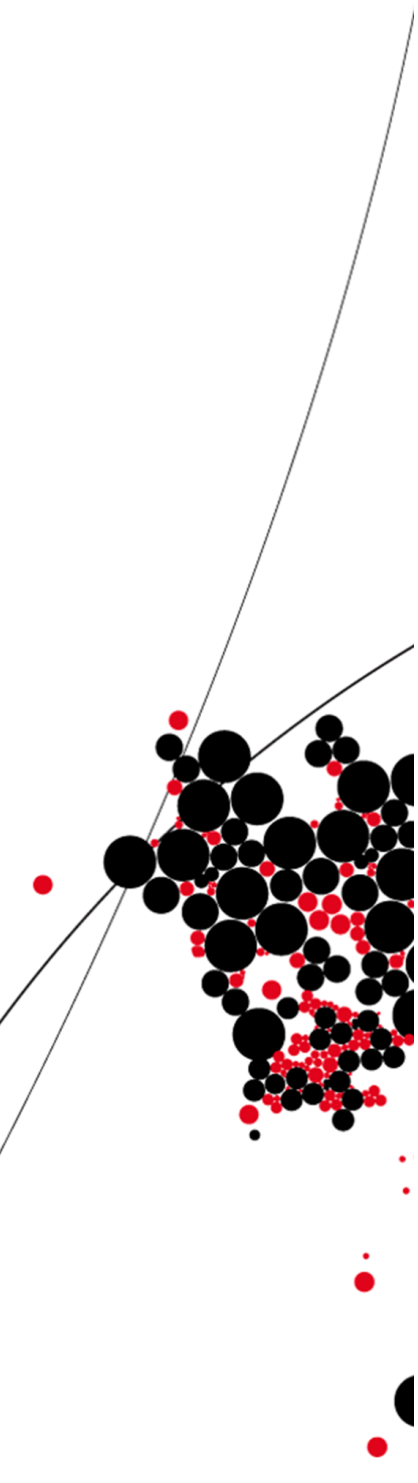
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science



A dynamic deployment framework for a Staging site in the Personal Health Train

Virginia Graciano Martinez

Master Thesis Report
March 2021

Supervisors:
dr. L. Ferreira Pires
dr. L.O. Bonino Da Silva Santos
dr. R. Guizzardi-Silva Souza

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my supervisors, dr. Luis Ferreira and dr. Luiz Bonino, for their academic guidance in the subject matter, all the discussions we had during our meetings and their continuous motivation, patience, and support during my thesis. Since the beginning of the thesis, they challenged me, helped me shape my research, and guided me with their precious questions, comments, and continuous feedback. Thanks as well to dr. Renata Guizzardi for participating in my thesis committee and her valuable feedback.

I would also like to thank friends and classmates with whom I spent these two years. It would take too long to mention you all, but I am sure you will recognize yourself as part of those. It was the first time I lived in a foreign country for a long time, and you all contributed to this unforgettable adventure, even in quarantine times.

From the bottom of my heart, thanks to my parents and my brothers. They supported me during this process, and I recognize I would have never come this far without you. To my friends in Mexico who were on the lookout and cheering me on all the time.

Last but not least, I wish to thank the Mexican National Council of Science and Technology (CONACyT) for the scholarship to pursue my studies.

Abstract

Healthcare data are absolutely necessary for increasing scientific and medical progress. However, patients' data are sensitive by nature, and it is not an easy task for healthcare organizations to share the data due to privacy, ethical and legal concerns. The Personal Health Train (PHT) is a novel approach that addresses the beforementioned problems by moving the analytical tasks towards the data, instead of moving the data to a central point. The PHT approach's rationale is that instead of requesting and receiving data, we expect to ask a question and receive an answer. PHT infrastructure is designed to deliver queries and algorithms that can be executed at healthcare institutes and provide just results to the person who asked. Consequently, sensitive data remain within the healthcare organization's control, and the end-user never has access to them, but he harnesses the data for analysis. However, some organizations may not have enough computing capacity to execute computation-intensive tasks, whereby a new computation-capable environment such as a cloud provider is required.

This research aimed to investigate how a new computation-capable environment can be deployed dynamically, respecting the PHT principles, complying with regulations, and integrating it with the current PHT architecture. To facilitate the analytics execution at the source, we proposed and designed an architecture for a Staging site that can be deployed dynamically in the cloud just when required. We employed Infrastructure as Code, APIs, and Event-based systems to achieve this. We implemented the architecture proposal using novel technologies and Amazon Web Services (AWS) and evaluated the proposal with a case study, analyzing datasets of ten thousand patients and one hundred thousand patients. The research showed that our work could alleviate the IT infrastructure constraints that the healthcare organizations can have, using the cloud and automation tools to ensure the PHT execution whilst respecting the PHT approach principles as much as possible. Although our design requires moving the data to the cloud, the data are still within the data source realm and control, keeping data privacy.

Keywords: *PHT, Cloud Computing, Distributed learning, Infrastructure as a Code, Cloud Federation,Hybrid infrastructure, Analytics.*

Contents

List of acronyms

AES-256	Advanced Encryption Standard
API	Application Programming Interface
AKS	Azure Kubernetes Service
AWS	Amazon Web Services
CapEx	Capital Expenditure
CRUD	Create, Read, Update, and Delete
DNS	Domain Name Server
DSL	Domain-Specific Language
EBS	Event-Based System
ECS	Elastic Container Service
EKS	Elastic Kubernetes Service
EC2	Elastic Compute Cloud
EHR	Electronic Health Records
EU	European Union
FAIR	Findable, Accessible, Interoperable, Reusable
FHIR	Fast Healthcare Interoperability Resources
GDPR	General Data Protection Rules
GKE	Google Kubernetes Engine
HCL	Hashicorp Configuration Language
HL7	Health Level 7

HTTP Hypertext Transfer Protocol

IAM Identity and Access Management

IaC Infrastructure as Code

IaaS Infrastructure as a Service

IT Information Technology

JSON JavaScript Object Notation

LOINC Logical Observation Identifiers Names and Codes

MFA Multi Factor Authentication

NIST National Institute of Standards and Technology's

ODM Operational Data Model

OPEX Operating Expenses

PaaS Platform as a Service

PHT Personal Health Train

REST Representational State Transfer

RQ research question

SaaS Software as a Service

SDK Software Development Kit

SNOMED CT Systematized Nomenclature of Medicine – Clinical Terms

SNS Simple Notification Service

S3 Simple Storage Service

VLP Varian Learning Portal

VPC Virtual Private Cloud

VPN Virtual Private Network

XML Extensible Markup Language

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Background

In recent years, vast amounts of structured and unstructured data have been generated by people and various institutions worldwide. This situation is known as *big data* and has become popular in almost every sector [11]. Historically, the healthcare industry has generated large amounts of data; while most data used to be stored in hard copy form, the tendency is toward digitizing these massive amounts of data nowadays [11]. These data require proper management and analysis to derive meaningful information. Therefore, scientists can use these data like never before, accelerating medical progress, improving a wide range of medical functions and providing healthcare delivery quality such as disease surveillance, clinical decision support, and population health management.

Traditional data analysis requires data sharing and centralization; however, this is not a realistic approach. From a technical perspective, it is unlikely that someone would collect all the relevant data. It would be expensive to host all the data and maintain the infrastructure. Besides, it would require too much time to move these potentially massive amounts of data to a central point to be processed. Moreover, sharing privacy-sensitive data out of the organizational boundaries is often not feasible for ethical and legal restrictions. Regulations such as the EU General Data Protection Rules (GDPR) impose strict requirements concerning the protection of personal data [12]. To comply with these regulations and harness the massive amount of data generated, advanced analytics and a distributed learning approach can be used.

Distributed learning, first introduced by Google in 2016 [13], analyzes distributed databases at different data sources location. Data source organizations control the entire execution and return just the results, without sharing information and keeping sensitive data privacy [14]. Therefore, it allows the use of data from several healthcare organizations while complying with the regulations. New approaches

based on distributed learning are emerging to analyze data in their original databases, limiting access to third parties. One of these approaches is the Personal Health Train (PHT), which aims to bring analytics to data rather than bringing data to the institutions that perform analytics. Scientists should be able to run analytics, learning from all the data, including sensitive data, without the data leaving organizational boundaries, preserving data privacy and control, whereby overcoming ethical and legal concerns [14].

The PHT provides an infrastructure to support distributed and federated solutions that utilize the data at the original location. Moreover, it is based on the Findable, Accessible, Interoperable, Reusable (FAIR) Principles, guaranteeing that the involved digital data are findable, accessible, interoperable and reusable [12]. The main design principle is to give data owners the authority to decide which data they want to share, and monitor their usage. Regarding privacy and security, PHT's main benefit is that data processing happens within the data owner's administrative realm. Besides, the researcher would be able to get valuable information from different sources without directly accessing the data. It targets maximal interoperability between diverse systems by focusing on machine-readable and interpretable data, metadata, workflows and services.

The PHT approach follows a train metaphor. The main concepts of this approach are [14]:

- **FAIR Data Stations:** These are data access points containing FAIR data, mainly healthcare organizations. They are conceptualized as the *Data Stations*, and they provide data sets, metadata, interaction mechanisms to these datasets and required computational power to execute analytics tasks.
- **Trains:** These are the components that interact with data at the Stations. These components carry the algorithms and data queries from the data consumer to the Data Station.
- **Track:** This is the metaphor used to describe all communication between the user interested to learn from data and the Data Stations.

1.2 Problem Statement

One of the PHT characteristics is that Trains visit the data at the Data Stations, i.e., algorithms carried by the Trains, move to the Data Stations where they are executed, and process the data available at the station. However, to perform the analysis at the source, computational resources are necessary. More specifically, a sandboxed environment should be available within the healthcare organization

where Trains are received and executed without interfering with the organization's regular processing needs. However, the computing resources required for processing the data may exceed the available healthcare IT infrastructure processing power. The healthcare providers' IT infrastructure has been designed to cover their regular processing needs; some of these stakeholders are currently using infrastructure that could be adjusted to comply with the PHT specifications and requirements. When they cannot adjust or provide the infrastructure required, they cannot support third-party algorithms' processing, missing opportunities to get valuable data.

Even though the PHT can provide scientific progress from which the healthcare sector can benefit, the Train execution cannot disrupt the hospitals' daily technological activities. In this scenario, Trains should be able to use other computing environments if new Data Stations can be dynamically staged in a computation-capable environment, such as a cloud provider, and keeping the data still within the data source realm and control. Only the required data are temporarily moved to the new station while preserving the sensitive data protected, and the algorithm is routed there for execution.

Cloud computing provides a flexible approach to how IT infrastructures, applications, and services are designed, deployed and delivered. It provides a scalable, on-demand, elastic provisioning and distributed computing infrastructure on a pay-per-use basis [15]. It facilitates computing deployment and orchestration and can be used just when required, resulting in a cheaper option than buying an entire big data solution. Therefore, Cloud computing should enable Trains to employ scalable resources dynamically. We will refer to Staging Data Station from now on as a temporary set up in a cloud environment that a Train can use to process data.

1.3 Research Questions

Based on our problem statement, we formulated the following research question (RQ):

> **RQ:** How to implement a Staging Data Station dynamically in the cloud while keeping private information protected?

To answer this question, we formulated the following sub-questions:

> *RQ1:* How can we keep private information protected when the PHT approach is run in a public cloud?

> *RQ2:* Which are the relevant technologies/tools to execute processes automatically and migrate data to the cloud dynamically?

RQ3: How does the modification of the PHT approach impact its principles?

SQ1: To what extent the integrity of PHT suffers from this modification?

1.4 Objectives

This thesis aims to design and implement a dynamic deployment framework for a Staging Data Station in the Cloud without compromising data and respecting the PHT approach's principles. To achieve this goal, we extended the PHT's current architecture, and we analyzed the processing of personal data regulations to define the requirements that our design should fulfil. Additionally, this system was validated according to several aspects that determine its quality, such as functionality, regulation compliance, performance and security.

1.5 Approach

We followed the *Design Science Research* methodology defined by Hevner et al. [16] and by Peffers et al. [17] to answer our research questions. The methodology revolves around a problem that can be solved by designing an innovative artifact. Following the design, the artifact follows two processes, *build and evaluate* to reflect on whether or not the problem has been solved. Based on the methodology presented in [17], our work follows five steps:

1. **Problem identification and motivation:** We performed a systematic literature review to identify the gaps in the existing PHT approach and indicated how cloud technology can improve some computing limitations.
2. **Defining the objectives of the solution:** We defined the goals of the desired artifact. These objectives can be translated into a list of functional and non-functional requirements.
3. **Design and development:** This step includes the artifact's design and implementation,so that a Staging Data Station can be executed dynamically, keeping the private data protected.
4. **Demonstration and Evaluation:** The methodology prescribes two different steps for demonstration and evaluation. However, in this project, these steps

are related and are combined in a single step. The artifact implementation is presented and demonstrates the suitability of the artifact to solve the problem.

5. **Communication:** We communicate the reference of our solution by discussing our findings, results, and contributions.

The build-and-evaluate loop, represented by steps three and four, is done several times iteratively to assess and refine the solution before the final artifact is generated. We executed the following tasks based on the steps mentioned above:

- Compiled a list of requirements based on regulations compliance.
- Defined a sequence of actions for the entire PHT execution process.
- Developed a preliminary design of the system based on the assessment of state-of-the-art approaches, the sequence of actions and the requirements. We discussed this design with supervisors to validate that the requirements are met.
- Developed a reference architecture based on the preliminary design and detailed discussion on the various building blocks that compose it.
- Selected the tools and technologies to be used in the development step.
- Implemented some fundamental building blocks to perform proof of concept evaluation.
- Evaluated regulations compliance, testing the proof of concept of the implemented building blocks. This evaluation was done through a case study.
- Discussed the solution's suitability, summarized the research, discussed limitations, future improvements and possible directions.

1.6 Contributions

The contributions of this thesis are:

i. Reviewing the current regulations in the processing of personal data in a public cloud.
ii. Proposing an architectural design for a Staging Data Station.
iii. Proposing an approach that allows an automated deployment of a Staging Data Station in the cloud while keeping the private data secure.
iv. Extending the current PHT architecture to support future growth.
v. Implementing a prototype that everyone can reuse.

1.7 Thesis Outline

This document is further structured as follows. Chapter 2 explains in detail the required background knowledge of the PHT architecture and technologies to be used and implemented. Chapter 3 gives a brief explanation of the related work. Chapter 4 identifies and analyze the functional and non-functional requirements for the design. Chapter 5 is the main body of the research and presents the proposed architecture as well as the integration into the current PHT architecture. Chapter 6 provides an implementation of the main building blocks of our solution to show the feasibility of our approach. Chapter 7 evaluates the solution through a case study. Finally, our findings and future work are discussed in Chapter 8.

Chapter 2

BACKGROUND

The PHT approach's core idea is the distributed learning concept introduced by Google in 2016 [13]. Distributed learning is a more sustainable medical data analysis approach and can unlock much more data without violating privacy. To this end, the PHT infrastructure was designed to deliver algorithms and questions that can be run at data source organizations. Consequently, such an infrastructure where analytics are run at the source requires appropriate definitions of where we can find data (Findable), how we can access these data (Accessible), how we can interpret the data (Interoperable), and how we can reuse the data (Reusable). Hence, the PHT infrastructure should rely on the FAIR principles [14]. Also, the current PHT architecture requires a sandboxed environment at the source to execute Trains. However, this Information Technology (IT) environment is limited; when the source does not have enough computational resources, Trains can use other computing environments if new Data Stations can be dynamically staged in a computation-capable environment. We use the word *dynamically* to refer to the creation and destruction of computing resources without human intervention. Nevertheless, shifting to another IT environment ushers in new challenges, such as configure and set up the infrastructure, transfer the data, configure the network, which computing instances to use and how many resources, besides security and legal compliance.

According to [5], the latest technologies and platforms such as virtualization, cloud, containers, automation and Infrastructure as Code can simplify the IT environment deployment. Therefore, adopting cloud and automation tools in the PHT architecture can lower barriers for making infrastructure deployments dynamic.

This chapter provides background information on the current PHT architecture and the FAIR principles that it follows. It presents the concepts and technologies

that can enable a dynamic deployment for a PHT Staging site, essential to understanding the design and implementation explained in the rest of this thesis.

2.1 FAIR Principles

The FAIR principles were first published in 2016 [4]. They consist of guidelines for best data management practices that aim to make data FAIR: Findable, Accessible, Interoperable and Reusable. These guiding principles seek to enhance the machines and individuals' ability to find and use data automatically [12]. These principles define characteristics that current data resources, vocabularies, tools, and infrastructures should publish to facilitate discovery and reuse by third-parties.

Although FAIR principles were initially designed for data management, they can be applied to any digital object to create an integrated domain to support reusability [4]. The PHT approach encourages improving the reuse of data by sharing analytics, interacting with the data and completing its task without giving the end-user access. Within the PHT, the FAIR principles apply to both the Train and Station, keeping in mind that the objective is to enhance distributed data's reusability with distributed analytics.

The medical data stored in healthcare organizations can be explored for both clinical and research purposes. It is essential to manage the data in a manner that there is always a single meaning of the data no matter where and by whom the data are being used. The PHT does not dictate any specific standard or technology for data, and instead, it only requires publishing single choices as metadata. The PHT focuses on making data, processes, tasks, and algorithms FAIR [14]. Thus, it enables data providers and data users to match FAIR data to FAIR analytics, and entitles them to make informed decisions about participating in specific applications.

FAIR principles also become relevant for analytics tasks. Interoperability and accessibility can be provided by applying FAIR principles to the analytics tasks and system components that interact with these tasks. FAIR guidelines are described in Table 2.1. In the following section, we explain the PHT architecture and how it adheres to the FAIR principles.

Table 2.1: FAIR Guidelines [4].

Principle	Guidelines
Findable	• Meta(data) are assigned globally unique and persistent identifiers. • Data are described with rich metadata. • Metadata include the identifier of the data they describe. • Meta(data) are registered in a searchable resource.
Accessible	• Meta(data) are retrievable by their identifier using a standardized communication protocol. • The protocol is open, free and universally implementable. • The protocol allows for authentication and authorization when required.
Interoperable	• Meta(data) use a formal, accessible, shared and broadly applicable language for knowledge representation. • Meta(data) use vocabularies that follow FAIR principles. • Meta(data) include qualified references to other meta(data)
Reusable	• Meta(data) are richly described with a plurality of accurate and relevant attributes. • Meta(data) are associated with detailed provenance. • Meta(data) meet domain-relevant community standards.

2.2 PHT Overview and Roles

The Personal Health Train proposes an approach that encompasses technological and legal aspects of sensitive data reuse. When data sharing is not feasible, using distributed analytics on distributed data becomes an appropriate solution. This approach requires discovering, exchanging and executing analytics tasks with minimal human intervention. In [1], [12], [18], authors have proposed an architectural design used on recent proofs of concepts to achieve these requirements. Figure 2.1 depicts the high-level PHT architecture.

