Agendas/Alianças Verdes para a Reindustrialização
02-C05-i01.02-2022.PC644874240-00000016

Work Packages 1

PROJECT #1
Architecture of MaaS

P1_E3
Architecture and technologies MaaS Part3
Physical View

PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU

Executive Summary

This report presents the architectural design and physical deployment strategy for the Mobility-as-a-Service (MaaS) Platform, with a focus on the IT perspective within a multi-viewpoint architectural framework, following the prior development architecture phase, documenting the transition from logical service design to physical deployment, aligning technical decisions with a scalable, resilient, high-performance platform's operational needs.

The approach centers on the physical view of the architecture, as defined in Kruchten's 4+1 model, emphasizing the mapping of software components to execution environments across compute, storage, and network resources, particularly critical for microservice-based platforms, where flexibility and modular deployment are essential. The design process followed an iterative methodology, incorporating the conclusions and recommendations of the national strategy for smart territories concerning constructing and operationalizing an Urban Platform, cloud provider service selection, orchestration strategies, and deployment optimizations to meet non-functional requirements such as scalability, performance, and availability. The result is a detailed deployment architecture using Unified Modeling Language representation to illustrate system distribution and runtime configurations, emphasizing open-source technologies to support both cloud-based and on-premises deployment scenarios. These architectural descriptions are the projections from a unified model for the physical view, providing concrete guidance on selecting and configuring components at each architectural layer, ensuring consistency with the platform's broader technical and business goals, and establishing a comprehensive blueprint for the physical realization of the MaaS Platform. Its Data Model Analysis refines entity relationships using FIWARE and GTFS-based models, using NGSI-LD-compliant Smart Data Models to support semantic interoperability. These models guide the creation of service-specific schemas deployed via FIWARE and other open-source technology, enhancing scalability and modularity. Such a resulting architecture enables a modular, scalable, and resilient platform, leveraging microservice principles and cloud-native practices through container-based orchestration, open-source technology choices, and multi-environment configurations, and is thus capable of supporting the evolving demands of urban mobility ecosystems.

Contents

List of Figures

List of Tables

Table of contributions

Chapter	Specific Content	Contributions		
		UMinho	CCG	CEiiA
	First version	X	X	

1 Introduction

This document presents the architectural design approach and the resulting physical architecture for developing and deploying the Mobility-as-a-Service (MaaS) Platform, focusing on the IT perspective within a multi-viewpoint architectural framework. The work detailed herein builds upon the results of the previous development architecture phase, strongly aligned with the reference architecture of the National Strategy for Smart Territories [1], and provides an explicit mapping between logical components and their physical realizations, supporting deployment in both development and production environments.

The architectural approach is leveled in the physical view of the system, in line with Kruchten's 4+1 architectural model [2], focusing on the relationship between software components and their execution environments, describing how services are deployed across hardware resources such as computing nodes, storage systems, and networks. This perspective is especially relevant in microservice-based platforms, where the dynamic nature of services requires flexible, scalable, and resilient deployment strategies.

The technical architecture defines the elements required to support business services, covering the IT infrastructure, communication networks, processing capabilities, and adherence to relevant standards. These components are then instantiated in a physical architecture specifying the actual configuration and deployment of software elements across physical and virtualized resources—tailored to different usage scenarios, including development and multi-stakeholder production sites.

To address the non-functional requirements—scalability, performance, and availability—the document follows an iterative design process, encompassing the selection and organization of cloud provider services, definition of orchestration strategies, and optimization of deployment topologies. The architectural blueprint uses UML deployment and component diagrams to reflect the distributed and modular nature of the platform.

The physical architecture section demonstrates the application of the followed IT approach, providing detailed recommendations for open-source-based components at the computing, storage, and communication levels. It defines the technological instantiations aligned with each architecture layer presented in the previous development architecture and presents the deployment structure, including container orchestration, database placement, network configurations, and runtime environments. Also, a conceptual data model is developed and further detailed into a logical and physical data model, leveraging NGSI-LD mappings and JSON-LD schemas to support interoperability and implementation within the platform's deployment architecture. Thus, these mappings ensure that the MaaS Platform's architecture is adaptable and ready to meet the evolving requirements of urban mobility services.

2 Approach to architectural design

2.1 IT Perspective

The IT infrastructure comprises the technical, deployment, and physical architectures from a Viewpoint framework. The technical architecture provides the technical components that enable the business strategies and functions (D. Chen et al., 2008) [3] to support the deployment of services in terms of IT infrastructure, networks, communications, processing, and standards (Booch, 2010) [4], usually seen as the physical view of Kruchten's 4+1 framework. A physical architecture describes the system's physical layout, revealing which pieces of software run in what pieces of hardware (Fowler, 2004) [5].

Focusing on the physical view from the IT perspective involves mapping the software to the hardware, which primarily takes the previously analyzed non-functional requirements of the platform. The various components need to be identified and mapped onto the various nodes (e.g., Devices and Execution Environments), using different physical configurations, ones for development and testing and others for the various sites of different stakeholders for production. The connectors must map to the communication medium, local or wide networks, bus, and containers or packages mapped to physical subsystems.

