


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



## D5.12 – Data Management Workbench and Open APIs - III

<b>Revision Number</b>	3.0
<b>Task Reference</b>	T5.5
<b>Lead Beneficiary</b>	UBI
<b>Responsible</b>	Konstantinos Perakis
<b>Partners</b>	AGRO AKTIF BOI BOS CP GFT INNOV ISPRINT JSI LXS NBG PRIVE RB UBI WEA
<b>Deliverable Type</b>	Report (R)
<b>Dissemination Level</b>	Public (PU)
<b>Due Date</b>	2022-03-31
<b>Delivered Date</b>	2022-03-28
<b>Internal Reviewers</b>	ATOS, GFT
<b>Quality Assurance</b>	INNOV
<b>Acceptance</b>	WP Leader Accepted and Coordinator Accepted
<b>EC Project Officer</b>	Beatrice Plazzotta
<b>Programme</b>	HORIZON 2020 - ICT-11-2018
	This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no 856632

## Contributing Partners

Partner Acronym	Role <sup>1</sup>	Author(s) <sup>2</sup>
<b>UBI</b>	Lead Beneficiary	Konstantinos Perakis, Dimitris Miltiadou, Stamatis Pitsios
<b>LXS</b>	Contributor	Alejandro Ramiro, Luis Miguel Garcia, Javier Pereira
<b>ATOS</b>	Internal Reviewer	Ignacio Elicegui
<b>GFT</b>	Internal Reviewer	Ernesto Troiano
<b>INNOV</b>	Quality Assurance	Filia Filippou

## Revision History

Version	Date	Partner(s)	Description
0.1	2022-01-17	UBI	ToC Version
0.2	2022-01-21	UBI	Initial contribution to Section 3
0.3	2022-01-31	UBI, LXS	Contributions to Section 4
0.35	2022-02-04	UBI	Updated contribution to Section 3
0.40	2022-02-10	UBI, LXS	Updated contribution to Section 4
0.45	2022-02-18	UBI, LXS	Updated contribution to Section 4
0.50	2022-02-23	UBI	Updated contribution to Section 4
0.70	2022-02-28	UBI, LXS	Updated contributions to Sections 4 and 5
0.80	2022-03-04	UBI	Finalisation of sections 3 and 4
1.0	2022-03-07	UBI	First Version for Internal Review
2.0	2022-03-17	UBI	Version for Quality Assurance
3.0	2022-03-28	UBI	Version for Submission

---

<sup>1</sup> Lead Beneficiary, Contributor, Internal Reviewer, Quality Assurance

<sup>2</sup> Can be left void

## Executive Summary

The document at hand, entitled “Data Management Workbench and Open APIs - III”, constitutes the final report of the efforts and the produced outcomes of Task 5.5 “OpenAPI for Analytics and Integrated BigData/AI WorkBench”. The scope of Task 5.5 is to enable access to the added-value analytics functionalities of INFINITECH, that are offered via microservices implementations, and their respective Open APIs in an integrated way through a single entry-point. The main object of the current report is to deliver **the final fully functional version of the INFINITECH Open API Gateway** which is delivered on M30 per the INFINITECH Description of Action. The INFINITECH Open API Gateway has been implemented in accordance with the **detailed design specifications** that were documented in the previous versions of the deliverable building on top of the first prototype version that has been presented in deliverable D5.11. The final fully functional version of the INFINITECH Open API Gateway is delivered with a rich set of features that effectively address the accessibility and consumption challenges raised from the modular nature of the microservices implementations of added-value offerings in INFINITECH.

Towards this end, the current deliverable reports the final and full documentation of the INFINITECH Open API Gateway, encapsulating the updates from the previous version that were reported with deliverable D5.11. In particular, the advancements and optimisations from the previous version are highlighted and the supplementary complete documentation of the technical details of the implementation of the final fully functional version of the INFINITECH Open API Gateway is provided. The deliverable builds on top of the outcomes and knowledge extracted so far in order to provide the updated and final report of the work performed until M30.

Hence, the scope of the current report can be summarized in the following axes:

- To document the results of the detailed and comprehensive **analysis of the challenges imposed by the microservices architecture**, in the consumption of the offered – by the underlying INFINITECH microservices – business functionalities from any client applications or services. During this analysis, the commonly applied approaches were presented and analysed, focusing on their advantages and disadvantages. In addition to this, the rationale for the selection of the API Gateway pattern as the basis for the design of the INFINITECH Open API Gateway is presented. The results of the analysis remained unchanged from the previous iteration, however they are included in the deliverable for coherency reasons.
- To present the **final design specifications of the INFINITECH Open API Gateway** which constitute the basis for the implement of the delivered solution of the single entry-point for the added-value analytics functionalities and other core offerings of INFINITECH, as one of the core ingredients of the INFINITECH Reference Architecture. The detailed documentation of the design specifications includes the business need which is covered and the rationale behind the design of the INFINITECH Open API Gateway. Furthermore, it includes the list of core functionalities of the component which were driven by the core design decisions. The design specifications document also the high-level architecture of the component is defined and the two distinct modules that compose this architecture. Finally, the documentation of the design specifications concludes with the details of the supported use cases from each module and their respective sequence diagrams. In terms of updates in this iteration, the documentation was extended with the analysis of the covered business need and rationale behind the designed solution.
- To deliver the **final and fully functional version of the INFINITECH Open API Gateway** and provide the **final and complete technical documentation of the delivered solution**. To this end, the deliverable extends the previous documentation of the delivered solution in order to accommodate all the technical details of the produced and delivered final version. It includes the complete technical documentation of the two core modules of the INFINITECH Open API Gateway, with the final list of implemented functionalities and the complete integration details of these two module towards the delivery of the final solution. Finally, it provides a walkthrough from the user’s perspective of the delivered solution focused on the offered user interface.

- To document the final list of well-established **open-source technologies, libraries and frameworks** which are leveraged during the implementation phase of the INFINITECH Open API Gateway component. The list remained unchanged from the previous version; however it is included in the deliverable for coherency reasons.

The current deliverable provides the required updates and optimisations from the previous iteration and constitutes the final report of the Task 5.5. This concludes the activities of the specific task as per the INFINITECH Description of Action.

## Table of Contents

1	Introduction.....	8
1.1	Objective of the Deliverable .....	8
1.2	Insights from other Tasks and Deliverables.....	9
1.3	Structure .....	10
2	Motivation and Challenges .....	11
3	The INFINITECH Open API Gateway .....	14
3.1	Business need & Rationale .....	14
3.2	Design overview.....	15
3.3	Design Specifications .....	17
3.4	Use Cases and Sequence Diagrams .....	19
3.4.1	Gateway.....	19
3.4.2	Service Registry .....	23
4	Implementation of the INFINITECH Open API Gateway .....	27
4.1	Gateway .....	27
4.1.1	Gateway Backend .....	30
4.1.2	Gateway Frontend.....	33
4.2	Service Registry.....	35
4.3	The INFINITECH Open API Gateway Solution .....	37
5	Baseline technologies and tools.....	43
6	Conclusions.....	45
	Appendix A: Literature .....	47

## List of Figures

Figure 2-1:	Direct Client-to-Microservices communication.....	12
Figure 2-2:	API Gateway pattern.....	13
Figure 3-1:	INFINITECH Open API Gateway high-level architecture .....	18
Figure 3-2:	Open APIs discovery sequence diagram .....	20
Figure 3-3:	Request Handling (1-to-1).....	21
Figure 3-4:	Request Handling (1-to-many).....	23
Figure 3-5:	Self-registration sequence diagram .....	24
Figure 3-6:	Self-deregistration sequence diagram.....	25
Figure 3-7:	Periodic Health Check sequence diagram .....	26
Figure 4-1:	Gateway Module Architecture and interactions .....	28
Figure 4-2:	Gateway Request Processing Handling.....	29
Figure 4-3:	Gateway Request Processing Handling (1-to-many) .....	30
Figure 4-4:	Discovery of microservices’ Open API documentation.....	35
Figure 4-5:	List of registered microservices .....	41
Figure 4-6:	Open API Documentation of a registered microservice .....	42

## List of Tables

Table 3-1: INFINITECH Open API Gateway design aspects .....	17
Table 3-2: Gateway – Open APIs discovery.....	19
Table 3-3: Gateway – Request Handling (1-to-1).....	20
Table 3-4: Gateway – Request Handling (1-to-many).....	22
Table 3-5: Service Registry – Self-registration .....	23
Table 3-6: Service Registry – Self-deregistration .....	24
Table 3-7: Service Registry – Periodic Health Check.....	25
Table 5-1: INFINITECH Open API Gateway list of technologies.....	44
Table 6-1: Conclusions (TASK Objectives with Deliverable achievements) .....	46
Table 6-2: (map TASK KPI with Deliverable achievements) .....	46

## Abbreviations/Acronyms

Abbreviation	Definition
<b>AI</b>	Artificial Intelligence
<b>API</b>	Application Programming Interface
<b>DL</b>	Deep Learning
<b>GDPR</b>	General Data Protection Regulation
<b>IP</b>	Internet Protocol
<b>HTTP</b>	Hypertext Transfer Protocol
<b>JSON</b>	JavaScript Object Notation
<b>JWT</b>	JSON Web Token
<b>KPI</b>	Key Performance Indicator
<b>ML</b>	Machine Language
<b>N/A</b>	Not Applicable
<b>PSD2</b>	Payment Services Directive Two
<b>RA</b>	Reference Architecture
<b>REST</b>	Representational State Transfer
<b>UI</b>	User Interface
<b>URL</b>	Uniform Resource Locator

# 1 Introduction

D5.12 is released in the scope of WP5 “Data Analytics Enablers for Financial and Insurance Services”, which enumerates the associated activities and documents the updated outcomes of Task 5.5 “OpenAPI for Analytics and Integrated BigData/AI WorkBench”. The specific deliverable is prepared in accordance with the INFINITECH Description of Action and constitutes the third and final report which documents the work performed within the context of Task 5.5 till M30. To this end, the deliverable presents the complete detailed documentation of the designed solution for the effective and efficient access to the added-value analytics functionalities and other core offerings of INFINITECH through a sophisticated, integrated and single-entry point manner.

The deliverable builds directly on top of the results documented in the previous iterations, namely deliverables D5.10 and D5.11, in order to provide the complete documentation of the design specifications and the implementation details of the delivered final version of the INFINITECH Open API Gateway. An initial analysis of the modular nature of the microservices architecture in conjunction with the virtualised infrastructures where these microservices are usually deployed, revealed the challenges and restrictions imposed during the consumption of these microservices from the possible client services in an effective and efficient manner. The Gateway pattern has been selected after a comprehensive analysis of the available solutions as the most suitable solution and well-established approach to overcome these challenges. Hence, this pattern has been the basis of the design of a full solution which enables the different microservices to be discoverable and consumable by the clients. As the microservices constitute the main ingredient of the INFINITECH platform, and the various analytics and other main functionalities of INFINITECH are based on microservices, the delivered solution is considered of the utmost importance. The goal of the designed solution is to eliminate the barriers of accessing these offered by microservices functionalities from different clients. The solution is designed taking into consideration the easy integration with the INFINITECH platform and constitute the single-point-of-entry for those microservice-based functionalities.

In this third iteration, the documentation of the previously delivered prototype version of the INFINITECH Open API Gateway is extended and supplemented with the details of all the implemented functionalities. The implementation of the final version is based on the delivered prototype and is also driven by the design specifications documented in the previous iterations. The final and fully functional delivered solution enables the INFINITECH project’s pilots, as well as the stakeholders of the financial sector, to consume the offered functionalities in a straight-forward and easy manner, and without requiring knowledge of the underlying microservices architecture. It delivers the implementation of the complete set of features of the INFINITECH Open API Gateway as scheduled per the INFINITECH Description of Action.

## 1.1 Objective of the Deliverable

The purpose of this deliverable is to report the outcomes of the work performed within the context of Task 5.5 from M23 till M30 that the task concludes its activities. As the deliverable constitutes the third and final iteration, its main goal is to provide the final and comprehensive documentation of the fully functional version of the INFINITECH Open API Gateway, as well as the advancements from the previous iteration.

During this third and final period (from M23 till M30), the activities mainly focused on the enhancement and optimisation of the first prototype version of the INFINITECH Open API Gateway into the fully functional version which offers all the designed features and functionalities. To this end, the document of the implementation details of each module has been extended to include the new functions implemented as well as the updates on the existing ones.