As shown in Figure 2.1, several entities are involved in the entire PHT workflow. Therefore, the architecture defines four main roles representing various stakeholders and their responsibilities, which are represented by the blue elements of Figure 2.1.

i. **Curator:** It has authority over the data. This role can be played by the data

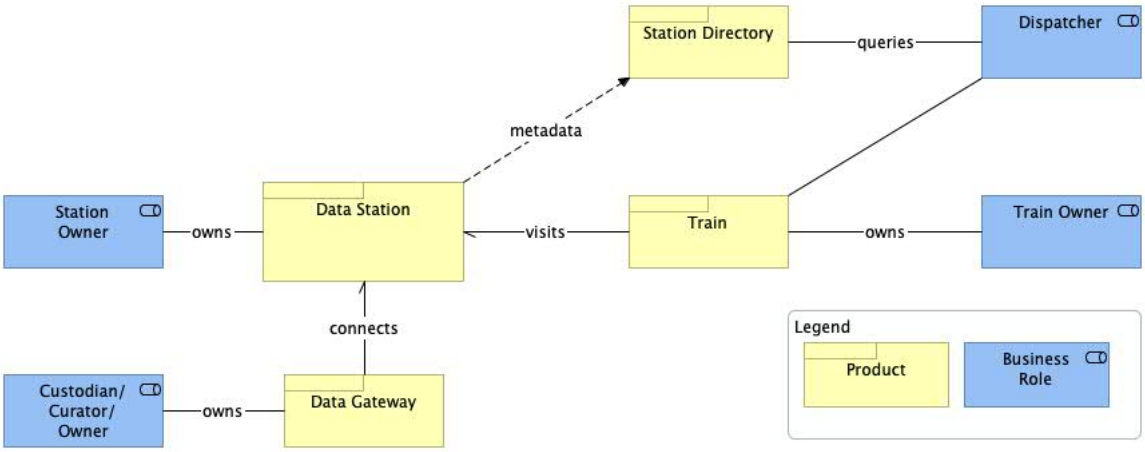


Figure 2.1: High-level PHT architecture [1].

owner or by any other actor who controls the data [1]. It provides enough metadata to be ***findable (F)*** and published by Station Registry as a curator.
Responsibilities:

- Publish data on Data Station.
- Control authority over the data.
- Grant or deny access to the data when requested.

ii. **Station Owner:** It is the entity responsible for the operations of a Data Station.
Responsibilities:

- Run and maintain the Station available to its users, both the Train Owner and the Curator.

iii. **Train Owner:** It is the entity responsible for its Trains, i.e., a scientific entity. A given Train interacts with data in a Data Station on behalf of the Train Owner.
Responsibilities:

- Build and deploy Trains in the system.
- Provide the required Train metadata.

iv. **Dispatcher:** The entity responsible for dispatching Trains on behalf of their Train Owners to the appropriate Data Stations. The Dispatcher interacts with the Station Directory to discover which Stations provide access to the required data and orchestrates the Trains to the target Stations.
Responsibilities:

- Discover Data Stations by interacting with the Station Directory.

- Check Train's metadata.
- Route Trains to the appropriate Data Stations.
- Orchestrate the dispatch of Trains to multiple Data Stations when necessary.

2.3 PHT Core Elements

The core elements are managed by the aforementioned roles and are represented by the yellow elements in Figure 2.1. They were designed as FAIR objects to allow full integration, and their interfaces are data access points.

2.3.1 Data Stations

These are data access points containing FAIR data related to healthcare organizations. They are conceptualized as the Data Stations, and they provide data sets, metadata, interaction mechanisms to these data sets, and the required computational power to execute analytics tasks.
Responsibilities:

- Provide metadata about itself and the datasets accessible through it.
- Provide access to its datasets.
- Allow eligible Trains to interact with the data.
- Allow eligible Data Curators to publish their metadata and data in the Station.
- Allow Data Gateways to establish secure connections between the Gateway and the Station.

Figure 2.2 depicts the Data Station architecture in terms of its interface and internal structure. A Data Station presents its functionality through its interface. The interface exposes three groups of services:

i. **Metadata Services:** The Data Station's Metadata Component provides access to the Data Station's metadata and metadata of all datasets made available through this Station. External applications willing to retrieve metadata from the Data Station invoke these metadata Services to accomplish the task.

ii. **Data Interaction Services:** They are supported by the Data Interaction Component, which provides the functionality for external users to access the data

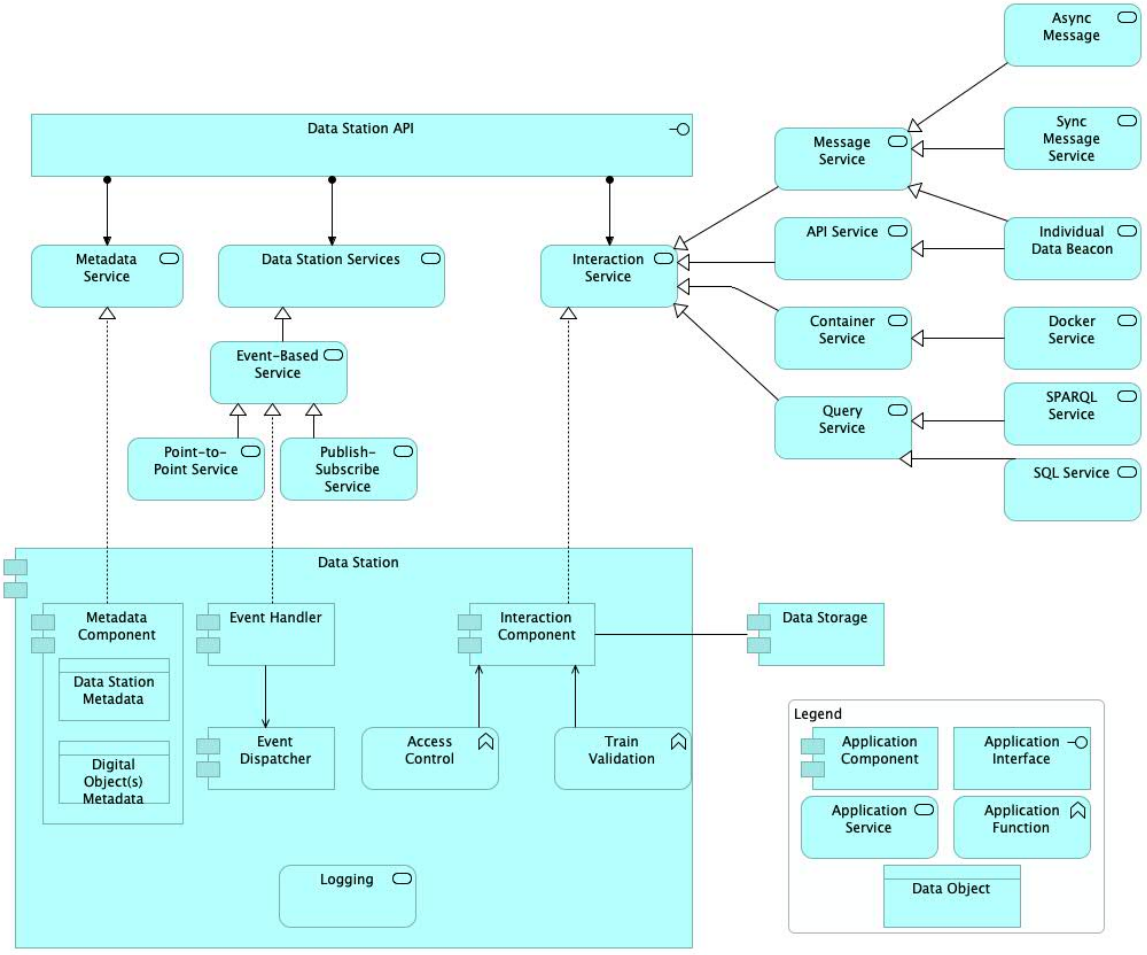


Figure 2.2: Data Station architecture [1].

made available through the Station. Data access can happen through messages, Application Programming Interface (API) calls, container execution and queries. For each of the cases, the Data Interaction Service is specialized in a Message Service, API Service, Container Service and Query Service. More than one Interaction Service can be provided in a given Data Station specified in the Data Station metadata record set in Metadata Services.

The Data Interaction Component also performs validation on the incoming Trains, via the Train Validation function, to assess that they behave according to the Station's requirements and the Train description defined in the Train's metadata. Whenever the data required by a Train has access restriction, the Data Interaction Component also enforces the required access control.

The data made available through a Data Station can be either external or internal to the Station. On that basis, the Station can include a Data Storage component in its deployment or secure access to an external Data Storage Component. A hybrid setup, in which some data is kept in an internal data

storage and other data in an external data storage, is also possible. These options improve flexibility and scalability.

iii. **Data Station Services:** Data Stations provides event-based services. For instance, a Station Directory can subscribe to be notified when the Station updates its metadata, keeping information up-to-date.

The **Logging Service** logs the interactions of a Data Station. The logs allow for traceability and also make the Station auditable.

Each Station must be identified with a persistent identifier and registered in the Station Registry with its metadata. It is ***findable (F)*** by publishing the metadata about the data sets repositories and the computational environment. It is ***accessible (A)*** by following standardized communication protocols to discover and receive Trains. Trains can be delivered with open and universal implementable protocols that follow standard authentication and authorization procedures. Data access control is under the Stations' exclusive control, but results are communicated with open protocols.

Data and Metadata Layer

Healthcare data can be structured, such as laboratory results, or unstructured, such as clinical notes. For completeness, the scientific knowledge that can be established needs both structured and unstructured data to be harnessed. Trains portability and interoperability rely on the runtime environment, and that can process any data at the different Data Stations. Data interoperability relies on healthcare data exchange standards, such as Health Level 7 (HL7) versions 2 and 3, Operational Data Model (ODM), OpenEHR, more recently, architects improved HL7 by basing it on RESTful principles, and released a new specification called Fast Healthcare Interoperability Resources (FHIR) [14]. They defined a specific information format, while the information structure remains the same. Hence, structuring data at the source can be an option to provide interoperability between the Trains and data at Data Stations. Moreover, it is essential to keep semantic consistency, using a vocabulary system based on terminology and coding standards such as Systematized Nomenclature of Medicine – Clinical Terms (SNOMED CT) and Logical Observation Identifiers Names and Codes (LOINC) [19]. The FHIR resources and semantics terminology makes data syntactically and semantically ***interoperable (I).***

2.3.2 Station Directory

The Station Directory is the metadata registry for all Stations in the system, including the datasets' metadata accessible through each Data Station. It allows users to discover data, and from which Station the data can be accessed. It is the vital component to make data and Data Stations findable (F).
Responsibilities:

- Harvest and index metadata from all Data Stations in the system.
- Allow users to search for data based on the indexed metadata.

The Station Directory is a specialization of the Data Station, which means it implements the same architecture and behaviour as the Data Station plus its specific features. The data available in the Station Directory are the other Data Stations' metadata; therefore, it implements API and Query Services for other client applications to access its data.

It also implements Publish-Subscribe services to allow client applications to subscribe for specified changes in the harvested metadata. It can also subscribe to these services to be notified by Data Stations whenever an updated their metadata.

2.3.3 Data Gateway

The Data Gateway aggregates access to all data under one Data Curator's authority that is available through different Data Stations. Like the Station Directory, the Data Gateway is a specialization of a Data Station; it presents the same functionality as the Data Station extended with its specific functionality. In the healthcare industry, patient's data may be stored in different Stations for different reasons, such as suitability according to specialization or gathering by various hospitals or medical devices. Nevertheless, the Data Curator must have access to its data, regardless of location, and this is done through the Gateway.
Responsibilities:

- Provide access to all data under the authority of a Data Curator distributed in different Data Stations.
- Allow the control and curation of the data by the Data Curator.

2.3.4 Train

The Train interacts with the data in a Data Station. A Train is dispatched to a Station by its Dispatcher and can be implemented using different technologies such as queries, API calls, containers. Trains also have their metadata describing who is responsible for the Train, what type of data the Train requires, what it does with the data, and for which purposes.
Responsibilities:

- Identify itself to the Data Station and request access to data.
- Access and process data in the Data Station.
- Return to its Dispatcher with the results.

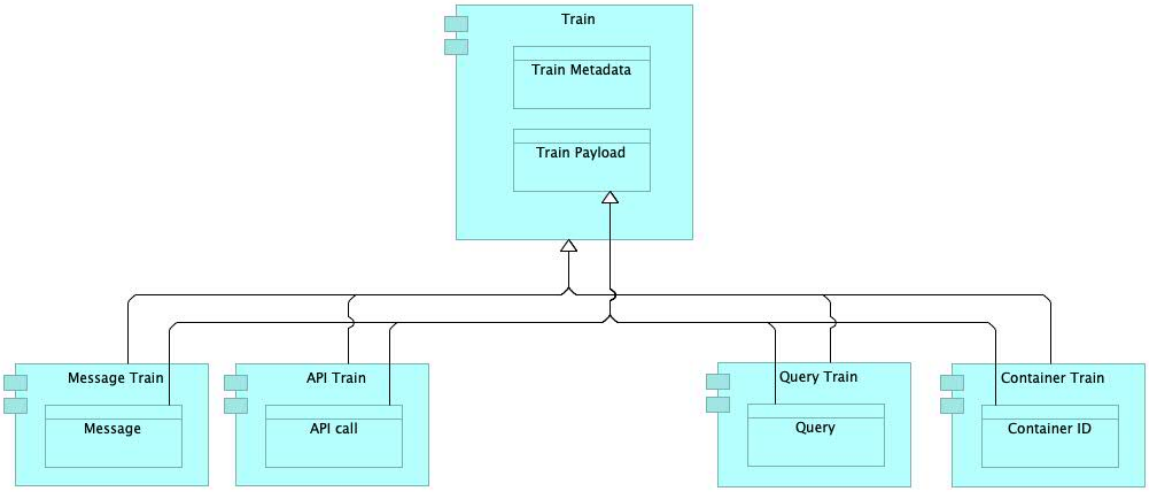


Figure 2.3: Train architecture [1].

The Train acts on behalf of its Train Owner and access and process data in Data Stations. As explained in the Data Station Section 2.3.1, different interactions forms are supported, ranging from messages to container execution, including API calls and data queries. Figure 2.3 depicts the Train architecture. As it is shown, a Train is composed of two main elements: the Train metadata and the Train Payload.

The *metadata* contains information such as who is responsible for the Train, the type of the Train (message, API call, query or container), the data it requires, and its purpose. Train *Payload* depends on the kind of Train. For instance, container trains have the identifier of the container image as their payload. API trains use API calls; the message trains payload is the message itself. Finally, query trains have the query as their payload.

Trains follow the four dimensions of FAIR principles. They are ***findable***, as they are persistently and uniquely identified and are registered in a Train Registry as digital objects. They are ***accessible*** through open, interoperable and free implementable protocols allowing authorization and authentication. They are ***interoperable*** since every Train described by metadata uses a formal, accessible, shared and broadly applicable language. The metadata defines both the content and provenance of the analytics task.They are also self-contained and can be executed in multiple locations, and as a consequence, they are ***Reusable***. Moreover, Trains can be stored in Train registries allowing other end-users to reuse Trains anytime.

2.4 PHT and Cloud Computing

The National Institute of Standards and Technology's (NIST) provides the most widely used definition of Cloud computing [20]. It states :

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

Cloud computing led to a flexible approach to how IT infrastructures, applications, and services are designed, deployed and delivered. It provides a scalable, on-demand, elastic provisioning and distributed computing infrastructure on a pay-per-use basis [15]. A cloud delivery model serves an explicit, pre-packaged combination of IT resources offered by Cloud providers. There are three models on which Cloud services are offered [21]:

- **Infrastructure as a Service (IaaS):** It delivers on-demand components for building IT infrastructures such as storage, virtual servers, and networks.
- **Platform as a Service (PaaS):** It provides ready-to-use environments for applications hosted on the cloud.
- **Software as a Service (SaaS):** It offers applications and services on-demand, accessible through the web.

Some of the benefits of cloud computing that contribute to the PHT approach are:

- On-demand access to pay-as-you-go computing resources on a short-term basis, such as vCPU per hour, and the ability to release the resources that are no longer required.
- The perception of having unlimited computing resources available on-demand and anytime, dealing with a lack of on-premises resources.
- An abstraction of the infrastructure so that applications are not locked into devices and can be moved if required.

2.5 Infrastructure as Code

Cloud computing offers computing, network, and storage resources through services that abstract the underlying hardware. Currently, a range of tools exists that handles the infrastructure provisioning and use scripts to define the hardware's final state to be provisioned in the cloud. These scripts are part of what is called Infrastructure as Code (IaC). It is an approach to infrastructure automation based on practices from software development. A formal definition for IaC given by Kief Morris [5] is:

> "*Infrastructure as Code is an approach to managing IT infrastructure for the age of the cloud, microservices and continuous delivery that is based on practices from software development* ".

The key elements of Infrastructure as Code [5] are the dynamic infrastructure platform, infrastructure definition tools, and the application programming interfaces. These elements are described in more detail below. Furthermore, to have a deployment with no human intervention, some event-based service or action is required, which is also explained further in this section.

2.5.1 Dynamic Infrastructure Platform

A dynamic infrastructure platform is a system that grants computing resources, principally servers, storage, and networking to be programmatically allocated and

managed. The most known dynamic infrastructures are IaaS Cloud services. Table 2.2 lists several dynamic infrastructure platforms with the most representative examples.

Table 2.2: Dynamic Infrastructure platforms [5].

Type of Platform	Providers
Public IaaS Cloud	AWS, Azure, Digital Ocean, Google Cloud
Private IaaS Cloud	OpenStack, CloudStack, VMware vCloud
Bare-metal Cloud	Foreman, Cobbler

The dynamic infrastructure platform includes private and public cloud services. However, for the PHT Staging site, we will focus exclusively on the Public IaaS Cloud option because a vendor runs the infrastructure, and we do not need to worry about Capital Expenditure (CapEx) but only in Operating Expenses (OPEX). Additionally, certain features have to be considered in IaC. The platform needs to be:

- *Programmable:* the platform must be programmable through scripts or any piece of software. These must be able to interact with the cloud platform.
- *On-demand:* the platform yields users with the capability to create and destroy resources instantly in a matter of minutes or seconds.
- *Self-Service:* the platform allows changing and customizing resources based on user needs.

The three main building blocks provided by a dynamic platform are: compute, networking and storage. There is a broader service offer, but nearly all of them are variations of the main building blocks. Table 2.3 shows the leading cloud provider offers, according to Gartner [22], for each building block that is relevant to the PHT Staging site deployment.

2.5.2 Application Programming Interfaces (API)

APIs are programming interfaces that allow software components to be used by other software components. They have emerged from the need to exchange information with data providers, and they are the building blocks that allow interoperability for platforms on the web. APIs usually exhibit an interface through

Table 2.3: Cloud providers' offers [6], [7], [8].

	AWS	**Azure**	**Google Cloud**
Compute	Elastic Compute Cloud (EC2)	Azure Batch, Azure Dedicated Host	Compute Engine
Storage	Simple Storage Service (S3)	Azure Blob Storage	Google Storage
Containerization	Elastic Container Service (ECS), Elastic Kubernetes Service (EKS)	Azure Kubernetes Service (AKS)	Google Kubernetes Engine (GKE)
Networking	Virtual Private Cloud (VPC), Route 53	Azure Virtual Network, Azure Domain Name Server (DNS)	VPC, Cloud DNS

an Hypertext Transfer Protocol (HTTP) web server; so that clients make HTTP requests for data, and the web server replies. These responses can be JSON or XML documents [23]. The API is implemented in programming languages like Python, NodeJS, Java or Ruby.

Representational State Transfer (REST)-based APIs are currently the most popular. REST is about resources, can be identified, addressed, and named on the web. REST APIs expose data as resources and use standard HTTP verbs to perform Create, Read, Update, and Delete (CRUD) actions on these resources [24].

2.5.3 Event-based Systems

Proper management of resources is crucial when they are on the public cloud for security and cost optimization purposes. Monitoring services and event-based systems can facilitate the (healthcare) organization to manage resources properly in a cost-effective manner and automate the workflow execution to deploy the Staging station without human intervention.