The notation for our physical blueprint will follow the UML [6] deployment diagram or component diagram to reflect its distributed aspect. As per Kruchten's advice, not all software architecture needs the full 4+1 views, and some can be omitted, for example, due to being very small systems. However, this view is of great interest because of the increased complexity of many dynamic parts, a characteristic of microservices. Services may be realized by implementations that can run in several execution environments. Separating the logical or development implementation from its possible physical realization on various platforms keeps the services models simpler and more resilient to changes in underlying execution environments, supporting a variety of technology implementations. This report, therefore, aims to define the implementation of the technology and describe a logic to support the automation of these technological mappings, proposing options for physical realization in development and production environments.

Through an iterative process of sketching, organizing, specifying, and optimizing, based on the results of previous development architecture analysis, and at the same time selecting a cloud provider and its services catalog, among the design decisions taken during the logic architecture conception, the report presents a deployment architecture, also having in mind the main concerns of scalability, performance, and availability.

3 Physical Architecture

This section describes the decisions to select application components from the cloud provider's catalog at the compute, storage, and communication level to determine concerns of the platform design, namely scalability, performance, and fault tolerance requirements. We highlight the technological instantiation of the components identified in each architecture layer, with recommendations as to which typologies/technologies to use for each component. These are summarized in the table technologies selection.

This section additionally describes software deployment on hardware nodes, focusing on hardware requirements described in the orchestration and physical configuration subsection. Finally, it presents the deployment architecture, showing where components are deployed and the context of runtime platforms, networks, and mapping components to infrastructure (containers and databases).

3.1 Technologies Selection

Table 1 - Technological instantiation of the components identified in the architecture

Layer	Component	Package	Recommendation
Interoperability	Collection Interface	{P20} Traffic Real-Time Interface	For custom solutions, select Python, Node.js, or .NET Core. For solutions with no-code/low-code, select solutions like Apache Nifi, providing built-in processors for REST APIs (GET, POST). To implement the API Gateway, select Fiware-specific building blocks like Wilma PEP Proxy or KrakenD (used unofficially). Other lightweight setups with Fiware are Nginx and API Umbrella.
		{P33} Stratify Location Real-Time Interface	
		{P17} Parking Real-time Interface	
		{P37} TSP Static Interface	
		{P12} TSP Real-Time Interface	
		{} Public Collector API Gateway	
	Sharing Interface	{P16} Sharing Data Interface	
		{P29} User Location Real-Time Interface	
		{P7} SaaS Interface	
		{P6} Ticketing Validation Interface	
		{} Public Sharing API Gateway	
		{} Web API Gateway	
		{} Mobile API Gateway	
Data Processing	Context Broker	{P10} Real-Time Processor	NGSI-LD compatibility. Technology suggestion: Orion Context Broker from FIWARE.
		{P8} SaaS Information	
	History	{P11} Real-Time History	Choose the solutions offered by FIWARE such as Quantumleap and Mintaka that use a time-series database like TimescaleDB.

Layer	Component	Package	Recommendation
	Adapters	{P13} TSP Adapter	To ensure data transformation into the expected normalized format, select solutions like Apache NiFi or Apache AirFlow. FIWARE also offers Draco to transform data with connectors for NGSI-LD.
		{P18} Parking Adapter	
		{P21} Traffic Adapter	
		{P32} User Adapter	
		{P34} Stratify Location Adapter	
	Machine Learning and Analytics Engines	{P31} Plan Trip Travel Processor	Solutions can range from commercial products and/or cloud based to on-premises environments in Python or R. In some cases, it may make sense to opt for AutoML solutions. The use of OpenTripPlanner (OTP) technology is also recommended.
	Specific Support Services, ETL, Event Managers, BI	{P14} TSP Processor	To ensure data transformation into the expected normalized format, select solutions like Apache NiFi or Apache AirFlow. FIWARE also offers Draco for data transformation and Perseo for rule processing and trigger actions, both with connectors for NGSI-LD. For custom solutions, select Python, Node.js, or .NET Core.
		{P19} Parking Processor	
		{P22} Traffic Processor	
		{P24} Accessibility Processor	
		{P25} Alerts processor	
		{P26} Privacy Settings Processor	
		{P28} Feedback Processor	
		{P30} User Proximity Processor	
		{P35} Stratify Location Processor	
		{P36} Map Processor	
		{P2} Payment Management	Opt for solutions based on the container per service pattern in a microservices architecture and defined per each use case, based on Python, Node.js, or .NET Core.
		{P4} Ticketing Service Management	
		{P5} User Management	
		{P9} SaaS Management	
		{P15} TSP Information	
		{P27} Feedback Information	
		{P38} Maps Information	
	Storage	{P2} Payment Management	If a database per service pattern is used, select PostgreSQL, MySQL, or MongoDB. Use SmartDataModels and OpenStreetMaps (OSM) to support the information models. If a single schema is followed, unify the private schemas and use Neo4J. And use MongoDB for TSP and maps information.
		{P4} Ticketing Service Management	
		{P5} User Management	
		{P15} TSP Information	
		{P27} Feedback Information	
		{P38} Maps Information	