The deliverable aims at providing the complete documentation of the provided solution, extending the information that has been documented in the previous iteration. In addition to this, for coherency reasons, it contains the information included in the previous iteration, highlighting the updates and optimisations that were introduced where needed.



Thus, the first objective of the deliverable is to present the results, which remained unchanged from the previous iteration, of the performed analysis for the challenges imposed by microservices architecture to the client that aim at consuming the offered by the microservices business functionalities. The analysis presents the comparison of the two most common approaches, namely the direct client-to microservices and the API Gateway pattern, highlighting the advantages and disadvantages of each approach.

The second objective of the deliverable is to document the complete details of the design specifications of the INFINITECH Open API Gateway component. While the design specifications remained unchanged from the previous iteration also, the addressed business need and rationale behind the design of the delivered solution is documented. In detail, the design specifications present: a) the business need and rationale behind the designed solution, b) the design overview of the solution highlighting its main functionalities, c) the core decisions taken during the design phase of the component, d) the detailed specifications of the core modules of the modular architecture of the INFINITECH Open API Gateway component, namely the Gateway and the Service Registry, describing their offered functionalities in accordance with the design decisions and e) the complete list of supported use cases accompanied by the respective sequence diagrams for each use case which depict the interactions of the involved clients and modules.

The third objective is the completed and detailed documentation of the implemented final version of the INFINITECH Open API Gateway. In this final iteration, the complete list of functions that were implemented by each module is presented together with the details of their integration and interactions towards the delivery of the end-to-end functionalities of the INFINITECH Open API Gateway.

The fourth objective of the deliverable to document the complete list of baseline technologies and tools which are leveraged for the implementation of the INFINITECH Open API Gateway. The list remained unchanged from the previous version and it includes multiple technologies, libraries and frameworks which are successfully integrated in order to realise the fully functional version of the INFINITECH Open API Gateway.

The specific deliverable concludes the activities of Task 5.5 in accordance with the INFINITECH Description of Action. Hence, it constitutes the final documentation of the work performed and includes all the necessary updates and optimisations that were introduced from M23 till M30.

## 1.2 Insights from other Tasks and Deliverables

Deliverable D5.12 is released in the scope of WP5 “Data Analytics Enablers for Financial and Insurance Services” and documents the updated outcomes of the work performed within the context of Task 5.5 “OpenAPI for Analytics and Integrated BigData/AI WorkBench”. The task is directly related to the rest of the Tasks of WP5, enabling access to the added-value analytics functionalities of INFINITECH which are formulated by the library of ML/DL algorithms for Financial and Insurance Services (as produced by T5.4), that incorporates the incremental and parallel data analytics (produced by T5.2) as well as the declarative real-time analytics (produced by T5.3) and leverages the data collection process for the algorithm’s training and evaluation (produced by T5.1).

In addition, the task builds on top of the outcomes of WP2 “Vision and Specifications for Autonomous, Intelligent and Personalized Services” in which the overall requirements of the INFINITECH platform are defined. Specifically, Task 5.5 received as input the collected user stories of the pilots of the project and the extracted user requirements, as documented in deliverables D2.1 and D2.2 that report the work performed in Task 2.1. Furthermore, the inputs of the task included the elicited technical requirements and the fundamental building blocks of the INFINITECH platform and their specifications, as reported in deliverables D2.5 and D2.6. Finally, the task received as input the outcomes of T2.7, in which the INFINITECH Reference Architecture (INFINITECH RA) was formulated, as documented in deliverables D2.13, D2.14 and D2.15, during the design specifications definition of the INFINITECH Open API Gateway component that constitutes a part of the INFINITECH platform.

## 1.3 Structure

This document is structured as follows:

- Section 1 (current one) introduces the document, describing the context of the outcomes of the work performed within the task and highlights its relation to the rest of the tasks and deliverables of the project.
- Section 2 documents the motivation and challenges in the access of the deployed microservices of a microservices-based platform by the clients. This particular section remained unchanged from the previous iteration of the deliverable.
- Section 3 presents the design overview and design specifications of the INFINITECH Open API Gateway component, as well as the use cases addressed, along with the corresponding sequence diagrams. The particular section has been extended to document the addressed business need and rationale behind the designed solution.
- Section 4 presents the implementation details of the delivered version of the INFINITECH Open API Gateway component. The particular section has been extended to document the necessary updates related to the advancements on the implementation of the INFINITECH Open API Gateway.
- Section 5 presents the list of baseline technologies and tools that are utilised in the implementation of the INFINITECH Open API Gateway component. This particular section remained unchanged from the previous iteration of the deliverable.
- Section 6 concludes the document.

## 2 Motivation and Challenges

### Updates from D5.11:

*This particular section remained unchanged from the previous version. It presents an analysis of the challenges imposed in the consumption of business functionalities in the microservices architectures and commonly applied approaches to overcome them.*

In traditional monolithic applications, the application is designed and implemented with the aim of executing a specific, commonly domain-specific, business logic and interacts with the client-side or any third-party client (applications) via an exposed API. In this sense, the clients will usually retrieve the required data or invoke any specific operation by executing a single REST call, that is received and handled by the underlying server-side (backend) application. While the monolithic application approach has a number of benefits, such as simplifying development and deployment, as well as horizontal scalability by multiple running instances behind a load balancer, it also introduces a large number of drawbacks that are very critical and create severe barriers to application execution and evolution. The monolithic architecture has a limitation when it comes to application size and complexity. When the application grows in size, the number of functionalities, complexity and sustainability issues arise. The source code becomes difficult to understand and maintain, while also changes in the functionalities offered (and/or the introduction of new functionalities) require great development effort, and add multiple dependencies that should be met, hence the complexity grows. Additionally, the scaling and continuous deployment becomes problematic, since a small update might require a redeployment of all the applications once extensive testing has been successfully performed.

The introduction of the microservices architecture approach solves the drawbacks of the traditional monolithic architecture approach by structuring the underlying application as a collection of microservices that are loosely-coupled, highly maintainable and testable, independently-deployable, organised by business capabilities and can be owned by different development teams [1]. Nevertheless, while the microservices architecture enables the effective and efficient development of large and complex applications in a rapid, frequent and reliable manner, it also introduces a number of challenges that need to be addressed.

In the microservices architecture, the application is partitioned in multiple microservices, where each one of them undertakes a specific business capability of the application and a set of responsibilities. However, this business-capabilities decomposition imposes several requirements on the underlying microservices. Each microservice should provide an API that enables the needed intercommunication between the various services. As the context of each microservice is clearly defined, the provided API is usually fine-grained and restricted to the assigned business capability. Hence, the granularity of the APIs of the microservice is usually different than the one required by any client of the application, and there are cases where the client is required to invoke multiple microservices in order to obtain all the required data or execution results. Furthermore, the evolution of the application usually requires the introduction of new microservices or even the refinement of existing ones to support a new business capability. The dynamic nature of the virtualised infrastructures that are hosting the applications, and the microservices that compose these architectures, enables the deployment, upgrade, scaling and restart of each microservice independently of the rest of the microservices of the application. Hence, both the number of microservices, as well as their connection details, such as the hostname and their IP addresses might change and vary dynamically. Thus, the transition from a traditional monolithic application to a microservice application, imposes the problem of how the clients of this application can effectively and efficiently consume the offered business capabilities by either a specific microservice or a combination of microservices.

To this end, the problem can be narrowed down to the client-to-microservices communication. Two approaches can be followed to solve this problem. In the first approach, usually referenced as direct client-to-microservice communication [2], the client is able to directly make requests to the microservices of the application. In this case, each microservice provides an exposed API endpoint and can be reached through a specific URL that maps to a load balancer which in turn distributes the incoming requests to the requested microservice instances (see Figure 2-1).

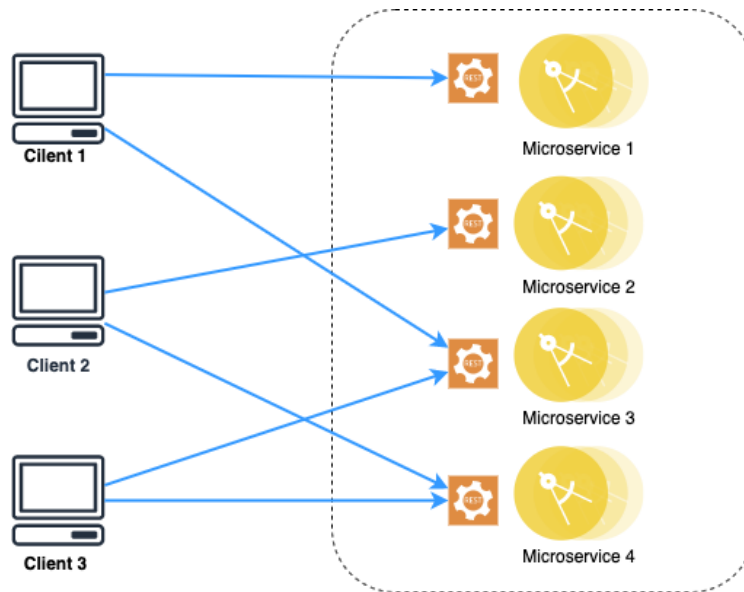


Figure 2-1: Direct Client-to-Microservices communication

However, this approach has several limitations and challenges that should be taken into consideration. The first challenge arises due to the fact that, by design, each microservice offers a fine-grained API with a specific context. Thus, there are cases where the client is forced to make multiple requests to different microservices, hence multiple server round trips, to collect the information needed to complete a single operation. This is inefficient and increases the latency in some cases, while it also increases the complexity in the source code of the client. An additional challenge is that microservices might use different and diverse communication protocols (e.g. HTTP, AMQP, Thrift) and as a consequence the client should be capable of interacting with all these diverse communication protocols. Moreover, common functionalities such as authorisation, authentication and logging, need to be taken care of at the microservices level, rather than at a common cross-cutting level on top of the microservices. Finally, as the implementation of the client is directly connected with the underlying microservices, any changes or updates in the existing microservices, such as the merging or splitting of an existing microservice, or the introduction of new ones, usually propagates changes on the clients that need to adapt to the evolution of the microservices.

The second approach that addresses all the above-mentioned challenges is the API Gateway pattern [3]. The API Gateway is introduced as an intermediate layer between the clients and the underlying microservices, and acts as a single entry-point for all clients that consume these microservices. All incoming requests are generated towards the API Gateway which serves them in two core ways. The API Gateway usually routes the request to the appropriate microservice acting as a reverse proxy, however in some cases it handles the request by invoking multiple microservices and aggregating the results, which are provided back to the requestor (see Figure 2-2). In addition to this, the API Gateway undertakes several cross-cutting functionalities such as the authorisation, authentication, monitoring and load balancing.

The API Gateway encapsulates the underlying system architecture, hiding the details of how the application is partitioned into microservices, providing different APIs for diverse clients if needed. It eliminates the need for clients to discover or keep track of the network locations of the microservices and their exposed APIs that can change dynamically. Furthermore, it reduces the problem of multiple server round trips, as it handles the invocation of multiple microservices and aggregation of the results. The API Gateway is able to handle the diverse communication protocols problem, since it exposes a well-defined and web-friendly API, undertaking the protocol conversion internally. Thus, the API Gateway is able to simplify both the communication with the clients, as well as the complexity of the client's source code. The client is only interacting with the API Gateway, hence the update or evolution of the application's microservice is hidden from the client. Furthermore, it handles cross-cutting functionalities, while the development of the underlying microservices is also simplified.