To achieve dynamic PHT execution in the Staging Data Station, a conventional request-reply model is not suitable. In contrast, an Event-Based System (EBS) is

more suitable since it communicates by generating and receiving event notifications, where an *event* is an occurrence that represents a state change. The affected component announces a *notification* that describes an event. A *publish-subscribe* service mediates between the EBS components, and send notifications from publishers to subscribers that have registered their interest in these events with a previously issued subscription. One of the main benefits is that the components are decoupled, and as a consequence, they can be scalable and deployed independently [25]. Moreover, the EBS components exchange messages using different transportation channels, depending on the message communication pattern employed. The principal message communication pattern employed is [26]:

Asynchronous Communication: It is a message communication where the publisher sends a notification message to the subscriber and proceeds without waiting for the response. This pattern uses transportation channels such as topics, queues to create loose coupling between publishers and subscribers. This pattern can benefit current cloud architectures, whereby it is considered a must in the architecture.

2.5.4 Infrastructure definition tools

An infrastructure definition tool specifies what infrastructure resources a user wants to implement and how they should be configured. There are two types of definition tools for setting up the infrastructure: ***provisioning*** tools and ***configuration*** tools. *Provisioning* tools specify and allocate the desired resources, and use the dynamic infrastructure platform to provision the resources that are specified. *Configuration* tools configure and manage the resources provisioned by the provisioning tools. We explore further the provisioning tools but not the configuration tools as our proposal will not require set up the resources created by the provisioning tool.

Provisioning Tools

Provision is the first phase to deploy any functional infrastructure and more specifically, the PHT Staging Station. The infrastructure is defined in configuration definition files, which can be seen as text files written in standard formats like JavaScript Object Notation (JSON), Extensible Markup Language (XML) and YAML, or in a proprietary Domain-Specific Language (DSL). The tool uses these files to provision, modify, or remove components of the infrastructure. It does this by interacting with the dynamic infrastructure platform API. Some examples of these tools are vendor-specific, such as CloudFormation for AWS, Azure Resource Manager

for Microsoft and Cloud Deployment Manager for Google Cloud. Whereas, some tools support multiple cloud providers, such as Terraform and Cloudify. Table 2.4 compares the different IaC provisioning tools.

Table 2.4: Provisioning Tools [9].

	Supported Platforms	**Configuration Language**
Terraform	AWS, Azure, Google Cloud, Digital Ocean, OpenStack, vSphere, vCloud and more.	DSL
CloudFormation	AWS	JSON, YAML
Azure Resource Manager	Azure	JSON
Cloud Deployment Manager	Google Cloud	YAML
Heat	AWS, OpenStack	HOT, YAML
Cloudify	AWS, Azure, Google Cloud, OpenStack, vSphere and vCloud	TOSCA, YAML

Terraform, Heat and Cloudify support multiple cloud vendors, including vendors that do not have proprietary solutions, enabling several resources from different cloud providers to be combined. A drawback of Terraform is that it employs a proprietary language for the configuration files, whilst the rest use standard languages such as JSON and YAML. In [9], the authors evaluated the different available IaC tools listed in Table 2.4. Among the tools analyzed, only Terraform and Cloudify comply with the requirements for working with leading cloud providers. The study concluded that Terraform is a much better tool by far than Cloudify because Cloudify consumes more computing resources and takes longer to receive results than Terraform. Besides, even though Cloudify supports multiple vendors, it does not support many services used in the market. Terraform is also more mature, and it is very well documented. For that reason, it is the tool that we will study and use for the architecture implementation.

Terraform

Terraform is an open-source infrastructure automation tool developed by HashiCorp [27]. It supports over 500 different providers, including public cloud vendors such as AWS, Azure, Google Cloud and Digital Ocean, as well as private Clouds like

OpenStack and VMware. Terraform allows the infrastructure to be described through definition files written in its proprietary DSL called Hashicorp Configuration Language (HCL). The language is declarative, which means the user specifies how the infrastructure should look like, and it does not worry about the state of the system [2]. Figure 2.4 depicts a declarative Terraform definition file.

```
variable "environment" {
  type = "string"
}

variable "subnets" {
  type = "map"

  default = {
    qa = "subnet-12345678"
    stage = "subnet-abcdabcd"
    prod = "subnet-a1b2c3d4"
  }
}

resource "aws_instance" "web" {
  instance_type = "t2.micro"
  ami = "ami-87654321"
  subnet_id = "${lookup(var.subnets, var.environment)}"
}
```

Figure 2.4: Terraform definition file [2].

Terraform also allows execution plans to be produced that detail the steps followed to reach the infrastructure desired state. The execution plan first gives an overview of what happens by the time it is called, and then Terraform sets up the infrastructure by running this plan. Moreover, Terraform stores the state of the managed infrastructure in a file called "terraform.tfstate", which is used to create execution plans and make the necessary infrastructure changes when required. After each operation is performed, Terraform refreshes the state to match the actual real-time infrastructure. Figure 2.5 illustrates the Terraform workflow; it shows that from the definition file, Terraform interacts with the cloud vendor and provisions the resources and services specified in the files.

Figure 2.5: Terraform workflow.

Chapter 3

RELATED WORK

This chapter presents and discusses the current proofs of concepts to achieve a distributed learning approach. There are few papers around this topic due to the novelty of the approach; however, they provide the learning from previous related work that fosters developing a robust PHT architecture. The following sections discusses the different proof of concepts implemented so far and a brief discussion about limitations.

3.1 Varian Medical Systems

The Varian Learning Portal (VLP) by Varian Medical Systems is the most popular technology used by most current PHT implementations [28], [29], [3], [30], [19].Varian Medical Systems is an American manufacturer of oncology software and treatments, for that reason the papers using this technology have done research only around cancer.

The VLP is a cloud-based system that implements user, data station, and project management. It is composed of two elements, a **master** and a **learning** connector. A learning connector is installed at each Data Station to connect the VLP master to a local Data Station. The end-user uploads his application to the VLP web portal, which can be done in MATLAB, MathWorks, Natick, MA, USA. VLP and Data Stations communicate via file-based, asynchronous messaging. The iterative execution of applications and communication between them is known as a learning run, and each Data Station can accept or deny a learning run. Figure 3.1 depicts the VLP structure.

In [28], authors demonstrated that it is feasible to use the distributed learning

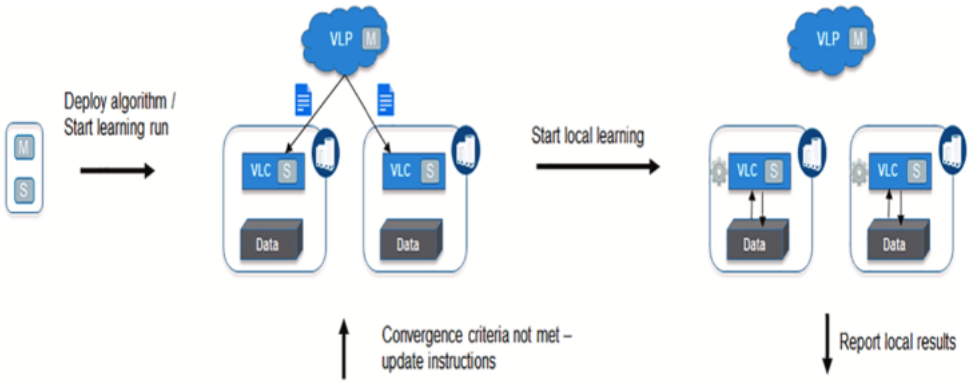


Figure 3.1: VLP [3]

approach to train a Bayesian network model on patient data coming from several medical institutions. Data were extracted from the local data sources in each hospital and then mapped to codes. Besides, in the Varian learning portal, the researcher uploaded his or her Bayesian network model application for learning. The Varian learning portal transmits the model application and validation results between the central location and the hospitals. In [29], authors built-in MATLAB R2018 a logistic regression model to predict post-treatment two-year survival. The VLP connector was installed in 8 healthcare institutions. In [19], the authors used the VLP to run a study to develop a radiomic signature; the authors pointed out the preference for VLP because it had already implemented the essential technical overhead(logging, messaging and Internet security).

In these proofs of concepts, authors have concentrated on the algorithms to evaluate and train the data distributed geographically. They tried to demonstrate that the results are just as accurate as when data are centralized. Therefore, they have harnessed the PHT approach using VLP technology, but they do not improve the PHT architecture. Also, the solution is sponsored by VLP as it is a solution that usually incurs a cost. Moreover, the applications are not reusable by everyone, and only the users from each project can see what they have done. For case studies beyond cancer, this solution will not be suitable; then, other options must be explored or developed.

Other technologies similar to the VLP are dataSHIELD, and ppDLI [19], which follow a similar client-server schema. An agent is installed in the data station and connects to a central server. However, VLP entirely focuses on the healthcare sector; in contrast, dataSHIELD and ppDLI have not been used in the healthcare sector so far, but they are primarily used in other sectors. All of them have a cost.

3.2 Open-source Technology

In [12], and [18], the authors leveraged containerization technologies for sending applications to the Data Stations, more precisely Docker containers. The former created a Train containing an FHIR query and an algorithm to calculate summary statistics, then wrapped them as a Docker image and submitted it to a private Docker registry. The latter initially used a phenotype design web client to create Docker images containing query, the metadata and the script and then submitted them to a public Docker registry.

In [12], the authors used a central server with a master algorithm. The master algorithm coordinates the task among two Data Stations and aggregates the results obtained from each Data Station before sharing the final work. The Docker image includes an algorithm to query data that loads the data locally and temporarily inside the Docker container. The data station's infrastructure component controls the Train's execution at the data station. They retrieved data about patients born before *01-01-1990*, diagnosed with **hypertension** and fetched **age** and **body mass index**. They proved that existing healthcare standards and containerization technology could be leveraged for achieving distributed data analysis. Besides, their goal is to train a machine learning algorithm later on, but first, they needed to prove that open-source tools were feasible to build a PHT infrastructure.

In [18], the authors used a simple Spring Boot Framework to do the routing task; it knows all current Data Stations in advanced and then tags the Docker images when they arrive. Docker tag designates the station that should pull the image. The Data Stations run a cron job to scan the Docker namespace continuously and identify via the tag system if there is a Train that they have to run. Besides, they used the Portus authorization service and Docker registry frontend to manage the station's authentication. They harnessed the Portus interface to monitor the Train images that are running. At each Data Station, the Train is run inside a Docker container. After the computation is finalized, results are pushed back to the Docker registry. The Docker image version is updated, and the station invokes the resulting API and posts the results to the end-user.

In these papers, the authors focused more on the architecture and infrastructure than on the algorithm. They used Docker container as leading technology to wrap the algorithm and run the Train in each Data Station. Besides, other frameworks and open-source tools were used to build different parts of the architecture. These options do not consider the lack of resources and what to do in that case. In these implementations, the Trains can be reused as they use the Docker registry

and allow more interoperability with any environment than the VLP technology.

None of the previous proof of concepts has implemented the entire PHT architecture; however, they have demonstrated it is a feasible approach, and simple queries or sophisticated machine learning algorithms can be run on top of the infrastructure. We are more aligned with the work done by [12], and [18] as we consider we can contribute with further improvements in the architecture and implementations than the work done by the VLP, which is a vendor solution. We used the work done by [12], and [18] as base for our workflow development and the main proposal of this research.

Chapter 4

REQUIREMENTS ANALYSIS

Cloud computing can enable Trains to employ scalable resources dynamically when the main Data Station does not have enough computing resources. This chapter specifies the requirements needed to deploy a Staging Data Station in the cloud that fulfills the PHT desired functionality.

4.1 Motivation

Shifting to the cloud ushers in several new challenges, such as transfer the data, the network required to migrate, which computing instances to use and how many resources, besides security and legal compliance. On [31], the authors mentioned the life cycle of big data and its technological challenges. Data storage, data transmission, data management, data processing, data visualization and data integration are the phases reviewed. The PHT should cope with each of these phases; however, special attention should be given to data transmission, data management, and data processing when migrating to cloud environments. Besides, it is essential to consider how to deploy the infrastructure dynamically.

The first step towards designing and implementing a fully Staging Data Station system is to define a list of requirements that the Station should fulfil to function properly and support all the mandatory use cases. A PHT general workflow and sequence of actions were defined to identify the Staging Data Station requirements. In [32], [33], the authors explained that the requirements for a system fall into the following two categories:

- **Functional requirements:** They describe the business capabilities that the system must supply and its behaviour at run-time.

- **Non-functional requirements:** They describe the "Quality Attributes" that the system must meet in delivering functional requirements.

We want to implement an architecture to support the PHT approach effectively. In order to share privacy-sensitive data out of the organizational boundaries, regulations compliance needs to be analyzed. Consequently, we consider regulation rules as part of non-functional requirements. We present the non-functional and functional requirements to design, implement and validate a compliant PHT Staging Data Station.

4.2 Non-Functional Requirements

The GDPR is the European regulation on data protection and privacy for all European Union (EU) citizens that became enforceable on May 25, 2018 [34]. The aims of the GDPR are mainly to give individuals more control over their data and harmonize the regulatory environment throughout the EU.

The GDPR applies to organizations established in the EU that process personal data and organizations outside the EU that process EU residents' data. ***Personal data*** is any information relating to an identified or identifiable natural person [34]. The GDPR defines two major actors that play a role in processing data subjects' data, namely the data controller and data processor.

The **data controller** is entitled to decide the purpose of the processing, which data should be processed, how long, who can access them, and what security measures need to be taken [34]. The data controller must exercise control over the data processor, and it is responsible for the processing, including legal liability.

The **data processor** is the authority that processes personal data on behalf of the controller. The processor's existence depends on a decision taken by the controller, who can decide to process data within the organization or to delegate the activity to an external processor [34]. Processors must record all processing activities to demonstrate their compliance, implement organizational and technical measures to secure the processing mechanism and notify data breaches to the controller.

GDPR Articles

This section presents the most important articles that must be considered to use the cloud while complying with data regulations. These articles are regarded as a basis to define our non-functional requirements.

Article 3 defines a territorial scope; the regulation applies to personal processing data belonging to EU citizens regardless of where the processing occurs. Personal data can be moved to third countries, i.e., countries outside the EU, only after an accurate evaluation of the safeguards.

Article 5 introduces the principle of storage limitation. It states that data should be stored as briefly as possible subject to the processing purposes for which it has been collected.

Article 25 states that the controller shall implement appropriate technical and organizational measures to ensure that only personal data necessary for each specific purpose of the processing are processed.

Under ***Article 32***, controllers and processors must implement appropriate technical and organizational measures to ensure a security level appropriate to the risk. The GDPR provides suggestions for what types of security actions might be required, including:

- The encryption of personal data and pseudonymization.
- Organizations must safeguard against unauthorized access.
- The ability to ensure the ongoing integrity, confidentiality, availability and resilience of processing systems and services.

We identified the GDPR as the primary regulation entity. Data controller and data processor are two essential roles identified by the GDPR regarding personal data processing accountability. Both are obligated to implement appropriate security measures and demonstrate that processing operations are compliant with the regulation's principles. In our scenario, it is possible to identify both roles clearly and objectively. GDPR prescribes that a data controller must be identified, as the Data Station is the primary organization in the PHT architecture; it plays this role. Data processor should be recognized as well. Unlike the data controller role, a data processor might not exist at all. Indeed, its existence depends on the data controller's decision to outsource personal data processing; for instance, when the Staging Data Station is required; therefore, the cloud plays the data processor role. Having clear roles helps to build a compliance architecture and to look for the most

suitable cloud provider. We identified the most crucial articles, 3, 5, 25 and 32, that have to be considered when the data processor is run in a cloud environment.

4.2.1 Quality Attributes

A quality attribute is a measurable characteristic of a system used to indicate how well the system satisfies stakeholders' needs. The quality attributes are not standing alone. Mainly those are tightly coupled with required functionality and other architectural constraints. In [32], the authors proposed using the Utility Tree technique from the Architecture Tradeoff Analysis Method (ATAM) to define and gather these quality attributes. The ATAM utility tree uses the following structure:

- **Highest level**: Quality Attribute requirement (security, performance, cost-effectiveness configurability).
  - Next level: Quality Attribute requirement refinements. For instance, "latency" is one of the refinements of "performance".
- **Lowest level**: Architecture scenarios—at least one architecture scenario per Quality Attribute refinement. The architecture scenarios include the following three attributes:
  - **Stimulus**: It describes what a user of the system would do to initiate the architecture scenario.
  - **Response**: It describes how the system would be expected to respond to the stimulus.
  - **Measurement**: It quantifies the stimulus's response.

We will use the quality attributes for designing the architecture but primarily for evaluation purposes explained in Chapter 7. The features can have different meanings depending on the context; for that reason, we defined the attributes derived from ISO 25010 [35], which is the primary standard for evaluating a software system's quality. The Figure 4.1 shows the utility tree with the quality attributes chosen.

4.3 Functional Requirements

The Staging Data Station has to be deployed automatically and transparently to the end-user. It has to be invoked just when required and automatically deployed.

Figure 4.1: Quality Attributes.

When created, the Staging Data Station has to exhibit the necessary behaviour to allow a given Train's execution. Following the data life cycle and the literature review [1], [12], [31], Figure 4.2 depicts the PHT workflow diagram that includes the delegation of processing to a cloud computing environment.

4.4 Sequence of Actions

Based on Figure 4.2, we assigned each action of the workflow to a phase. PHT execution is composed of four phases, which are depicted in Figure 4.3.

i. **TRAIN CREATION**

→ A Train Owner creates a Train.
→ The Train Owner creates an image of the Train.
→ The Train image is stored in a repository.

ii. **STATION DISCOVERY**

→ Dispatcher queries the Station Directory to discover which Stations contain the required data.
→ The Dispatcher routes the Train to the target Stations.

iii. **TRAIN NEGOTIATION**

→ Train identifies itself to the Data Station and requests access to data.
→ Data Station applies access control to the incoming Train to assess that it behaves according to the Station's requirements.
→ Data Station assesses Trains metadata to evaluate which computing resources are required.
→ Data Station allows eligible Trains to access the data.

