readinessProbe:
            initialDelaySeconds: 80
            failureThreshold: 200
            periodSeconds: 5
            successThreshold: 1
```

```
            httpGet:
              path: /health/ping
              port: http
              scheme: HTTP
    graph:
      children: []
      implementation: MLFLOW_SERVER
      modelUri: s3://mlflow/LatestModel/artifacts/model
      envSecretRefName: minio-secret
      name: seldon-serve
    name: default
    replicas: 1
```

The following two lines specify where the Seldon Core framework can retrieve the model saved by the previous step:

```
modelUri: s3://mlflow/LatestModel/artifacts/model
envSecretRefName: minio-secret
```

Once Seldon Core has downloaded the model from the previous location, it creates the *SeldonDeployment* kubernetes resource, which is basically a special kind of *deployment* that controls a pod that is able to answer to predictions. The following code snippet shows how to manually create the *SeldonDeployment* resource using the standard *kubectl* command and check the results:

```
kubeflow@eubox:~$ kubectl apply -f seldon.yaml
seldondeployment.machinelearning.seldon.io/mlflow-model-matteo created

kubeflow@eubox:~$ kubectl -n kubeflow-user-example-com get seldondeployment
NAMESPACE                  NAME                  AGE
kubeflow-user-example-com   mlflow-model-matteo   27d

kubeflow@eubox:~$ kubectl -n kubeflow-user-example-com    get pod | grep model
mlflow-model-matteo-default-0-seldon-serve-6c5db6fffc-cm29p   3/3     Running   0          27d
kubeflow@eubox:~$
```

When a *seldondeployment* is created, Kubeflow automatically creates the *Istio Gateway* and the *Istio Virtualservice* to securely expose the service externally:

```
kubeflow@eubox:~$ kubectl -n istio-system get gateway
NAME                    AGE
cluster-local-gateway   46d
istio-ingressgateway    46d
seldon-gateway          39d

kubeflow@eubox:~$ kubectl get virtualservice -n kubeflow-user-example-com
NAME                  GATEWAYS                           HOSTS   AGE
mlflow-model-matteo   ["istio-system/seldon-gateway"]   ["*"]   27d
kubeflow@eubox:~$
```

Dex [45] is responsible for the authentication and authorization of the request.

The endpoint for a prediction is:

http://<istio-endpoint>/seldon/<namespace>/<modelname>/v1.0/predictions

To call this endpoint it is necessary to be authenticated and authorized by Dex: the easiest way to do so is using the browser cookie in the following way:

- Open the Kubeflow Dashboard (using the Browser).
- Insert username and password.
- Retrieve the browser cookie (setting -> security and privacy setting).
- Export this cookie in an environment variable called SESSION

```
export SESSION  =
MTY1MjEwNjY4OHxOd3dBTkZNelQwSk5UVXMyUjBOVVRUYzNWRUpXUkVwVldsTlZNMDlRVkROVldWbFhWRFpYTjB0TVJWUkZORTVW
VGt4SFNrZEpTRkU9fOr-nMh1mNXOG65quzpnkEe7y4bTaEQPJjGTjkbL0Ch
```

The cookie will be passed into the header: as an example, the following snippet makes a call to the endpoint using the Linux utility *curl* [46]

```
curl -X POST -H 'Content-Type: application/json' -H "Cookie: authservice_session=${SESSION}"  -d
'{"data":{"ndarray": [[1, 2, 1], [1, 2, 3]]}}' http://a5b3e77eecbf24dae941312c6aeb9fc7-454945384.eu-
central-1.elb.amazonaws.com/seldon/kubeflow-user-example-com/mlflow-model-
matteo/api/v1.0/predictions
{"data":{"names":["t:0"],"ndarray":[[0.39328518509864807],[0.4882642924785614],[0.4323967695236206]]
},"meta":{"requestPath":{"seldon-serve":"seldonio/mlflowserver:1.13.1"}}}
```

In the previous call the input (data to predict) is `[[1, 2, 1], [1, 2, 3]]}` while the output is `[[0.39328518509864807],[0.4882642924785614],[0.4323967695236206]]`.

Coming back to the pipeline, what the *seldondeploy* component does it is exactly what has been described in this paragraph, but automatically and inside the pipeline.

Referring to the pipeline file in Figure 59 the lines from 41 to 49

```
41      # Open the file and load the file
42      with open("seldon.yaml") as f:
43          seldon_config = yaml.load(f, Loader=SafeLoader)
44
45      deploy_step = dsl.ResourceOp(
46          name="seldondeploy",
47          k8s_resource=seldon_config)
48
49      deploy_step.after(training_task)
```

Figure 60 - SeldonDeployment pipeline code

deliver this task using the *kubernetes.client.models* library and the *dsl.ResourceOp* function from the DSL library: such lines of code are able to deploy the *SeldonDeployment* in the same way it has been done manually in this paragraph (clearly everything is configured at Kubernetes level to have the rights to create this kind of resource in the namespace `kubeflow-user-example-com.)`

6.5.5 Running the pipeline

As already mentioned at the end of chapter 6.5.2, the pipeline in Figure 59 is a python program. The output of this program is the compiled pipeline yaml file that is what Kubeflow understands as a pipeline. The last two lines of the program

```
if __name__ == '__main__':
    kfp.compiler.Compiler().compile(first_pipeline, compiled-pipeline.yaml')
```

define the path of compiled pipeline yaml file (in this case *compiled-pipeline.yaml)*.

compiled-pipeline.yaml

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: pipeline-for-matteo-modeld-
  annotations: {pipelines.kubeflow.org/kfp_sdk_version: 1.8.11,
pipelines.kubeflow.org/pipeline_compilation_time: '2022-04-12T14:51:50.215341',
    pipelines.kubeflow.org/pipeline_spec: '{"description": "implement the movie project
      ML", "name": "Pipeline for Matteo Modeld"}'}
  labels: {pipelines.kubeflow.org/kfp_sdk_version: 1.8.11}
spec:
  entrypoint: pipeline-for-matteo-modeld
  templates:
  - name: download-data-function
    container:
      args: []
      command: [python, /pipeline/download_data.py, --data-bucket, /tmp/outputs/DataBucket/data]
      env:
      - {name: AWS_ACCESS_KEY_ID, value: minio}
      - {name: AWS_SECRET_ACCESS_KEY, value: *********}
      - {name: MLFLOW_S3_ENDPOINT_URL, value: 'http://s3.mlflow.svc.cluster.local:9000'}
      - {name: MINIO_BUCKET, value: mlflow}
      image: harbor.infinitech-h2020.eu/test/download-data:v6
    outputs:
      artifacts:
      - {name: download-data-function-DataBucket, path: /tmp/outputs/DataBucket/data}
    metadata:
      labels:
        pipelines.kubeflow.org/kfp_sdk_version: 1.8.11
        pipelines.kubeflow.org/pipeline-sdk-type: kfp
        pipelines.kubeflow.org/enable_caching: "true"
      annotations: {pipelines.kubeflow.org/component_spec: '{"description": "Download
          data from OneDrive and upload on kubeflow MINIO", "implementation": {"container":
          {"command": ["python", "/pipeline/download_data.py", "--data-bucket", {"outputPath":
          "DataBucket"}], "image": "harbor.infinitech-h2020.eu/test/download-data:v6"}},
          "name": "Download Data Function", "outputs": [{"description": "Name of Minio
          data bucket", "name": "DataBucket", "type": "String"}]}',
pipelines.kubeflow.org/component_ref: '{"digest":
          "63f3cc8f422bd2abbb5711ced11099b34fa9c72e2f72ff43bf030a1298fcb80a", "url":
          "download_data/download_data_pipeline_component.yaml"}'}
  - name: pipeline-for-matteo-modeld
    dag:
      tasks:
      - {name: download-data-function, template: download-data-function}
      - name: seldondeploy
        template: seldondeploy
        dependencies: [training-funcion]
      - name: show-results
        template: show-results
        dependencies: [training-funcion]
        arguments:
          parameters:
          - {name: training-funcion-ModelBucket, value: '{{tasks.training-
funcion.outputs.parameters.training-funcion-ModelBucket}}'}
          - {name: training-funcion-ModelKey, value: '{{tasks.training-
funcion.outputs.parameters.training-funcion-ModelKey}}'}
      - name: training-funcion
        template: training-funcion
        dependencies: [download-data-function]
        arguments:
          artifacts:
          - {name: download-data-function-DataBucket, from: '{{tasks.download-data-
function.outputs.artifacts.download-data-function-DataBucket}}'}
  - name: seldondeploy
    resource:
```

```
      action: create
      manifest: |
        apiVersion: machinelearning.seldon.io/v1alpha2
        kind: SeldonDeployment
        metadata:
          name: mlflow-model-matteo
          namespace: kubeflow-user-example-com
        spec:
          name: wines
          predictors:
          - componentSpecs:
            - spec:
                containers:
                - livenessProbe:
                    failureThreshold: 200
                    httpGet:
                      path: /health/ping
                      port: http
                      scheme: HTTP
                    initialDelaySeconds: 80
                    periodSeconds: 5
                    successThreshold: 1
                  name: seldon-serve
                  readinessProbe:
                    failureThreshold: 200
                    httpGet:
                      path: /health/ping
                      port: http
                      scheme: HTTP
                    initialDelaySeconds: 80
                    periodSeconds: 5
                    successThreshold: 1
            graph:
              children: []
              envSecretRefName: minio-secret
              implementation: MLFLOW_SERVER
              modelUri: s3://mlflow/LatestModel/artifacts/model
              name: seldon-serve
            name: default
            replicas: 1
    outputs:
      parameters:
      - name: seldondeploy-manifest
        valueFrom: {jsonPath: '{}'}
      - name: seldondeploy-name
        valueFrom: {jsonPath: '{.metadata.name}'}
    metadata:
      labels:
        pipelines.kubeflow.org/kfp_sdk_version: 1.8.11
        pipelines.kubeflow.org/pipeline-sdk-type: kfp
        pipelines.kubeflow.org/enable_caching: "true"
  - name: show-results
    container:
      args: [--key, '{{inputs.parameters.training-funcion-ModelKey}}', --bucket,
'{{inputs.parameters.training-funcion-ModelBucket}}']
      command:
      - sh
      - -ec
      - |
        program_path=$(mktemp)
        printf "%s" "$0" > "$program_path"
        python3 -u "$program_path" "$@"
      - |
```

```
        def show_results(key,
                         bucket):
            print(f"Key: {key}")
            print(f"Bucket: {bucket}")

        import argparse
        _parser = argparse.ArgumentParser(prog='Show results', description='')
        _parser.add_argument("--key", dest="key", type=str, required=True,
default=argparse.SUPPRESS)
        _parser.add_argument("--bucket", dest="bucket", type=str, required=True,
default=argparse.SUPPRESS)
        _parsed_args = vars(_parser.parse_args())

        _outputs = show_results(**_parsed_args)
      image: python:3.7
    inputs:
      parameters:
      - {name: training-funcion-ModelBucket}
      - {name: training-funcion-ModelKey}
    metadata:
      labels:
        pipelines.kubeflow.org/kfp_sdk_version: 1.8.11
        pipelines.kubeflow.org/pipeline-sdk-type: kfp
        pipelines.kubeflow.org/enable_caching: "true"
      annotations: {pipelines.kubeflow.org/component_spec: '{"implementation": {"container":
          {"args": ["--key", {"inputValue": "key"}, "--bucket", {"inputValue": "bucket"}],
          "command": ["sh", "-ec", "program_path=$(mktemp)\nprintf \"%s\" \"$0\" >
          \"$program_path\"\npython3 -u \"$program_path\" \"$@\"\n", "def show_results(key,\n
bucket):\n    print(f\"Key:
          {key}\")\n    print(f\"Bucket: {bucket}\")\n\nimport argparse\n_parser =
          argparse.ArgumentParser(prog=''Show results'', description='''')\n_parser.add_argument(\"-
-key\",
          dest=\"key\", type=str, required=True,
default=argparse.SUPPRESS)\n_parser.add_argument(\"--bucket\",
          dest=\"bucket\", type=str, required=True, default=argparse.SUPPRESS)\n_parsed_args
          = vars(_parser.parse_args())\n\n_outputs = show_results(**_parsed_args)\n"],
          "image": "python:3.7"}}, "inputs": [{"name": "key", "type": "String"}, {"name":
          "bucket", "type": "String"}], "name": "Show results"}',
pipelines.kubeflow.org/component_ref: '{}',
        pipelines.kubeflow.org/arguments.parameters: '{"bucket": "{{inputs.parameters.training-
funcion-ModelBucket}}",
          "key": "{{inputs.parameters.training-funcion-ModelKey}}"}'}
  - name: training-funcion
    container:
      args: []
      command: [python, /work/main_my_INFINITECH_pod.py, --data-bucket, /tmp/inputs/DataBucket/data,
        --model-key, /tmp/outputs/ModelKey/data, --model-bucket, /tmp/outputs/ModelBucket/data]
      env:
      - {name: AWS_ACCESS_KEY_ID, value: minio}
      - {name: AWS_SECRET_ACCESS_KEY, value: *********}
      - {name: MLFLOW_S3_ENDPOINT_URL, value: 'http://s3.mlflow.svc.cluster.local:9000'}
      - {name: MLFLOW_TRACKING_URI, value:
'postgresql+psycopg2://mlflow:mlflow@postgresql.mlflow.svc.cluster.local:5432/mlflow-db'}
      image: harbor.infinitech-h2020.eu/test/training:v5
    inputs:
      artifacts:
      - {name: download-data-function-DataBucket, path: /tmp/inputs/DataBucket/data}
    outputs:
      parameters:
      - name: training-funcion-ModelBucket
        valueFrom: {path: /tmp/outputs/ModelBucket/data}
      - name: training-funcion-ModelKey
        valueFrom: {path: /tmp/outputs/ModelKey/data}
      artifacts:
      - {name: training-funcion-ModelBucket, path: /tmp/outputs/ModelBucket/data}
```

```
      - {name: training-funcion-ModelKey, path: /tmp/outputs/ModelKey/data}
    metadata:
      labels:
        pipelines.kubeflow.org/kfp_sdk_version: 1.8.11
        pipelines.kubeflow.org/pipeline-sdk-type: kfp
        pipelines.kubeflow.org/enable_caching: "true"
      annotations: {pipelines.kubeflow.org/component_spec: '{"description": "Train
          the model", "implementation": {"container": {"command": ["python",
"/work/main_my_INFINITECH_pod.py",
          "--data-bucket", {"inputPath": "DataBucket"}, "--model-key", {"outputPath":
          "ModelKey"}, "--model-bucket", {"outputPath": "ModelBucket"}], "image":
          "harbor.infinitech-h2020.eu/test/training:v5"}}, "inputs": [{"description":
          "Name of Minio data bucket", "name": "DataBucket", "type": "String"}], "name":
          "Training funcion", "outputs": [{"description": "Key of model in Minio bucket",
          "name": "ModelKey", "type": "String"}, {"description": "Name of Minio model
          bucket", "name": "ModelBucket", "type": "String"}]}',
pipelines.kubeflow.org/component_ref: '{"digest":
          "a02eb723cee1bda6d6e918a5642e4bba5d8ea44d5eca3a453efec61f97171f69", "url":
          "training/training_pipeline_component.yaml"}'}
  arguments:
    parameters: []
  serviceAccountName: pipeline-runner
```