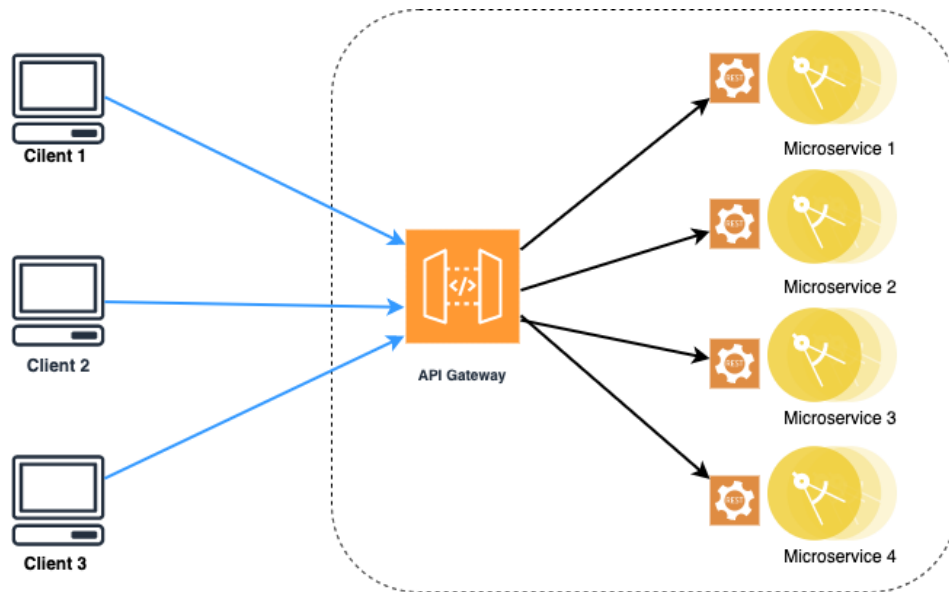


Figure 2-2: API Gateway pattern

However, the API Gateway pattern also has some drawbacks. As it constitutes a component with high availability, it should be designed, developed, deployed and maintained in a proper way in order to avoid becoming a bottleneck for the application. Hence, it is crucial that several aspects such as performance, adaptability and fault tolerance are taken into consideration to benefit from the API Gateway pattern solution. Nevertheless, despite these drawbacks, the API Gateway pattern is considered to be the proper approach in the case of INFINITECH, as it effectively solves all the fundamental issues and challenges described in the direct client-to-microservice communication.

## 3 The INFINITECH Open API Gateway

### Updates from D5.11:

*The particular section presents the design overview of the INFINITECH Open API Gateway, documenting the design specifications of the proposed solution, as well as the use cases addressed supplemented by the relevant sequence diagrams.*

*In terms of updates from the previous version, it includes the documentation of the business need and rationale behind the developed solution (section 3.1) while the rest of section remained unchanged.*

### 3.1 Business need & Rationale

The introduction and enforcement of the Revised Payment Service Directive (PSD2) [4] that became effective as of December 2020 as an updated and enhanced regulation, extending and expanding the previously established from 2017 Payment Service Directive (PSD) regulation, is reshaping the entire banking and financial services. The introduction and enforcement of PSD2 resulted on the emerge of the Open Banking initiative which has driven the new digital banking and financial era that modernizes the financial sector as a whole. It has introduced a disruptive growth in the banking and finance sector with the rise of novel financial services offered by financial institutions that enable customers to leverage a whole new set of innovative offerings. In this era, financial technology (Fintech) is booming with multiple new and constantly evolving technologies which aim to optimise and automate the delivery and use of financial products and services. This new and rapidly growing industry is offering solutions and services across different sectors of the banking environment such as consumer and business banking, insurance, e-payments, investments and portfolio management and more. The financial institutions are investing and leveraging Fintech solutions in order to enhance their existing services with new, faster and more efficient offerings but also to achieve growth, strengthen their position in the sector and reduce their operational costs with the introduction of new business models and services.

At the heart of the Open Banking initiative and Fintech stands the deployment and usage of APIs. APIs enable the integration, interoperability, interconnection and intercommunication of various services and solution in an effortless, accelerated and secure manner. As explained also in section 2, the usage of APIs is supplemented with the rise of the microservices architecture approach that provides the means to overcome the difficulties imposed by the previous approaches in the implementation of services such as the traditional monolithic architecture. To this end, the combination of the microservices architecture approach and the extensive usage of APIs enables faster implementation and reduced time to market, high reusability and extensibility of services while at the same time the reduce of development and operational costs.

The cornerstone of this innovation is the effective solution of an API Gateway which, as explained also in section 2, fosters innovation by undertaking the task of the interconnection of multiple services and clients and many more required cross-cutting functionalities. To this end, an API Gateway solution facilitates the development and exploitation of cutting-edge Fintech APIs that are transforming the banking and financial sector with the explosion of new apps, services and business models. As Fintech APIs provide the required means for the exchange of information and integration of multiple services and offerings, multiple opportunities are leveraged by the financial innovators which are currently completely reshaping the financial sector. In addition to this, an API Gateway solution enables the various banking and financial institutions to comply with the PSD2 regulation in a secure and effective manner, while also expanding their services and increase their competitiveness. The API Gateway along with the respective Fintech APIs are considered the drivers of the current success of many fintech and financial services tech leaders. Both are considered as building blocks of this innovative era and are leveraged across different sectors such as ePayments, eCommerce, Banking and Banking as a Service (BaaS), accounting, financial market data, personal and corporate finance management, credit and risk assessment, customer onboarding and Know-Your-Customer.

## 3.2 Design overview

The **INFINITECH Open API Gateway** is a sophisticated API Gateway that encompasses the Open API specification in order to provide a single point of entry for the added-value functionalities of INFINITECH which are based on microservices. The work that was performed during this phase was mainly focused on the Machine Learning (ML) / Deep Learning (DL) analytics functionalities of INFINITECH, which are implemented as microservices. Following the API Gateway pattern, as presented in Section 2, it is capable of effectively handling and determining the network location of dynamically-deployed microservice instances, while at the same time hiding the internal application's architecture from the clients of the application.

Hence, the main functionalities of the INFINITECH Open API Gateway are as follows:

- To act as a single point of entry for the ML/DL analytics functionalities of INFINITECH, handling the incoming requests towards the dynamically-deployed microservice instances;
- To facilitate the communication between the underlying microservice instances;
- To enable the self-registration of the microservice instances;
- To facilitate the discovery of the microservice instances;
- To enable the discovery of the microservice instance Open APIs;
- To effectively handle the unavailability or unreachability of the underlying microservice instances;
- To provide authentication, monitoring and logging functionalities to the microservice instances.

During the design phase of the INFINITECH Open API Gateway, several design decisions were taken based on the specifications set by the INFINITECH Reference Architecture, as documented in deliverable D2.15; the elicited technical requirements are reported in deliverable D2.6, as well as the requirements of the pilots and stakeholders of INFINITECH, as documented in deliverable D2.2. Within the INFINITECH Reference Architecture, the INFINITECH Open API Gateway is positioned in the Interface layer of the architecture undertaking the task of providing the required information from the Analytics layer to the Presentation layer where the clients reside.

With regards to the *Request Handling* processes of the INFINITECH Open API Gateway, it was decided that it should support both core ways of the API Gateway pattern. Hence, the INFINITECH Open API Gateway should be able to handle one to one (1-to-1) microservices invocation, meaning it can handle requests that are simply translated by routing the incoming request to the appropriate microservice. On the other hand, the INFINITECH Open API Gateway should handle one to many (1-to-many) microservices invocation, thus a request could be translated into the invocation of multiple microservices, whose results are aggregated and returned to the requestor. In the latter case, a sophisticated logic will be employed to effectively handle the multiple invocations, dependencies between the invocations, and failure handling through modern approaches, such as reactive programming with the promises pattern instead of the traditional asynchronous call-back pattern.

As the underlying microservices are operating in a distributed manner, it is required to establish a set of *Communication Mechanisms* across the different processes that enable the invocation of the microservices in both an asynchronous and synchronous manner. To this end, both approaches will be supported with the proper mechanisms for asynchronous message-based communication and synchronous communication.

One of the core aspects of the INFINITECH Open API Gateway is the effective and dynamic *Service Registry* and *Service Discovery* processes. The INFINITECH Open API Gateway is designed to be able to find and keep track of the network location of each microservice that it handles the requests for. As explained in Section 2, microservices are usually operating in a cloud infrastructure, within virtual machines or containers, that change dynamically in terms of IP addresses and ports. Hence, it is imperative that the INFINITECH Open API Gateway incorporates the appropriate Service Registry and Service Discovery processes that effectively handle the changes of the microservice instances in a dynamic manner.

Service Registry is a database that maintains the updated network locations of the microservices. Hence, the microservice instances are registered with the Service Registry on start-up and deregistered on shutdown. In the meanwhile, the Service Registry performs periodic health checks on the registered microservices to

ensure their availability. For the service registration, there are two approaches, the self-registration pattern and the third-party registration pattern. In the Self-Registration pattern [5], the microservice instance is responsible for the registration and deregistration of itself with the service registry, while in the third-party registration pattern [6] the registration and deregistration is handled by a third party application called registrar, by performing polling operations or event subscriptions. In the INFINITECH Open API Gateway, the self-registration pattern is followed as it keeps the complexity of the component at the lower levels, while the coupling of the registration process with the microservices requires less effort, as there is a large number of available libraries performing these operations with out-of-the-box integration.

With regards to Service Discovery there are two approaches, namely the client-side service discovery and the server-side discovery. In the client-side service discovery [7], the client retrieves the latest network location of a microservice by querying the Service Registry. While this approach is considered more efficient in terms of network requests, it bounds the client implementation with the Service Registry. On the other hand, in the server-side discovery [7] the client makes requests to the API Gateway and the API Gateway queries the Service Registry before it finally routes the request to the latest network location of the requested microservice. In the INFINITECH Open API Gateway, the server-side discovery is followed as it eliminates the need to couple the client's implementation with the Service Registry, and additionally it hides the implementation details of the API Gateway and the registered microservices from the client.

A crucial aspect of the INFINITECH Open API Gateway is the embracement of **Open APIs** to enhance the accessibility of the underlying ML / DL microservices. The Open API specification is proposed by the Open API initiative, which is an open-source collaboration project of the Linux foundation, and defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service, without access to the source code, documentation, or through network traffic inspection [8]. The INFINITECH Open API Gateway will embrace the Open API specification which all underlying ML / DL microservices are required to follow in order to help the clients understand and consume their exposed services without the need for knowledge of the implementation details or access to the source code of the microservices. Hence, the INFINITECH Open API Gateway will facilitate publishing the offered Open APIs of the microservices in order to enable their easy and effortless discovery by the clients of the INFINITECH Open API Gateway.

Another aspect of the INFINITECH Open API Gateway is *Fault Tolerance*. For either 1-to-1 or 1-to-many microservices invocation, a mechanism performing efficient failure handling should be employed. Towards this end, the Circuit Breaker pattern [9] will be followed in the INFINITECH Open API Gateway. The specific pattern is utilised to handle the case where one of the invoked microservices is unavailable or unreachable and the whole process is not stalled, while the allocated resources are properly managed. In this pattern, if a microservice reaches a threshold of consecutive failures, then the circuit breaker that acts as a proxy will immediately reject all upcoming requests to this microservice for a predefined testing period. When this testing period ends, a limited number of test requests are made; if they succeed, the circuit breaker allows the invocation of the microservice again, else a new testing period is started again.

One final aspect of the INFINITECH Open API Gateway is the offering of cross-cutting functionalities such as authentication, monitoring and logging. With regards to the authentication, the INFINITECH Open API Gateway will embrace the Access Token pattern and specifically the JSON Web Token (JWT). The Open API Gateway that acts as the single point of entry for the ML/DL functionalities of INFINITECH will authenticate the incoming requests from the clients and will provide a JWT that securely authenticates the requestor back to the microservices. Furthermore, the INFINITECH Open API Gateway performs continuous monitoring and detailed logging of all the received requests and executed operations for security, malicious activity and performance analysis purposes.

The following table summarizes the main design decisions that were taken during the design phase of the INFINITECH Open API Gateway.



Table 3-1: INFINITECH Open API Gateway design aspects

Design Aspect	INFINITECH Open API Gateway
<b>Request Handling</b>	Both 1-to-1 and 1-to-many microservices invocation
<b>Communication Mechanisms</b>	Both asynchronous message-based and synchronous communication
<b>Service Registry</b>	Self-registration pattern
<b>Service Discovery</b>	Server-side discovery pattern
<b>API Specifications</b>	Open API specification
<b>Fault Tolerance</b>	Circuit Breaker pattern
<b>Cross-cutting functionalities</b>	Authentication, Monitoring, Logging

It should be noted that the presented design aspects of the INFINITECH Open API Gateway do not limit the support for additional added-value functionalities which are based on microservices. On the contrary, the described approach facilitates the expansion of the list of supported microservices implementations in an effortless and effective manner as per the project's needs.