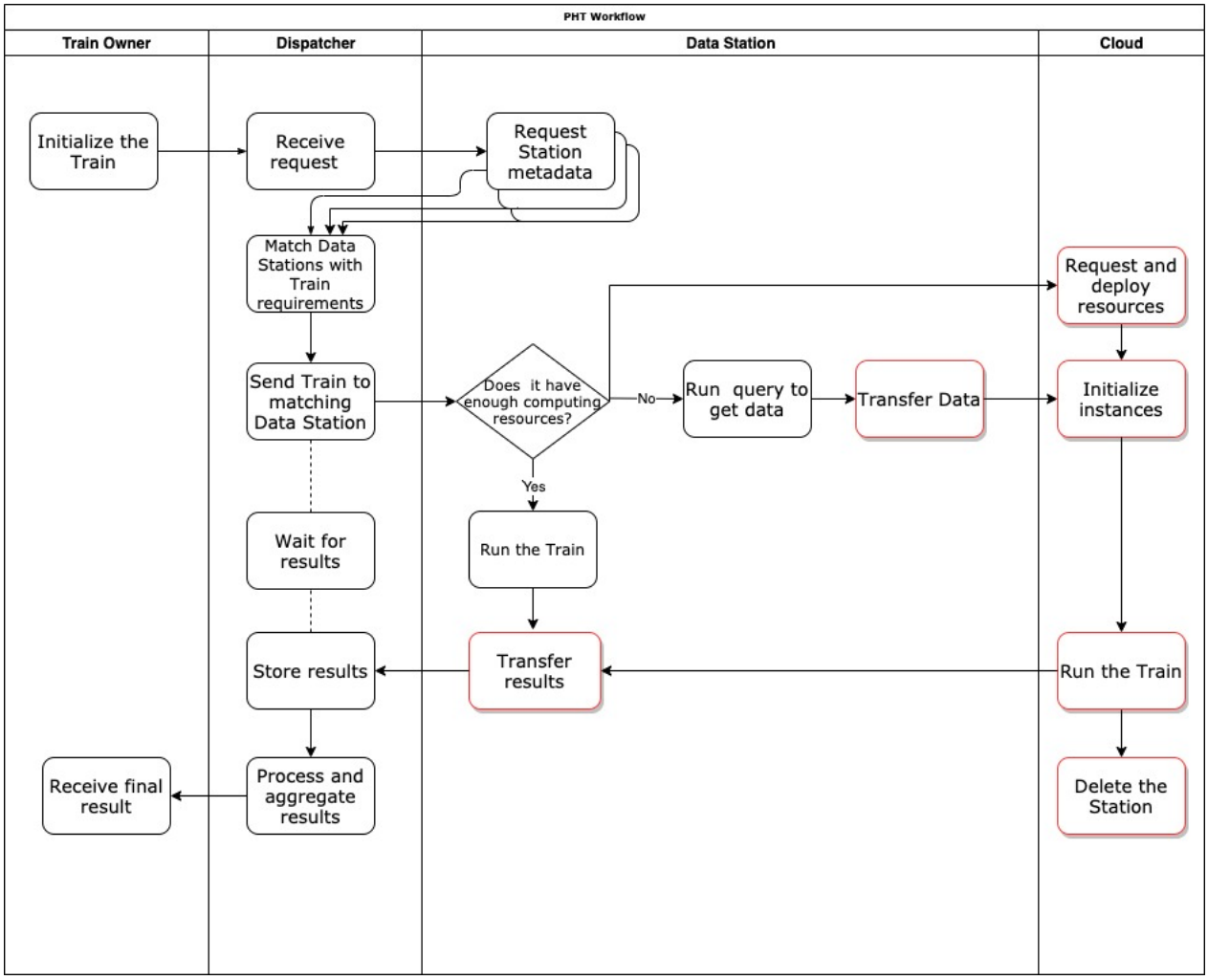


Figure 4.2: Complete PHT Workflow.

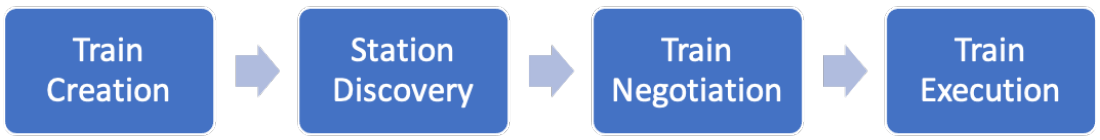


Figure 4.3: Sequence of Actions

iv. **TRAIN EXECUTION**

→ Data Station executes the Train in a secure environment.

→ If the Data Station does not have enough computing resources, then it creates a Staging Data Station in the cloud:

i Data Station executes the Train to get only the necessary data.

ii Create a storage bucket in the cloud.

iii Move data from step "i" to the cloud.
iv Create a Computing Engine VM in the Cloud.
v The new station executes the Train.
vi Copy the output of the execution to the cloud storage bucket.
vii Send output file to the main Data Station.
viii Delete output files after they are no longer required in the Cloud.
ix Delete the VM or compute resources created in the Cloud.

→ Main Data Station receives the output file from the cloud.
→ The main Data Station inspects the output file to validate that no privacy-sensitive information is left behind.
→ The Main Data Station sends the result to the Dispatcher.
→ The Dispatcher possibly aggregates results from different Data Stations.
→ The Dispatcher sends the final product to the Train Owner.

Table 4.1 depicts the resulting list of requirements; these requirements are used later to choose the appropriate technologies and tools to design and implement a Staging system prototype.

Table 4.1: List of Requirements.

Requirements	Type
1.Cloud provider must comply with articles 3, 5, 25 and 32 of GDPR.	Non-functional
2.Only the necessary information must be moved to the cloud.	Functional & Non-functional
3.The Infrastructure layer of the system must be deployed automatically.	Functional
4.The Network layer of the system must be deployed automatically.	Functional
5.The software components of the system must be implemented using open source technologies.	Functional
6.The system has to keep data privacy and comply with legal concerns.	Non-functional
7.The system must encrypt data in transit and at Rest.	Functional & Non-functional
8.The system must establish access control.	Functional & Non-functional
9.Cloud provider must have several sites in the EU.	Functional
10.The system must notify the main Station when the Train execution is done.	Functional
11.The output files have to be downloaded automatically by the main Data Station.	Functional
12.The Staging Data Station has to be destroyed once it is not required.	Functional & Non-functional
13.The system should be scalable.	Functional
14.The system must provide event-based services.	Functional

Chapter 5

DESIGN

In previous chapters, we focused on building a solid understanding of the current PHT architecture, Infrastructure as code approach and cloud technologies, and on defining the requirements for our work. We concluded that adopting cloud and automation tools lowers barriers for dynamically deploying infrastructures, facilitating the development of an architecture that complies with the PHT approach.

This chapter describes our Staging Data Station architecture, which leverages the cloud and Infrastructure as Code technologies. We begin by introducing the general elements of our system. We continue with our architectural design based on the PHT architecture presented in Chapter 2 and the new proposed building blocks.

5.1 Overview

The purpose of this section is to combine the PHT architecture and the requirements identified in Chapters 2 and 4 respectively, to design and deploy a Staging Data Station. An essential aspect of the Staging Data Station deployment is the automation of data processing. The aim is to provide autonomous data processing: instead of having human operators mediate between the workflow steps, these steps should initiate automatically. We identified four general elements that we believe are crucial to design and deploy a Staging Data Station: a dynamic platform, provisioning tools, APIs and event-based services.

- **Dynamic Platform**: The main building block of the Infrastructure. It is responsible for setting up the infrastructure resources of the Staging system. The dynamic platform should be compatible with an IaC provisioning tool to

achieve dynamic deployment and system management. The dynamic platform can be a cloud platform or a software tool that can provide computing and storage resources. It has to comply with GDPR and all the requirements listed in Table 4.1.

- **Provisioning Tool**:The user describes the desired infrastructure resources in the definition files in the IaC tool that depend on the selected tool. The provisioning tool translates these files via API calls into instructions for a dynamic platform that sets up the requested resources. The provisioning tool has to support the selected dynamic platform.
- **APIs**: Trains and tools should communicate and exchange information with the underlying infrastructure resources through REST APIs.
- **Event-driven services**: Infrastructure components communicate by generating and receiving event notifications. An event is any occurrence of interest, such as a state change in some resource. The affected component issues a notification describing the observed event, and a target resource triggers an action based on that. Event-based services can work with resources within or outside the dynamic platform, facilitating the separation between communication and computation.

5.2 Architectural Design

Depending on the compute resources available at the main Data Station and the Train requirements, the Station platform would run the Train locally or use a Staging platform in the cloud. We begin by introducing the various building blocks of our solution, and we proceed by combining them into the current architecture.

5.2.1 Data Station

The Infrastructure required to run a Train can be either internal or external to the Station. If necessary the Station can use a Staging site, having secure access to an external infrastructure such as a cloud provider. Consequently, this will allow us to have a hybrid setup, using only the Staging component or using some local resources and others from the staging component when required. These options allow scalability and flexibility. The Figure 5.1 depicts the proposed Data Station architecture.

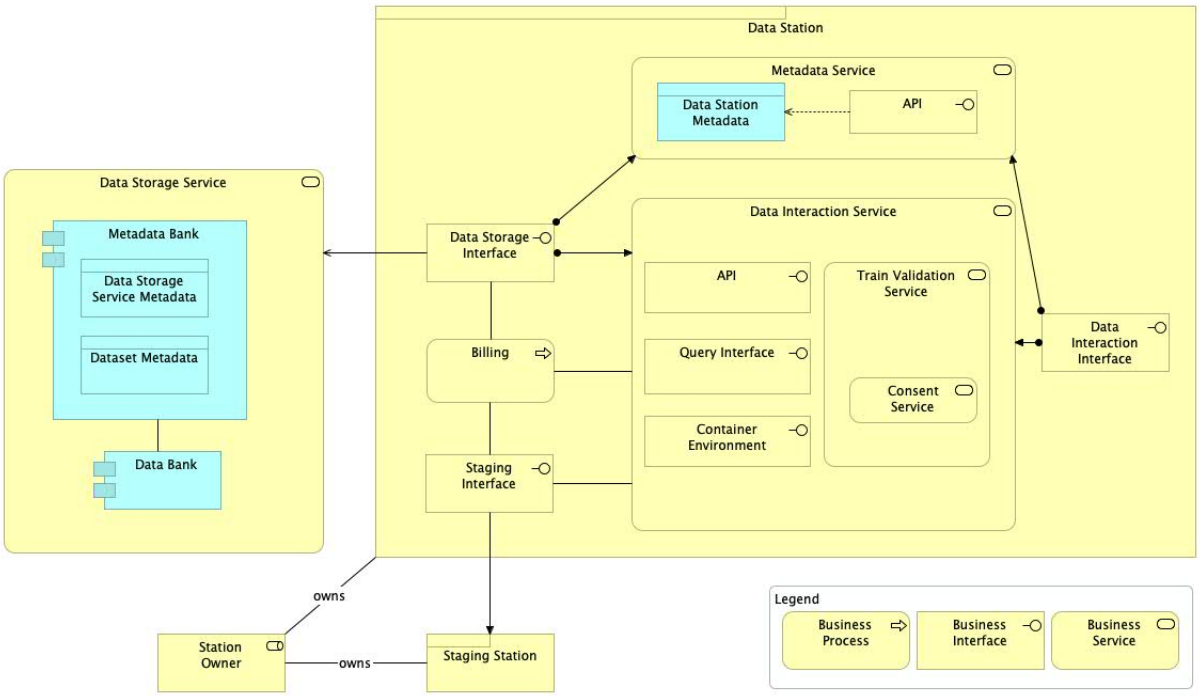


Figure 5.1: Data Station Architecture.

Interaction Component

The interaction component at the main Data Station provides functionality for external clients to access the data. However, API communication modifications have to be done, and the Staging component has been added to the architecture.

In the current architecture, the existing Station platform executes two phases, namely **Describe** and **Run**, as explained in Chapter 2. This solution is suitable for a single Station with the computing resources to run a Train. In this scenario, if the Data Station does not comply with the Train requirements, it does not execute the Train. Therefore, we had to define additional phases to launch the Staging Data Station. Figure 5.2 depicts the proposed communication system between the Station Platform and Train.

Proposed communication structure: In the **Describe phase**, the Station platform runs the arriving Train image. A **Train Description** is returned from the Train to the platform as a *response*. If the Data Station does not have the required resources, it enters the **Query** phase and runs the arriving Train. A query is executed to get the needed data. The resulting query returns a *query response*, which declares that the required data are ready for being processed in a Staging Data Station. The platform will then enter the **Stage** phase through an API call, launching the Staging Data Station. If the Station platform has the resources to run the Train,

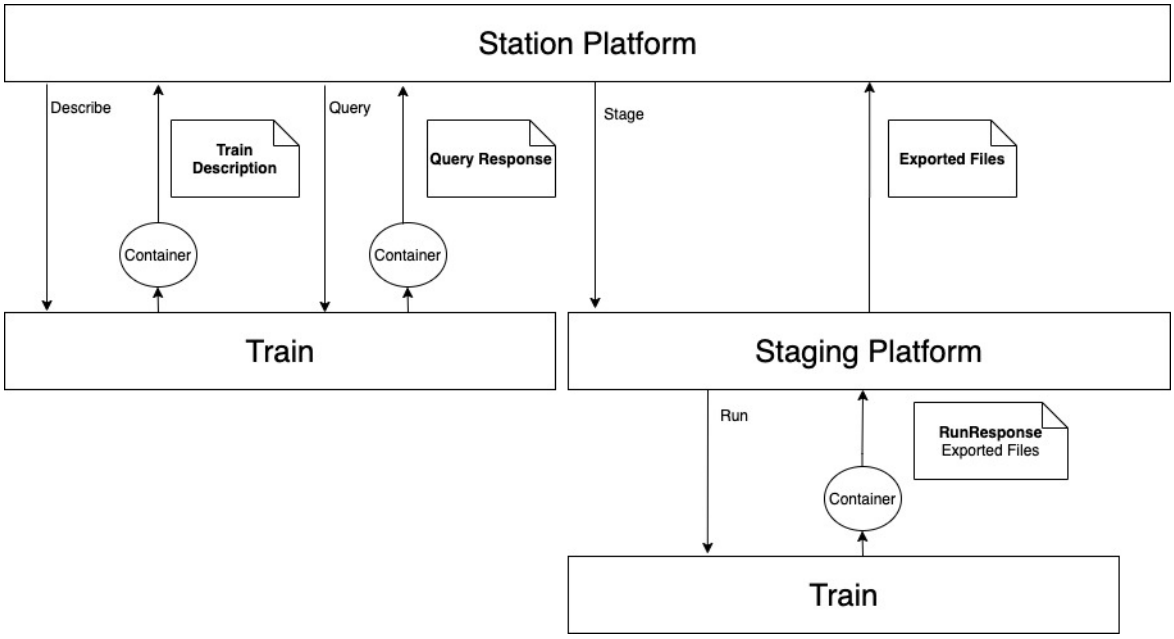


Figure 5.2: Proposed communication structure.

it works as it does currently, by performing Describe and Run phases.

The proposed architecture defines the following new **new responsibilities**:

- Deploy the Staging Data Station.
- Provide the required datasets.
- Expose the services required to use an external execution environment.
- Establish secure connections between the Staging platform and the Station.
- Allow the Staging platform to establish communication between the Staging Station and the main Data Station.
- Perform billing on data storage, data access, computing and data staging and computing capacity.

As depicted in Figure 5.1, we included the Staging Station, which communicates with the leading Data Station through an API, represented by the Staging Interface. This interface defines the Stage phase depicted in Figure 5.2. We assume that the role responsible for this new component is the *Station Owner*, as the staging process should be transparent for the Train Owner. However, the use of an external platform can incur an extra cost; so for that reason, a Billing component is added to the architecture and can be used for the convenience of the Station Owner.

5.2.2 Staging Data Station

The Staging Data Station is an extension of the Data Station, and it behaves like the original Data Station with some additional features. Figure 5.3 depicts the Staging Data Station architecture, which is composed of the following services:

- **Access Control:** It allows access control to the cloud environment; and only the Data Station owner should have access to this service. However, if needed, more users can be added and get specific permissions and policies to execute particular tasks. To provide a better secure environment, the Identity Management Service can provide roles to be used to allow or deny access to the other resources in the Cloud.
- **Data Storage:** It stores the input and output data. The input data are selected at the healthcare Data Station and moved by the Staging Station to the cloud. The output data is the result of the Train execution given in the Run Response, and it is sent to the Healthcare Data Station as exported files.
- **Event-based services:** The Staging Platform provides Event-based services to automate the execution steps. For instance, when the data are entirely moved to the cloud, the Staging Station notifies the computing instance that the Train can be run. Besides, the Station platform may subscribe to be notified when the Train execution finishes, to harvest the output files promptly. Events and trigger actions are achieved through the event handler and event dispatcher.
  - **Event Handler:** It listens to the events issued by infrastructure components to create rules and trigger actions.
  - **Event Dispatcher:** It executes the actions provided by the Event Handler.
- **Logging:** This service logs the data access interactions. This service enables regulatory compliance and security, but also operational tasks. It identifies which actions were taken by whom, what resources were acted upon, who accessed which data when the event occurred, and other details to help analyze and respond to an activity. This is a requirement for GDPR compliance, but also it is used to communicate with the Event-based services to launch tasks when an event occurs.

Our architecture proposal aims at deploying a Staging Data Station in the cloud

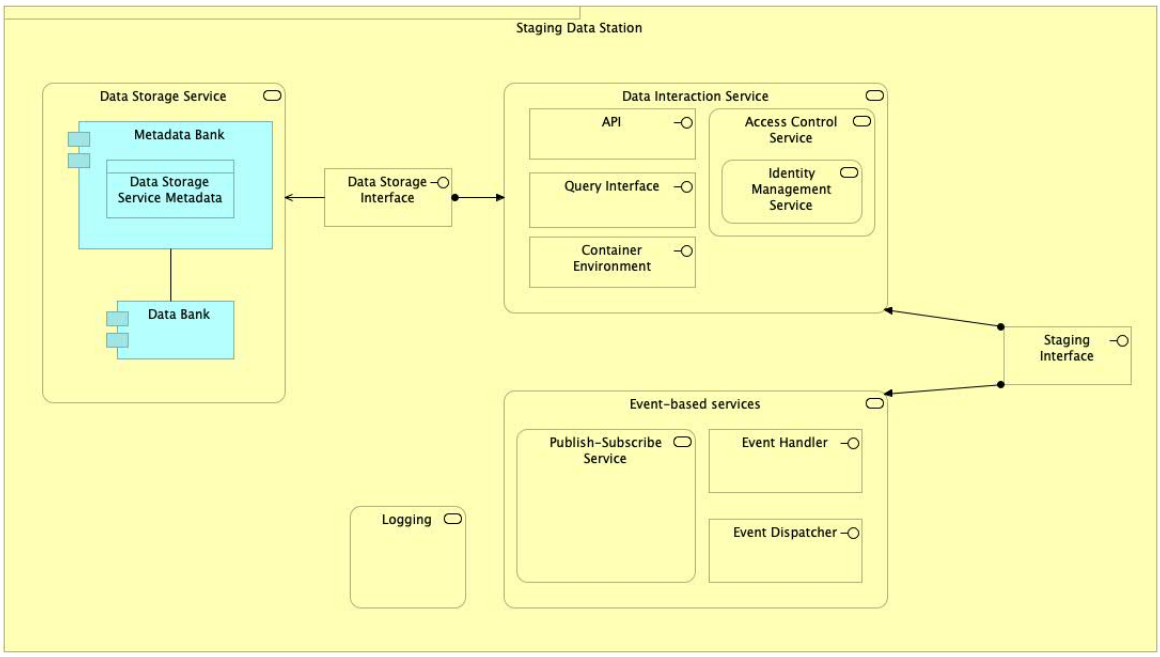


Figure 5.3: Staging Data Station Architecture.

while keeping private information protected. Therefore, some assumptions are considered, including:

- The Train is trusted.
- Trains are checked by the Data Station.
- Data Stations are trusted parties.
- Contracts and access to a cloud provider are made in advance.

5.3 Extended Design

The current architecture does not include some components that are needed for further architectural improvements and scalability. These components are not part of the Staging platform, but they can provide a more robust architecture and future enhancements. Figure 5.4 depicts the main elements of the extended PHT architecture and their relations.