One possible way to run the pipeline on Kubeflow is to upload the previous file using the Kubeflow Dashboard at http://a5b3e77eecbf24dae941312c6aeb9fc7-454945384.eu-central-1.elb.amazonaws.com/: once authenticated the main page looks as follows:

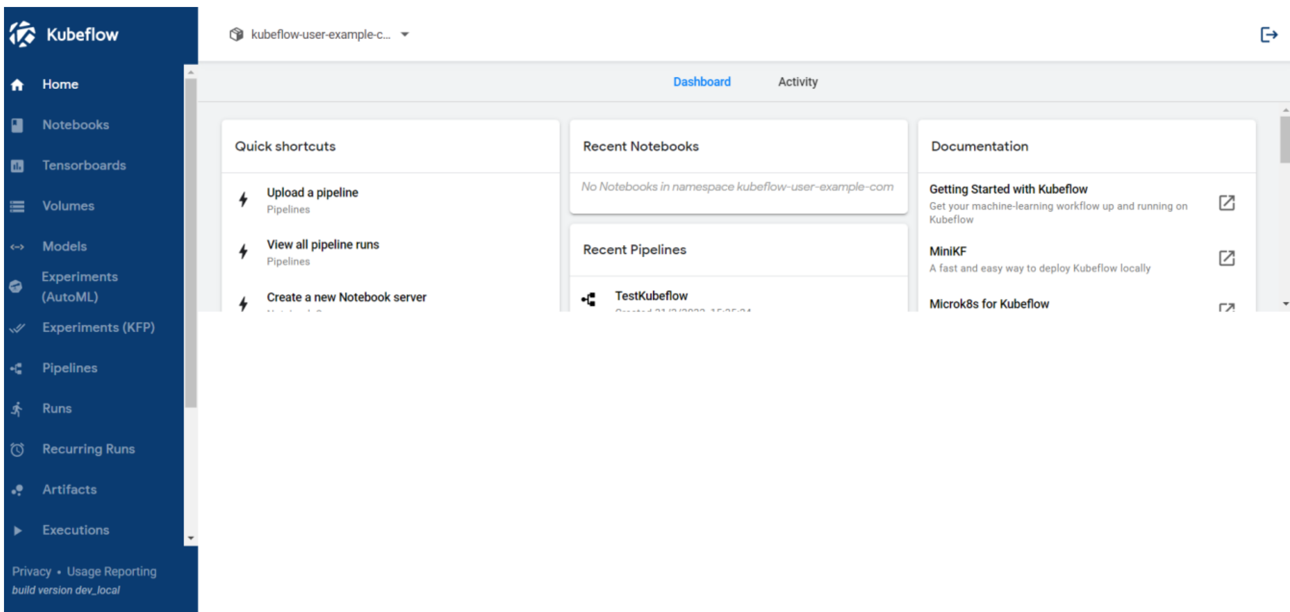


Figure 61 - Kubeflow Dashboard main page

To upload the pipeline file, it is necessary to click on *Upload a pipeline* (see figure below).

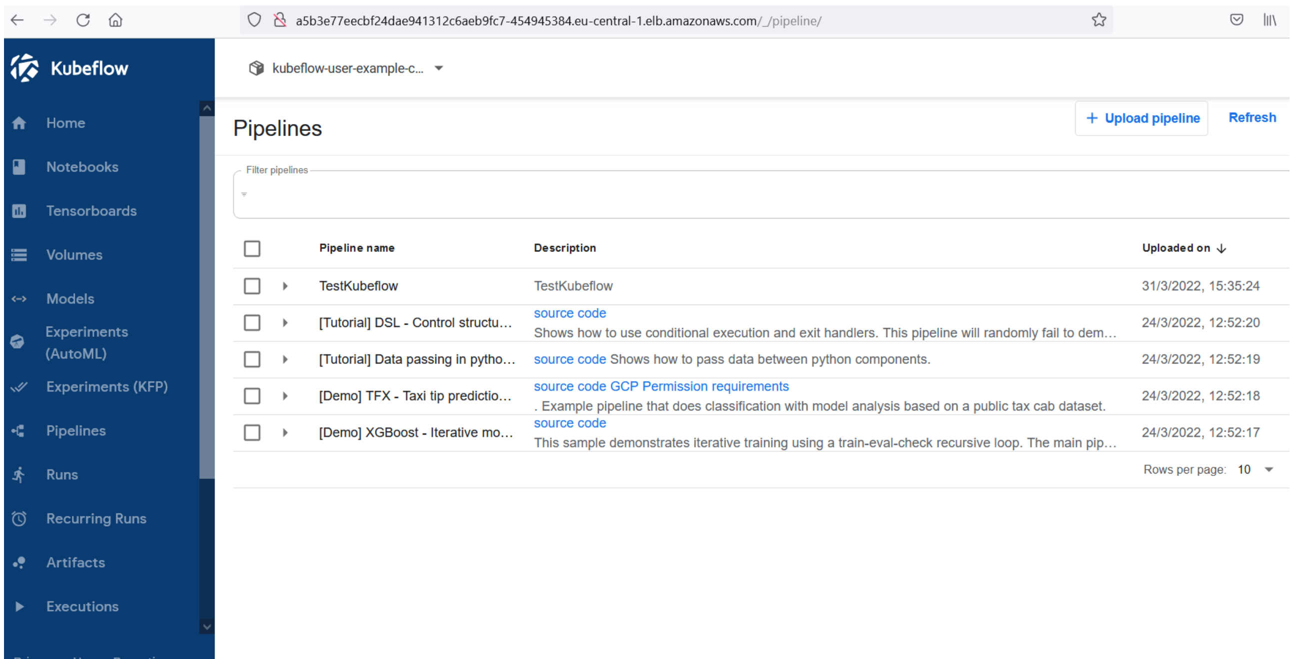


Figure 62 - Kubeflow Dashboard Upload a pipeline

And then click on *+ Upload a pipeline* button on the top right of the screen and choose the *compiled-pipeline.yaml* file (see figure below)

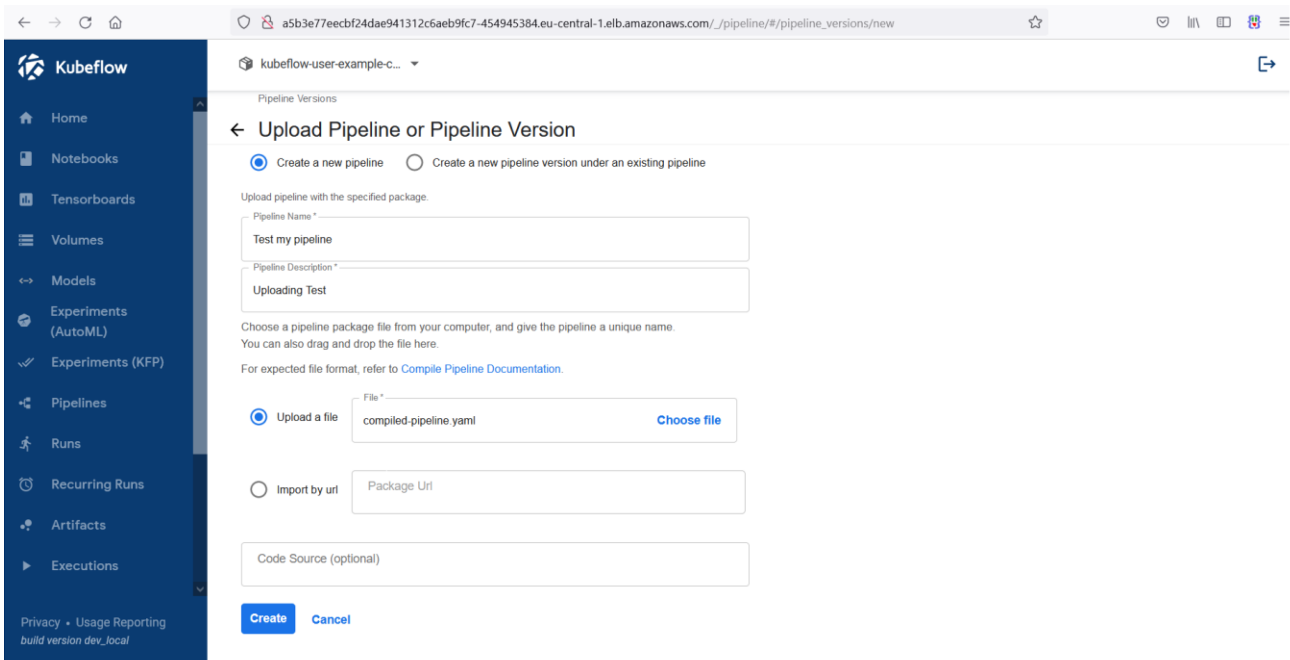


Figure 63 - kubeflow choose a pipeline yaml file

Clicking on *Create,* the following page appears:

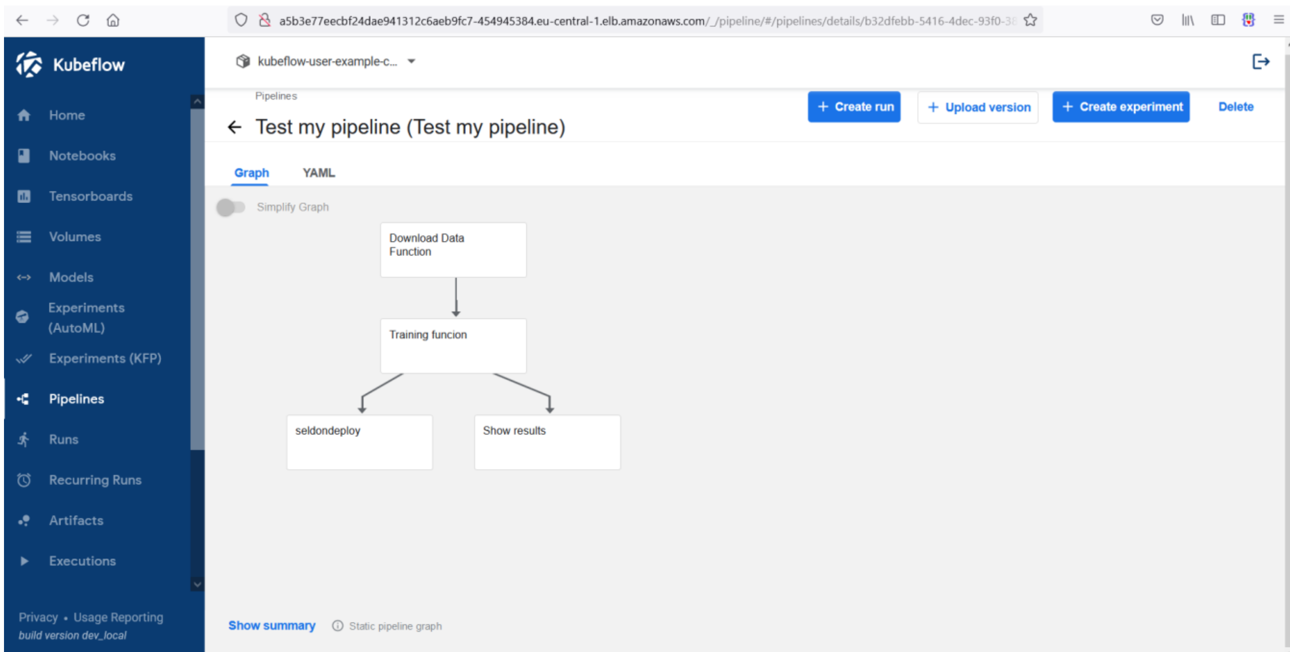


Figure 64 - Uploaded pipeline

A pipeline runs inside an *experiment*, so an experiment is the place where it is possible to find the pipeline results after its running. If it is the first time that the pipeline runs it is necessary to create a new experiment with the *+ Create experiment button,* otherwise it is possible to run the pipeline into an existing experiment. Assuming there is already an experiment created, click on the *+ Create run* button and choose that experiment like in the following picture:

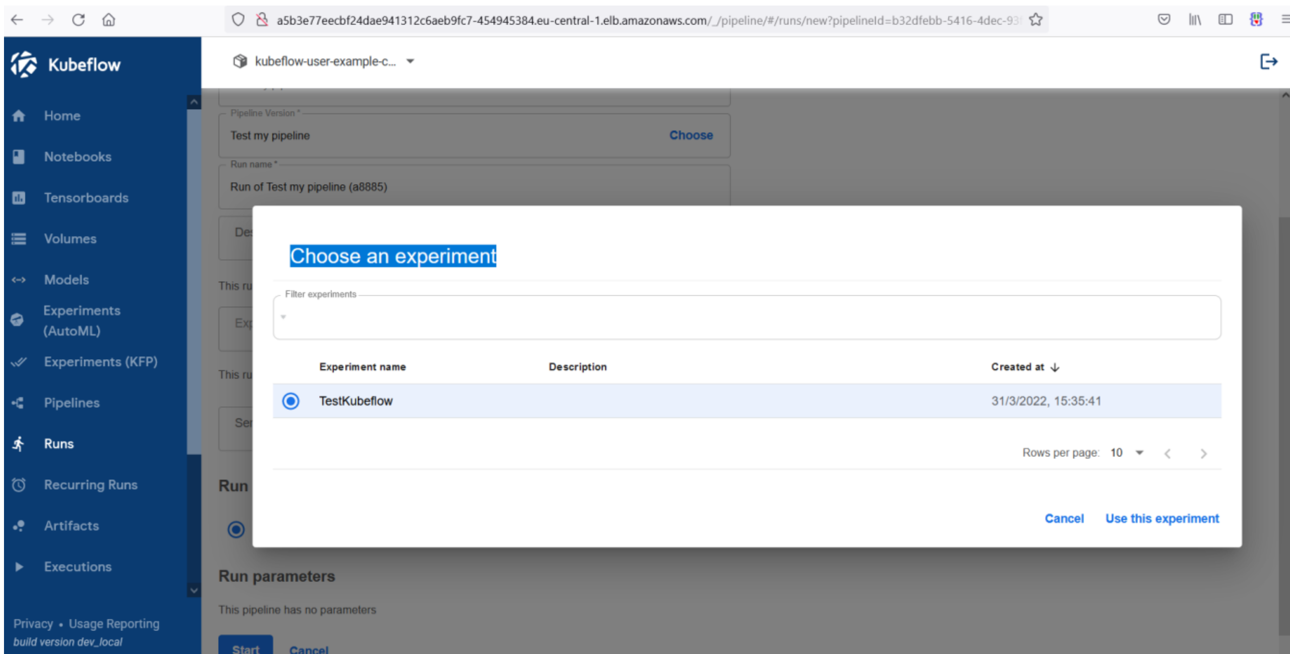


Figure 65 - Choose an experiment to run a pipeline

Once an experiment has been chosen click on *Use the experiment*:

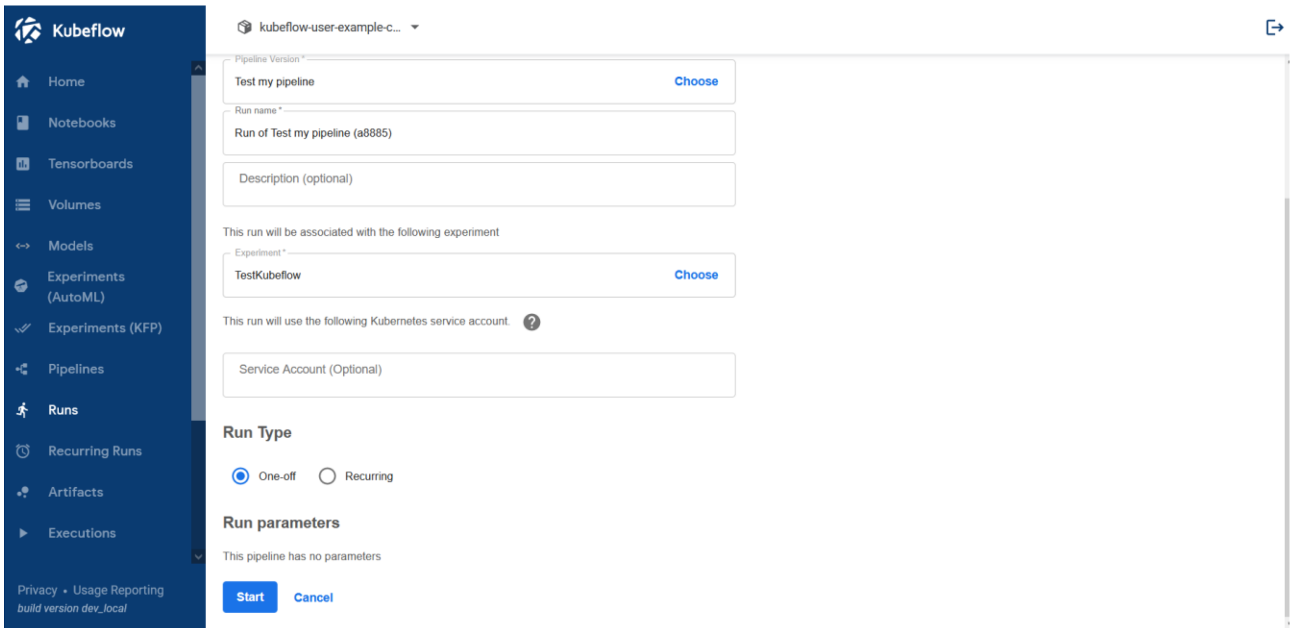


Figure 66 - Start a pipeline

At this stage it is possible to start the pipeline pressing the *Start* button (see figure above). Once the pipeline has started it is possible to access the specific run (a8885) and monitor each specific step (logs, input, output, etc.) as in the following pictures. All data will remain inside the experiment unless they are manually deleted.

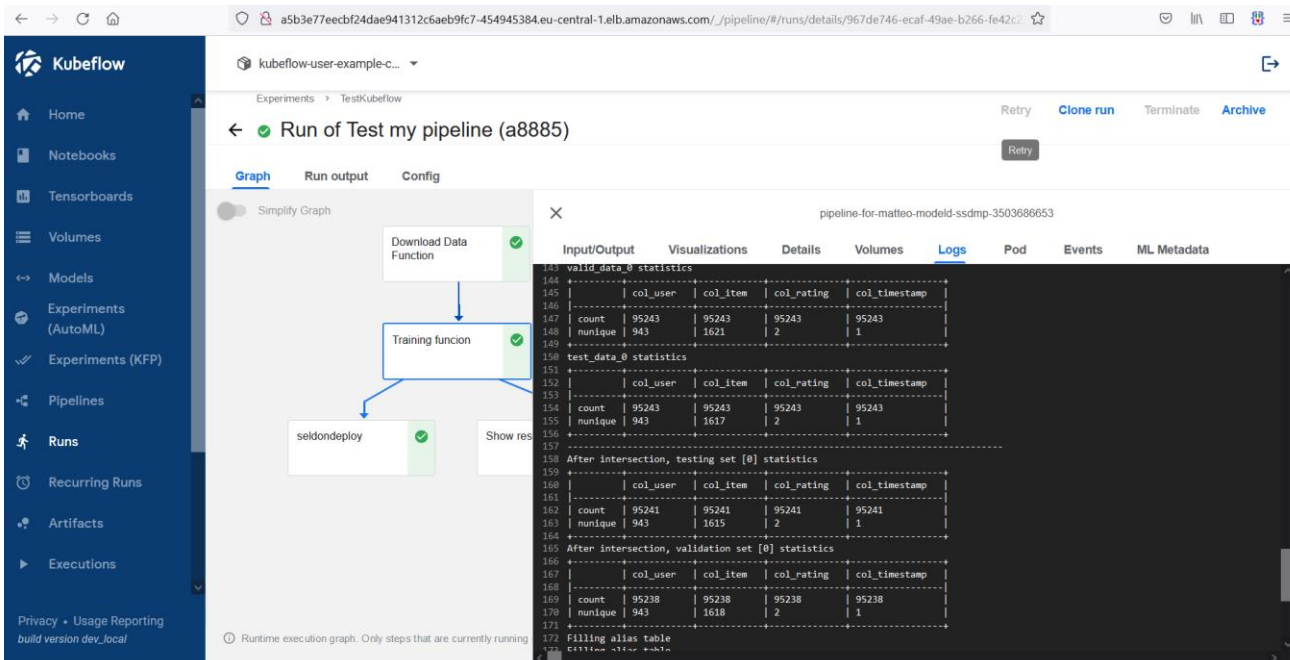


Figure 67 - Pipeline running view

When the pipeline running has been terminated, the model is trained and the Seldon Deploy has been automatically created to serve it, so it is possible now to ask for predictions using the Seldon Deploy API (refer to the previous chapter 6.5.4).