### 3.3 Design Specifications

During the analysis of the client-to-microservices communication problem, it was clear why the API Gateway pattern is considered the dominant solution for this problem, as it effectively and efficiently resolves the problem in a solid and robust manner. Hence, the consortium decided to design and implement the INFINITECH Open API Gateway that is based on the API Gateway pattern, aiming at providing the required access to the underlying ML/DL microservices. However, as presented in the overview of the INFINITECH Open API Gateway in the previous section, there are several aspects that were carefully addressed in the design phase towards the implementation of the optimal (most effective and efficient) solution that will address the requirements of INFINITECH stakeholders.

The INFINITECH Open API Gateway has a modular architecture composed of two core modules, namely the *Gateway* and the *Service Registry*. Each module has a clear scope and a distinct context undertaking a specific set of responsibilities. Additionally, the modules are interacting through well-defined REST APIs in order to perform the main operations of the INFINITECH Open API Gateway. Their role in the INFINITECH Open API Gateway is as follows:

- The Gateway provides the single-entry point for all incoming requests from the pilots and undertakes the responsibility of invoking the appropriate microservices while also ensuring the required inter-communication between the microservices where needed.
- The Service Registry provides the database that maintains the updated network locations of the microservices. Furthermore, it provides the mechanism for self-registration and deregistration, while also performing the health check operations.

Figure 3-1 depicts the high-level architecture of the INFINITECH Open API Gateway. As illustrated, the Gateway can receive new requests from the clients for the invocation of the underlying ML/DL microservices. Upon receiving a new request, the Gateway consults the Service Registry in order to discover the updated network location of the requested microservice. Once the information is retrieved by the Service Registry, the Gateway, acting as a reverse proxy, routes the request to the appropriate microservice by invoking the respective endpoint of the microservice. The selected microservice handles the request and provides the results back to the Gateway. At this final step, the Gateway replies to the client with the results as provided by the respective ML/DL microservice.

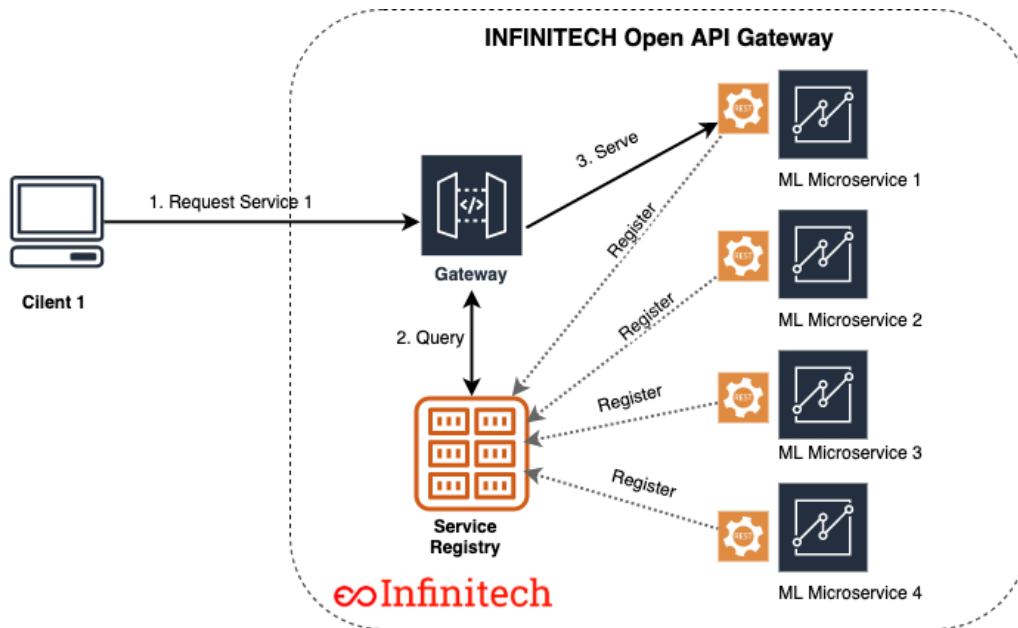


Figure 3-1: INFINITECH Open API Gateway high-level architecture

The execution is slightly different in the case where a request requires the invocation of multiple microservices. In this case, the Gateway upon receiving the request, consults the service registry and retrieves the list of microservices. Then, multiple requests are initiated to the respective microservices. Each microservice provides the results back to the Gateway and when all results are received the Gateway aggregates the results and sends them back to the client.

The scope of the Gateway is to facilitate the invocation of ML/DL microservices, hiding the details of the deployed microservices from the client. It undertakes all the core functionalities of the INFINITECH Open API Gateway utilising the services of the Service Registry to fulfil its purpose. Furthermore, the Gateway provides cross-cutting operations to the respective ML/DL microservices decoupling their implementation from these operations.

In accordance with the INFINITECH Open API Gateway overview presented in section 3.1, the main functionalities of the Gateway module are as follows:

- It performs the handling of incoming requests, acting as a reverse proxy and routing the incoming requests to the respective ML/DL microservices either for 1-to-1 microservices invocation or 1-to-many microservices invocation.
- It provides the communication mechanism for the microservices invocation in both asynchronous and synchronous ways.
- It performs the service discovery operations by interacting with the Service Registry during the request handling process.
- It enables the publishing of the Open APIs of the ML/DL microservices.
- It implements the fault tolerance mechanism for the effective handling of failures from the microservices side implementing the Circuit Breaker pattern.
- It undertakes the authentication of the requestor operations utilising JWT which are passed to the respective microservice.
- It performs the continuous monitoring and logging of all operations performed in the INFINITECH Open API Gateway

The scope of the Service Registry is to support the operations performed by the Gateway for the service discovery. In particular, the Service Registry is the complementary module that is capable of maintaining the updated network location of each microservice, and providing the appropriate information to the Gateway. Additionally, it decouples the service registration and deregistration from the INFINITECH Open API Gateway.

In accordance with the INFINITECH Open API Gateway overview presented in section 3.1, the main functionalities of the Service Registry module are as follows:

- It provides the mechanism for the self-registration and self-deregistration of the ML/DL microservices by employing the mechanism that handles these requests, and the registration client that should be integrated in the ML/DL microservices.
- It supports the service discovery operations by providing the required information to the Gateway.
- It performs periodic health checks upon the registered ML/DL microservices to ensure the availability of the registered microservices.

### 3.4 Use Cases and Sequence Diagrams

As explained in Section 3.2, the INFINITECH Open API Gateway is composed of two core modules, namely the Gateway and the Service Registry. In this section, the detailed use cases that each specific module addresses are documented, presenting in detail the relevant information. Additionally, for each use case, the corresponding sequence diagram that depicts the interactions of the modules and the involved clients is presented.

#### 3.4.1 Gateway

##### 3.4.1.1 Open APIs discovery

The specific functionality enables the discovery of the Open APIs of the registered microservices by the client. To achieve this, the Gateway maintains and displays a dynamic list where all registered microservices are listed. For each microservice, the list of Open APIs can be retrieved by the user. The list of Open APIs is made available to the Gateway during the self-registration process of each microservice.

Table 3-2: Gateway – Open APIs discovery

<b>Stakeholders involved:</b>	Client’s User
<b>Pre-conditions:</b>	1. The existence of at least one registered microservice with Open APIs
<b>Post-conditions:</b>	1. The client’s user is able to retrieve the list of Open APIs of a registered microservice
<b>Data Attributes</b>	None
<b>Normal Flow</b>	<ol style="list-style-type: none"> <li>1. The client’s user navigates to the page where the dynamic list of registered microservices is displayed by the Gateway</li> <li>2. The client’s user selects one of the microservices from the list</li> <li>3. The list of Open APIs is displayed to the user following the Open API specification</li> </ol>
<b>Pass Metrics</b>	1. The client’s user is able to retrieve the list of Open APIs for the selected microservice
<b>Fail Metrics</b>	1. The client’s user cannot retrieve the list of Open APIs for the selected microservice

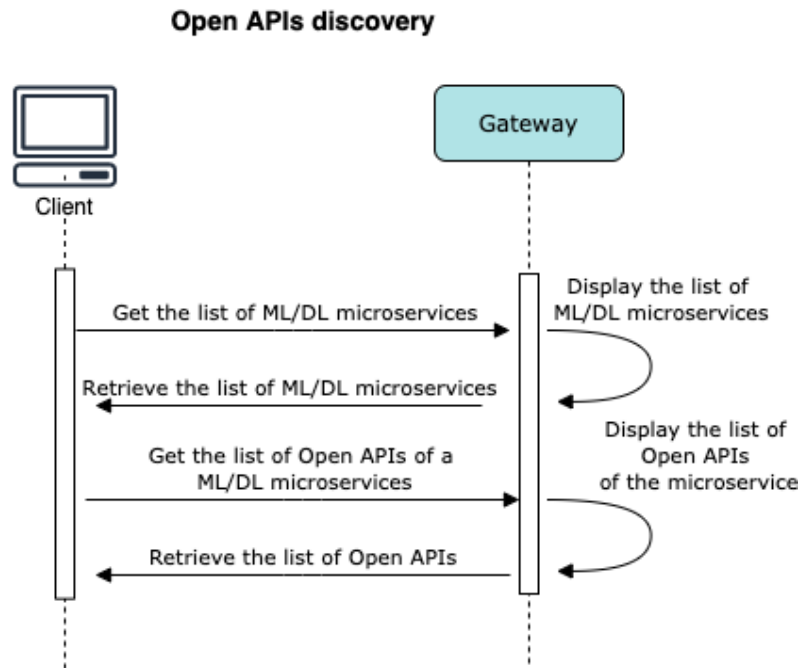


Figure 3-2: Open APIs discovery sequence diagram

### 3.4.1.2 Request Handling (1-to-1)

The specific functionality handles the incoming request for the invocation of the ML/DL microservice. The Gateway receives the request from the client and consults the Service Registry to retrieve the updated network location of the requested microservice. Upon retrieving the network location, the Gateway routes the request to the requested ML/DL microservice. The microservice receives and handles the request. The results are provided back to the Gateway which in turn returns them to the client. The specific use case also handles the case where the requested ML/DL microservice is unreachable or irresponsible.

Table 3-3: Gateway – Request Handling (1-to-1)

<b>Stakeholders involved:</b>	Client, Gateway
<b>Pre-conditions:</b>	1. The existence of at least one registered microservice with Open APIs
<b>Post-conditions:</b>	1. The client retrieves the results of the executed ML/DL microservice
<b>Data Attributes</b>	1. The endpoint of the desired microservice, prefixed with the name that the corresponding microservice used to register with the registry. 2. The data that needs to be forwarded to the desired microservice 3. The JWT that will be used to authenticate the client
<b>Normal Flow</b>	1. The client initiates a request for a specific ML/DL microservice to the Gateway 2. The Gateway consults the Service Registry to retrieve the updated network location of the ML/DL microservice 3. Upon the retrieval of the updated network location, the Gateway routes the request to the specific ML/DL microservice

	<p><i>In the case where the ML/DL microservice is available and operational:</i></p> <ol style="list-style-type: none"> <li>The ML/DL microservice receives and handles the request. The results are provided to the Gateway</li> <li>The Gateway propagates the results to the client</li> </ol> <p><i>In the case where the ML/DL microservice is unavailable and irresponsible:</i></p> <ol style="list-style-type: none"> <li>The request is not properly received by the requested ML/DL microservice</li> <li>The Gateway is informed and reports the failure to the requestor</li> </ol>
<b>Pass Metrics</b>	<ol style="list-style-type: none"> <li>The requested ML/DL microservice is invoked successfully and the results of the execution are returned to the client</li> <li>In the case where the requested ML/DL microservice is unavailable and irresponsible, the client is informed of the failure of the request</li> </ol>
<b>Fail Metrics</b>	<ol style="list-style-type: none"> <li>The requested ML/DL microservice cannot be invoked and the request fails</li> <li>The client is not informed when a failure occurs</li> </ol>