5.3.1 Train Handler

As the healthcare network grows, the Train Handler is expected to play a crucial role because more intelligence has to be used to control and manage the Train

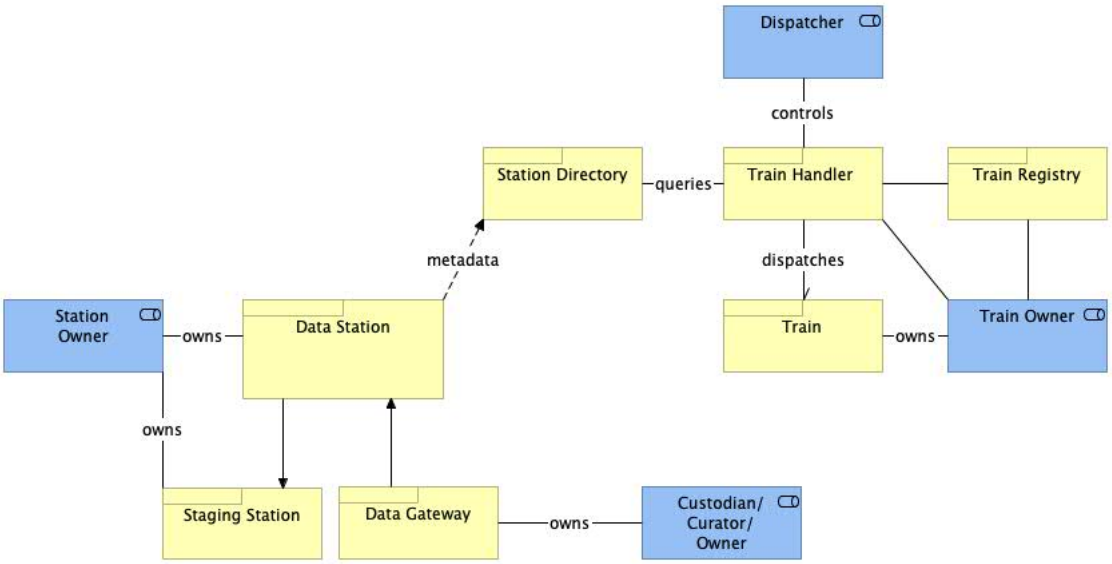


Figure 5.4: Extended PHT High-level Architecture.

schedule and results. It subscribes to be notified by the Train Registry whenever a new Train is added to the Train repository. It offers two services:

- **Routing Service:** It can create a routing plan for the Train, including the Stations that need to be visited and the choreography analysis. This choreography specifies if the Stations will be visited in a sequence or in parallel, among others. Once the plan is finalized, it is executed, sending the Train to its target Stations.
- **Aggregator Service:** It is responsible for handling the results before sending them to the user. It aggregates results from different Data Stations when required.

The Train Handler has the following responsibilities:

- Evaluate the Train metadata.
- Discover Data Stations by interacting with the Station Directory.
- Create a routing plan based on the Train requirements and available data.
- Dispatch Trains to the appropriate Data Stations.
- Orchestrate the dispatch of Trains to multiple Data Stations when necessary.

5.3.2 Train Registry

The Train Registry stores and publishes Trains as images in different versions. It primary purpose is to enable the reusability and portability of Trains. The Data Stations can connect to the registry to get Trains and push Trains with the results. In the current implementations of the PHT, the Data Stations scan the Train registry to know if there is a new Train for them. Although this approach works, some improvements are still possible. With short polling intervals, resources are wasted handling unnecessary requests, whereas long intervals increase update latency.

The Train Registry has the following responsibility:

- Implement Publish-Subscribe services to allow the Train Handler to subscribe for changes in the Train registry, e.g., when a new Train arrives.

Figure 5.5 depicts how the building blocks previously mentioned are integrated in the final PHT architecture proposal.

Summary

We can leverage the cloud and Infrastructure as Code technologies for building a Staging Data Station. We identified four general elements that are crucial. Using a dynamic platform, we can use theoretically unlimited resources to execute our Train. The dynamic platform solves the problem of limited resources. Therefore, to provide automated deployment, we can use provisioning tools via Infrastructure as Code that allows us to define the Staging site desired state defined in advanced in a piece of code. APIs allow the communication between the various architecture components, inside the Staging site and between the Data Station and the cloud. The Station Owner should create the definition files that describe the infrastructure resources through a provisioning tool that, via an API, instructs the dynamic platform to create and manage the infrastructure resources. Finally, we used event-based services for full automation, so that events can trigger specific workflow tasks instead of triggering them manually.

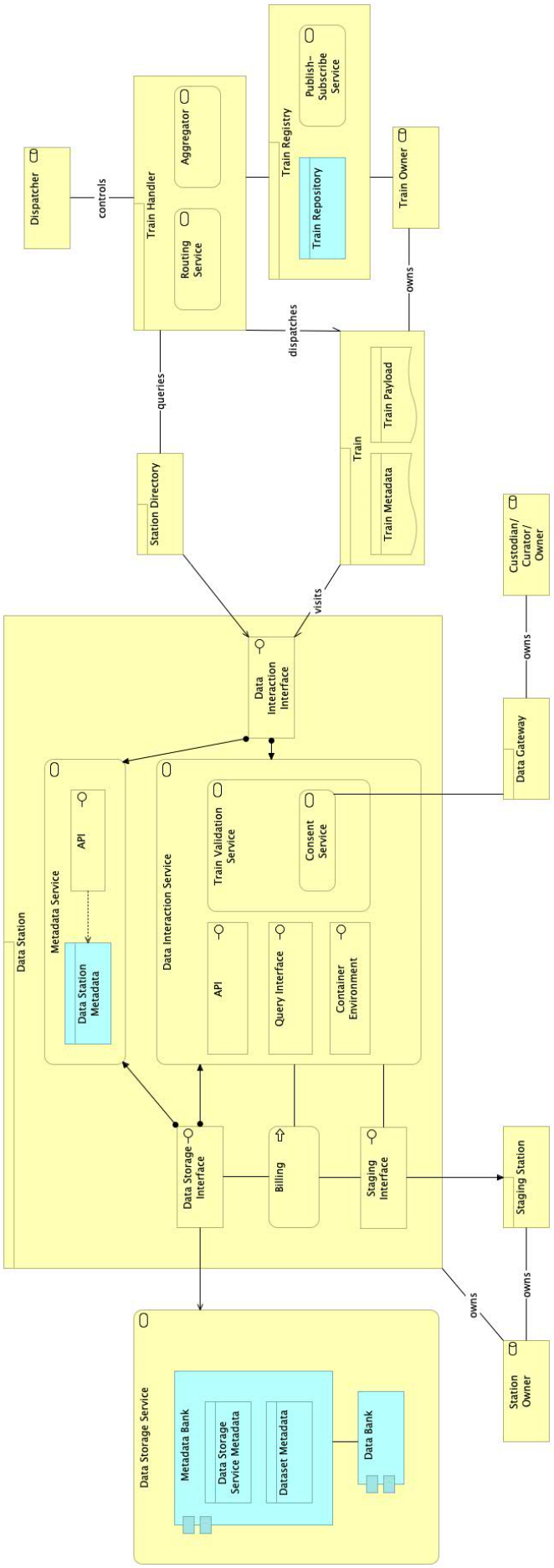


Figure 5.5: PHT Extended Architecture.

Chapter 6

IMPLEMENTATION

In this section, we present our implementation of the design. We begin with the selection of tools, following by the implementation in the dynamic infrastructure platform. We provide the implementation of the Staging Data Station in the cloud, in which only the steps highlighted in red in Figure 4.2 are implemented.

6.1 Selection of Tools

We selected tools for the implementation mainly based on the requirements listed in Chapter 4 and the elements in Chapter 5. Table 6.1 shows a list of the selected tools and a summary of the reasons for this selection.

6.1.1 Dynamic Infrastructure Platform

The dynamic platform offers the programmable resources used by the provisioning tool to set up the IT infrastructure. The most suitable platform for our use case is AWS, since it provides flexible and affordable infrastructure components on which we can build the PHT Staging Data Station. It offers a breadth of services aligned with our architecture requirements, allowing us to configure policies for a suitable security architecture. AWS is fundamentally a cloud middleware for our solution, not just a hosting business platform.

The Staging platform must be capable of running container images of Trains. AWS provides different computing solutions, from VMs to entire container management orchestrators, making it suitable for a heterogeneous IT infrastructure. It allows us to run a cluster of VMs or entire clusters of containers managed by

Table 6.1: Chosen tools.

Element	Tool	Reasoning
Dynamic Platform	AWS	•GDPR compliance •Extensive integration and support by Terraform •Free-tier resources for testing •Plenty of options for infrastructure resources •Global infrastructure, multiple locations worldwide •Requirements compliance
Provisioning	Terraform	•Open-source •Supports multiple dynamic platforms •Declarative configuration •Other options are vendor-specific solutions

supported orchestrators.

Providing services that comply with standards and regulations help us accomplishing compliance for our Staging platform. Compliance meets a checklist of standards and regulations; however, it involves a two-way partnership between the Data Station and AWS. The Data Station owns the data and uses them with legal and ethical obligations, and AWS must use data in compliance with regulations. AWS complies with several compliance programs in its infrastructure and data regulations [36], including GDPR, which is one of our primary requirements to host a Data Station in the cloud. The infrastructure and services meet the privacy and security aspects of the PHT specification. Besides the regulation's compliance, AWS provides services to keep data secure; these aspects fulfil the requirements listed in Chapter 4.

It also offers Software Development Kit (SDK), which allows the use and interaction with AWS resources. Consequently, it is possible to integrate and communicate with a wide range of external resources and external infrastructure. Multiple AWS SDK toolkits are available for Python, Nodejs, JavaScript, C++, Go, PHP and Ruby, which enable us to create our implementation with our preferred language. Moreover, AWS also supports tools to implement IaC, providing security from the beginning of the architectural design. Most of the AWS resources are supported by

Terraform, the provisioning tool chosen for automated deployment, a fundamental reason for this platform's choice.

AWS is available in multiple locations worldwide. Resources are not replicated across regions unless we choose to do so. In the first instance, the PHT approach was developed for the European market, but it can be a worldwide solution as it grows up. We proposed using the closest Cloud Data Center for legal and ethical purposes to implement the Staging Data Station in the cloud. Therefore, we need a platform with multiple locations to support it. Today, AWS has 24 regions worldwide, of which six are in Europe, namely in Frankfurt, Paris, London, Ireland, Stockholm and Milan. Spain and Switzerland are coming soon, which covers a vast part of Europe [37].

Although AWS is not a free platform, it offers a free usage tier for one year, which allows us to do tests and use all the services without incurring an extra cost [38]. Moreover, it offers an affordable pricing model for some resources, such as spot instances, and we pay for what we use per hour or resource depends on the service. Table 6.2 shows the AWS's services used for the implementation, as well as the PHT component that each service implements.

6.1.2 Provisioning Tool

The initial step towards deploying a Staging Data Station is the provisioning of the required components and resources dynamically, such as the storage, VMs and containers. Component provisioning is the most critical step for deploying a Staging platform automatically. Therefore, the choice of provisioning tool is vital for the entire system and should focus on offering high integration, scalability, and maintainability. A comparison of the available provisioning tools is available in Table 2.4. Terraform is the most suitable tool from the list, as it is open-source, it uses declarative files to specify the desired state of the infrastructure and then uses APIs to reach that state. Besides, Terraform supports custom-in-house solutions, which gives us more options for custom PHT configurations. Other provisioning tools are closed source, i.e. CloudFormation, Azure Manager and Cloud Deployment Manager, which automatically bonds the system to a specific platform. Although AWS offers CloudFormation, we chose Terraform to demonstrate that our solution is not bound to AWS. The provisioning tool could simultaneously be used with other cloud providers, allowing future improvement like the freedom to change the cloud environment or the possibility to build a federated Staging site.

Table 6.2: AWS's services [10].

Service	Description	PHT Component
S3	It provides object storage through a web service interface.	Data Storage
CloudWatch	It is a monitoring service that provides data and actionable insights for AWS infrastructure resources	Event-Handler, Event Dispatcher
Simple Notification Service (SNS)	It is a notification service. Using SNS topics, publisher systems can fanout messages to many subscriber systems, including HTTP endpoints.	Publish-Subscribe Service
ECS Fargate	It is the compute runtime environment. It is a serverless technology that facilitates the deployment, and we do not need to be concerned about how many resources assign in advance.	Container environment
Identity and Access Management (IAM)	It manages access to AWS services and resources securely.	Access Control
VPC	It establishes a custom networking environment	Networking
ECR	It is a fully-managed Docker container registry.	Train Registry

Terraform is the chosen provisioning tool and many infrastructure components can be implemented as Terraform resources. Terraform provisions the resources of a dynamic platform, and a Terraform provider is used to interact with the APIs and expose the resources from the corresponding dynamic platform. In this use case, the AWS provider is used for provisioning all the resources necessary. Besides, we choose the closest AWS region, Frankfurt, to comply with GDPR. Terraform is installed in the machine that runs the Data Station. See Appendix A for the Terraform configuration files used in our implementation.

6.2 Infrastructure Details

The implementation comprises two parts, *the Data Station*, which runs in a laptop and the *Staging Data Station*, which runs in the dynamic platform AWS. Figure 6.1 depicts the interaction diagram that shows the interactions between components in the implementation to support the deployment and execution of a Train.

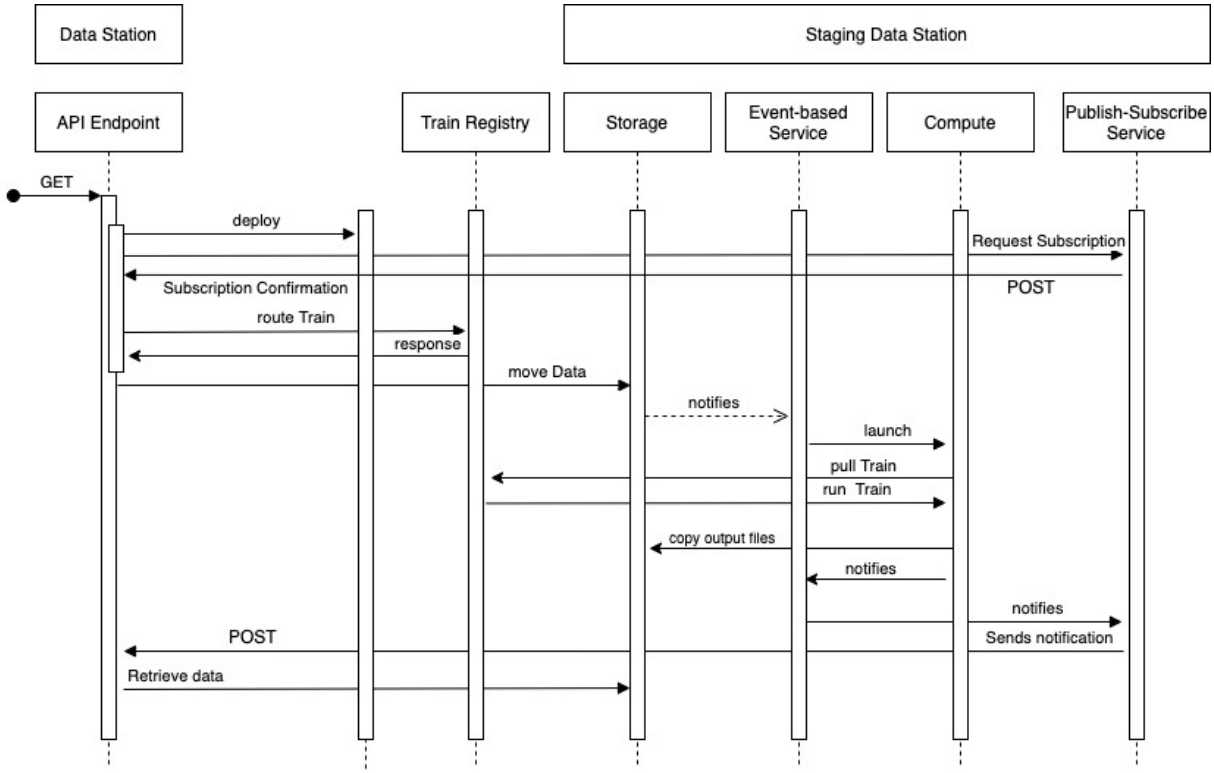


Figure 6.1: Interaction Diagram.

In Figure 4.2, in the Data Station swim lane the required data should be queried by the Train and sent to the cloud. The Data Station is also responsible for retrieving the final results from the cloud. For that reason, we implemented an API that plays the Data Station role, which interacts with the Staging Data Station and the Train.

Our implementation assumes that the data necessary to run the Train were already queried and stored in a specific path in the Data Station. At this point, the Train has already been sort to the Data Station, which means the Train is in the Data Station. Therefore, it is routed to a Train registry again as part of the workflow as we are not implementing a Train Handler with a routing protocol. The API is configured in NodeJS and exposed to the Internet via the localtunnel npm tool.

A **GET** request simulates the **Stage phase** proposed in Figure 5.2. The API call

launches the Staging Data Station through the Terraform definition files. The Terraform files allows the provision of the infrastructure components in the AWS Cloud all at the same time as depicted in Figure 6.1. During deployment, the Data station subscribes to receive a notification when the Train execution is completed. Parallel, the Train is routed to the Train Registry, after that the data are moved to the cloud storage. By using of logs and rules the event-based service detects when the data have been uploaded and immediately launches the computing resources via a task. The task pulls the Train from the Train registry and executes it. Once the Train execution is completed, the data are copied to the output storage. The event-based service detects a change in the computation state and announces it to the notification publisher. Finally, the notification publisher sends a notification message via a **POST** request to the API Endpoint, downloading the result files from the Output storage.

Figure 6.2 illustrates the high-level Staging Data Station architecture of the infrastructure we deployed in the AWS Cloud in accordance with the interaction diagram in Figure 6.1. In this way, the architecture is capable of processing a Container Train.

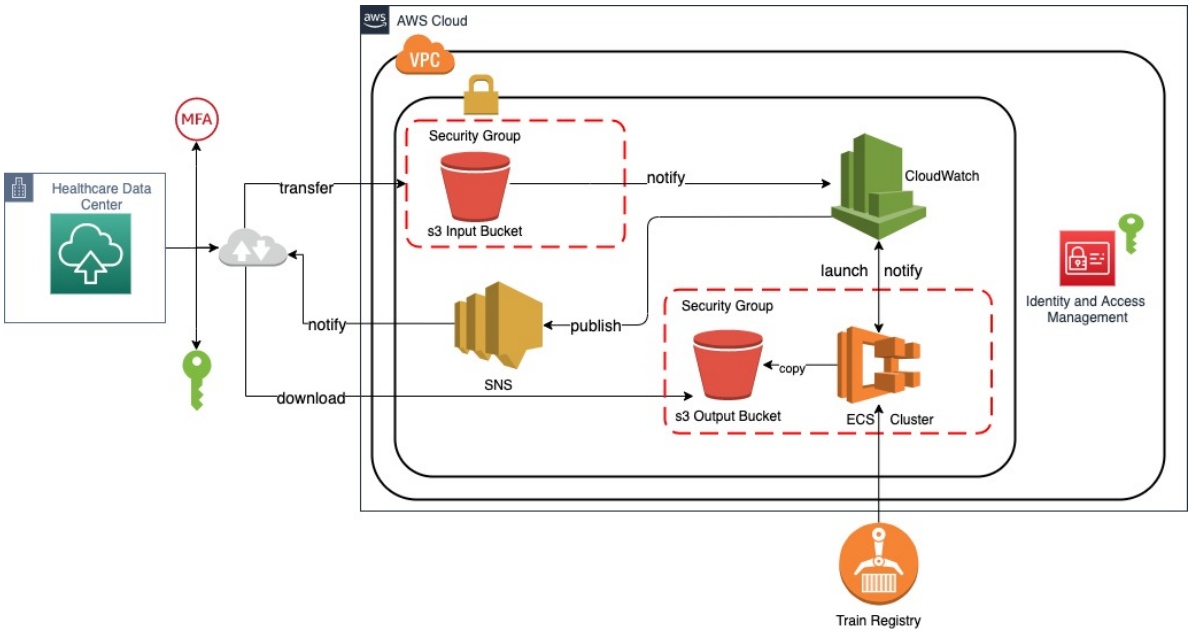


Figure 6.2: Implementation in AWS.