7 Implementation of the Blueprint guidelines to INFINITECH blueprint pilots

This chapter provides the description of the actual and concrete implementation of the "INFINITECH Way" Blueprint guidelines described in the previous chapter, applied to three (they were two in the previous release) of the official INFINITECH pilots, the so-called "INFINITECH blueprint pilots", namely:

- **Pilot #5b: Business Financial Management tools delivering Smart Business Advice**, owned by the partner **Bank of Cyprus (BoC)**;
- **Pilot #2: Real-time risk assessment in Investment Banking**, owned by partner **JRC**.
- **[NEW] Pilot #10: Platform for Real-time cybersecurity analytics on Financial Transactions' BigData**, owned by partners **Poste Italiane** and **ENG** (3rd and additional blueprint pilot added in this deliverable)

Those three pilots act as reference and "drive by example" for the future implementation of the guidelines to (possibly) all the official INFINITECH blueprint pilots (actually 15 pilots at the time of writing), planned for the final phase of the INFINITECH project.

7.1 Blueprint environment for Pilot #5b: Business Financial Management tools delivering Smart Business Advice

This section describes the final concrete INFINITECH blueprint environment associated to one of the official INFINITECH pilots, the **Pilot #5b: Business Financial Management (BFM) tools delivering a Smart Business Advice**, owned by partner **Bank of Cyprus (BoC)**, and also set up the basis for the future applications of the concepts described in Chapter 6 for all the INFINITECH target pilots' environments.

7.1.1 Pilot Objectives

Most of today's Financial Management tools for Small Medium Enterprises (SMEs) are geared towards analysing only past transactions, making such tools inadequate in today's world. Today, SMEs and their customers alike, demand just-in-time processing, transparency and personalized services to assist SME owners not only in understanding better their SME business/financial health but also to be able to model their options and to decide on the next best action to take.

Thus, Pilot 5b aims to assist SME clients of Bank of Cyprus in managing their financial health in the areas of cash flow management, continuous spending/cost analysis, budgeting, revenue review and VAT provisioning, all by providing a set of AI-powered Business Financial Management tools and harnessing available data to generate personalized business insights and recommendations. Machine learning algorithms, predictive analytics and AI-based interfaces will be utilized to develop a kind of smart virtual advisor with the aim to minimize SME business analysis effort, to focus on growth opportunities and to optimize cash flows performance.

7.1.2 Pilot Workflow

The pilot workflow can be presented starting from the datasets that it has to manage. For the data collection process, tokenized data from designated BoC databases, as well as data from open sources and SME ERP/Accounting software will be migrated to the data repository of the BoC testbed, following a reverse pseudonymization technique before returning to BoC premises and be displayed to the SME clients. Some of the available datasets require real time data collection, while in others historical data collection is sufficient

to provide actionable business insights. In more detail, transaction and account data related to the respective SME will be drawn from BOC's repository by a real time/historical data collector as well as transaction and account data from Open Banking (PSD2), as well as BOC's customer data, will utilize a historical data collector. In addition, an external data collector will be used to integrate other related Open Banking/macroeconomic data. The SMEs' data source (e.g. ERP/Accounting system) utilization remains optional as consent is required for the collection and processing of such data and its cloud availability being required. The involved data sources involved are summarized in the following list:

- Transaction Data from Open Banking (PSD2).
- Transaction Data from SMEs (optional).
- Other Data (Market).
- Other Data from SMEs (optional).
- Accounts Data from BOC.
- Accounts Data from Open Banking.
- Customer Data from BOC.
- Direct Input from SMEs.

In the target pilot PoC, all data except external macroeconomic data will be pseudonymised (by tokenization) before being uploaded to the IRA.

The cloud Data Repository (within the IRA) will then store all collected data (along with the generated insights), past SME financial actions (to measure to what degree an SME's actions reflect the recommended insights), as well as the minimum user input that is required. A continuous data streaming will connect the Data Repository with the various deployed BFM tools (machine learning algorithms), which would allow the retraining of the respective AI models and the generation of useful insights and recommended actions. A reverse data pseudonymisation will then be applied before the processed data move to the bank middleware component that contains composite APIs and produces push notifications, all of which will be offered to the SMEs via Android, iOS and web applications. Upon SME user login, the IRA is also accessed, insights/recommendations picked up from the cloud data repository and provided to the SME user.

The pilot workflow described is depicted in Figure 68.

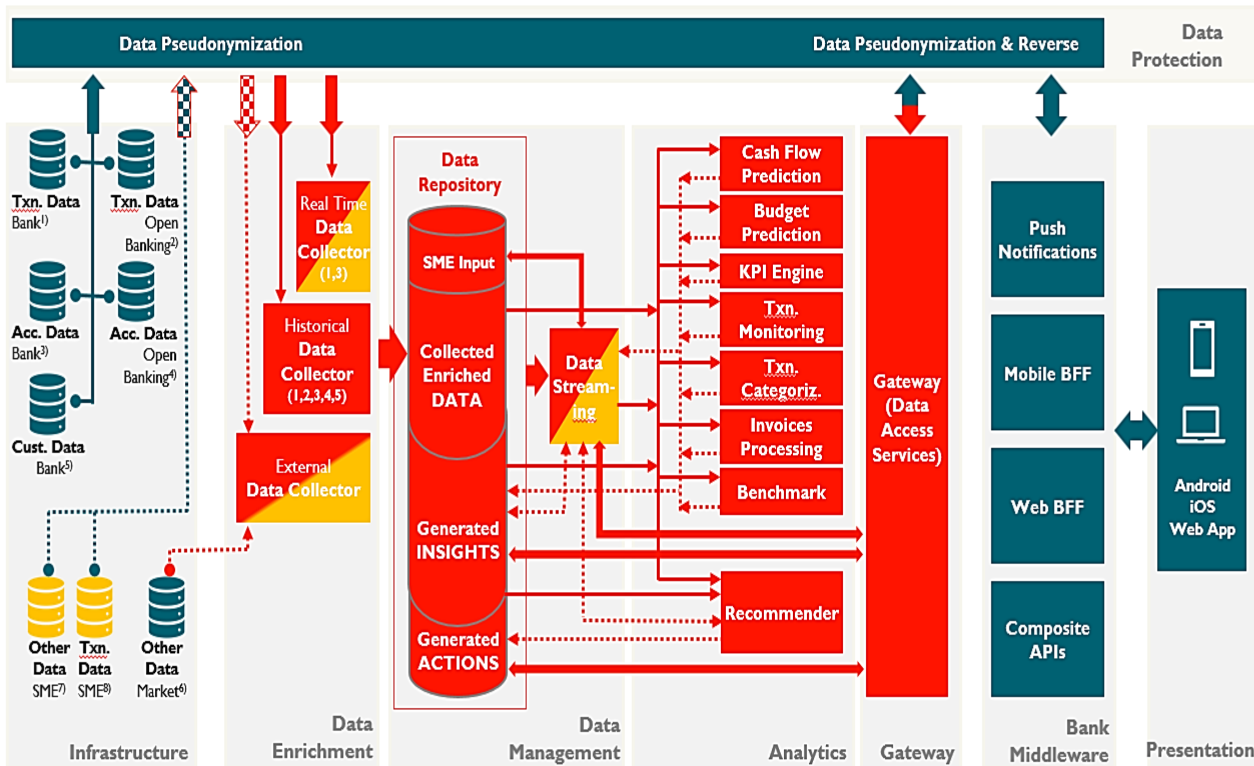


Figure 68 - Pilot #5b workflow

7.1.3 Blueprint reference testbed implementation

Within the context of this deliverable the third version of blueprint reference testbed (Blueprint_v3) and PoC version (Prototype_v3) is introduced. Accordingly, the PoC will not use all available components of the INFINITECH platform, as illustrated in Figure 68, but just a selected subset.

In more detail, the blueprint testbed implementation for pilot #5b is based on the following assumptions:

- The PoC version of the pilot (Prototype_v3) does not fully represent the BFM advisor, since some of the pilot-specific data analytics services are still under development (i.e. 4 analytics services are integrated till now) while the implementation of the invoice processing engine remains optional, depending on other bank activities for its development and implementation.
- The pilot does not utilize Blockchain technology, so all the Blockchain-oriented components will not be integrated.
- Data migrated to the blueprint testbed is owned by BOC and will be pseudonymised (using the tokenization technique) before entering the INFINITECH ecosystem. There is no need for an INFINITECH anonymizer for the Prototype_v3 PoC.
- Regarding the Data Management layer, the core component is the Infinistore where both historical and new data are stored. In this direction, both Kafka queue and zookeeper (provided by INFINITECH) and JDBC connections are used in terms of data handling.
- Finally, as implemented the components that comprise the logic of the BFM tool are pilot-specific, but of course following the INFINITECH way.

Prototype_v3 and Blueprint_v3 versions will include the components listed in Table 2.

Group	Prototype_v3	Blueprint_v3

On-Premises Data source Layer and Data Management	An FTP server will provide historical data Batch of data in csv format loading Stream of new transaction data (under development)	1) A data loader thread within the transaction-categorization POD 2) A REST api connected to the Kafka queue 3) Infinistore will persistent volume 4) JDBC connections between the infinistrore and the analytics pods
Analytics Layer	Cash Flow Prediction, Transaction Categorization Benchmarking	Cash Flow Prediction -pod Transaction Categorization - pod Benchmarking - pod
Data Security and Privacy	Encrypted SQL queries (under development)	Data Anonymization (by BoC) Encrypted queries are tested as infinistore feature
Interfaces	REST API	REST API NodePorts: exposes the service on a static port on the node IP address.

Table 2 - Prototype_v3 vs Blueprint_v3 components

The next and final version of the PoC prototype (Prototype_v4) will include stream processing and some complementary components of the Analytics group. Based on the pilot #5b development roadmap, Prototype_v4 will be hosted on the private AWS cloud of BoC, where all progress will be migrated from the Project's reference testbed currently being utilized.

The ML/DL models of the Analytics Components (Cash Flow Prediction, Transaction Categorization) are trained offline once a day and saved in the persistent volume provided by the testbed. It is also noted that the data provided till now are already cleaned so the trivial pre-processing needed, is included in each component's pipeline in each specific container. Also in the next version, an incremental way of updating the ML models will be explored in order not to request the whole dataset for retaining the models.

Transaction-categorization: In this Analytics component both a rule-based and ML-based model were incorporated in a hybrid model. The main reason for that is that we provide the interaction of the user by enabling the user to re-categorize a given transaction category. To this end, an ML model is retrained every day in order to update the new information provided by the users. For retraining the model, the whole dataset is being retrieved from the Infinistore, this will be optimized in the next version.