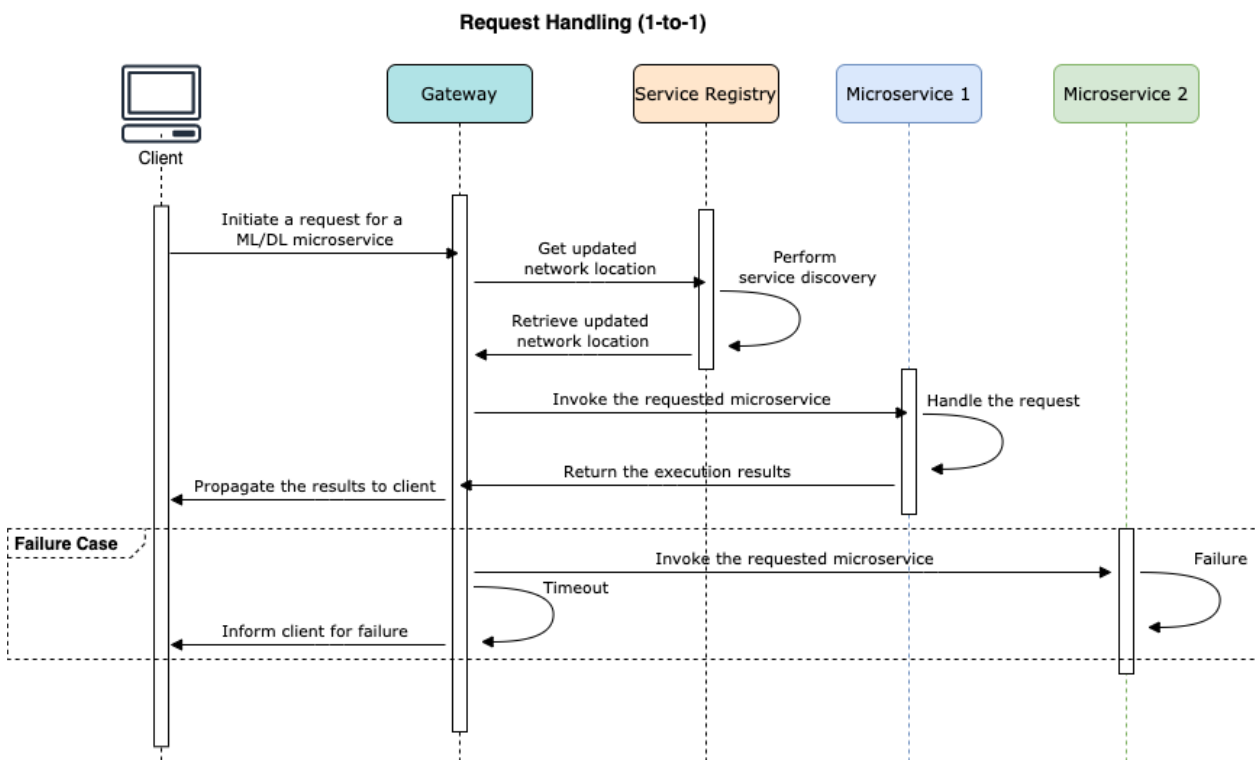


Figure 3-3: Request Handling (1-to-1)

### 3.4.1.3 Request Handling (1-to-many)

The specific functionality handles the incoming request for the invocation of a set of ML/DL microservices. The Gateway receives the request from the client and consults the Service Registry to retrieve the updated network location of the requested microservices. Upon retrieving the network location of the microservices, the Gateway initiates one request to each of the ML/DL microservices involved in the request. The respective microservices receive and handle the requests. The results from each microservice execution are provided back to the Gateway. Upon receiving all the results, the Gateway aggregates the results and returns them to the client. The specific use case also handles the case where one of the requested ML/DL microservices is unreachable or irresponsible.

Table 3-4: Gateway – Request Handling (1-to-many)

<b>Stakeholders involved:</b>	Client, Gateway
<b>Pre-conditions:</b>	1. The existence of at least two registered microservices with Open APIs
<b>Post-conditions:</b>	1. The client retrieves the results of the executed ML/DL microservices
<b>Data Attributes</b>	<ol style="list-style-type: none"> <li>1. A list of the endpoints of the desired microservices, prefixed with the names that the corresponding microservices used to register with the registry.</li> <li>2. The data that needs to be forwarded to each of the desired microservices</li> <li>3. The JWT that will be used to authenticate the client</li> </ol>
<b>Normal Flow</b>	<ol style="list-style-type: none"> <li>1. The client initiates a request to the Gateway.</li> <li>2. The Gateway retrieves the request which involves the execution of the multiple microservices. It consults the Service Registry to retrieve the updated network location of the ML/DL microservices</li> <li>3. Upon the retrieval of the updated network locations of the microservices, the Gateway initiates a request to each specific ML/DL microservice</li> </ol> <p><i>In the case where all ML/DL microservices are available and operational:</i></p> <ol style="list-style-type: none"> <li>4. Each ML/DL microservice receives and handles the request. The results are provided to the Gateway</li> <li>5. Upon receiving all the results from all invoked microservices, the Gateway aggregates the results</li> <li>6. The Gateway propagates the aggregated results to the client</li> </ol> <p><i>In the case where one of the ML/DL microservices is unavailable and irresponsive:</i></p> <ol style="list-style-type: none"> <li>4. One of the requests is not properly received by the requested ML/DL microservice</li> <li>5. The Gateway is informed, stops the processing and reports the failure to the requestor</li> </ol>
<b>Pass Metrics</b>	<ol style="list-style-type: none"> <li>1. The requested ML/DL microservices are invoked successfully and the aggregate results are returned to the client</li> <li>2. In the case where one of the requested ML/DL microservice is unavailable and irresponsive, the client is informed of the failure of the request</li> </ol>
<b>Fail Metrics</b>	<ol style="list-style-type: none"> <li>1. The requested ML/DL microservices cannot be invoked and the request fails</li> <li>2. The client is not informed when a failure occurs</li> </ol>

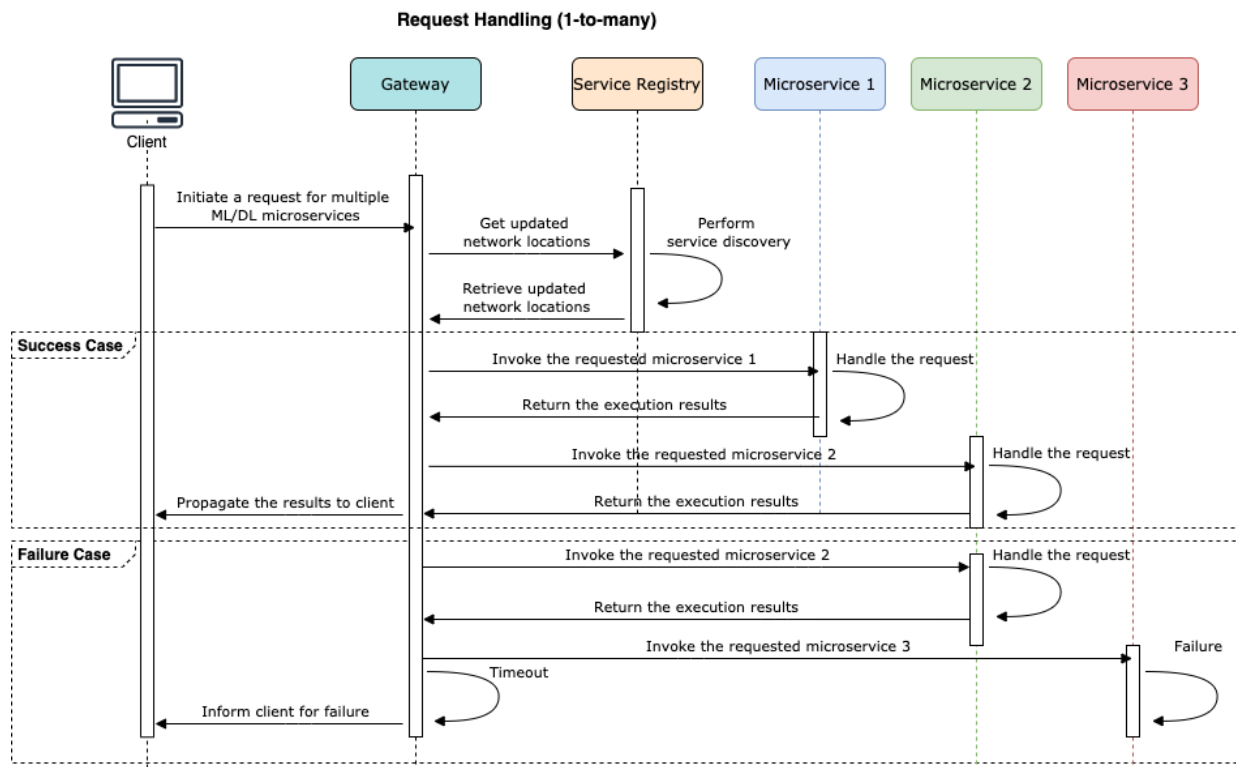


Figure 3-4: Request Handling (1-to-many)

### 3.4.2 Service Registry

#### 3.4.2.1 Self-registration

The specific functionality handles the self-registration of the ML/DL microservice in the Service Registry of the INFINITECH Open API Gateway. The prerequisite for the registration is the integration of the registry client in the microservice implementation. During the microservice start-up, the microservice is self-registered to the Service Registry.

Table 3-5: Service Registry – Self-registration

<b>Stakeholders involved:</b>	ML/DL microservice, Service Registry
<b>Pre-conditions:</b>	1. The existence of a ML/DL microservice that has integrated the registry client and its implementation
<b>Post-conditions:</b>	1. The ML/DL microservice is registered in the Service Registry
<b>Data Attributes</b>	<p>Upon self-registration, the microservice needs to send the following information to the service registry:</p> <ol style="list-style-type: none"> <li>1. The name that the microservice will use as a unique identifier to register to registry</li> <li>2. The host of the microservice</li> <li>3. The port of the microservice</li> <li>4. The health-check endpoint</li> </ol>

	5. The health-check interval
<b>Normal Flow</b>	<ol style="list-style-type: none"> <li>1. The ML/DL microservice is started. The integrated registry client is invoked.</li> <li>2. The registry client communicates with the Service Registry and self-registers the microservice instance.</li> </ol>
<b>Pass Metrics</b>	1. The ML/DL microservice is successfully registered in the Service Registry and it is ready to receive new requests
<b>Fail Metrics</b>	1. The ML/DL microservice failed to register in the Service Registry

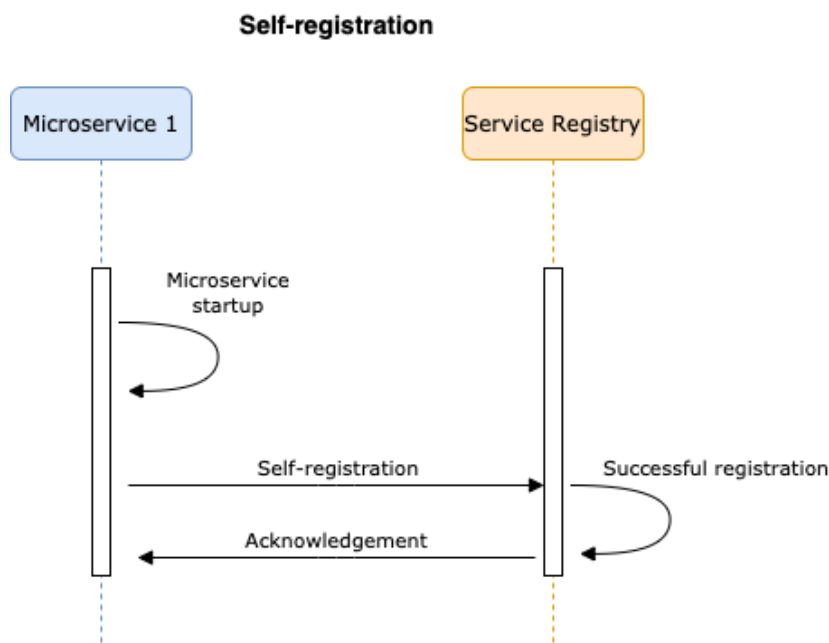


Figure 3-5: Self-registration sequence diagram

### 3.4.2.2 Self-deregistration

The specific functionality handles the self-deregistration of the ML/DL microservice in the Service Registry of the INFINITECH Open API Gateway. As a prerequisite, the registry client should be integrated with the microservice implementation. During the microservice shutdown, the microservice is self-deregistered from the Service Registry.