6.2.1 Authentication

In order to create an environment, we first need an Amazon Web Service Account and also a special authentication method to authenticate. We use Multi Factor

Authentication (MFA) to access the AWS console. We use an Admin role that represents the healthcare organization, which in addition to a name, has two keys, namely the **public** assess key and the **secret** key. In this way, the desired connection to the environment is done in an absolutely reliable and secure way.

6.2.2 Notification Service

Amazon SNS sends an HTTP POST request when it confirms a subscription, sends a notification message or confirms a request to unsubscribe. The POST message contains SNS header values that allow us to identify the type of message and run specific jobs in the API Endpoint. The most important value of the message is:

x-amz-sns-message-type; which indicates the type of message either *SubscriptionConfirmation*, *Notification*, or *UnsubscribeConfirmation*. We take this header value to confirm the subscription and to receive a notification as depicted in Figure 6.1. Figure 6.3 illustrates these steps when the API Endpoint receives a **Subscription Confirmation** message; it confirms that it wants to receive notification messages from SNS. On the other side, when the message type is **Notification**, the API knows it is time to download the results; therefore, it executes a Python script using the AWS SDK to download the output files.

```
if (req.header('x-amz-sns-message-type') === 'SubscriptionConfirmation') {
    const url = payload.SubscribeURL;
    await request(url, handleSubscriptionResponse)
}else if (req.header('x-amz-sns-message-type') === 'Notification') {
    console.log('RECEIVED');
    const childPython = spawn ('python', ['./down.py']);
```

Figure 6.3: POST request.

6.2.3 Storage

The Data Storage component is implemented with Amazon S3, so from now on it is referred to as Bucket. It is one of the first resources that we need to create to move the required data to the cloud. We used three buckets by design, to store input data, output data and log files. Respectively the use of several buckets provides more granular security and facilitates automation, triggering different actions on each of them; input data triggers the Train execution while output data triggers the download and faster data retrieval at the end of the execution. Table 6.3 shows the Buckets and their use.

Table 6.3: Buckets.

Bucket	Use
Input	It stores the data from the healthcare organization.
Log	It stores event history log files of the AWS account activity in the region. It helps the event-based service launch other resources dynamically.
Output	It stores the results of the Train execution. The main Data Station retrieves these files at the end of the Train execution.

The implementation supports *client-side encryption* and *server-side encryption* for protecting data at rest in the cloud and in transit against unauthentic and unauthorized access while ensuring that remain intact and available. The data are protected all-time through:

1. **Protect Data in Transit**

Client-side encryption is used to protect data in transit by encrypting data before sending it to AWS S3. To have secure connections, HTTPS protocol is used. If the healthcare organization has Virtual Private Network (VPN) infrastructure, it is recommended to establish a private connection to the cloud.

2. **Protecting Data at Rest in the cloud**

For the server-side encryption, a unique encryption key is generated for each object. Then, data are encrypted by using the 256-bits Advanced Encryption Standard (AES-256). After that, a securely stored and regularly rotated master key encrypts the encryption key itself. Users can choose between mutually exclusive possibilities to manage the encryption keys. The input and output Buckets use unique Amazon S3-Managed Keys (SSE-S3) with strong multifactor encryption. Data are encrypted using AES-256 encryption.

Data Transfer

The Data transfer is configured with the Terraform files too. We exploit the **depends_on** meta-argument provided by Terraform to express dependencies between components. In this case, the data are transferred once the Train has been already routed to the Train Registry as it is the process that takes more time to finish, we can assure that the rest of the resources will be already created once the process

ends. As a consequence, we can move the data without concerns. Besides, to detect file corruption, we verify the integrity of the uploaded data with MD5 checksum.

6.2.4 Event-based Services

Usually, AWS Services generate notifications when an event occurs. These events are used to trigger an action. However, these actions have to be stored somewhere and rules and targets should be defined based on them. The bucket for log files stores all the input bucket actions. We created a CloudTrail that reports the activities of objects in the input bucket. These activities are seen as events by the CloudWatch service. Then, we configured an *upload S3 event* rule in CloudWatch. A CloudWatch target means once a rule is fulfilled, it triggers an action in the target. Accordingly, when data finish being uploaded to the Input Bucket, which is the configured rule , it launches a compute engine for containers using ECS, and in this way the Train is executed.

6.2.5 Computing

Another critical resource created by Terraform is the compute resource. Amazon ECS makes it easy to launch containers and scale rapidly to meet changing demands. One of the challenges we have to face when we execute an application is the provision and management of computing and memory resources. There are several mechanisms to predict the resources required and scale when appropriate. However, the Staging Data Station is a temporary deployment which has the main task of providing the appropriate computing resources for the Train. ECS Fargate is a serverless solution that allocates the required amount of computing capabilities, avoiding the need to choose instances in advance and scaling cluster capacity if the application requires it. We use an entity called a **task definition** to describe to ECS how to run the containers. ECS task definition can be thought of as a prototype for running an actual task. The task definition allows for one or more containers to be specified. In our implementation, each Train is mapped onto one task definition. The task definition describes that the Train should be pulled from the Train Registry when the CloudWatch rule matches the uploading event. Unlike Virtual Machines in the cloud, ECS Fargate is charged by vCPU and memory and not by the hour.

6.2.6 Security

Beyond authentication and encryption mechanisms, in a cloud infrastructure, a good strategy for security is to classify, split, and divide everything, whether it is by roles, permissions, regions, networks or firewall. In a cloud environment we can implement security at different levels. We created a Virtual Private Cloud to isolate our components from other customers in AWS. However, the resources cannot interact with each other, if we do not configure a policy to allow them to interact. Therefore, we provide security via an Identity and Access Management service with access control through user definitions, roles and permissions to users on each step of the workflow. For instance, the ECS Cluster has read access to the S3 input bucket, but it does not have write permission as it only requires to get data from it. On the other side, ECS has write-access to the S3 output bucket.

The implementation uses a collection of network access rules to limit traffic types that can interact with a resource. This collection of rules is called a **security group**. A resource can have one or more assigned security groups. The rules in a security group control the traffic that is allowed to an instance. If incoming traffic does not match a rule from the group, access is denied by default.

Chapter 7

CASE STUDY

This chapter describes the process to evaluate the design proposed in this research. To choose the appropriate approach for our evaluation, we followed the framework of Venable et al. We followed the analytical evaluation method [16] to assess the architecture using the static and dynamic analysis. This option is the most suitable evaluation strategy since our proposal involves functional and non-functional aspects. In particular, we begin by choosing the Checklist-Driven Validation and the Scenario-Based Validation approaches presented in [32], [33].

7.1 Approach

We modified the utility tree built in Chapter 4 to organize the attributes to be evaluated and show the results. We defined several quality attributes in the utility tree: *performance, security, reusability, functional suitability, configurability, cost-effectiveness and portability*. These attributes are derived from ISO 25010 [35], which is the primary standard for evaluating the quality of a software system.

We focus on quality attributes features for evaluation because they are defined on top of the functional requirements. We decided to focus on the performance attribute for the dynamic analysis; therefore, other quality attributes are for the static analysis. We used the lowest level of the utility tree depicted in Figure 7.1 for the scenario-based validation, and the first two levels for the checklist-driven validation. Checklist and scenario-based approaches complement each other. The checklist helps us with the static analysis, and it is complemented with the scenario-based process, which helps with the dynamic analysis that measures how the system responds to the questions and answers from the checklist.

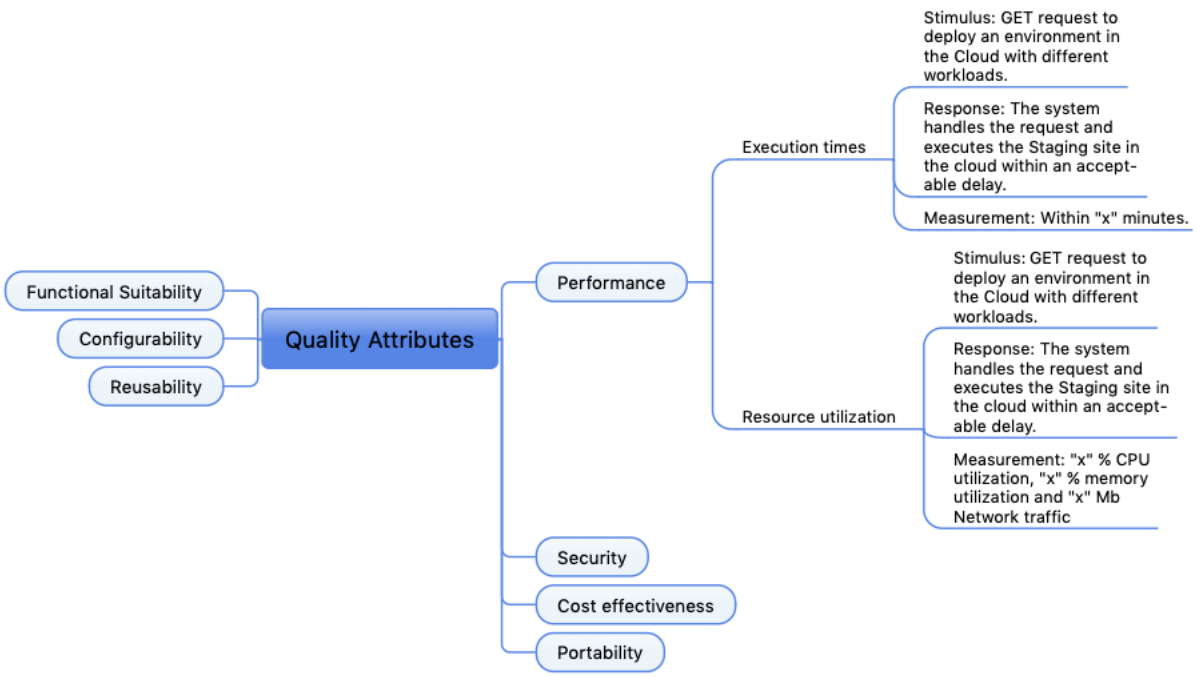


Figure 7.1: Utility tree.

We used the implementation explained in Chapter 6, and we built a Container Train with an algorithm to process and analyze COVID-19 patients' information. The implemented architecture comprises a real environment and simulated data. On one side, the Staging site is deployed in a real cloud environment (AWS), taking advantage of AWS's free tier to avoid extra costs. Besides, the use of open-source tools allowed us to configure the infrastructure also without additional costs. On the other side, the patient's data needed to be simulated; otherwise, it would have taken a long time to obtain real patient's data with their respective consent. For the static analysis, we analyzed through the checklist method the set of qualities we wanted to assess based on the requirements established in Chapter 4.

7.2 Dynamic Analysis

This section provides a clear insight into how our case study has been carried out from datasets to perform an exploratory data analysis for a dynamic evaluation, followed by evaluation metrics. In a dynamic analysis, the main goal is to study the system using dynamic qualities, such as performance.

We have evaluated our architecture, using the scenario-based approach by simulating a COVID-19 case study. We used datasets created in the literature [39],

where the authors generated synthetic data using the open-source Synthea tool, resulting in datasets containing synthetic Electronic Health Records (EHR). Moreover, we created a Container Train containing an algorithm in Python to analyze and aggregate results to calculate and measure the impact of COVID-19 on society.

Data Station handles the execution process. We used a computer with 1.8GHz Dual-Core Intel Core i5 and 8GB memory, representing the Data Station for this research. The computer ran the following software: Terraform, Docker client, AWS SDK for Python, Nodejs and Express. To run the implemented architecture, we executed the following steps:

1.- Created an AWS account.

2.- For security reasons, it is not recommended to add the access keys in the code; therefore, we configured the authentication credentials in a named profile in a configuration file in the computer (.aws/config).

3.- Initialized the Docker client.

4.- Added the path where the datasets are stored in the Terraform file (see Figure A.6 in Appendix A).

5.- Initialized the API.

6.- Ran the localtunnel npm tool to expose the API to the Internet and receive an HTTP URL.

7.- Ran the "curl <API HTTP URL>" command.

7.2.1 Datasets

From March 2020 to May 2020, a model of the Corona virus (COVID-19) disease progression and treatment was built for the open-source Synthea patient simulation, as explained in [39]. When properly constructed and validated, synthetic data used in Data Analytics and Machine Learning tasks have been shown to have as accurate results as real data, without compromising privacy [40]. The study outputs data from 124,150 synthetic patients. We chose two bundles of datasets to compare execution times; these include information about the disease, symptoms, treatment, mortality and care plan. Indeed, the only difference is that the first one has information of ten thousand patients and the second one hundred thousand patients, actually the former is a scale of the later.

In Chapter 2, we mentioned that it is essential to manage the data in a manner that data are always interpreted in the same way, no matter where and by

whom the data are being used. The datasets contain FHIR data and codes that specify each feature; this follows the FAIR principles and allows us to create an algorithm based on the analysis we want to do and not to be concerned about specific datasets structures. Table 7.1 depicts an example of a dataset where we use the information from the **CODE** column to configure the algorithm. For instance, to know if a patient got COVID, we use the code **840539006**. Although our research does not cover how data should be curated and how different data formats can be transformed into consumable data standards, it should be clear that codes and vocabularies facilitate the development of algorithms that can be used anywhere. Therefore, our Train is **reusable** and **interoperable** with any Station.

Table 7.1: Datasets.

START	STOP	CODE	DESCRIPTION
2020-03-15	2020-04-01	65363002	Otitis media
2020-03-01	2020-03-30	386661006	Fever (finding)
2020-03-01	2020-03-05	840544004	Suspected COVID-19
2020-03-01	2020-03-30	840539006	COVID-19
2020-02-12	2020-02-26	44465007	Sprain of ankle

7.2.2 Evaluation metrics

Performance is the most suitable quality attribute to evaluate the architecture using dynamic analysis. We used two sets of measurements for this quality attribute, based on the ISO standard and the validation technique presented in [33].

1.- **Time behaviour** - Defined as the degree to which the response and processing times and throughput rates of a system meet requirements when performing its functions. We measure execution times from when the GET method is invoked until resources are destroyed, in one instance of execution of a Train.

2.- **Resource utilization** - Defined as the degree to which the amounts and types of resources used by a system meet requirements when performing its functions. We measured the CPU Average use and RAM average use in the cloud. Moreover, we measured network traffic in the Data Station during the execution process.

7.2.3 Validation

The datasets are stored in the computer, that plays the Data Station role. We executed the process explained in Figure 6.1. In [33], authors recommend to use at least one scenario validation per quality attribute. Our scenario performs to do a simple analysis with information stored in datasets of various sizes that represent different workloads to measure the system behaviour, mainly the network and computing resources. The experiment aimed to calculate all matching patients diagnosed with COVID-19 and get measures about them using the 10K and 100K bundles. For the patients diagnosed with COVID-19, we got summary statistics of patients who recovered and died, and the care plan of the people infected.

We ran the execution of the system five times per bundle. After these executions, we got an average calculation for the analysis of the system. This prevents any data disturbance caused by isolated events from having significant effects on the results. We used the tool *iftop* installed on the computer that plays the Data Station to collect network traffic information. Besides, we harnessed the CloudWatch monitoring tool in AWS to get the CPU and memory utilization.

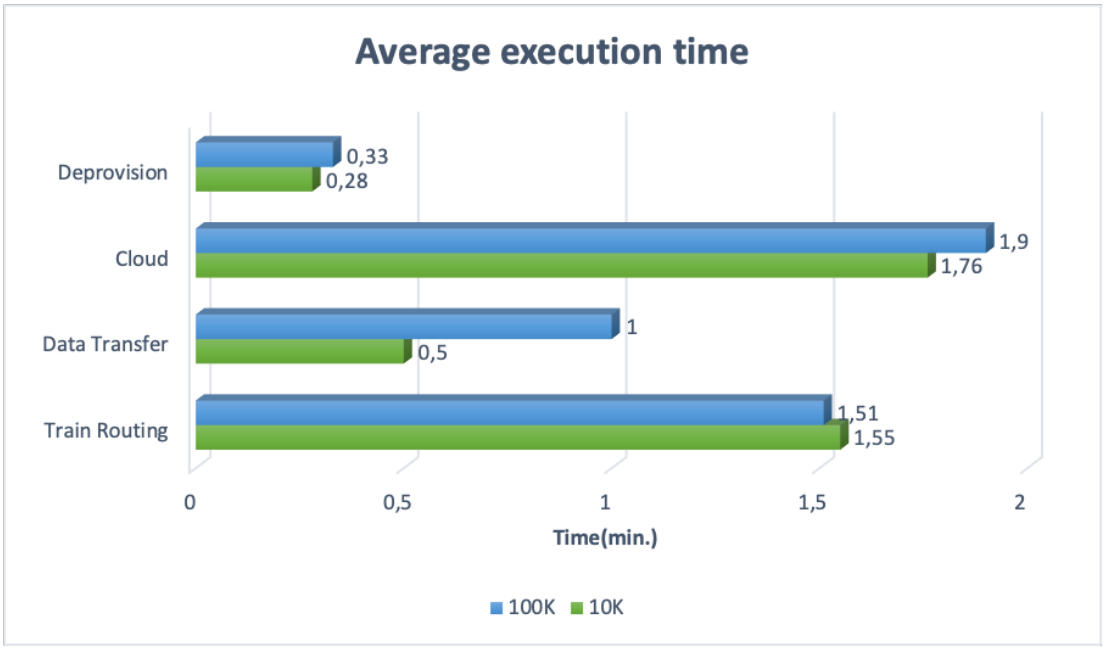


Figure 7.2: Average execution time.

Figure 7.2 shows the execution time for the provision and de-provisioning process. The provisioning process comprises the Terraform files execution, data transfer, Train routing, Train processing in the cloud, and downloading the results. The de-provision process covers just the deletion of the entire cloud resources created by Terraform. The entire execution process is the sum of all the values shown in

Figure 7.2 per dataset bundle. We split the provisioning measurements to get the execution time for Train routing, data transfer and cloud; the cloud measure includes all the AWS processes. The Terraform files are executed so fast that it is not very easy to be measured. We can observe that the difference between the two bundles' execution time is slight; there is only around 15% of difference. Figure 7.3, Figure 7.4 and Figure 7.5 explain the reason behind this behaviour.

Figure 7.3 depicts the network traffic during the provisioning process; the 100K bundle consumes on average around 70 Mb while the 10K consumes around 62 Mb; as a consequence, the transmission times were different but not ten times bigger as the amount of data.