The prototype component diagram is represented in the figure below:

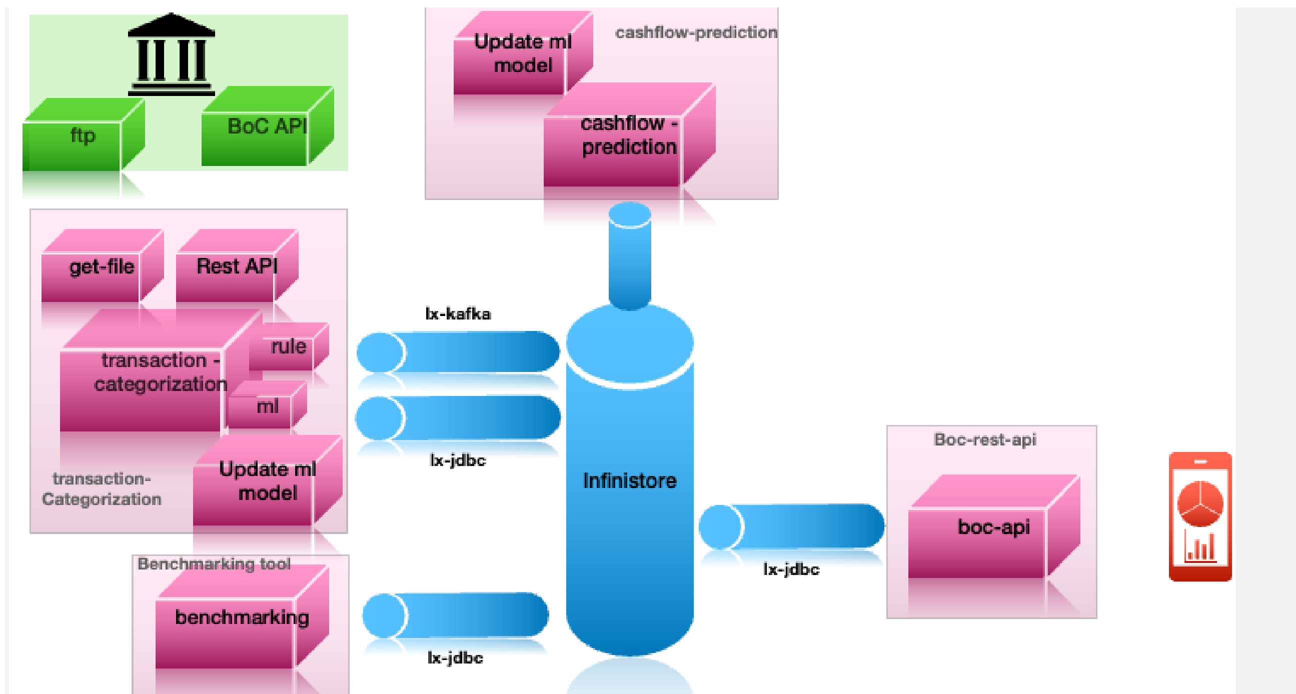


Figure 69 - Prototype_v3 components diagram

Cashflow-prediction: The second microservice implemented in the Prototype_v3 is the Cash Flow Prediction Engine, which aims to accurately predict the cash inflows and outflows of the given categories produced by our hybrid transaction categorization model for each SME. To this end, a probabilistic Deep RNN is used, applied on each time series (per account per category) as a multivariate time series. Given that the prediction horizon is extended to 3 months, both resampling and aggregating the amount of the transactions based on each specific account and date have been applied to our data sample. These processes were included in the containers. The DL model is updated once a day. In the next version online aggregates of Infinistore will replace this pre-processing step.

Benchmarking tool: In the scope of pilot5b we developed a benchmarking tool that can be used by SMEs to evaluate themselves against their competitors, based on their size and operating sector. This tool is a pilot specific tool and can be applied only to BOC clients.

Transaction-monitoring: This tool is under development, till now an approach of representing the transactions as Graphs leveraging Neo4j is used. As the transaction monitoring is still under development, it is not yet deployed in the Prototype_v3.

Boc-api: Given the above-mentioned points, a REST API (boc-api) is developed on top, which delivers real-time information when invoked. This data is forwarded directly to the Infinistore via JDBC connections, it is likely in the next version of the Prototype this data could be forwarded to the analytics component if needed.

infinistore: Is the datastore on a persistent volume provided from the INFINITECH.

lx-kafka: Additionally, an interface between the transaction categorisation and Infinistore is developed to handle the real time data streams ingested in the application from the data provider. As these microservices need to communicate asynchronously, ensuring high availability and fault-tolerance, the most popular intermediate is the use of data queues. Apache Kafka is the most dominant when it comes to data items.

Demo_android_app: Within INFINITECH, an android app has been developed to present and serve the created data from the aforementioned components.

In order to run the pilot inside the blueprint environment, a dedicated sandbox named **pilot5b** has been created. The components deployed in such a sandbox are:

- Analytics (transaction-categorisation, cashflow-prediction, benchmarking)
- A REST API (boc-api).
- Infinistore (with persistent volume).
- lx-kafka.

All the components are packaged following the INFINITECH way as docker containers deployed in a Kubernetes cluster (i.e. AWS EKS) [15]. More specifically, every component is deployed in a POD with specific features within the cluster. The most important one is the capability of auto-scaling when needed in terms of the demand. Thus, as a sandbox is defined all the components deployed under the same namespace allow the interaction and connection between them.

The client connections from the external world towards such a sandbox and all sandboxes deployed in the blueprint are managed through NodePorts (see Figure 2), which is a key component of the IRA Interface layer.

The process workflow implies that when a user sends a (HTTPS) request it is forwarded by the boc-api to the Infinistore POD, from there the required data will be retrieved and return to the application in json format response. The proposed blueprint architecture is illustrated in Figure 70.

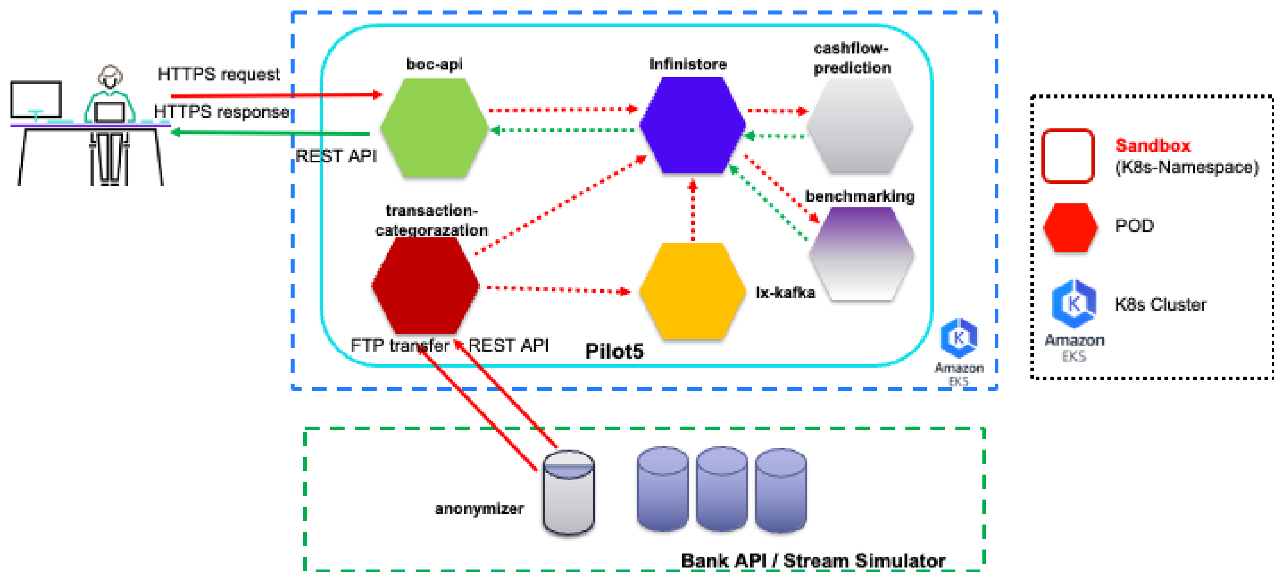


Figure 70 - Pilot #5b blueprint reference architecture

7.2 Blueprint environment for Pilot #2: Real-time risk assessment in Investment Banking

This section describes the final concrete INFINITECH blueprint environment associated to one of the official INFINITECH pilots, the **Pilot #2: Real-time risk assessment in Investment Banking**, owned by the partner **JRC**, and also, like for the previous pilot, setup the basis for the future applications of the concepts described in the previous section 6 for all the INFINITECH target pilots' environments.

7.2.1 Pilot Objectives

Risk assessment is of high importance when it comes to trading, investments, and other financial activities, as poor risk monitoring could lead to inefficient investments, loss of capital and penalties by regulatory authorities. Thus, robust risk models, capable of yielding real-time results, are valuable assets for investment banking. Pilot2 introduces a financial tool which is able to provide risk assessment on Forex portfolios in (near) real-time. Financial risk is measured in terms of both Value-at-Risk and the Expected Shortfall, with the respective models utilizing not only statistical but also deep learning techniques (i.e., DeeVaR [47]) that achieve state of the art results. The pilot, based on INFINITECH data management technologies, provides real-time risk assessments, utilizing the latest market data [48]. In addition, it provides a state of the art model for sentiment analysis in financial texts based on transfer learning techniques. These features along with the provided pre-trade analysis make this solution a valuable tool for practitioners in high frequency trading and investment banking in general.

7.2.2 Pilot Workflow

The pilot workflow could be divided into five distinct layers as they are depicted in Figure 71. Starting with data sources, there is the infrastructure layer related to the bank side providing the input data which consists of real time prices, historical prices, trading positions and news in a text form. Then, the input data is injected to the INFINITECH ecosystem through the data management layer, which is able to handle both stream and batch data. Data processing and analytics layers are responsible for performing all the required data transformations and calculations in order to predict all the required risk metrics of the input trading positions and the current market sentiment. Finally, for the pilot-specific needs an appropriate User Interface has been developed not only to visualize the risk predictions but also to perform pre-trade analysis leveraging the developed risk models at rest.

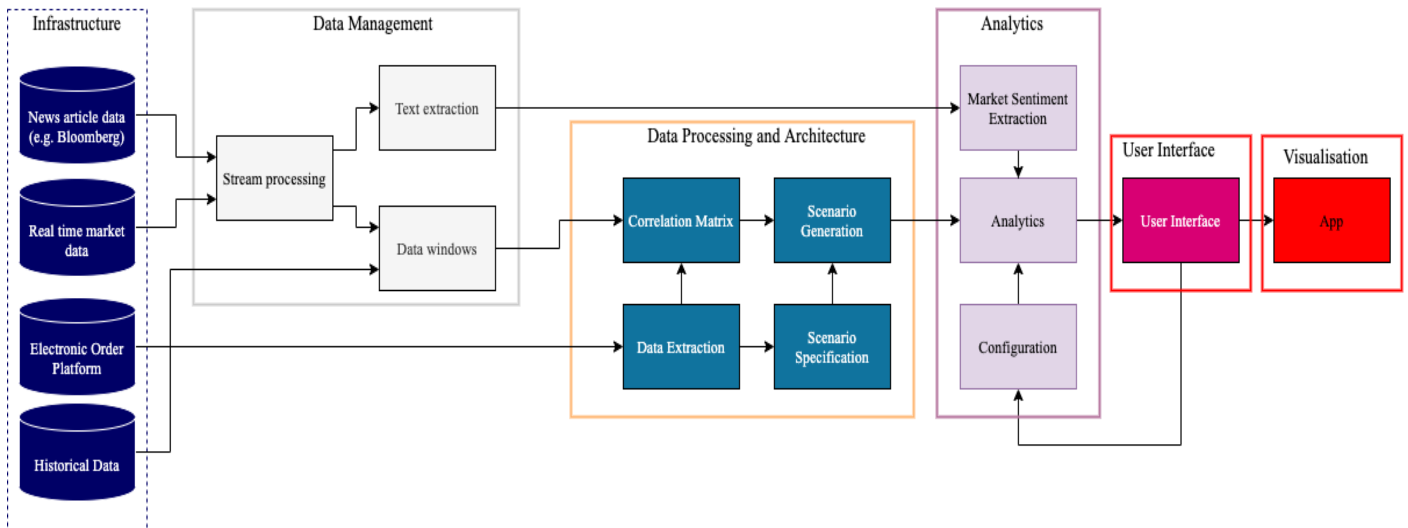


Figure 71 - Pilot #2 workflow

It is noted that, as this use case is under continuous development and in order to test the implemented components, the market data (i.e., real-time prices and trades) of the infrastructure layer is simulated by stream generators. The latter are producing both market and trades data at the same frequency as the provided datasets. The financial news is obtained directly from Twitter's API, however, any other data feed could be integrated to the Pilot solution leveraging its RESTful interface. In such a way, it is feasible to simulate the real-time process and prove that the utilized components are integrated properly.

7.2.3 Blueprint reference testbed implementation

This sub-section describes the mapping of the aforementioned workflow to the actual implementation of Pilot 2, according to the IRA, to the reference testbed. The current version of the pilot consists of five components, three (AI for VaR Prediction, Sentiment Analysis for Financial News, Infinistore, Kafka) of which are provided by INFINITECH, while the remaining one (UI based on VaR) was developed for the pilot-specific needs. These are provided as containerized microservices leveraging the relevant advantages described in Chapter3. Pilot's Prototype and Blueprint include the components listed in Table 3 - Prototype vs Blueprint componentsand are further analysed in this section.