Table 3-6: Service Registry – Self-deregistration

<b>Stakeholders involved:</b>	ML/DL microservice, Service Registry
<b>Pre-conditions:</b>	1. The existence of a ML/DL microservice that has integrated the registry client on its implementation and has been successfully registered in the Service Registry.
<b>Post-conditions:</b>	1. The ML/DL microservice is successfully deregistered from the Service Registry
<b>Data Attributes</b>	1. The identifier of the microservice that needs to be deregistered



<b>Normal Flow</b>	<ol style="list-style-type: none"> <li>1. The ML/DL microservice initiates shutdown. The integrated registry client is invoked.</li> <li>2. The registry client communicates with the Service Registry and self-deregisters.</li> </ol>
<b>Pass Metrics</b>	<ol style="list-style-type: none"> <li>1. The ML/DL microservice is successfully deregistered from the Service Registry</li> </ol>
<b>Fail Metrics</b>	<ol style="list-style-type: none"> <li>1. The ML/DL microservice failed to deregister from the Service Registry</li> </ol>

**Self-deregistration**

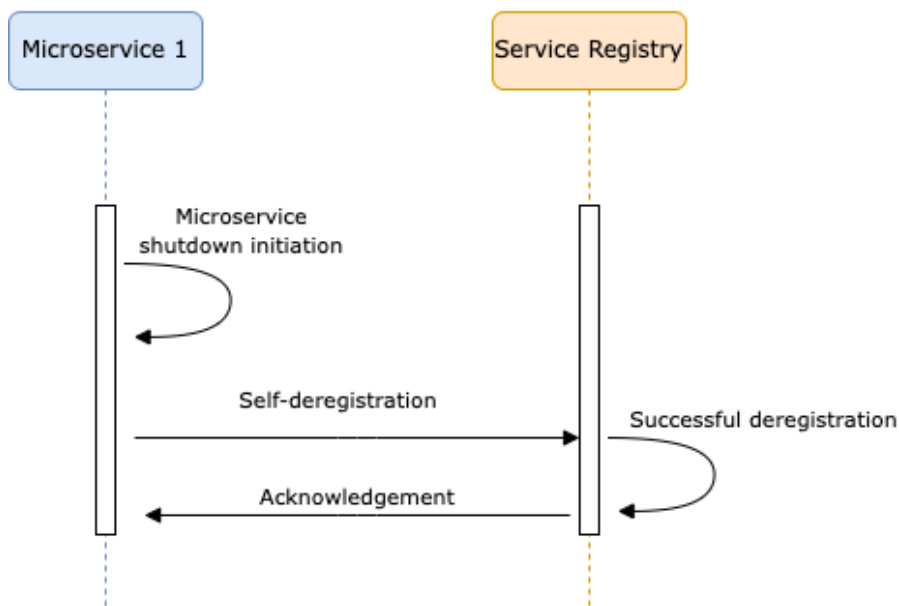


Figure 3-6: Self-deregistration sequence diagram

**3.4.2.3 Periodic Health Check**

The specific functionality handles the need for periodic health check execution on the registered microservices to ensure their availability. In this context, the Service Registry in every period checks if each registered service is reachable and operating as expected. In the case where a microservice is not responding, then the microservice is marked as in an “Unhealthy” state. At this point, the microservice can properly self-deregister when it comes back into service or it can be removed by the administrator of the INFINITECH Open API Gateway at any time.

Table 3-7: Service Registry – Periodic Health Check

<b>Stakeholders involved:</b>	ML/DL microservice, Service Registry
<b>Pre-conditions:</b>	<ol style="list-style-type: none"> <li>1. The existence of a ML/DL microservice that has been successfully registered in the Service Registry and is operating as expected.</li> <li>2. The existence of a ML/DL microservice that has been successfully registered in the Service Registry and is not responding or is not reachable anymore.</li> </ol>

<b>Post-conditions:</b>	1. The operational ML/DL microservice remains in the Service Registry in a “Healthy” state and the unresponsive ML/DL microservice is flagged as “Unhealthy”
<b>Data Attributes</b>	N/A
<b>Normal Flow</b>	<ol style="list-style-type: none"> <li>1. The Service Registry initiates the health check operation for a specific microservice that is not reachable or irresponsive.</li> <li>2. The specific microservice is flagged as “Unhealthy”</li> <li>3. The Service Registry initiates the health check operation for a specific microservice that is operational.</li> <li>4. The specific microservice remains in the Service Registry in a “Healthy” state</li> </ol>
<b>Pass Metrics</b>	1. The operational microservice remains in the Service Registry in a “Healthy” state, while if unresponsive is in an “Unhealthy” state.
<b>Fail Metrics</b>	1. The operational microservice is flagged as “Unhealthy” and / or the unresponsive is flagged as “Healthy”

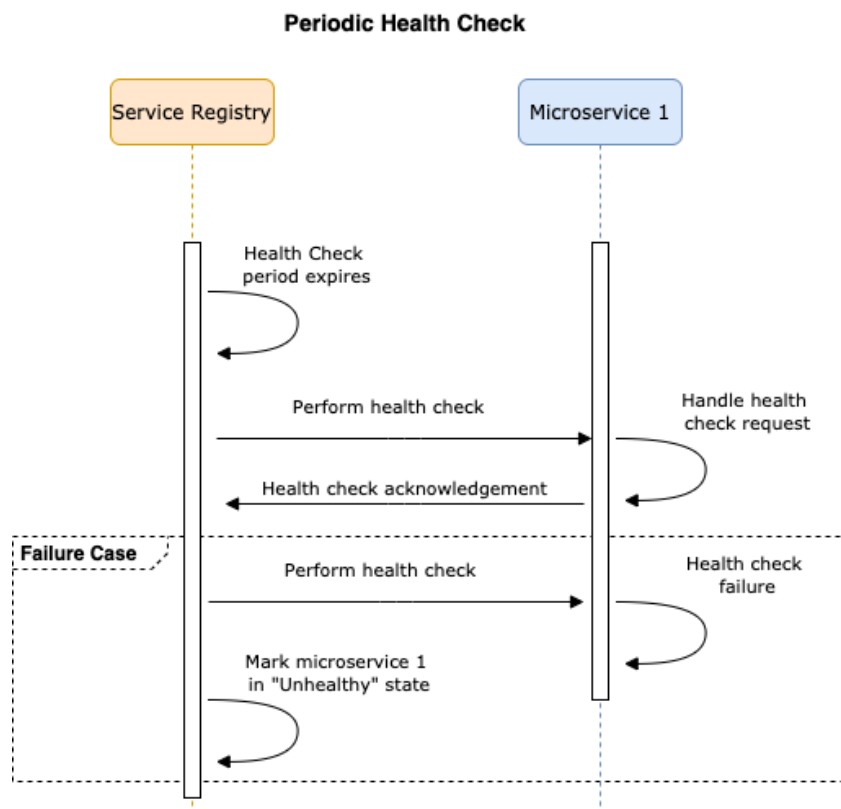


Figure 3-7: Periodic Health Check sequence diagram

## 4 Implementation of the INFINITECH Open API Gateway

### Updates from D5.11:

*The particular section documents the necessary updates related to the advancements on the implementation of the INFINITECH Open API Gateway. In detail, the following documentation is introduced:*

- *The implementation details of the advanced request handling operations which are performed in order to facilitate 1-to-many microservices invocation*
- *The enhancement of the service registry for high availability, fault tolerance and flexibility*
- *A walkthrough of the provided solution from the user's perspective*

As described in section 3.3, the INFINITECH Open API Gateway adopts a modular architecture and is composed of two core modules, namely the Gateway and the Service Registry. The integration of those two modules formulates the overall solution that provides the single-entry point for the underlying added-value analytics functionalities which are made available in the form of microservices. The implementation of both modules is based on the formulated design specifications which are also documented in section 3.2 of the current deliverable.

In the following subsections, the implementation details of the final fully functional version of the INFINITECH Open API Gateway are documented in detail. In particular, for each module the functionalities which were implemented in the course of development of the specific solution are documented providing an overview of the delivered module along with the details on how these modules are interacting in order to provide the end-to-end functionalities of the INFINITECH Open API Gateway.

### 4.1 Gateway

The main role of the Gateway module is to provide the single-entry point for all incoming HTTP requests and to properly handle them by invoking the appropriate microservices as defined in the parameters of the request, ensuring the discovery and intercommunication of the underlying microservices where needed.

To meet its goals, the Gateway is divided into two main components namely the *Gateway Backend* and the *Gateway Frontend* components. The Gateway Backend component undertakes all the core functionalities of the Gateway module by performing all the required request-handling operations. As the Gateway module acts as an advanced reverse proxy which offers additional functionalities than just forwarding the request to the appropriate microservice, multiple operations are performed and orchestrated in the background in order to effectively handle and process all incoming HTTP requests. The Gateway Frontend is providing the single user interface of the INFINITECH Open API Gateway through which the clients can discover and explore the details of the registered microservices, providing them with access to the corresponding Open API documentation of each registered microservice. Both components interact through well-established APIs in order to realise the workflows designed in Section 3 of the current deliverable.

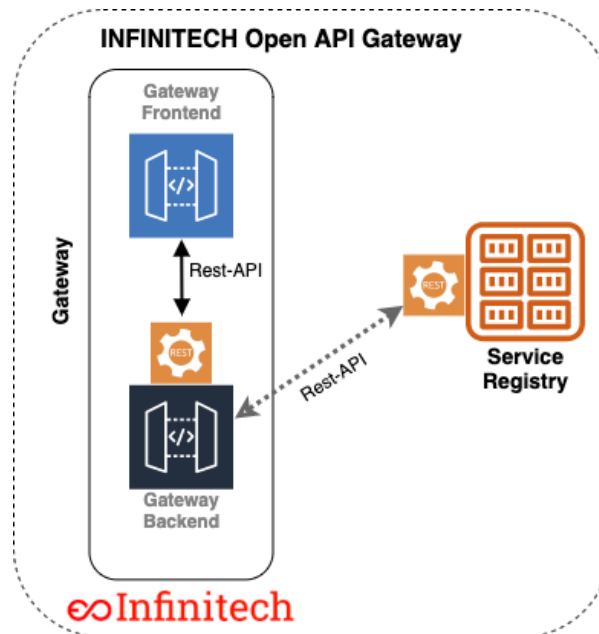


Figure 4-1: Gateway Module Architecture and interactions

The core part of the Gateway Backend component is based on the Spring Cloud Gateway library<sup>3</sup> that is offered as part of the Java-based Spring Cloud Ecosystem<sup>4</sup>. The basic concepts of this library are routes and filters. Route constitutes a basic concept of the Spring Cloud Gateway as it defines the routing of a specific request to the underlying microservice upon a successful match. However, before this request is proxied to the appropriate microservice a set of filters are applied which are able to apply modifications in the incoming HTTP request or in the outgoing HTTP response. Filters are divided into the ones applied prior the routing of the request to the requested microservice, hence applying “pre” filter logic, and the ones applied on the microservice’s response, which are applying “post” filter logic.

For the purposes of the INFINITECH Open API Gateway, the specific implementation has been extended taking the implementation of the standard API Gateway offered by Spring Cloud Gateway one step further to being more dynamic with the introduction of the dynamic routing and dynamic filtering features, which were introduced in the prototype version as documented in deliverable D5.11, as well as the Advanced Request Handling operations feature which is introduced on top of the existing features in the final fully functional version as documented in the current deliverable. To this end, the complete list of features is as follows:

- a) **Dynamic routing** which is not relying on “hardcoded” routes in the configuration but leverages the dynamic service registry that is offered by the Service Registry module. Dynamic routing is implemented taking into consideration the changes in the availability and network access information of microservices in the cloud and containerised environments.
- b) **Dynamic filtering** which is applied on all incoming HTTP requests with the dynamic configuration of the relevant filters. This dynamic configuration is compiled based on the updated and accurate metadata of each microservice which are dynamically updated and maintained during their self-registration in the Service Registry.
- c) **Advanced Request Handling operations** which are performed in order to facilitate 1-to-many microservices invocation. The specific novel feature enables the processing of requests which are translated internally as the invocation of multiple microservices whose results are aggregated and returned to the requestor. In case where multiple dynamically-deployed microservice instances should be invoked to proper address the requestor’s needs, this can be achieved via a request on a

<sup>3</sup> Spring Cloud Gateway, <https://spring.io/projects/spring-cloud-gateway>

<sup>4</sup> Spring Cloud Ecosystem, <https://spring.io/projects/spring-cloud>

single advanced endpoint that undertakes the responsibility of the collection and aggregation of the results.

These new features, extending the Spring Cloud Gateway, are effectively integrated into the Gateway module’s request handling process, which constitutes the core offering of the Gateway module, extending its capabilities and facilitating the implementation of the designed features of the INFINITECH Open API Gateway. Figure 4-2 displays the implemented request handling process and how routes and filters are leveraged in this process.