Layer	Component	Package	Recommendation
Services	Dashboards	{P1} Administration Interface	Choose open-source data visualization tools like Grafana for time series analytics, or use web framework like Angular or Library React, for custom forms, charts, and dashboards.
		{P3} User Interface	
		{P23} User Registration Interface	
Support	Management and Safety	{P39} Cloud Administration	Choose cloud solutions offered by Google Cloud, Microsoft Azure or AWS, or open-source cloud management platforms (e.g., OpenStack, Apache CloudStack) with microservices orchestrators like Kubernetes/docker.
		{P40} Cloud Configuration	
	Shared Services	{} IAM	Select Keycloak for the Identity Access Management as a building block for prototyping. For production, the integration with the autenticação.gov is recommended. This integration is available to both the public and private sectors and must be filed with *AMA - Agência para a Modernização Administrativa, IP*.

3.2 Orquestration and Physical Configuration

1. Authentication and authorization components are vital in modern system architectures to ensure secure, controlled access to APIs and services. A leading open-source solution in this space is Keycloak, originally developed by Red Hat. Keycloak provides robust support for OAuth2, OpenID Connect, and SAML and integrates easily with identity providers (LDAP, Active Directory, social login). It features user management, single sign-on (SSO), multi-factor authentication, and fine-grained access control, making it highly suitable for both enterprise and public-sector applications. In the FIWARE ecosystem, the primary components for auth are Keyrock, Wilma PEP proxy, and AuthZForce. These components are open source and designed to work together, forming a full authentication and authorization stack. While Keyrock provides user and client registration, Wilma intercepts and secures API calls, and AuthZForce enforces detailed access policies. Nginx (open-source) can also act as a lightweight API gateway or reverse proxy, integrating with Keycloak or Keyrock via OAuth2 middleware (e.g., oauth2-proxy or lua-resty-openidc) to enforce token validation and forward headers downstream. For development, these components typically need one vCPU, 512 MB–1 GB RAM, and 100–500 MB disk. Production deployments require

2–4 vCPU, 2–4 GB RAM, and 1 GB+ disk, particularly when running persistent user databases or analytics features. They are all container-friendly and well-suited for Kubernetes-based cloud-native deployments, where they can be horizontally scaled and integrated with ingress controllers and service meshes for centralized auth enforcement.

2. Open-source API gateways provide essential features such as request routing, authentication, rate limiting, logging, and service aggregation, making them a critical part of secure, scalable architectures. In the FIWARE ecosystem, Wilma PEP Proxy serves as the core API gateway for enforcing OAuth2-based access control and integrating with identity and policy services like Keyrock and AuthZForce. Additional tools like KrakenD, Traefik, and API Umbrella offer lightweight, flexible gateway options, with some being used in European smart city projects or to complement FIWARE components. OTher widely used if the NGIX (plus LUA modules), used as HTTP server with gateway capabilities via reverse proxy with Lua, and Express Gateway, a Node.js-based API gateway, developer-friendly for microservices. For development, most gateways require minimal resources: 1 vCPU, 512 MB RAM, and 100–300 MB of disk space. In production, it's recommended to provision 2 vCPU, 1–2 GB RAM, and up to 1 GB disk for high availability and moderate traffic. Lightweight gateways like KrakenD and Traefik are ideal for microservices and container-based setups, while Wilma should be used in FIWARE deployments needing fine-grained access control. Deployment is best handled through Docker or Kubernetes, enabling easy scalability and integration with other FIWARE components.

3. To ensure interoperability, openness, and compatibility in API design, the selected technology should support data sharing through open, RESTful APIs aligned with DCAT-AP standards for metadata exchange and provide NGSI-LD interfaces for accessing real-time and contextual information. APIs exposing other data types should be defined using open standards like OpenAPI, support JSON for data serialization, be openly documented, and enforce secure communication protocols (e.g., HTTPS). REST should be the primary architectural style, providing standard HTTP verb semantics and response codes to ensure interoperability across client types and programming environments.

4. For REST APIs, especially over HTTP, it is recommended to implement asynchronous I/O and patterns like async methods or producer-consumer, ensuring better scalability and responsiveness. While REST over HTTP using JSON is the standard for public and private APIs, performance testing should be conducted early to assess whether a binary protocol is needed for efficiency. Public APIs intended for web or mobile applications should adopt REST with well-defined idempotent behavior (using PUT over POST where possible) and comply with the OpenAPI specification for defining methods, parameters, and response structures.

5. The data is normalized using the NGSI-LD standard and Smart Data Models, thus guaranteeing the sharing of detailed information on data structures and their areas of operation. Using the NGSI-LD also ensures greater interoperability and efficiency from the outset, as well as semantics

associated with data types. The following smart data models are used to instantiate the database per service schemas:

Table 2 - Database per service schemas: data types

Package	Smart Data Model	Source
{P2} Payment Management	Connection	https://github.com/smart-data-models/dataModel.S4SYST/tree/master/Connection
{P4} Ticketing Service Management	KeyCardSwitch	https://github.com/smart-data-models/dataModel.OCF/tree/master/KeyCardSwitch
	FareCollectionSystem	https://github.com/smart-data-models/dataModel.Transportation/tree/master/FareCollectionSystem
	Booking	https://github.com/smart-data-models/dataModel.MarineTransport/tree/master/Booking
{P5} User management	KeyCardSwitch	https://github.com/smart-data-models/dataModel.OCF/tree/master/KeyCardSwitch
	FareCollectionSystem	https://github.com/smart-data-models/dataModel.Transportation/tree/master/FareCollectionSystem
	UserID	https://github.com/smart-data-models/dataModel.OCF/tree/master/UserID
	ResourceReport	https://github.com/smart-data-models/dataModel.OSLO/tree/master/ResourceReport
	GtfsAccessPoint	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsAccessPoint/doc/spec/index.html
	GtfsRoute	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsRoute/doc/spec/index.html
	TransportStation	https://github.com/smart-data-models/dataModel.Transportation/tree/master/TransportStation
	Activity	https://github.com/smart-data-models/dataModel.User/tree/master/Activity
	Alert	https://fiware-datamodels.readthedocs.io/en/stable/Alert/doc/spec/index.html
	System_alerts	https://github.com/smart-data-models/dataModel.GBFS/tree/master/system_alerts
{P8} SaaS Information	Connection	https://github.com/smart-data-models/dataModel.S4SYST/tree/master/Connection
	CO2	https://github.com/smart-data-models/dataModel.OCF/tree/master/CO2
{P10} Real-Time Processor	Connection	https://github.com/smart-data-models/dataModel.S4SYST/tree/master/Connection
	Vehicle	https://fiware-datamodels.readthedocs.io/en/stable/Transportation/Vehicle/Vehicle/doc/spec/index.html

	ParkingSpot	https://fiware-datamodels.readthedocs.io/en/stable/Parking/ParkingSpot/doc/spec/index.html
	TrafficFlowObserved	https://fiware-datamodels.readthedocs.io/en/stable/Transportation/TrafficFlowObserved/doc/spec/index.html
	RestrictedTrafficArea	https://github.com/smart-data-models/dataModel.Transportation/tree/master/RestrictedTrafficArea
	UserContext	https://github.com/smart-data-models/dataModel.User/tree/master/UserContext
{P11} Real-Time History	Vehicle	https://fiware-datamodels.readthedocs.io/en/stable/Transportation/Vehicle/Vehicle/doc/spec/index.html
	ParkingSpot	https://fiware-datamodels.readthedocs.io/en/stable/Parking/ParkingSpot/doc/spec/index.html
	TrafficFlowObserved	https://fiware-datamodels.readthedocs.io/en/stable/Transportation/TrafficFlowObserved/doc/spec/index.html
	RestrictedTrafficArea	https://github.com/smart-data-models/dataModel.Transportation/tree/master/RestrictedTrafficArea
	UserContext	https://github.com/smart-data-models/dataModel.User/tree/master/UserContext
{P15} TSP information	GtfsAgency	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsAgency/doc/spec/index.html
	GtfsStop	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsStop/doc/spec/index.html
	GtfsRoute	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsRoute/doc/spec/index.html
	GtfsTrip	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsTrip/doc/spec/index.html
	GtfsStopTime	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsStopTime/doc/spec/index.html
	GtfsCalendarDateRule	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsCalendarDateRule/doc/spec/index.html
	GtfsCalendarRule	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsCalendarRule/doc/spec/index.html

	GtfsFrequency	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsFrequency/doc/spec/index.html
	GtfsTransferRule	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsTransferRule/doc/spec/index.html
	GtfsAccessPoint	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsAccessPoint/doc/spec/index.html
	GtfsShape	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsShape/doc/spec/index.html
	GtfsStation	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsStation/doc/spec/index.html
	GtfsService	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsService/doc/spec/index.html
{P38} Maps information	GtfsAgency	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsAgency/doc/spec/index.html
	GtfsStop	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsStop/doc/spec/index.html
	GtfsRoute	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsRoute/doc/spec/index.html
	GtfsTrip	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsTrip/doc/spec/index.html
	GtfsStopTime	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsStopTime/doc/spec/index.html
	GtfsCalendarDateRule	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsCalendarDateRule/doc/spec/index.html
	GtfsCalendarRule	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsCalendarRule/doc/spec/index.html
	GtfsFrequency	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsFrequency/doc/spec/index.html
	GtfsTransferRule	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsTransferRule/doc/spec/index.html