GROUP	Prototype	Blueprint
Analytics	AI for VaR Prediction	AI for VaR Prediction
Analytics	Sentiment Analysis for Financial News	Sentiment Analysis for Financial News
Data Management	LXS-Datastore	Infinistore
Injection	LXS-Kafka	LXS-Kafka
Interface	UI based on VaR	UI based on VaR

Table 3 - Prototype vs Blueprint components

The **AI for VaR Prediction** component, provided by the analytics group with the AI & ML algorithms, reads every minute the latest aggregated prices and trades data from the Infinistore, in order to predict VaR and ES. Therefore, every minute it provides updated risk estimations according to the latest market prices.

However, new trading positions are posted with a HTTP request directly to this component and then are saved to the datastore. As a result, the risk assessments are updated instantly when a new trading position takes place. The predictions along with the input trades are stored to the Infinistore to be used by the User Interface.

The **Sentiment Analysis for Financial News** component also provided from the analytics group provides the sentiment analysis feature using a state-of-the-art pretrained AI model named FinBert [49]. FinBERT is a pre-trained NLP model that analyses the sentiment of financial texts. It is built by further training the BERT language model in the finance domain, using a large financial corpus and thereby fine-tuning it for financial sentiment classification. Pilot 2 integrates this model coupled with the required data pre-processing workflows in a REST API that infers the sentiment of the input financial text. The predicted sentiment, the input and the cleaned text as well as its metadata (e.g., timestamp, creator, location etc.) are stored to the Infinistore and visualised in the pilot's UI

The third utilized component is the **Infinistore** from Data Management group. The JRC pilot relies on this and on its dual interface to firstly being able to ingest operational data at any rate and also to perform analytical query processing on the live data that have been added. Moreover, the historical data is stored by this component using the static data injection described is Section 4.1.1. The Infinistore also incorporates the "Online Aggregates" providing real time data analysis using a declarative way with standard SQL statements. In this way, the required aggregate operations, such as average price per instrument per minute are defined, and the result of the execution is pre-calculated on the fly. As a result, it can be retrieved instantly with a GET operation, enabling the utilization of the latest data.

The next component of Pilot 2 in the **LXS-Kafka** which is an interface between the data provider (e.g. Bank API) and the datastore, inside the sandbox, handling the real-time input data. In this use case, the stream of tick data is automatically pushed into the Infinistore, taking advantage of its dual query interface.

UI based on VaR is the last component of the current version of Pilot 2 being an API interface between the user and the provided application. On the one hand, UI based on VaR queries every minute the Infinistore in order to retrieve both the latest VaR/ES prediction, aggregated market data as well as market sentiment predictions. This is achieved via a JDBC connection using the components' internal IP addresses known only inside the sandbox. On the other hand, the user is able to send HTTP requests to this component inside the sandbox in order to perform pre-trade analysis.

Each component is deployed as a docker container in an automated manner, using Kubernetes container orchestration system. In such a way, all software artifacts of the integrated solution are given a common namespace, and they can interact with each other inside the namespace as a sandbox. The latter consists of the software artifacts deployed under the given namespace.

A high-level overview of this implementation is illustrated in Figure 72. Each component is running in a POD which is the simplest unit in the Kubernetes object. A POD encapsulates one container, having its own storage resources, a unique network IP, access port and options related to how the container should run. This setup enables the auto-scaling of the underlying services according to the volume of external requests. For instance, in case of influx of clients (i.e. traders) requiring risk assessment of their portfolios, the corresponding (red) POD will automatically scale-out as needed.

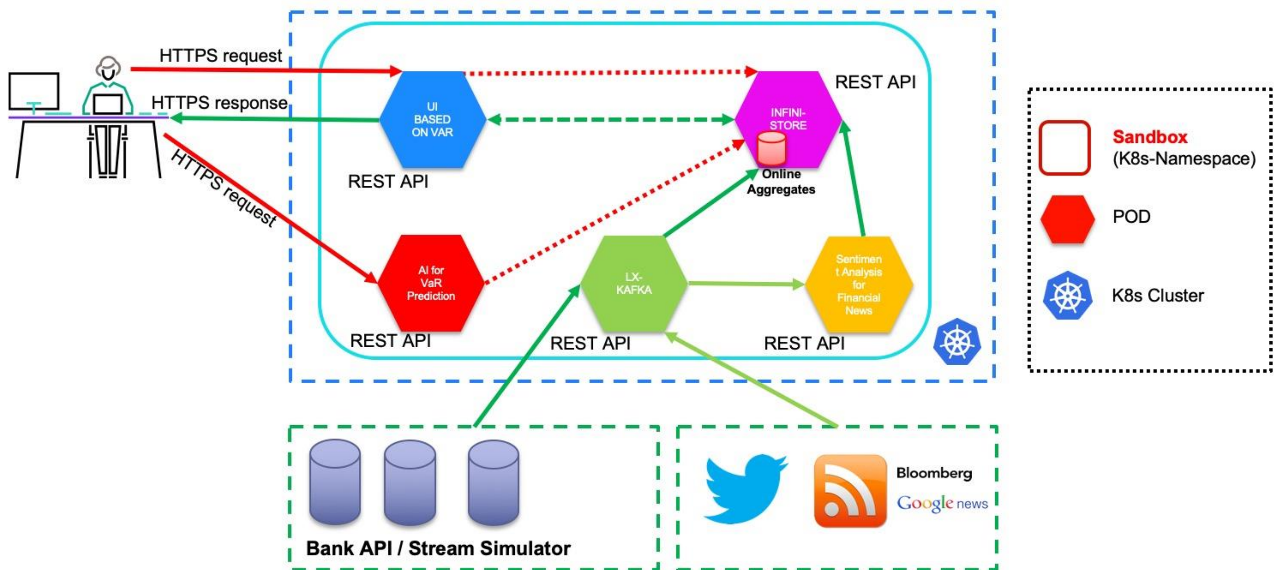


Figure 72 - Pilot #2 blueprint reference architecture

Using such a concept allows for the portability of the sandbox, as it can be deployed in different infrastructures. Towards this direction, Pilot 2 was initially deployed in INFINITECH blueprint reference testbed, hosted in AWS and then was successfully migrated from AWS to NOVA testbed which is its target infrastructure. From the developer's point of view, this process does not require any extra effort and can be completed in a timely manner, as it is irrelevant to the sandbox in which data center it will be hosted.

It is also noted that, as each pilot's service is provided as a REST API, additional features could be added as separate components to the existing sandbox. This could be easily achieved by assigning to the new microservice the respective namespace and by following INFINITECH CI/CD guidelines.

7.3 Blueprint environment for Pilot #10: Platform for Real-time cybersecurity analytics on Financial Transactions' BigData

This section describes the final concrete INFINITECH blueprint environment associated to one of the official INFINITECH pilots, the **Pilot #10: Platform for Real-time cybersecurity analytics on Financial Transactions' BigData**, owned by the partners **Poste Italiane** and **ENG**, and also, like for the previous pilots, setup the basis for the future applications of the concepts described in the previous section 6 for all the INFINITECH target pilots' environments.

7.3.1 Pilot Objectives

The objective of Pilot #10 is to craft a distributed detection system aimed at discovering new types of frauds in the financial context, leveraging AI and ML mechanisms that work at the same time on a big amount of data and live data. The software stack for real-time cybersecurity analytics, introduced by Pilot #10, is devised to reduce, in the detection process, the occurrences of the false positives by iteratively training, validating, and testing the involved ML models, to obtain in this way, more accurate and rapid notifications about suspicious financial movements and fraudulent patterns. The pilot demonstrates how the proposed real-time anomaly detection system can fit explainability characteristics leaving the control of the entire process to

human operators who can decide to customize, supervise, and integrate the system with third-party solutions for applying crucial mitigation measures.

7.3.2 Pilot Workflow

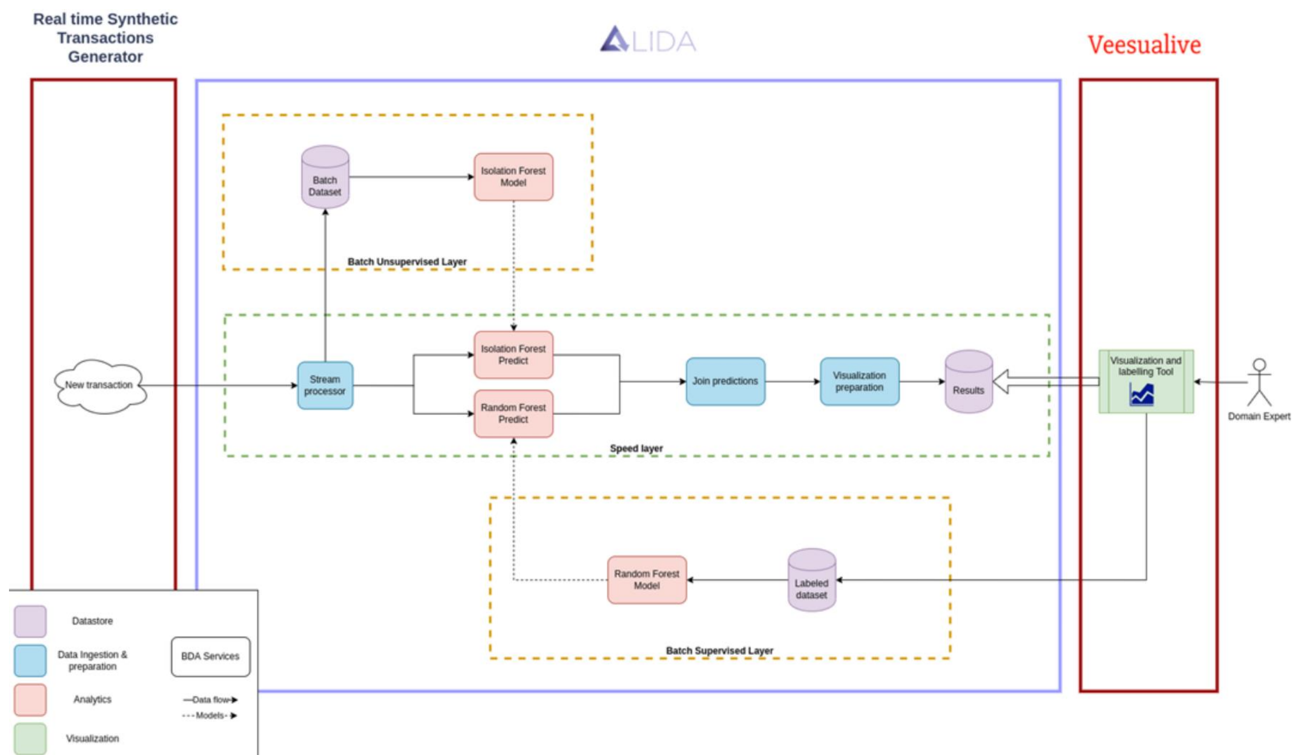


Figure 73 - Pilot #10 workflow

As depicted in Figure 73, Pilot #10 workflow consists of three logical tiers deployed in a hybrid testbed, meaning that the system relies on two different infrastructures: the data source tier and visualization/alerting tier run on AWS, and the processing tier is running on a dedicated infrastructure powered by ALIDA, a Data Science and Machine Learning platform based on advanced frameworks and open source technologies for design, deployment, execution, and monitoring of big data analytics workflows.

The data source tier has one instance of the Real-Time Synthetic Transactions Generator that is configured to produce SEPA transaction data (for further information regarding the component, please refer to the deliverable D7.10 - Predictive Financial Crime and Fraud Detection). Transactions are published in a queue available in a Kafka broker secured in the same tier.