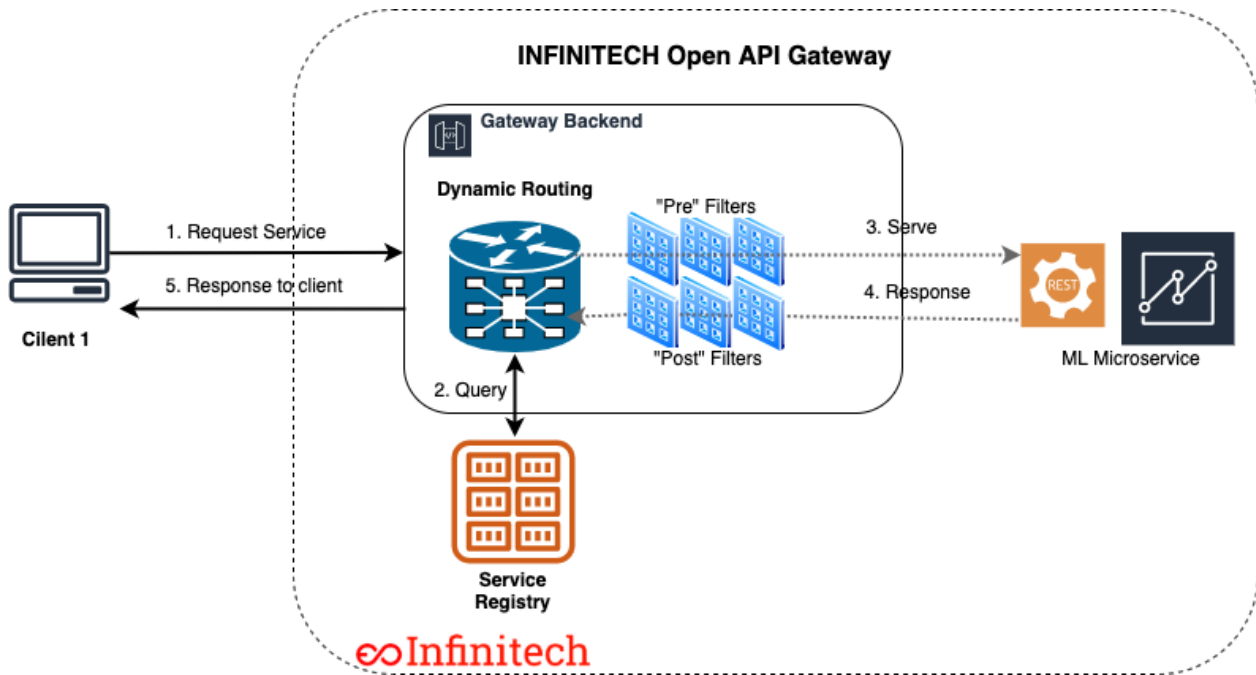


Figure 4-2: Gateway Request Processing Handling

As documented above, the final fully functional version of the INFINITECH Open API Gateway introduces the Advanced Request Handling operations leveraging the already available from the prototype version dynamic routing and dynamic filtering features. In the case of 1-to-many microservices invocation, the request processing handling is altered in order to accommodate the requirement of interacting with multiple microservices. At first, the INFINITECH Open API Gateway provides an advanced endpoint which is capable of receiving and handling requests that result in the invocation of multiple microservices, orchestrating their invocation as well as the collection and aggregation of the respective results as provided by the invoked microservices. Hence, the requestor can make a single request to the specific endpoint and receive the required response in an effortless manner. The Gateway Backend encapsulates a sophisticated logic, as per the design specifications presented in section 3, that translates the requests into multiple internal requests to the requested microservices, taking care at the same time any possible dependencies and fault tolerance aspects of the performed operations. The Gateway Backend consults the Service Registry in order to retrieve the details of the dynamically-deployed microservice instances which will be invoked and generates the respective requests as instructed by the requestor. Once all the processing is finished and the respective results are collected and aggregated in an asynchronous manner they are sent back to the requestor. Figure 4-3 displays the implemented request handling process in the case of 1-to-many microservices invocation.

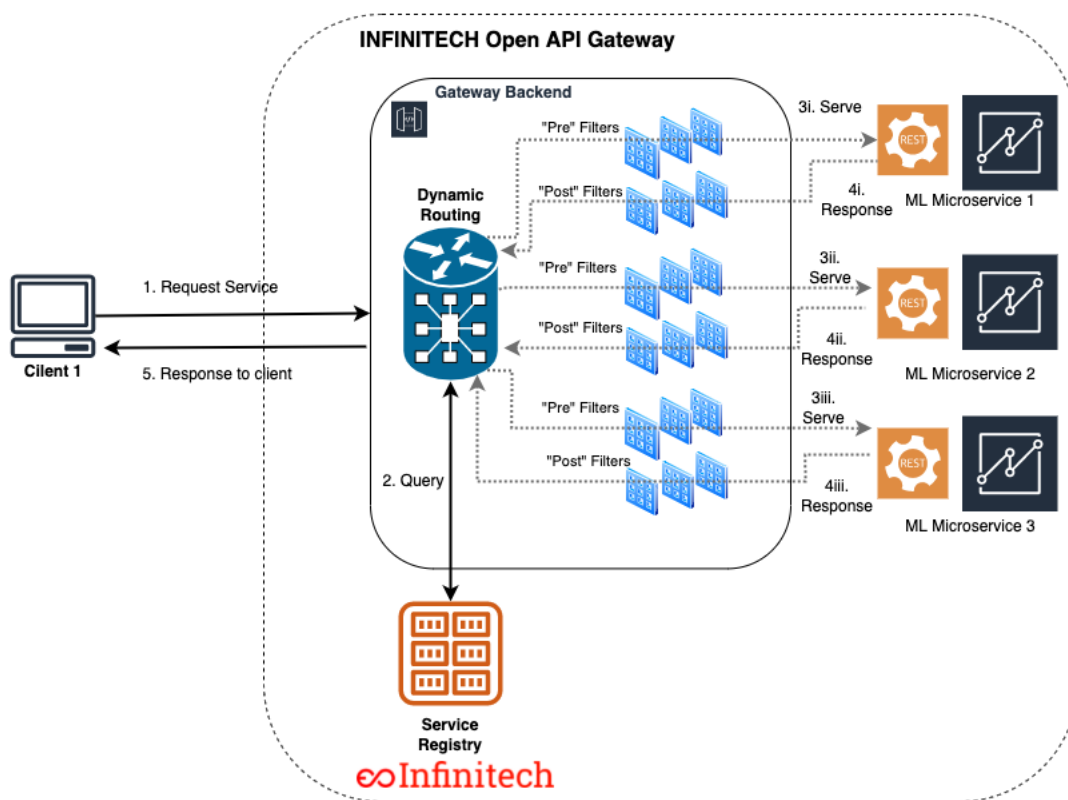


Figure 4-3: Gateway Request Processing Handling (1-to-many)

It should be noted however that the 1-to-many microservices invocation process has two limitations. The first one is related to the nature of the response types of the microservices. In detail, the scenario of the different response types of the invoked microservices cannot be supported (i.e. the case where one microservice responds with a JSON file and while the second microservice responds with a multipart file) as this limitation is imposed by the HTTP protocol where mixing of different response types is not allowed by definition. The second limitation is related to the sequential execution of the requests received by the INFINITECH Open API Gateway. As described in section 3, the INFINITECH Open API Gateway adopts the reactive programming on its design principles that enable its non-blocking operation where multiple requests can be handled simultaneously. Hence, the INFINITECH Open API Gateway operates in the optimal manner without the problem of bottlenecks when heavy or time-consuming requests are handled. In the case where one request should be finished first before proceeding to the next one, it is the responsibility of the client to handle this sequential processing.

### 4.1.1 Gateway Backend

The Gateway Backend interacts with the Service Registry module via the well-defined APIs which are exposed by the Service Registry in order to automatically and dynamically retrieve the list of registered microservices as well as their appropriate metadata in order to compile the list of dynamic routes for which the Gateway serves as an advanced reverse proxy. This dynamic routing is supplemented with the dynamic filtering with a set of filters that are applied on all incoming HTTP requests. In detail, the Gateway Backend component is leveraging a series of configurable, built-in filters, which are offered by the Spring Cloud Gateway in order to implement the design workflow of the Gateway module. In addition to this, it exploits the offer by the Spring Cloud Gateway library functionality of developing and employing custom filters where needed.

Towards this end, the list of filters currently implemented and utilised in the Gateway Backend component are as follows:

#### Logging Filter:

The Logging filter is a custom filter which intercepts all incoming HTTP requests and logs the following information: the request type (GET, POST, PUT, DELETE, etc.), the original incoming URI and the transformed URI/service name that will eventually handle the request. It should be noted that for privacy preservation reasons and compliance with GDPR, sensitive information items such as headers, parameters and body are not being logged, since they may contain sensitive information i.e. tokens.

Auth Filter:

The Auth Filter constitutes a custom filter which intercepts all incoming requests and checks whether the request should be authenticated or authorized before reaching the appropriate microservice. During service self-registration, each microservice can define two parameters, namely the authentication and the authorization parameters as described in section 4.2, that control this behaviour. It should be noted that the specific filter is implemented but currently not leveraged by the underlying ML/DL microservices of INFINITECH, since it requires an additional authentication/authorization which is not provided yet.

Rewrite Path Filter:

The Rewrite Path Filter<sup>5</sup> is a built-in filter which is configured and utilised within the Gateway Backend to appropriately modify the URI of the incoming HTTP request in order to be proxied to the underlying microservice.

Prefix Path Filter:

The Prefix Path Filter<sup>6</sup> is built-in filter which is configured and utilised within the Gateway Backend to automatically prepend the context path of the underlying microservice in the request URI of the incoming HTTP request, so that the client does not have to include it explicitly when triggering the gateway.

Add Request Header Filter:

The Add Request Header Filter<sup>7</sup> is a built-in filter which is dynamically configured and utilised within the Gateway Backend to properly set the X-Forwarded-Prefix header, so that when the request arrives to the appropriate service, it knows that it was being reverse proxied before arriving. The specific approach is utilised in the case where the Swagger user interface is utilised to trigger an endpoint of a microservice.

Retry Filter:

The Retry Filter<sup>8</sup> is a built-in filter which is dynamically configured and utilised within the Gateway Backend in order for the gateway to automatically retry upon a failed request. The specific filter is dynamically configured for each separate microservice independently based on the appropriate metadata values of each microservice propagated during self-registration, as described in section 4.2.

Circuit Break Filter:

The Circuit Break Filter<sup>9</sup> is a built-in filter which is dynamically configured and utilised within the Gateway to apply the Circuit Breaker pattern, a mechanism for properly handling requests to “slow” or “failed” services, by immediately returning an *HTTP 503 - Service Unavailable* code, preventing a further burden to that service.

---

<sup>5</sup> Rewrite Path Filter, <https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#the-rewritepath-gatewayfilter-factory>

<sup>6</sup> Prefix Path Filter, <https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#the-prefixpath-gatewayfilter-factory>

<sup>7</sup> Add Request Header Filter, <https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#the-addrequestheader-gatewayfilter-factory>

<sup>8</sup> Retry Filter, <https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#the-retry-gatewayfilter-factory>

<sup>9</sup> Circuit Breaker Filter, <https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#spring-cloud-circuitbreaker-filter-factory>

The specific filter constitutes a core filter of the implementation as it provides the required fault-tolerance feature. In order to be effective and efficient, the following parameters are set:

- *slidingWindowType*: The specific parameter configures the type of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed. Sliding window can either be count-based or time-based.
- *slidingWindowSize*: The specific parameter configures the size of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed.
- *minimumNumberOfCalls*: The specific parameter configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate or slow call rate. For example, if *minimumNumberOfCalls* is 10, then at least 10 calls must be recorded, before the failure rate can be calculated. If only 9 calls have been recorded the CircuitBreaker will not transition to open even if all 9 calls have failed.
- *permittedNumberOfCallsInHalfOpenState*: The specific parameter configures the number of permitted calls when the CircuitBreaker is half open.
- *failureRateThreshold*: The specific parameter configures the failure rate threshold in percentage. When the failure rate is equal or greater than the threshold, the CircuitBreaker transitions to open and starts short-circuiting calls.
- *waitDurationInOpenState*: The specific parameter configures the time that the CircuitBreaker should wait before transitioning from open to half-open.
- *slowCallDurationThreshold*: The specific parameter configures the duration threshold above which calls are considered as slow and increases the rate of slow calls.
- *slowCallRateThreshold*: The specific parameter configures a threshold in percentage. The CircuitBreaker considers a call as slow when the call duration is greater than *slowCallDurationThreshold*. When the percentage of slow calls is equal or greater than the threshold, the CircuitBreaker transitions to open and starts short-circuiting calls.