	GtfsAccessPoint	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsAccessPoint/doc/spec/index.html
	GtfsShape	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsShape/doc/spec/index.html
	GtfsStation	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsStation/doc/spec/index.html
	GtfsService	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/GtfsService/doc/spec/index.html
{P17} Parking real-time Interface	ParkingSpot	https://fiware-datamodels.readthedocs.io/en/stable/Parking/ParkingSpot/doc/spec/index.html
{P20} Traffic real-time Interface	RestrictedTrafficArea	https://github.com/smart-data-models/dataModel.Transportation/tree/master/RestrictedTrafficArea
	TrafficFlowObserved	https://fiware-datamodels.readthedocs.io/en/stable/Transportation/TrafficFlowObserved/doc/spec/index.html
{P12} TSP real-time interface	Vehicle	https://fiware-datamodels.readthedocs.io/en/stable/Transportation/Vehicle/Vehicle/doc/spec/index.html
{P29} User Location Real-time Interface	ArrivalEstimation	https://fiware-datamodels.readthedocs.io/en/stable/UrbanMobility/ArrivalEstimation/doc/spec/index.html
	Vehicle	https://fiware-datamodels.readthedocs.io/en/stable/Transportation/Vehicle/Vehicle/doc/spec/index.html
	UserContext	https://github.com/smart-data-models/dataModel.User/tree/master/UserContext
	Alert	https://fiware-datamodels.readthedocs.io/en/stable/Alert/doc/spec/index.html
{P33} Stratify Location real-time Interface	CrowdFlowObserved	https://github.com/smart-data-models/dataModel.Transportation/tree/master/CrowdFlowObserved
{P27} Feedback Information	Vehicle	https://fiwaredatamodels.readthedocs.io/en/stable/Transportation/Vehi cle/Vehicle/doc/spec/index.html
	VehicleModel	https://fiwaredatamodels.readthedocs.io/en/stable/Transportation/Vehi cle/VehicleModel/doc/spec/index.html
	IssueTracking	https://github.com/smart-datamodels/dataModel.IssueTracking

6. The logical packages, or services, will be containerized, with the following initial requirements per service instance (container):

Table 3 - Initial hardware requirements per service instance

Package	Tecnology/ Framework	vCPU [vcore]	RAM [GB]	Disc [GB]
{P1} Administration Interface	Grafana/React	1	0,5	1
{P2} Payment Management	Dotnet Core	1	1	1
{P3} User Interface	Grafana/React	1	0,5	1
{P4} Ticketing Service Management	Dotnet Core	1	1	1
{P5} User Management	Dotnet Core	1	1	1
{P6} Ticketing Validation Interface	Dotnet Core	1	1	1
{P7} SaaS Interface	Apache Nifi	1	1	1
{P9} SaaS Management	Dotnet Core	1	1	1
{P10} Real-Time Processor	Orion-LD	1	2	2
{P11} Real-Time History (write)	Quantumleap	1	1	1
{P11} Real-Time History (read)	Mintaka	1	1	1
{P12} TSP Real-Time Interface	Dotnet Core	1	1	1
{P13} TSP Adapter	Apache Nifi	1	1	1
{P14} TSP Processor	Apache Nifi	1	1	1
{P15} TSP Information	Node.JS	1	1	1
{P16} Sharing Data Interface	Dotnet Core	1	1	1
{P17} Parking Real-time Interface	Dotnet Core	1	1	1
{P18} Parking Adapter	Apache Nifi	1	1	1
{P19} Parking Processor	Apache Nifi	1	1	1
{P20} Traffic Real-Time Interface	Dotnet Core	1	1	1
{P21} Traffic Adapter	Apache Nifi	1	1	1
{P22} Traffic Processor	Apache Nifi	1	1	1
{P23} User Registration Interface	React	1	0,5	1
{P24} Accessibility Processor	Dotnet Core	1	1	1
{P25} Alerts processor	Perseo	1	1	1
{P26} Privacy Settings Processor	Dotnet Core	1	1	1
{P27} Feedback Information	Dotnet Core	1	1	1
{P28} Feedback Processor	Python	1	1	1
{P29} User Location Real-Time Interface	Dotnet Core	1	1	1
{P30} User Proximity Processor	Apache Nifi	1	1	1
{P31} Plan Trip Travel Processor	Python/R	1	1	1
{P32} User Adapter	Apache Nifi	1	1	1
{P33} Stratify Location Real-Time Interface	Dotnet Core	1	1	1
{P34} Stratify Location Adapter	Apache Nifi	1	1	1
{P35} Stratify Location Processor	Apache Nifi	1	1	1
{P36} Map Processor	Apache Nifi	1	1	1

Package	Tecnology/ Framework	vCPU [vcore]	RAM [GB]	Disc [GB]
{P37} TSP Static Interface	Dotnet Core	1	1	1
{P38} Maps Information	Node.JS	1	1	1
{} IAM	Keycloak	1	0,5	1
{} IAM database	PostgreSQL	1	1	1
{P2} Payment Management Database	MySQL/PostgreSQL	1	1	1
{P4} Ticketing Service Management Database	MySQL/PostgreSQL	1	1	1
{P5} User Management Database	MySQL/PostgreSQL	1	1	1
{P8} SaaS Information Database	MySQL/PostgreSQL	1	1	1
{P10} Real-Time Processor Registration Database	MongoDB	1	2	1
{P11} Real-Time History Database	TimeScaleDB	1	2	2
{P15} TSP Information Database	MongoDB	1	1	1
{P27} Feedback Information Database	MySQL/PostgreSQL	1	1	1
{P25} Alerts processor Database	MongoDB	1	1	1
{P38} Maps Information Database	MongoDB	1	1	1
{} Public Sharing API Gateway	Nginx	1	0,5	1
{} Web API Gateway	Nginx	1	0,5	1
{} Mobile API Gateway	Nginx	1	0,5	1
{} Public Collector API Gateway	Nginx	1	0,5	1