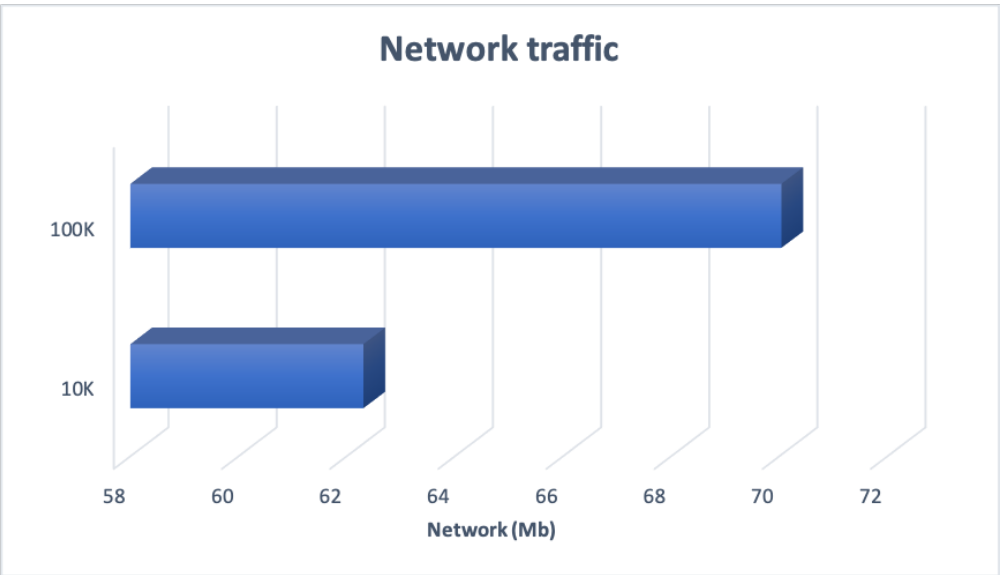


Figure 7.3: Network traffic.

Figure 7.4 depicts the CPU average utilization. We can observe the CPU average utilization for the 100K bundle was 85.6%, almost 30% more than the 10K bundle. Consequently, the cloud processing time was very similar among both bundles, but more resources were consumed by the 100K.

Figure 7.5 shows the average memory utilization. In general, both bundles did not consume much memory, and both used less than 20%. The two bundles' overall execution times were very similar despite the size difference due to the system consumes more network and compute resources for the biggest bundle.

We can conclude that the Data Station network and cloud computing instance play a crucial role in our system performance, more than the amount of data. The scalability of the computing resources is achieved in the cloud; on the other side, the network consumption depends on each healthcare organisation's net-

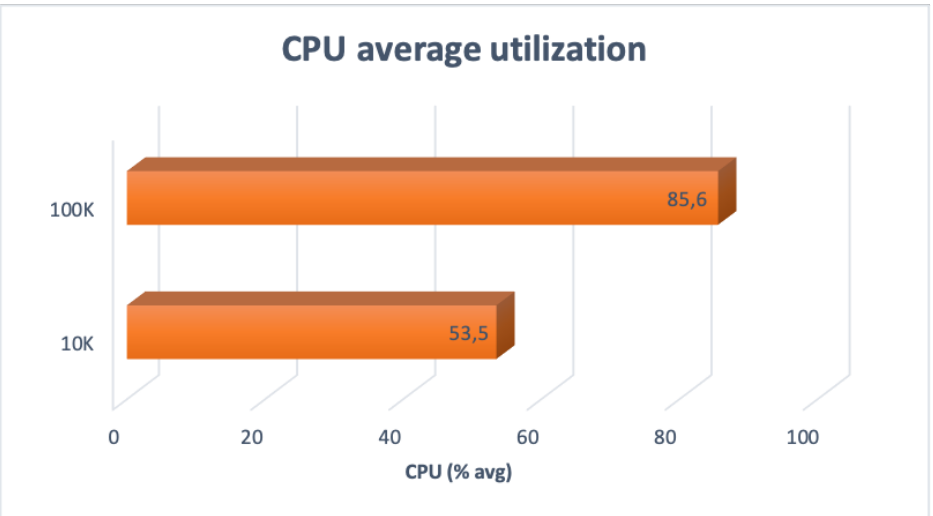


Figure 7.4: CPU average utilization.

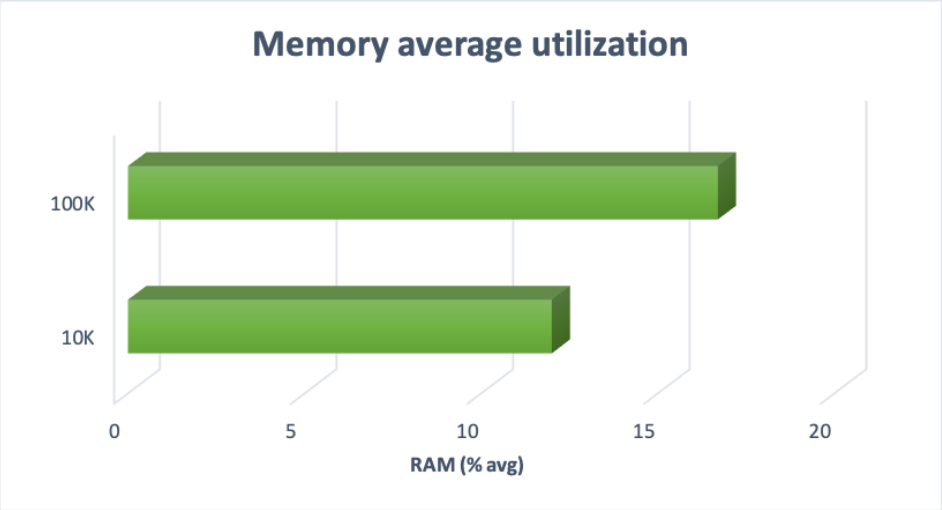


Figure 7.5: Memory average utilization.

work, and it can be improved with techniques such as multi-upload files. If we want to increase speed, we can use multithreading techniques, although it consumes much network resources. Train routing has a significant time in the entire execution process; we can improve it, adding the Train Handler proposed in Chapter 5 to avoid consuming much network resources from the Data Station and reducing the execution time.

We also present the results from the analysis of the 10K datasets, where table 7.2 depicts the mortality rate of the 8820 infections, 96% of people recovered, which is a high rate, demonstrating that it is highly contagious but not highly fatal.

The care plan for COVID has two values, namely staying at home or being hospitalized. Table 7.3 illustrates summary statistics of patients who recovered at

Table 7.2: Mortality rate.

	Recovered	**Death**	**Ventilated**
COVID-19 (n=8820)	0.9606	0.0404	0.0325

home and hospitals. The hospitalization rate is considered high for the period these data were gathered. However, still, the vast majority of people followed a care plan at home. Besides, in Table 7.4, we can observe that the ICU Admission rate was high, and almost everyone at the ICU required ventilation. The death rate for people in the ICU was high, and nearly all patients required ventilation. From these data, we can conclude that patients admitted to the ICU and who use ventilation have a high probability of dying.

Table 7.3: Care plan.

Care Plan	**Rate**
Home Isolation	0.7952
Hospitalised	0.2116

Table 7.4: ICU Admission Rate.

	ICU Admission	**Ventilation**
Required Ventilation	0.7653	1.0
Recovered	0.3573	0.1637
Death	0.6453	0.8362

These results mean the success of our architecture implementation. Our deployment enables analysis against privacy-sensitive data sources and successive evaluation of that analysis in a secure enclave. We could deploy the Staging site, the Train analyzed the data and got a final file with the information provided in tables 7.2, 7.3, and 7.4. directly in the Data Station. It demonstrates that the standardization of the data structures alongside a proper architecture facilitates data analysis in any environment.

7.3 Static Analysis

Some system features are essential to be aware of the system's quality level, but a dynamic analysis is not suitable for them. For these features, we decided to use a static analysis with a checklist validation. Figure 7.1 depicts the utility tree using as a reference for the analysis.

Functional Suitability: Our architecture proposal complies with the PHT principles and the requirements defined in Table 4.1. The primary benefit of the PHT is keeping the data within the Station Owner realm, which means that even though the architecture proposal requires to move the data to the cloud, the control and management are still the responsibility of the Station Owner. Hence the Train Owner never has access to the data even if the data are in the cloud. For that reason, the results are first downloaded by the Data Station. The PHT workflow can provide, where the results can be reviewed to check that they do not contain sensitive data before sharing the results with the Train Owner. The Staging site is deployed automatically, and the entire Train execution process is performed automatically without human intervention. We used a serverless container environment that supports scalability if more computing and memory are required, and we do not need to estimate resources in advance. For example, in our validation the system provided the required computation for analyzing both bundles without the need to change the configuration.

Configurability: Defined as the end-user support to customization. Besides the serverless solution, the implementation supports any other computing environments, such as VMs or Kubernetes; depending on the preferences of the Station Owner. Nothing changes from the architecture perspective; however, more configuration management tools would be required to configure the VMs, and these tools can be configured in the same Terraform files. The end-user can customize the resources' names, use any AWS region changing the region in the files and use any containerization technology.

Portability and **reusability**: We defined a territorial scope looking to choose the closest cloud region, Frankfurt, for moving the data, complying with Article 3 of the GDPR. The system is reusable; if there are not cloud preferences, anyone can use the configuration files alongside an appropriate AWS account and the required open-source software installed. Moreover, the solution can be used in any AWS region.

Security: One of the major concerns in this architecture is the security of the data. We must ensure that the data are not compromised from transferring from

the healthcare data center to the cloud and when data are already in the cloud. The implementation supports *client-side encryption* and *server-side encryption* for protecting data at rest and in transit against unauthentic and unauthorized access, while ensuring intact and available data. These security mechanisms make the architecture comply with article 32 from GDPR. Besides, the data are stored in the cloud just for the processing period. The Terraform configuration allows the resources and the data to be discarded as soon as the job is finished, complying with Article 5 of the GDPR.

AWS provides data protection by design; when a user or an application tries to use the AWS resources, the AWS service executes a few steps to determine whether to allow or deny the request. All requests are denied by default; this means that everything that is not explicitly allowed by the policy defined in the implementation is rejected. Therefore, every component is configured through the IAM roles and permissions to access only the resources required to complete its tasks, keeping private data. This approach complies with Article 25 of the GDPR [36].

Cost-effectiveness: We harnessed open-source tools such as NodeJS, Terraform, Python and Docker; this decision not only decreases the deployment costs but also supports the usability of the system, as more people are familiar with standards structures used by open-source software than with vendor-lock solutions. Moreover, the Terraform tool gives us the flexibility to use the cloud environment that we preferred. The serverless instance in the cloud provides us with the scalability and the computing power to run any Train. However, the incurring cost is by vCPU and RAM consumed, and not by hour. This can be a drawback of the solution, but more tests must be done to assess the cost-effectiveness of the solution. We did not reach that peak in consumption in our scenario as our data analysis was not intensive as what a complex machine-learning algorithm can be.

Chapter 8

CONCLUSIONS

In this work, we designed and presented a reference architecture of a Staging Data Station, i. e., a computation-capable environment that supports Train executions when the main Data Station does not have enough resources. We employed Infrastructure as Code, APIs, and Event-based systems to realize a dynamic deployment in the cloud. To ensure that the architecture complies with the regulations and the patients' data are secure, we reviewed the current regulations for processing personal data in a public cloud. We translated the most crucial articles into technical requirements. We described a PHT workflow and created a sequence of actions that uses the requirements and gives a better insight into the PHT approach to continue developing more parts of the system.

We implemented the architecture proposal using novel technologies and AWS. We evaluated the proposal with dynamic analysis through a case study, analyzing datasets of ten thousand patients and one hundred thousand patients. Besides, we did a static analysis checking that implementation complied with the GDPR articles alongside a set of quality attributes derived from the primary standard for evaluating a software system's quality, ISO 25010.

The research showed that we could deploy a computation-capable environment when required using the cloud and automation tools, complying with the PHT principles while providing a fitting and secure site. Although our design requires moving the data to the cloud, the data are still within the data source realm and control, keeping data privacy. Moreover, the proposal considered the principal regulation for processing personal data in the cloud to keep the information secure and private as possible; however, this cannot be fully guaranteed. The case study showed that the instantiation and processing times of the Staging site depend on the network in the Data Station and the computing resources consumed

in the cloud. The simulation showed similar execution times with different workloads sizes but a significant difference in the network and computing consumption, which can cause a bottleneck in the Data Station network. The case study worked adequately with a simple aggregation algorithm. In future, our results can be further validated and possibly improved by considering machine learning algorithms. We believe that our work can alleviate the IT infrastructure constraints that the healthcare organizations can have to ensure the PHT execution whilst respecting the PHT approach principles as much as possible.

8.1 Answer to the Research Questions

RQ1: How can we keep private information protected when the PHT approach is run in a public cloud?

Private data can be securely stored and accessed in a cloud provider by implementing appropriate procedures, safeguards, policies and following regulations for data security and privacy like the GDPR, which sets the security guidelines and responsibilities that the Station Owner and the cloud provider must follow. To further protect the integrity and confidentiality of sensitive data in the cloud, we need to use multiple user-enforced security levels to restrict access, such as multi-factor authentication between the Data Station and the cloud, and identity management, roles and permissions among the resources in the cloud. Moreover, we can limit access through network configuration using access lists that allow only specific ports to communicate with the Data Station. Lastly, encrypting the data at various points. When it is transferred from the Data Station to the cloud besides using a secure transport protocol, and when the data are already in the cloud using Advanced Encryption Standard AES or any advanced encryption mechanism offered by the cloud provider.

RQ2: Which are the relevant technologies/tools to execute processes automatically and migrate data to the cloud dynamically?

We can leverage the cloud and Infrastructure as Code technologies for building a Staging Data Station. We identified four general elements that are crucial. Using a dynamic platform, we can use theoretically unlimited resources to execute our Train. The dynamic platform solves the problem of limited resources. Therefore, to provide automated deployment, we can use provisioning tools via Infrastructure as Code to define the Staging site desired state specified in advanced in a piece of code. The data migration can be configured inside the same definition files, facili-

tating the entire workflow. The APIs allow the communication between the various architecture components inside the Staging site and between the Data Station and the cloud. The Station Owner should create the definition files that describe the infrastructure resources through a provisioning tool that, via an API, instructs the dynamic platform to create and manage the infrastructure resources. Finally, we used event-based services for full automation so that events can trigger specific workflow tasks instead of triggering them manually.

RQ3: How does the modification of the PHT approach impact its principles?
SQ1: To what extent the integrity of PHT suffers from this modification?

The PHT approach deals with the problem of sharing sensitive data. The PHT approach's rationale is that instead of requesting and receiving data, we expect to ask a question and receive only an answer but not the data used to answer. The modification of the current PHT architecture and proofs of concepts by adding the Staging site does not imply a considerable impact on the integrity as sensitive data remain within the healthcare organization's control, and the Train Owner never has access to them. Moreover, the Staging site is part of the PHT Architecture, respecting the PHT approach's rationale of asking a question and receive an answer. The Staging site's process should be transparent for the Train Owner, giving us greater certainty that the data will be secure if we follow the techniques indicated in RQ1. However, there is always a small risk that the data can be intercepted or stolen in using the cloud; for that reason, the Staging site is not the first environment recommended to be used, but if it is needed, it adheres to the PHT principles.

Main RQ: How to implement a Staging Data Station dynamically in the Cloud while keeping private information protected?
We designed and presented a Staging Data Station reference architecture that employs APIs to communicate the various architecture components inside the Staging site and between the Data Station and the cloud. To provide the automated deployment of this architecture, we conclude we can use provisioning tools via Infrastructure as Code to define the Staging site desired state, and we can use event-based services for full automation so that events can trigger specific workflow tasks while keeping private data protected by implementing appropriate encryption mechanisms, regulation and policies compliance, and limiting access to the cloud resources.

8.2 Limitations

The present research still has some limitations. We acknowledge that some components of our work could be susceptible to discussion.PHT is a novel solution proposed for the recent development around distributed learning concepts. Consequently, more extensive coverage across the literature of the reference architecture is lacking. The few current implementations followed very different methods; therefore, we developed our architecture, starting from unsettled arguments that could weaken the design if someone is not familiar with those implementations.

Another limitation is that we tested only one type of Train, a Container Train, deploying only the Staging site. While we tested a working implementation of the Staging Data Station using a real cloud environment with open-source tools and validated our architecture with a case study, implementation and integration of the complete architecture should be developed to assess whether the design is general enough. The different types of Trains should be tested, including machine learning algorithms in the Container Train.

We validated our solution with a case study that worked adequately with a simple aggregation algorithm. This research can be extended by including different types of data analysis, such as machine learning data training. Besides, the validation process can be reinforced with expert interviews that were not possible in this research because the work was not done in direct association with any of the organizations involved in the PHT project. A more sophisticated case study could be developed if the research could have been done with an organization involved to have better inputs and integrate the proposal with current proof of concepts.

8.3 Future Work

The limitations we described in the previous section could potentially steer future works. The vast majority of healthcare organizations should be better integrated to achieve the Internet of health data in the future. Most current implementations have used solutions that are not entirely interoperable with any computing infrastructure and cannot be used by everyone else. Therefore, we need to provide and test a more robust solution to run and operate in any environment without requiring much manual intervention. Future work must then strive to define mechanisms to automate as many processes as possible.

Future research should be further testing the solution by including machine

learning algorithms in the Train or dependent transactions; for instance, they can experience idle moments waiting for input data. The architecture should then be tested to measure how it behaves and adapts the architecture if necessary. As explained in the Section 2.3.1, various Train interactions are supported, ranging from messages to container execution, including API calls and data queries. Other trains with different interactions should be created and tested in the current architecture. Future work should further implement the extended architecture proposed in this research and evaluate the entire workflow.

Last, we propose as future work to assess the solution developed in this research, integrating the implementation to current proof of concepts developed by organizations in the PHT project. Some of these implementations already have deployed a vast majority of the PHT workflow and have elaborated more robust case studies. It would be ideal if the work can be joined and tested to measure the behaviour and measure other metrics like performance and execution time since the end-user sends the Train until the results are gotten.

Appendix A

Creating Cloud resources and an API

A.1 Creating Cloud resources with Terraform

Cloud provider

Terraform provider is used to interacting with the APIs and expose the resources from the corresponding dynamic platform. In this use case, the AWS provider is used for provisioning all the resources necessary. Besides, we choose the closest AWS region, Frankfurt, to comply with regulations. Figure A.1 illustrates the file for configuring the Cloud provider.

```
provider "aws" {
    region = "eu-central-1"
}
```

Figure A.1: Terraform file for configuring the Cloud provider.

Input/Output Bucket

S3 input bucket is used to stage the data. It is private, so the uploader (Healthcare Organization) needs to be authorized. Encryption is enabled at rest with the algorithm AES256. The force_destroy allows destroying the resource without the need to delete the stored files first [41]. Figure A.2 depicts the configuration. Output bucket to store the results, figure A.3 illustrates the configuration.

```
resource "aws_s3_bucket" "staging"{
    bucket = "staging-data-station"
    force_destroy = true
    acl = "private"

    server_side_encryption_configuration {
        rule {
            apply_server_side_encryption_by_default {
                sse_algorithm = "AES256"
            }

        }
    }
}
```

Figure A.2: Terraform file for creating input bucket.

```
resource "aws_s3_bucket" "output_bucket"{
    bucket = "output-pht-data"
    force_destroy = true
    acl = "private"

    server_side_encryption_configuration {
        rule {
            apply_server_side_encryption_by_default {
                sse_algorithm = "AES256"
            }

        }
    }
}
```

Figure A.3: Terraform file for creating output bucket.

Log Bucket

CloudTrail provides an event history of the AWS account activity in the region. Therefore S3 bucket for the Cloudtrail data is required; we referred to it as a log bucket. The policy allows CloudTrail to write to this bucket. Figure A.4 illustrates the configuration.

```
resource "aws_s3_bucket" "cloudtrail"{
    bucket = "pht-staging-cloudtrail"
    force_destroy = true

    policy = <<POLICY
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AWSCloudTrailAclCheck",
            "Effect": "Allow",
            "Principal": {
              "Service": "cloudtrail.amazonaws.com"
            },
            "Action": "s3:GetBucketAcl",
            "Resource": "arn:aws:s3:::pht-staging-cloudtrail"
        },
        {
            "Sid": "AWSCloudTrailWrite",
            "Effect": "Allow",
            "Principal": {
              "Service": "cloudtrail.amazonaws.com"
            },
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::pht-staging-cloudtrail/*",
            "Condition": {
                "StringEquals": {
                    "s3:x-amz-acl": "bucket-owner-full-control"
                }
            }
        }
    ]
}
POLICY
}
```

Figure A.4: Terraform file for creating the log bucket.