The processing tier, composed of ALIDA workflows, is divided into three layers or branches: a speed layer, a batch unsupervised layer, and a batch supervised layer. These branches cooperate as follows: a stream processor in the speed layer subscribes to the SEPA transaction queue through Kafka and listens to incoming data. Data are sent to two destinations, or rather to the batch unsupervised layer for historical persistence, and two ALIDA ML services, the “Isolation Forest Predict” and the “Random Forest Predict” that host two ML models. The two services emit predictions that are prepared to be consumed and sent into storage. The ML models above mentioned, are continuously updated by the batch layers: the “Isolation Forest Model“ service consumes historical data at a scheduled time to enrich the knowledge of the target “Isolation Forest Model” model, likewise, the Random Forest Model consumes the data labelled by the user in order to perform the same update process with the “Random Forest Predict” model.

The visualization/alerting tier runs Veesualive, a data visualization and exploration fork of Supersets by ENG with a financial domain-specific set of charts built-in. It is attached to the speed layer to receive fraud-

detection results and alerts in real-time and, at the same time, provides labelling features to support the batch supervised layer and let it work as expected.

7.3.3 Blueprint reference testbed implementation

All the components supporting pilot #10 are listed in the table below. This paragraph presents a description of the testbed and the installation process that allows to replicate it in any infrastructure.

GROUP	Prototype	Blueprint
Analytics	ALIDA Platform	ALIDA Platform
Injection	SEPA Transaction Generator	SEPA Transaction Generator
Injection	Kafka	Kafka
Presentation	Veesualive	Veesualive

Table 4 - Prototype vs Blueprint components

In the previous paragraph, we mentioned that pilot #10 runs in a hybrid testbed. It is worth noticing in Figure 74 that the hybrid testbed doesn't generate any inconvenience nor negative side-effects in the pilot execution: cross VPC connections are secured by proper authentication and authorization mechanisms and the data transfer relies on encrypted channels. VPC 1 contains the deployments of Veesualive, Kafka, and the SEPA transaction generator.

VPC 2 contains the ALIDA ecosystem through which pilot #10 crafts and runs the required workflows. Both the VPCs are compliant with the concepts of INFINITECH sandboxes, and both the environments are delivered through Kubernetes. There are two cross-connection points between them: one is for consuming user inputs during data exploration and for delivering output events through Veesualive. The other one is for consuming transaction events coming from the SEPA transaction generator that will be processed in the ALIDA ecosystem.

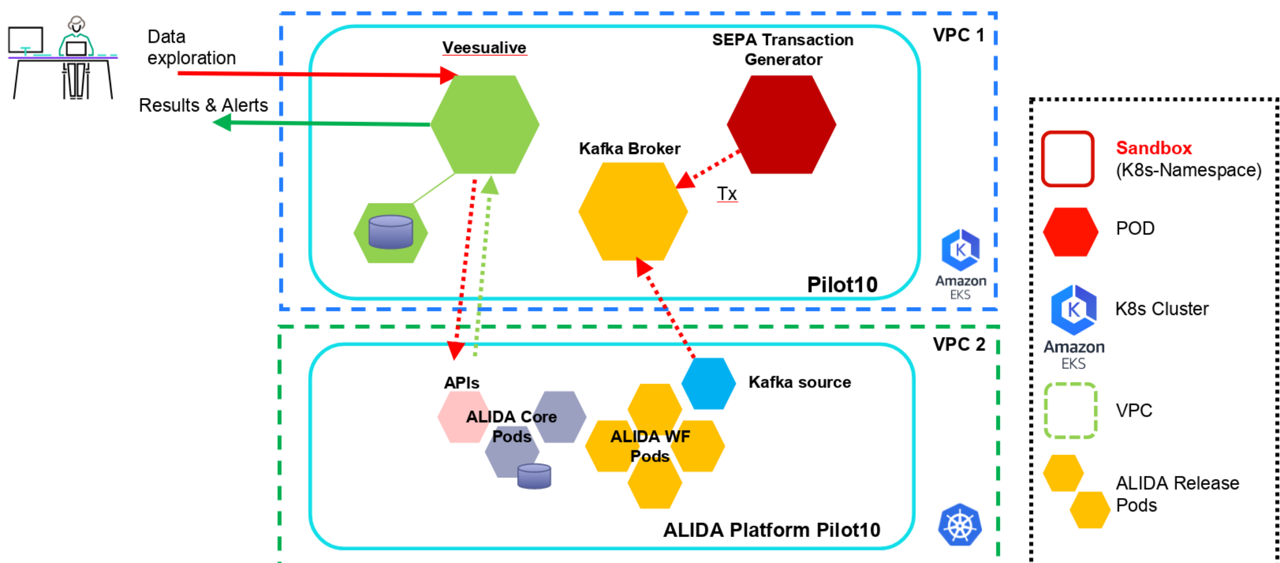


Figure 74 - Pilot 10 blueprint reference architecture

Reproducing the environment is straightforward, it is just a matter of replicating the following steps.

The testbed is implemented as a cloud-native solution that relies on kubernetes resources assets and helm charts. First thing to do is cloning Pilot #10 repository:

```
$ git clone https://gitlab.infinitech-h2020.eu/blueprint/pilot10.git
```

cd into the pilot10/kafka folder and start the kafka deployment process:

```
$ cd pilot10/kafka
```

Install the bitnami helm repository:

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
```

Deploy the release:

```
$ helm install —name kafka —namespace pilot10 bitnami/kafka -f values.yaml
```

Generate kafka keys and certificates. It is required to use the public FQDN of the service as Common Name when asked.

```
$ ./generate-jks.sh
```

Copy (or softlink) the kafka.keystore.jks file to kafka-n.keystore.jks where n is the number of the kafka broker replicas deployed. Then, create a secret from the certificates and keys:

```
$ kubectl create secret —namespace pilot10 generic kafka-jks —from-file=./kafka.truststore.jks —
from-file=./kafka-0.keystore.jks —from-file=./kafka-1.keystore.jks
```

Convert the jks keystore to pem using the script:

```
$ ./convert-jks.sh
```

The script will produce a keystore.pem file to grant kafka access to clients and 3rd party brokers and establish a secure connection. Then, cd to the parent folder to deploy the SEPA transaction generator:

```
$ cd ..
$ kubectl apply -f ./k8s-resources
```

This command will deploy the SEPA transaction generator and will create its own configmap.

For Veesualive installation, we can refer to the official Superset installation through the official Helm chart. The only thing to do is to override the official image with the INFINITECH’s Veesualive docker image in the values.yaml file.

Add the superset helm repository:

```
$ helm repo add superset https://apache.github.io/superset
```

Edit a values.yaml release customization file

```
image:
    repository: harbor.infinitech-h2020.eu/presentation/veesualive
    tag: latest
```

Then, install the veesualive release:

```
$ helm install veesualive --namespace pilot10 -f values.yaml superset/superset
```

Once every component is up and running we can add the analytics tier using the ALIDA platform. The speed layer is designed as represented in the below figure:

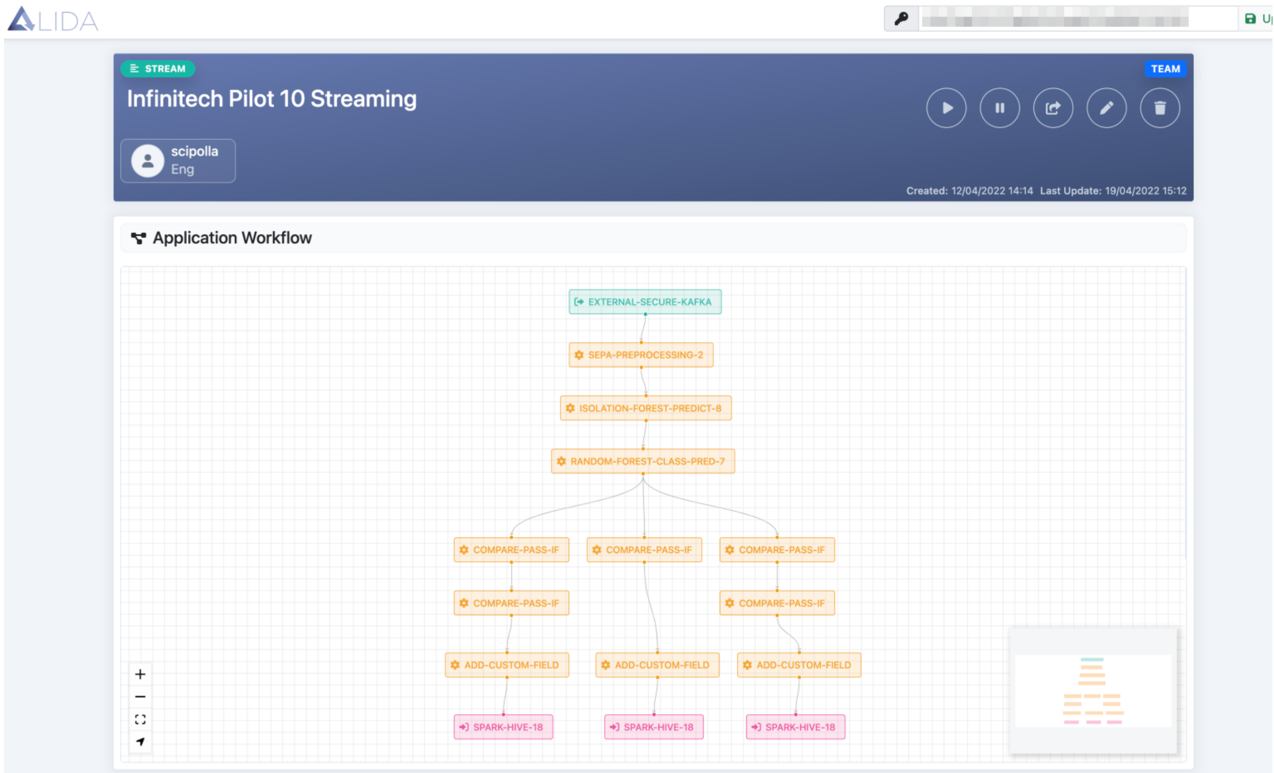


Figure 75 - Pilot #10 speed layer

The definition of the batch supervised layer is represented in the below figure:

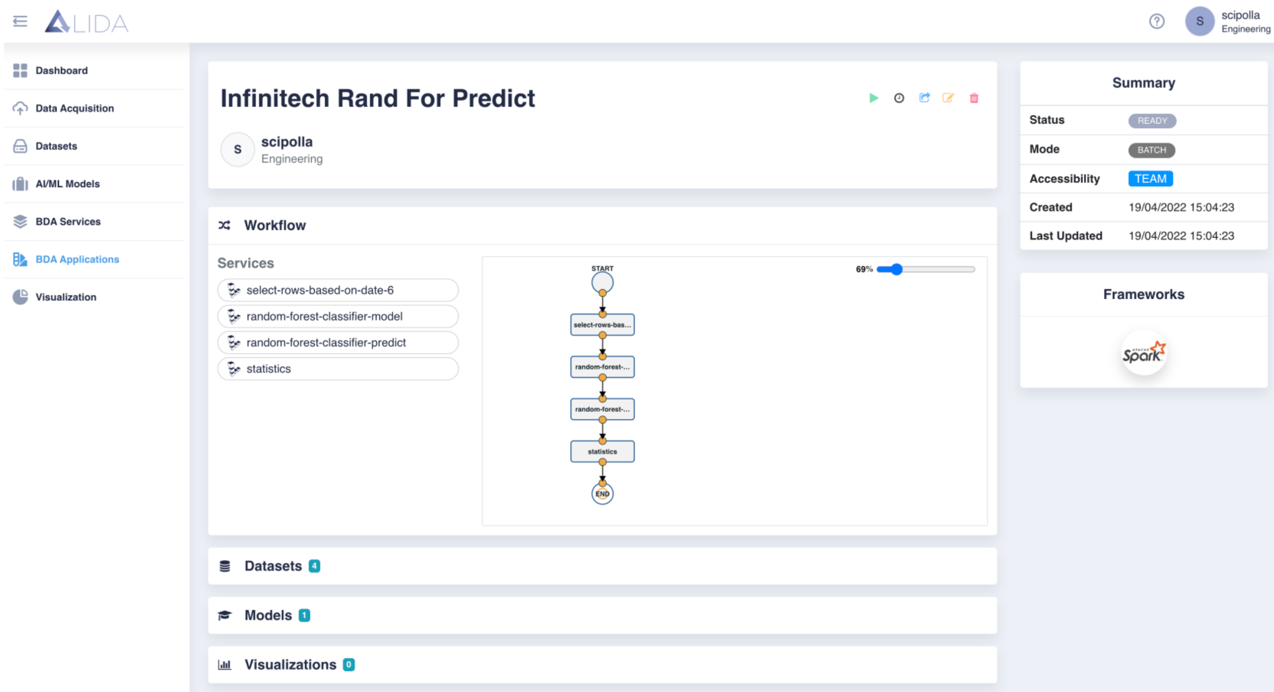


Figure 76 - Pilot #10 batch supervised layer

And the definition of the bath unsupervised layer is represented in the below figure:

Figure 77 - Pilot #10 batch unsupervised layer

The ALIDA workflows definition is made of building blocks that enable data source, data sink and data processing capabilities that can be combined to enable ETL and AI use cases. The definition and the execution of the ALIDA workflows completes reference testbed implementation for Pilot #10.

8 Conclusions

This document has reported the results of INFINITECH WP6 Tasks T6.2 "Mechanisms and Tools for Tailored Sandboxes Provision and Configuration" and T6.3 "Integrated Management of Testbeds' Datasets", achieved during the third and final phase of the project, i.e. the final specification of tools and techniques that will be leveraged to implement the testbeds and sandboxes concepts and the management of datasets within the INFINITECH project.

The document is the accompanying textual specification of the other major deliverable result: the third release of the proposed tools and techniques integrated and deployed into the INFINITECH blueprint reference testbed setup, designed and implemented upon two of the target INFINITECH infrastructures. The document and the developed INFINITECH blueprint reference testbed setup constitute the overall deliverable output.

The achieved results have provided key contributions for the fulfilment of the 2nd major WP6 milestone (MS14 – Second Version of ALL Sandboxes & Testbeds Available – achieved in M24 of the project) and also of the 3rd major WP6 milestone (MS17 – Final Version of ALL Sandboxes & Testbeds Available – foreseen for M33 of the project).

The work has been carried out in close cooperation and coordination with the other INFINITECH WP6 tasks and work packages 2-3-4-5 tasks and partners, taking into account and integrating the delivered results and concepts (e.g. the INFINITECH Reference Architecture proposed by WP2) in a coherent and uniform manner. Moreover, it has taken into account the feedback coming from the 2nd round of the INFINITECH work package dedicated to the Large Pilots Operations and Stakeholders' Evaluation of the proposed Financial and Insurance Services (WP7), and additionally will be one of the major drivers of the upcoming 3rd round of such WP7 Pilots.

The WP6 work on Tasks 6.2 and 6.3 will continue without any interruption also in the next period of the project (even after the release of this deliverable), evolving and enriching the proposed tools and techniques with additional capabilities and features that will take into account the latest evolutions of relevant technologies occurring during the related project timeframe. In particular we plan to apply the MLOps integration to some selected pilots: #7 ("Testbed for Avoiding Financial Crime", owned by the partner CAIXABANK) and the already mentioned pilot #10, which will act as blueprint pilots for this topic. Moreover, we'll provide continuous inputs and support to Tasks 6.4 and 6.5 and WP7 pilots activities, and applying the Blueprint guidelines implementation to possibly all the incumbent partners organizations (on private data centres/cloud providers accounts) and to all the FinTech/InsuranceTech partners organizations (in the shared data centre hosted by the partner NOVA), also taking into account feedbacks coming from all the pilots in the previous periods, as well as any relative corrective actions to be planned and deployed.

In the end, the development of the testbeds and the relative sandboxes life cycle will continue until the end of the project (so even afterwards the official end of Tasks 6.2 and 6.3) to provide dynamic upgrades in order to fulfill the INFINITECH project and the relevant pilots' Testbeds & Sandboxes needs.

Finally, with respect to the specific objectives and KPIs set for WP6 and related tasks in the project DoA, the Table 5 and Table 6 summarize how the work done in this deliverable addresses them.

Table 5 - Mapping of DoA/Tasks Objectives with Deliverable achievements

Objectives	Comment
OBJECTIVE 6 – TESTBEDS AND TAILORED SANDBOXES FOR BIGDATA & IOT EXPERIMENTATION,	The approach, mechanisms and tools described in D6.6 enable the definition and deployment of the testbeds and sandboxes envisioned by the objective.

TESTING, INNOVATION AND STANDARDIZATION IN FINANCE & INSURANCE	

Table 6 - Mapping of DoA/Tasks KPIs with Deliverable achievements

KPI	Comment
KPI 1 Testbeds to be Established >= 9 (>=8 in Incumbent organizations and >=1 (EU-wide) testbed for FinTech/InsuranceTech firms);	The approach, mechanisms and tools described in D6.6 incorporates by design the proper scalability features that enable the definition, deployment and provisioning of the target numbers of testbeds, sandboxes and datasets envisioned by the reported KPIs (KPI 1-2-3).
KPI 2 Distinct datasets (assets) to be shared through the testbeds >=25;	
KPI 3 Tailored Sandboxes to be developed & customized based on the project's tools >=14 (i.e. equal to the number of pilots);	

9 Appendix A: Literature

[1] [Online]. Available: Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance.

[2] [Online]. Available: Can be left void.

[3] K. Philippe, "“Architectural Blueprints - The “4+1” View Model of Software Architecture”," *IEEE Software,* vol. 12, pp. pp. 42-50, November 1995, pp. 42-50.

[4] R. M. M. M. M. A. Irakli Nadareishvili, Microservice Architecture: Aligning Principles, Practices, and Culture, O'Reilly Media, Inc., 2016.

[5] Google, “Containers,” Google, 2018. [Online]. Available: https://cloud.google.com/containers.

[6] Kubernetes, “What is kubernetes?,” Kubernetes, 5 August 2020. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[7] Docker, “What container,” 2020. [Online]. Available: https://www.docker.com/resources/what-container.

[8] AWS, “What is aws,” Amazon, 2020. [Online]. Available: https://aws.amazon.com/it/what-is-aws/.

[9] Wikipedia, “Bastion Host,” [Online]. Available: https://en.wikipedia.org/wiki/Bastion_host.

[10] HAproxy, “HAproxy,” HAproxy, [Online]. Available: http://www.haproxy.org/#desc.

[11] Hashicorp, “Introduction to Terraform,” Hashicorp, [Online]. Available: terraform.io/intro/index.html.

[12] Rancher, “Rancher overview,” SUSE, [Online]. Available: https://rancher.com/docs/rancher/v2.5/en/overview/architecture/.

[13] Gartner, “DevOps Gartener Glossary,” Gartner, 2020. [Online]. Available: https://www.gartner.com/en/information-technology/glossary/devops.

[14] “DevTest and DevOps for microservices,” Microsoft, 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/dev-test-microservice.

[15] AWS, “Amazon EKS,” Amazon, 2020. [Online]. Available: https://aws.amazon.com/eks/?nc1=h_ls.

[16] Git, “Git,” Software Freedom Conservancy, 2020. [Online]. Available: https://git-scm.com/about.

[17] GitLab, “Gitlab docs,” GitLab, 2020. [Online]. Available: https://docs.gitlab.com/charts/installation/deployment.html.

[18] Jenkins, “Jenkins,” [Online]. Available: https://jenkins.io/.

[19] S. Nexus, “Sonatype Nexus OSS,” [Online]. Available: https://www.sonatype.com/nexus-repository-oss.

[20] OpenLDAP, “OpenLDAP,” [Online]. Available: https://www.openldap.org/.

[21] HELM, “Quickstart,” HELM, 2020. [Online]. Available: https://helm.sh/docs/intro/quickstart/.

[22] Slack, “Slack features,” Microsoft, [Online]. Available: https://slack.com/intl/en-it/features.

[23] D. G. Neil MacDonald, “12 things to get right for successful devsecops,” Gartner, December 2019. [Online]. Available: https://www.gartner.com/en/documents/3978490/12-things-to-get-right-for-successful-devsecops.

[24] Sonarqube, “Sonarqube architecture,” SonarSource SA, 2020. [Online]. Available: https://docs.sonarqube.org/latest/architecture/architecture-integration/.

[25] Google, “MLOps: Continuous delivery and automation pipelines in machine learning,” Google, 2019. [Online]. Available: https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.

[26] Kubeflow, “About Kubeflow,” Kubeflow, 2020. [Online]. Available: https://www.kubeflow.org/docs/about/kubeflow/.

[27] A. W. Service, “Amazon Kubernetes Service,” [Online]. Available: https://aws.amazon.com/it/quickstart/architecture/amazon-eks/.

[28] L. Encrypt, “Let’s Encrypt,” [Online]. Available: https://letsencrypt.org/.

[29] Infinitech, “Gitlab Infinitech Main Page,” [Online]. Available: https://gitlab.infinitech-h2020.eu/,.

[30] Infinitech, “Gitlab Data Management,” [Online]. Available: https://gitlab.infinitech-h2020.eu/data-management.

[31] Infinitech, “Gitlab Analytics,” [Online]. Available: https://gitlab.infinitech-h2020.eu/analytics.

[32] Infinitech. [Online]. Available: https://gitlab.infinitech-h2020.eu/data-management/infinistore/-/blob/master/Dockerfile.

[33] Apache.org, “Maven,” [Online]. Available: https://maven.apache.org/.

[34] Infinitech, “Gitlab Personal Access Token,” [Online]. Available: https://gitlab.infinitech-h2020.eu/help/user/profile/personal_access_tokens.md.

[35] Infinitech, “Gitlab explore groups,” [Online]. Available: https://gitlab.infinitech-h2020.eu/explore/groups.

[36] Infinitech, “Pilot Ref on Gitlab,” [Online]. Available: https://gitlab.infinitech-h2020.eu/pilot-ref.

[37] Infinitech, “Gitlab-Infinitech-Blueprint,” [Online]. Available: https://gitlab.infinitech-h2020.eu/blueprint.

[38] Google, “Kubernetes-Kubectl,” [Online]. Available: https://kubernetes.io/docs/tasks/tools/install-kubectl/.

[39] “kfp kubeflow sdk,” [Online]. Available: https://www.kubeflow.org/docs/components/pipelines/sdk/sdk-overview/.

[40] “kfp.dsl package,” [Online]. Available: https://kubeflow-pipelines.readthedocs.io/en/latest/source/kfp.dsl.html.

[41] FBK, “ D5.9 – Library of ML/DL Algorithms - III,” p. 12.

[42] mlflow.org, “An open source platform for the machine learning lifecycle,” [Online]. Available: https://mlflow.org/.

[43] K. K. Framework. [Online]. Available: https://www.kubeflow.org/docs/external-add-ons/kserve/kserve/.

[44] K. M.-f. m. serving. [Online]. Available: https://www.kubeflow.org/docs/external-add-ons/serving/overview/.

[45] D. O. C. Provider. [Online]. Available: https://dexidp.io/.

[46] curl. [Online]. Available: https://curl.se/.

[47] G. M. D. K. J. S. M. F. D. K. Georgios Fatouros, "DeepVaR: a framework for portfolio risk assessment leveraging probabilistic deep neural networks," *Digital Finance,* pp. 1-28, 2022.

[48] G. e. a. Fatouros, "Addressing Risk Assessments in Real-Time for Forex Trading.," in *Big Data and Artificial Intelligence in Digital Finance: Increasing Personalization and Trust in Digital Finance using Big Data and AI*, Springer Nature, 2022, pp. 159-170.

[49] D. Araci, "Finbert: Financial sentiment analysis with pre-trained language models," *arXiv preprint arXiv:1908.10063,* 2019.

[51] Istio, "What is Istio," Istio, 2020. [Online]. Available: https://istio.io/latest/docs/concepts/what-is-istio/ .

[52] [Online]. Available: https://en.wikipedia.org/wiki/Bastion_host.