7. For a development scenario of the application architecture based on microservices, where there is a need to start containers fast and want a small footprint per container to achieve better density per hardware unit in order to lower costs, we will use Linux docker containers, along with programming frameworks better suit developers' knowledge, running on a Virtual Machine. We envision a main data model for Real-time context data and several others for the remaining services following a pattern of database per service, each one servicing its data through its private data planes REST API, resulting in the following minimum hardware requirements for the virtual machine:

   Table 4 - Calculated virtual machine hardware requirements for the docker containers development scenario

Calc vCPU [vcore]	Calc RAM [GB]	Calc Disc [GB]	Black-box services Orion-LD, Grafana, Mintaka, QuantumLeap, Perseo, Keycloak, Apache Nifi/ Apache Airflow	Custom dotnet core, python, angular, react, nodeJS	Black-box databases MySQL, PostgreSQL, MongoDB, TimescaleDB	Total containers
50	51	52	17	22	11	50

8. Still, in a development scenario, instead of using docker containers, a more complex Kubernetes development environment is considered, designed in a local Kubernetes cluster with three nodes, one master, and 2 workers. The Etcd server will run on the main master node. And 2 nodes will be used for the workers. In this way, we obtain the following topology for a minimal development cluster instantiated on a public cloud platform on Microsoft Azure or Google Cloud, running the option with the highest expected number of containers and up to two instances per service:

Table 5 – Kubernetes cluster development scenario

Module	**AKS Node pool**	**GKE Node Pool**
Services	50	50
POD	100	100
Duration	24h / day	24h / day
Master Nodes	1 Microsoft Azure managed node	1 Google Cloud managed node
Worker Nodes	2	2
VM Master	Microsoft Azure managed	Google Cloud managed
VM Worker	Standard D2s v3: 2 vCPU, 8 GB RAM, 50 GB SSD	e2-standard-2: 2 vCPU, 8 GB RAM, 50 GB SSD

9. Considering the installation environment and fault tolerance requirements, using kubernetes for application orchestration in a microservices architecture, the minimum master nodes in a high availability configuration are 3 nodes. The Etcd servers will run on the master nodes. And 2 nodes will be used for workers. This gives us the following topology for the minimum production cluster for a public cloud platform on Microsoft Azure or Google Cloud, running the option with the highest expected number of containers, and up to two instances per service:

Table 6 - Kubernetes cluster production scenario

Module	**AKS Node pool**	**GKE Node Pool**
Services	50	50
POD	100	100
Duration	24h / day	24h / day
Master Nodes	3 Microsoft Azure managed nodes	3 Google Cloud managed nodes
Worker Nodes	2	2
VM Master	Microsoft Azure managed	Google Cloud managed
VM Worker	Standard D2s v3: 2 vCPU, 8 GB RAM, 50 GB SSD	e2-standard-2: 2 vCPU, 8 GB RAM, 50 GB SSD

10. In Kubernetes, the Ingress controller serves as an API gateway, directing

client requests to backend services through a single endpoint while supporting advanced features like SSL termination, authentication, IP restrictions, and traffic limits. It reduces communication overhead by aggregating requests and can offload common functionalities from the services themselves. Ingress controllers such as Nginx, HAProxy, Traefik, Azure Application Gateway or Google Cloud API Gateway can be deployed within an AKS cluster, and operate as an edge router or reverse proxy. As this element can always be a point of failure, at least two replicas should always be installed for fault tolerance. While self-signed TLS certificates can be used for encryption in development, production environments require certificates signed by a trusted certificate authority (CA).

3.3 Deployment Architecture

Taking into account the decisions and design criteria described in the previous sections, the following diagram present the proposed option for the deployment architecture of the development version of the MaaS Platform solution.

Figure 1 - Deployment Diagram of the MaaS Platform development scenario

3.4 Physical Data Model Analysis

Building on the previous development architecture work and the initial mapping of logical data objects to Smart Data Models, this analysis extends to explore the relationships between entities using FIWARE best practices and GTFS-based models. By reasoning through real-world physical context and functions, smart data models are categorized into four groups: transit structure, user and system, mobility entities, and access and infrastructure, revealing clusters and conceptual relationships. These insights are used to refine the service-specific database schemas and are visually represented in the following diagram.