CloudTrail

The configuration depicted in Figure A.5 creates a new CloudTrail and instructs it to report what is going on with objects in the uploads bucket. Any activity in the S3 input bucket called Staging bucket will be available as a CloudWatch event [42].

Data Transfer

We exploit the **depends_on** meta-argument provided to express dependencies from the Train. In this case, all the data are transferred once the Train has been already

```
resource "aws_cloudtrail" "staging"{
    name = "pht-staging-cloudtrail-transfer"
    s3_bucket_name = aws_s3_bucket.cloudtrail.id
    s3_key_prefix = "staging"

    event_selector {
        read_write_type = "All"
        include_management_events = false

        data_resource{
            type = "AWS::S3::Object"
            values = ["${aws_s3_bucket.staging.arn}/"]
        }
    }
}
```

Figure A.5: Terraform file for creating a CloudTrail resource.

routed to the Train Registry. Besides, to prevent corrupted files, we verify the integrity of the uploaded data with MD5 checksum. The configuration iterates over each file in the path specified. Figure A.6 depicts the configuration.

```
resource "aws_s3_bucket_object" "object"{
    depends_on = [module.ecr_docker_build]
    for_each = fileset ("/DataStation/", "*")
    bucket = aws_s3_bucket.staging.id
    acl = "private"
    key = each.value
    source = "/DataStation/${each.value}"
    etag= filemd5 ("/DataStation/${each.value}")
}
```

Figure A.6: Terraform file for uploading a file to a bucket.

Event rule

This setup a rule that listens for the events stored in the CloudTrail, when files are uploaded to the S3 bucket, triggers an ECS job [43]. Figure A.7 illustrates the configuration.

Computing Environment

Figure A.8 depicts the definition of the computing components. The cluster for running containers, the container registry, and we used an available module [44] to push the train from the Data Station to the registry.

```
resource "aws_cloudwatch_event_rule" "uploads"{
    name = "staging-capture-uploads"
    description = "Capture S3 events when files are uploaded"

    event_pattern = <<PATTERN
{
  "source": [
      "aws.s3"
  ],
  "detail-type":[
      "AWS API Call via CloudTrail"
  ],
  "detail": {
      "eventSource":[
          "s3.amazonaws.com"
      ],
      "eventName": [
          "PutObject",
          "CompleteMultipartUpload"
      ],
      "requestParameters": {
          "bucketName": [
              "staging-data-station"
          ]
      }
  }
}
PATTERN
}
```

Figure A.7: Event-rule.

```
resource "aws_ecs_cluster" "dataStation"{
    name = "staging-data-station"
}

resource "aws_ecr_repository" "train_repository" {
    name = "train-repository"
}

module "ecr_docker_build" {
  source = "github.com/onnimonni/terraform-ecr-docker-build-module"
  dockerfile_folder = "${path.module}/dataframe"
  docker_image_tag = "latest"
  aws_region = "eu-central-1"
  ecr_repository_url = aws_ecr_repository.train_repository.repository_url
}
```

Figure A.8: Cluster and Train.

Figure A.9 shows the networking configuration. We created a VPC with a cidr_block: *10.0.0.0/16* that allows us to use the IP address that start with "*10.0.X.X*". We cre-

ated subnets with cidr_block:10.0.1.0/24, 10.0.2.0/24 and 10.0.3.0/24.

```
resource "aws_vpc" "pht_vpc"{
    cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "pht_subnet_a"{
    vpc_id = aws_vpc.pht_vpc
    cidr_block = "10.0.1.0/24"
    availability_zone= "eu-central-1a"
}

resource "aws_subnet" "pht_subnet_b"{
    vpc_id = aws_vpc.pht_vpc
    cidr_block = "10.0.2.0/24"
    availability_zone= "eu-central-1b"
}

resource "aws_subnet" "pht_subnet_c"{
    vpc_id = aws_vpc.pht_vpc
    cidr_block = "10.0.3.0/24"
    availability_zone= "eu-central-1c"
}

resource "aws_subnet" "default_subnet_efs"{
    availability_zone= "eu-central-1c"
}
```

Figure A.9: Networking definition.

Figure A.10 depicts the **task definition** configuration that runs in the ECS Cluster to indicate how to behave. The task_role permits to call other AWS services. The execution_role permits to get images and also permits to publish in CloudWatch.

Figure A.11 shows the target definition.The target is the glue between the ECS task definition and the event rule. It tells CloudWatch to run the ECS job when the rule matches an event.

Identity Access Management

Figure A.12 shows the configuration of the role and policy to allow Cloudwatch to act on behalf of the Station Owner to launch events to run an ECS Task.

Figure A.13 shows the task execution role and the policy to allow pulling a container image from the Train Registry. It grants the ECS cluster and Fargate agents permission to make AWS API calls on behalf of the Station Owner.

```
resource "aws_ecs_task_definition" "taskDefinition"{
    family = "pht-staging-task"
    container_definitions = <<DEFINITION
    [
        {
            "name": "staging-environment",
            "image": "${aws_ecr_repository.train_repository.repository_url}",
            "essential": true,
            "cpu": 256,
            "memory": 512
        }
]
DEFINITION
    task_role_arn = aws_iam_role.ecsTaskRole.arn
    execution_role_arn = aws_iam_role.ecsTaskExecutionRoles.arn
    network_mode = "awsvpc"
    cpu = 256
    memory = 512
    requires_compatibilities = ["FARGATE"]
}
```

Figure A.10: Task definition.

```
resource "aws_cloudwatch_event_target" "uploads"{
    target_id = "staging-process-uploads"
    arn = aws_ecs_cluster.dataStation.arn
    rule = aws_cloudwatch_event_rule.uploads.name
    role_arn = aws_iam_role.uploads_events.arn

    ecs_target {
        launch_type = "FARGATE"

        task_count = 1
        task_definition_arn = aws_ecs_task_definition.taskDefinition.arn
        network_configuration {
            subnets = [aws_subnet.pht_subnet_a.id, aws_subnet.pht_subnet_b.id, aws_subnet.pht_subnet_c.id, aws_default_subnet.default_subnet_efs.id]
            security_groups = [aws_security_group.service_security_group.id]
            assign_public_ip = true
        }
    }
}
```

Figure A.11: CloudWatch target.

Figure A.14 depicts the configuration to create a role and a policy that can send data to the S3 Output bucket.

Publisher-Subscriber

Figure A.15 shows the configuration to create a publisher-subscriber topic, the policy and the subscriber that receives notifications once the Train execution finishes.

```
resource "aws_iam_role" "uploads_events"{
    name = "staging-uploads-events"

    assume_role_policy = <<POLICY
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Principal":{
                "Service": "events.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
POLICY
}

resource "aws_iam_role_policy" "ecs_events_run_task_with_any_role"{
    name = "staging-uploads-run-task-with-any-role"
    role = aws_iam_role.uploads_events.id

    policy = <<POLICY
{
    "Version": "2012-10-17",
    "Statement": [
        {
           "Effect": "Allow",
           "Action": "iam:PassRole",
           "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "ecs:RunTask",
            "Resource": "${replace(aws_ecs_task_definition.taskDefinition.arn,"/:\\d+$/", ":*")}"
        }
    ]
}
POLICY
}
```

Figure A.12: IAM for Cloudwatch.

A.2 Creating an API with NodeJS and Express

Figure A.16 depicts the Data Station API. Child processes are used to call other codes written in other languages such as bash, where terraform instructions (init, apply and destroy) are configured, and Python for downloading the results. The GET request method invokes the terraform files, and the POST method subscribes to receive notifications from the cloud and downloads the results when it is invoked.

```
resource "aws_iam_role" "ecsTaskExecutionRoles" {
    name = "ecsTaskExecutionRoles"
    assume_role_policy = data.aws_iam_policy_document.assume_role_policy.json
}

data "aws_iam_policy_document" "assume_role_policy" {
    statement {
        actions = ["sts:AssumeRole"]

        principals {
            type = "Service"
            identifiers = ["ecs-tasks.amazonaws.com"]
        }
    }
}

resource "aws_iam_role_policy_attachment" "ecsTaskExecutionRole_policy"{
    role = aws_iam_role.ecsTaskExecutionRoles.name
    policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"
}
```

Figure A.13: IAM for task execution.

```
resource "aws_iam_role" "ecsTaskRole"{
    name = "ecs-task-role"

    assume_role_policy = <<POLICY
{
    "Version": "2012-10-17",
        "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Principal": {
                "Service": "ecs-tasks.amazonaws.com"
        },
        "Effect": "Allow",
        "Sid": ""
      }
    ]
}
POLICY
}

resource "aws_iam_role_policy_attachment" "taskS3"{
    role = aws_iam_role.ecsTaskRole.name
    policy_arn = "arn:aws:iam::aws:policy/AmazonS3FullAccess"
}
```

Figure A.14: IAM for sending data to the bucket.

```
resource "aws_sns_topic" "topic"{
    name = "pht"
}

resource "aws_sns_topic_policy" "policy"{
    arn = aws_sns_topic.topic.arn
    policy = <<POLICY
{
    "Version":"2012-10-17",
    "Statement":[{
        "Effect": "Allow",
        "Principal": {"Service":"s3.amazonaws.com"},
        "Action": "SNS:Publish",
        "Resource":  "${aws_sns_topic.topic.arn}",
        "Condition":{
            "ArnLike":{"aws:SourceArn":"${aws_s3_bucket.output_bucket.arn}"}
          }
      }]
}
POLICY
}

//Subscriber attached to the above topic
resource "aws_sns_topic_subscription" "http"{
    endpoint = "https://rotten-skunk-4.loca.lt"
    endpoint_auto_confirms = true
    protocol = "https"
    topic_arn = aws_sns_topic.topic.arn
}
```

Figure A.15: SNS.

```
const express = require('express');
const request = require('request');
const {spawn} = require ('child_process');
const router = express.Router();
var bodyParser = require('body-parser')

var app = express();

app.use(bodyParser.text());

app.use(router);
router.get('/', function(req,res){
    res.send('RUNNING');
    const terraform= spawn('./pht.sh');
});

app.use(router);
router.post('/',bodyParser.text(),handleSNSMessage);

var handleSubscriptionResponse = function (error, response) {
    if (!error && response.statusCode == 200) {
        console.log('YES! SUCCESSFUL CONFIRMATION, accepted the confirmation from AWS');
    }
    else{
        throw new Error(`NO! FAIL, unable to subscribe to given URL from AWS`);
            //console.error(error)
    }
}



async function handleSNSMessage(req, resp, next) {
    try {
        let payloadStr = req.body
        payload = JSON.parse(payloadStr)
        console.log(JSON.stringify(payload))
        if (req.header('x-amz-sns-message-type') === 'SubscriptionConfirmation') {
            const url = payload.SubscribeURL;
            await request(url, handleSubscriptionResponse)
        }else if (req.header('x-amz-sns-message-type') === 'Notification') {
            console.log('RECEIVED');
            const childPython = spawn ('python', ['./down.py']);
            childPython.on('close', (code) => {
                console.log('child process close all stdio with code');
            });
        } else {
            throw new Error(`Invalid message type ${payload.Type}`);
        }
    }catch (err) {
        console.error(err)
        resp.status(500).send('ERROR')
    }
    resp.send('OK')
}

app.listen (3000);
console.log('app listening on http://localhost:3000');
```

Figure A.16: API.

Bibliography

[1] L. B. D. S. SANTOS, “Farm data train blueprints,” 2018.

[2] Y. Brikman, *Terraform: Up & Running: Writing Infrastructure as Code*. O’Reilly Media, 2019.

[3] E. Czeizler, W. Wiessler, T. Koester, M. Hakala, S. Basiri, P. Jordan, and E. Kusela, “Using federated data sources and varian learning portal framework to train a neural network model for automatic organ segmentation,” *Physica Medica*, vol. 72, pp. 39–45, 2020.

[4] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne *et al.*, “The fair guiding principles for scientific data management and stewardship,” *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016.

[5] K. Morris, *Infrastructure as code: managing servers in the cloud.* " O’Reilly Media, Inc.", 2016.

[6] “Migrate to amazon ec2.” [Online]. Available: https://docs.aws.amazon.com/ec2/index.html?nc2=h_ql_doc_ec2#amazon-ec2-auto-scaling

[7] “High performance computing (hpc) on azure.” [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/topics/high-performance-computing

[8] “Using clusters for large-scale technical computing in the cloud.” [Online]. Available: https://cloud.google.com/solutions/using-clusters-for-large-scale-technical-computing

[9] L. R. de Carvalho and A. P. F. de Araujo, “Performance comparison of terraform and cloudify as multicloud orchestrators,” in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 380–389.

[10] "Aws." [Online]. Available: https://aws.amazon.com/?nc2=h_lg

[11] S. Kumar and M. Singh, "Big data analytics for healthcare industry: impact, applications, and tools," *Big Data Mining and Analytics*, vol. 2, no. 1, pp. 48–57, 2019.

[12] A. Choudhury, J. van Soest, S. Nayak, and A. Dekker, "Personal health train on fhir: A privacy preserving federated approach for analyzing fair data in healthcare," in *Machine Learning, Image Processing, Network Security and Data Sciences*, A. Bhattacharjee, S. K. Borgohain, B. Soni, G. Verma, and X.-Z. Gao, Eds. Singapore: Springer Singapore, 2020, pp. 85–95.

[13] N. Rieke, J. Hancox, W. Li, F. Milletari, H. Roth, S. Albarqouni, S. Bakas, M. N. Galtier, B. Landman, K. Maier-Hein *et al.*, "The future of digital health with federated learning," *arXiv preprint arXiv:2003.08119*, 2020.

[14] O. Beyan, A. Choudhury, J. van Soest, O. Kohlbacher, L. Zimmermann, H. Stenzhorn, M. R. Karim, M. Dumontier, S. Decker, L. O. B. da Silva Santos *et al.*, "Distributed analytics on sensitive medical data: The personal health train," *Data Intelligence*, vol. 2, no. 1-2, pp. 96–107, 2020.

[15] Y. Ahn and Y. Kim, "Auto-scaling of virtual resources for scientific workflows on hybrid clouds," in *Proceedings of the 5th ACM workshop on Scientific cloud computing*, 2014, pp. 47–52.

[16] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS quarterly*, pp. 75–105, 2004.

[17] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of management information systems*, vol. 24, no. 3, pp. 45–77, 2007.

[18] M. Karim, B.-P. Nguyen, L. Zimmermann, T. Kirsten, M. Löbe, F. Meineke, H. Stenzhorn, O. Kohlbacher, S. Decker, O. Beyan *et al.*, "A distributed analytics platform to execute fhir-based phenotyping algorithms," 2018.

[19] Z. Shi, I. Zhovannik, A. Traverso, F. J. Dankers, T. M. Deist, P. Kalendralis, R. Monshouwer, J. Bussink, R. Fijten, H. J. Aerts *et al.*, "Distributed radiomics as a signature validation study using the personal health train infrastructure," *Scientific data*, vol. 6, no. 1, pp. 1–8, 2019.

[20] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011.

[21] Z. Mahmood, *Fog computing: Concepts, frameworks and technologies*. Springer, 2018.

[22] "2020 magic quadrant for cloud infrastructure platform services." [Online]. Available: https://pages.awscloud.com/GLOBAL-multi-DL-gartner-mq-cips-2020-learn.html

[23] B. Jin, S. Sahni, and A. Shevat, *Designing Web APIs: Building APIs That Developers Love*. " O'Reilly Media, Inc.", 2018.

[24] M. Medjaoui, E. Wilde, R. Mitra, and M. Amundsen, *Continuous API Management: Making the right decisions in an evolving landscape*. O'Reilly Media, 2018.

[25] B. Christudas, *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*. Apress, 2019.

[26] "Introduction to messaging for modern cloud architecture." [Online]. Available: https://aws.amazon.com/blogs/architecture/introduction-to-messaging-for-modern-cloud-architecture/

[27] "Introduction to terraform." [Online]. Available: https://www.terraform.io/intro/index.html

[28] A. Jochems, T. M. Deist, J. Van Soest, M. Eble, P. Bulens, P. Coucke, W. Dries, P. Lambin, and A. Dekker, "Distributed learning: developing a predictive model based on data from multiple hospitals without data leaving the hospital–a real life proof of concept," *Radiotherapy and Oncology*, vol. 121, no. 3, pp. 459–467, 2016.

[29] T. M. Deist, F. J. Dankers, P. Ojha, M. S. Marshall, T. Janssen, C. Faivre-Finn, C. Masciocchi, V. Valentini, J. Wang, J. Chen *et al.*, "Distributed learning on 20 000+ lung cancer patients–the personal health train," *Radiotherapy and Oncology*, vol. 144, pp. 189–200, 2020.

[30] T. M. Deist, A. Jochems, J. van Soest, G. Nalbantov, C. Oberije, S. Walsh, M. Eble, P. Bulens, P. Coucke, W. Dries *et al.*, "Infrastructure and distributed learning methodology for privacy-preserving multi-centric rapid learning health care: eurocat," *Clinical and translational radiation oncology*, vol. 4, pp. 24–31, 2017.

[31] C. Yang, Q. Huang, Z. Li, K. Liu, and F. Hu, "Big data and cloud computing: innovation opportunities and challenges," *International Journal of Digital Earth*, vol. 10, no. 1, pp. 13–53, 2017.

[32] M. Erder and P. Pureur, "Chapter 6 - validating the architecture," in *Continuous Architecture*, M. Erder and P. Pureur, Eds.

Boston: Morgan Kaufmann, 2016, pp. 131–159. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128032848000063

[33] ——, "Chapter 3 - getting started with continuous architecture: Requirements management," in *Continuous Architecture*, M. Erder and P. Pureur, Eds. Boston: Morgan Kaufmann, 2016, pp. 39–62. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128032848000038

[34] P. Voigt and A. Von dem Bussche, "The eu general data protection regulation (gdpr)," *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.

[35] "Iso 25000, iso/iec 25010." [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[36] "Navigating gdpr compliance on aws." [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/navigating-gdpr-compliance/navigating-gdpr-compliance.pdf#document-revisions

[37] "Regions, availability zones, and local zones." [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html

[38] "Aws free tier." [Online]. Available: https://aws.amazon.com/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc

[39] J. Walonoski, S. Klaus, E. Granger, D. Hall, A. Gregorowicz, G. Neyarapally, A. Watson, and J. Eastman, "Synthea™ novel coronavirus (covid-19) model and synthetic data set," *Intelligence-based medicine*, vol. 1, p. 100007, 2020.

[40] N. Patki, R. Wedge, and K. Veeramachaneni, "The synthetic data vault," in *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2016, pp. 399–410.

[41] "Resource: aws_s3_bucket." [Online]. Available: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/s3_bucket

[42] "Resource: aws_cloudtrail." [Online]. Available: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/cloudtrail

[43] "S3 event based trigger mechanism to start ecs fargate tasks without lambda." [Online]. Available: https://medium.com/@piyalikamra/s3-event-based-trigger-mechanism-to-start-ecs-far-gate-tasks-without-lambda-32f57e

[44] “Ecr push module.” [Online]. Available: https://github.com/onnimonni/terraform-ecr-docker-build-module