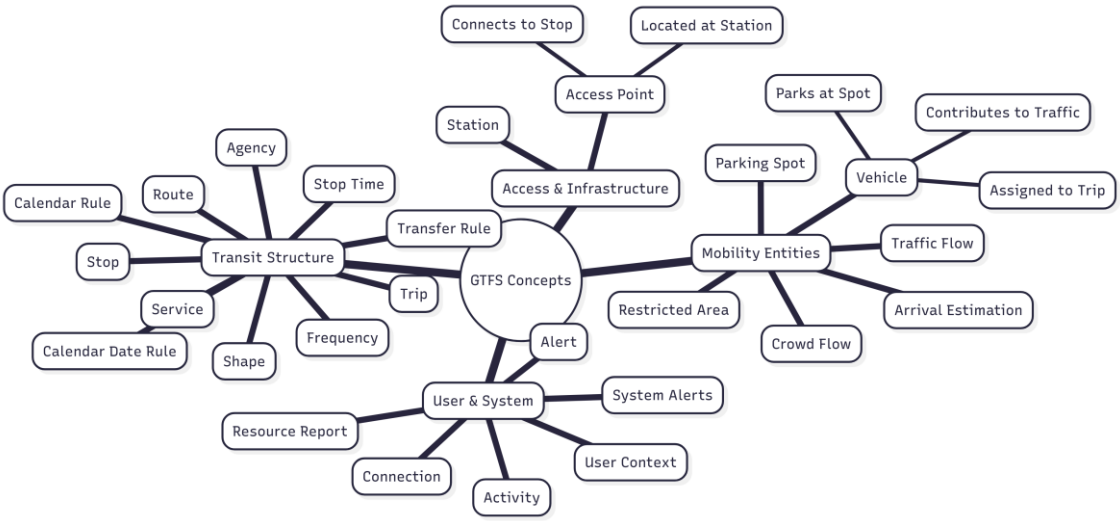

Figure 2 - Entities categorization

Using GTFS and FIWARE specifications as a foundation, we analyzed how entities interact to define associations and directional relationships, forming a conceptual data model. For instance, the entity GtfsAccessPoint is linked to GtfsStop through a "provides access to" relationship, reflecting real-world transit structures where access points connect passengers to specific stops. This logic extends to FIWARE models, which often relate entities via geolocation or functional roles. The resulting associations are summarized in a table using NGSI-LD specification practices, where relationships are expressed through action-oriented attributes defined with the usage of a verb (plus optionally an object) such as hasStop, operatedBy, hasTrip.

Table 7 - Data model structure

Entity 1	Relationship	Entity 2	Reasoning Description
GtfsService	provided by	GtfsAgency	A service (e.g. a bus line) is managed by a transit agency.
System_alerts	issued by	GtfsAgency	System alerts are created by the transit agency to notify of system-wide issues.

Entity 1	Relationship	Entity 2	Reasoning Description
GtfsAgency	manages	GtfsRoute	The agency defines and operates various routes.
GtfsRoute	includes	GtfsTrip	A route comprises multiple trips (vehicle journeys).
GtfsTrip	scheduled at	GtfsStopTime	A trip has a schedule that defines stop times.
GtfsStopTime	occurs at	GtfsStop	Each stop time occurs at a specific physical stop.
GtfsTrip	follows	GtfsShape	The trip follows a geographical shape or path.
GtfsTrip	operates by	GtfsCalendarRule	The service days of a trip are defined by calendar rules.
GtfsCalendarRule	exceptions	GtfsCalendarDateRule	Calendar rules may have date-specific exceptions.
GtfsTrip	has	GtfsFrequency	A trip may have frequency-based scheduling.
GtfsStop	connects to	GtfsTransferRule	Stops may have defined transfer connections to others.
Vehicle	assigned to	GtfsTrip	Vehicles are assigned to specific trips.
Vehicle	parks at	ParkingSpot	Vehicles may be assigned to park at specific locations.
Vehicle	enters	RestrictedTrafficArea	Vehicle movement may be restricted in certain areas.
Vehicle	contributes to	TrafficFlowObserved	Vehicle data helps inform traffic flow observations.
Vehicle	estimated by	ArrivalEstimation	Arrival estimates are calculated for the vehicle at a stop.
ArrivalEstimation	destination	GtfsStop	Arrival estimates are provided for specific stops.
GtfsAccessPoint	provides access to	GtfsStop	Access points lead passengers to/from specific stops.
GtfsAccessPoint	located at	GtfsStation	Access points are physically part of a station.
CrowdFlowObserved	observed at	GtfsStation	Crowd flow data is linked to station areas.
ResourceReport	reports on	GtfsStation	Reports provide data related to a station's status or issues.
Connection	uses	GtfsAccessPoint	Connections involve using physical access points.
UserContext	user of	Vehicle	User context reflects which vehicle the user is associated with.
Alert	affects	GtfsRoute	Alerts can impact specific transit routes.
Activity	performed by	UserContext	User activities are associated with a specific user context.

As FIWARE-based smart data models evolve, most entities now support NGSI-LD mappings, enabling JSON-LD-based interoperability for federated systems and data spaces. Applications must be capable of navigating these entity relationships through queries that access properties defined in each model. Starting from the conceptual data model, we add more details to refine it from the conceptual to the logical model. Built from smart data model schemas and the relationships just refined, the logical model defines both the structure and relationships between entities. Based on the proposed physical deployment architecture, we know exactly how to implement its databases and refine our logical model into the physical data model, which can directly map between the diagram and the actual database system, specifying how each property is stored and linked. For simplicity, the relationships and entities are illustrated in a conceptual data model diagram below, with the logical model defined using the JSON-LD smart data models schemas, modified with the relationships described previously. These definitions provide the foundation for each service's database schema when implemented in the deployment architecture's FIWARE Orion Context Broker component or other open-source storage components using a database-per-service microservice application pattern.

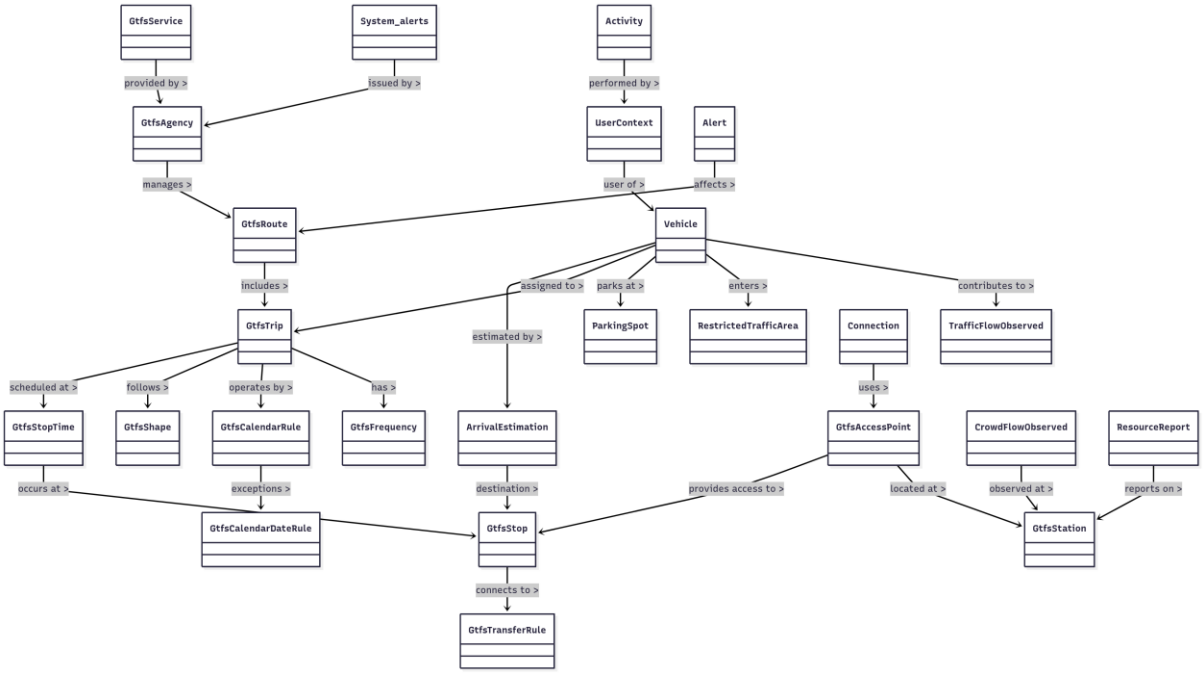

Figure 3 - Conceptual data model for the MaaS platform

4 Conclusions

This document's architectural design and physical deployment strategy provide a scalable basis for implementing the MaaS Platform from an IT perspective, establishing a clear and structured mapping between logical service components and their execution environments addressing critical non-functional requirements such as scalability, performance, availability, and fault tolerance.

The physical architecture enables modular and adaptable service deployment by leveraging the flexibility of microservice architectures and aligning with Kruchten's 4+1 framework. UML deployment diagrams facilitate the visualization of the system's distributed nature and the relationships between software components and hardware nodes. It supports an architecture that is resilient to changes in execution environments and promotes reuse across development and production stages.

The technological instantiation of the components identified in each layer of the development architecture was based on the recommendations of the National Strategy for Smart Territories, being a support tool for carrying out the construction and operationalization of Urban Platforms concerning the typologies/technologies to be used for each component. The selection of cloud provider services was guided by the platform's operational needs and aligned with best practices for cloud-native design, but with a strong effort to explore an on-premises deployment strategy based on open-source technology. They included container-based orchestration and multi-environment configurations, ensuring the platform scales efficiently and remains robust under varying loads and operational contexts. By leveraging GTFS-based models and FIWARE best practices, the data analysis identified meaningful relationships between entities, expressed using NGSI-LD semantics, guiding the development of a coherent conceptual data model, which is then refined into a logical model using JSON-LD schemas. Such an approach ensures seamless integration with FIWARE technology and other open-source storage technology supporting the microservice application patterns across the platform's deployment architecture. The report thus turns into a practical and strategic blueprint for the physical realization of a reliable and consistent technological MaaS Platform, ensuring it can support smart urban mobility scenarios.

5 Bibliography

[1] Government of Portugal, Digitalization and Administrative Modernization Government Area, Portugal Digital Mission Structure. (2023). National strategy for smart territories: Reference architecture for urban management platforms (ARPGU). Agency for Administrative Modernization (AMA)

[2] Kruchten, P. (1995). The 4+1 View Model of Architecture. IEEE Software, 12(6), 42–50. https://doi.org/10.1109/52.469759

[3] Chen, H. (2008). Towards service engineering: service orientation and business-IT alignment. In Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS). IEEE. https://doi.org/10.1109/HICSS.2008.462

[4] Booch, G. (2010). Enterprise Architecture and Technical Architecture. IEEE Software, 27(2), 96–96. https://doi.org/10.1109/MS.2010.4

[5] Fowler, M. (2004). UML distilled: a brief guide to the standard object modeling

language. Addison-Wesley Professional.

[6] UML, O. M. G. (2011). 2.4. 1 superstructure specification. document formal/2011-08-06. Technical report, OMG