The Circuit Break Filter implementation that is utilised for the fault-tolerance feature is supplemented with a lightweight, easy-to-use fault tolerance library named Resilience4j<sup>10</sup> from which the Time Limiter filter is used to force a timeout in the requests if they exceed a specific threshold.

#### Request Rate Limiter Filter:

The Request Rate Limiter Filter is a built-in filter which is dynamically configured and utilised within the Gateway Backend to optionally apply a rate limiting in the underlying microservice, in order to determine if the current request is allowed to proceed. If it is not, a status of *HTTP 429 - Too Many Requests* is returned. The algorithm behind the implementation of the rate limiting is the Token Bucket algorithm<sup>11</sup> and it uses a Redis<sup>12</sup> instance to keep track of the number of requests performed. The specific filter is also dynamically configured for each separate microservice independently based on the appropriate metadata values of each microservice propagated during self-registration, as described in section 4.2.

In addition to the filters, in the final fully functional version of the INFINITECH Open API Gateway the Gateway Backend has implemented the execution of advanced request handling operations which are enabling the 1-to-many microservices invocation. To support this operation four classes are implemented in total, namely the *RequestDto*, *ResponseDto*, *MultiRequestHandler* and *MultiRequestRouter*.

#### RequestDto:

This class represents a Data Transfer Object (DTO) which is used to describe a single request that will be made during the 1-to-many invocation. This class has the following attributes:

- *endpoint*: *String*: A string value representing the endpoint to be triggered.

<sup>10</sup> Resilience4j, <https://resilience4j.readme.io/docs>

<sup>11</sup> Token Bucket algorithm, [https://en.wikipedia.org/wiki/Token\\_bucket](https://en.wikipedia.org/wiki/Token_bucket)

<sup>12</sup> Redis, <https://redis.io/>



- `httpMethod: HttpMethod`: The HTTP method to use for this call (GET, POST, etc..).
- `headers: Map<String, String>`: The headers to include in this call, if any.
- `body: Map<String, Object>`: The body in JSON format to include in this call, if any.
- `queryParams: Map<String, List<String>>`: Any additional query parameters that need to be included in the call.

#### ResponseDto:

This class represents a Data Transfer Object (DTO) which is used to describe a single response of a single request during the 1-to-many invocations. It holds the following attributes:

- `statusCode: int` : The status code of the response (e.g. 200, 500, etc..)
- `headers: Map<String, String>`: The response headers.
- `data: Map<String, String>`: The data of the response in JSON format (if any).
- `requestOrder: int`: This attribute is used to associate the response with the original requests. Since requests are being performed in an asynchronous manner and the responses are being received in the same way, it could be hard to associate which response corresponds to which request. To do so, this attribute is being used to match the response with the original order of the RequestDTOs, as those were received in the 1-to-many invocation endpoint.

#### MultiRequestHandler:

The specific class implements the functionalities related to the request handling and its translation into the invocation of multiple microservices. In this class, the following functions were implemented:

- `handleMultiRequest(serverRequest: ServerRequest): List<ResponseDto>`: This is the main function of the class. It accepts as a parameter a `ServerRequest` object which encapsulates the `RequestDto` objects and iterates over those DTOs, performs the corresponding requests and aggregates the returned responses before sending them back to the user, as a collection of `ResponseDto` objects.
- `checkInput(requestDto: RequestDto): String`: This is an internal function that is used to check if the structure of the incoming `RequestDto` objects is correct. If no errors are identified, the procedure continues, otherwise, a response status with code 400 Bad Request is returned.

#### MultiRequestRouter:

The specific class is a request router (a.k.a REST Controller) that exposes the necessary endpoints to allow the initiation of the 1-to-many request handling. In this class, the following function was implemented:

- `multiRequestRoutes(multiRequestHandler: MultiRequestHandler)` :
- `RouterFunction<ServerResponse>`: This function accepts as parameter a list of `MultiRequestDto` objects that represent the multiple requests to be made and calls the `handleMultiRequest` function of the `MultiRequestHandler` class to handle the process.

## 4.1.2 Gateway Frontend

The Gateway Frontend is offering the single user interface of the INFINITECH Open API Gateway acting as the single point of reference of the documentation of the Open APIs of the registered microservices. The Gateway Frontend leverages the embracement of the Open API specification in order to present and publish the documentation of the offered Open APIs of the microservices enabling their easy and effortless discovery and consumption from the clients of the INFINITECH Open API Gateway. In particular, the Gateway Frontend acts as the mediator between the clients of the INFINITECH Open API Gateway and the provided by the registered microservices documentation of their Open APIs by providing the means for all registered microservices to publish their Open API documentation.

To this end, the purpose of the Gateway Frontend is two-fold: a) to display the list of registered microservices and their relevant metadata or properties and b) to display Open API documentation of each registered

microservice. It should be noted at this point that the Gateway Frontend is not responsible for the generation or undertaking the compilation of the aforementioned Open API documentation for each microservice, as this is provided by each microservice during their self-registration, but only for the publishing of the provided documentation.

The Gateway Frontend constitutes a Single Page Application (SPA) which is based on the React JS library<sup>13</sup> displaying the dynamic list of the registered microservices. In particular, the well-established Material-UI framework<sup>14</sup> is leveraged which offers a novel framework for the design and implementation of the user-friendly and fast user interfaces.

For each microservice, the following information is displayed:

- *Service Name*: The name of the registered microservice as provided during its self-registration.
- *Authorization*: A Boolean value that illustrates the existence of an authorisation restriction of the specific microservice.
- *Authentication*: A Boolean value that illustrates the existence of an authentication restriction of the specific microservice.
- *Open-API-Swagger Link*: A link URL to Open API documentation of the Open APIs of the specific microservice which is based on Swagger.

By navigating to a specific microservice of the list, the user is able to access the Open API documentation of the Open APIs via two different options (Figure 4-4). The first option generates a redirection to the selected Open API documentation where the user can read all the relevant information of the provided Open APIs. The second option displays the Open API documentation in an integrated manner by expanding the list item to display the relevant documentation while collapsing the rest of the items of the list.

The Gateway Frontend is composed of a set of functions which are utilised in order to collect, aggregate and display the dynamic list of the registered microservices along with their properties and metadata. The list of implemented functions are as follows:

- `fetchMicroservicesData(endpoint: Object)`: The specific function fetches microservices' information and attributes via the `'/actuator/gateway/routes'` endpoint.
- `createTable(headers: Object)`: The specific function undertakes the creation of the schema, headers, dynamic pagination, searching bar and all the functionalities which are supported by the final fulfilled Table.
- `dataTransformation(res: Object)`: The specific function transforms the fetched microservices' data into usable by user interface format before ingesting it into the Table.
- `dataParsing(data: Object)`: The specific function undertakes the parsing of the data into the Table and makes them reachable by the user.
- `swaggerIntegration(swaggerPath: Object)`: The specific function utilises an `iframe` component and custom Material Table's actions to integrate the OpenAPI-Swagger Documentation Page of the desired microservice.

---

<sup>13</sup> React, <https://reactjs.org/>

<sup>14</sup> Material UI, <https://material-ui.com/>

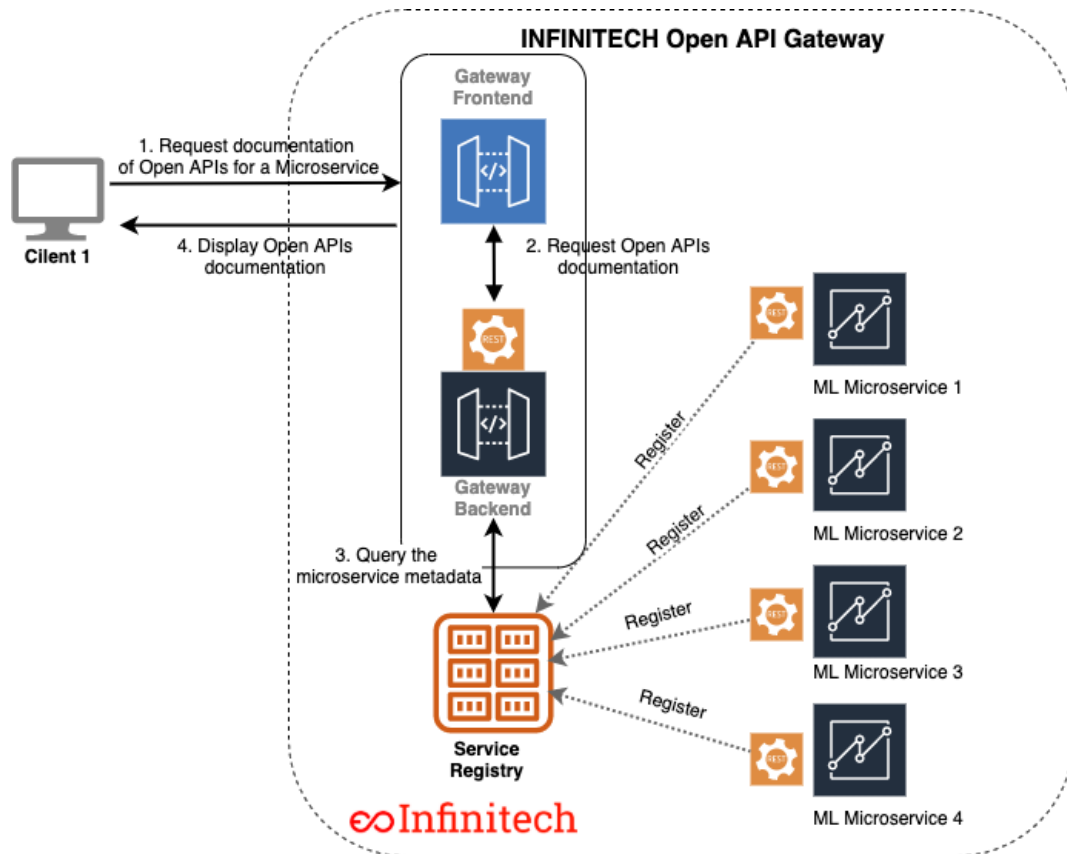


Figure 4-4: Discovery of microservices' Open API documentation

## 4.2 Service Registry

The main role of the Service Registry is to provide the database that effectively maintains the updated network locations of the registered microservices, offering the mechanism for self-registration and deregistration and the sophisticated mechanism that performs health check operations on these microservices.

Towards this end, the dominant open-source tool named Consul<sup>15</sup> is leveraged that provides out-of-the-box service registry, service discovery and health checking. Service Registry is realised by properly configuring Consul and integrated it with the Gateway module via well-defined API interfaces in order to realise the workflows designed in Section 3 of the current deliverable. In particular, the Service Registry, which is based on the design specifications that are documented in Section 3 also, is complementing the Gateway module by supporting the operations performed by the Gateway covering the service discovery aspects of the solution.

Consul undertakes the critical part of the integration of the microservices and provides the required service discovery operations. However, the successful registration and utilisation of any candidate microservice adheres to a specific set of rules and requirements that should be met. The integration requirements imposed by the Service Registry are as follows:

- *Self-registration and self-deregistration*: Each microservice should be able to self-register and deregister to the Service Registry utilising the service registry mechanism provided by Consul.
- *Required metadata on registration*: During service registration to consul, the following metadata should be provided:
  - *Authentication* (Boolean): It indicates if anyone accessing this service should be authenticated.

<sup>15</sup> Consul, <https://www.consul.io/>

- *Authorization* (Boolean): It indicates if the entity accessing the specific microservice should be authorized to do so.
- *ContextPath* (String): The context path of the microservice. It can be an empty string "", or something like "/microservice1", "/linear-regression-model", etc.
- *SwaggerPath* (String - Optional): The path holding the JSON representation of the Open API documentation. For instance, when using Spring's springdoc library, this defaults to "/v3/api-docs", or if using FastAPI "/openapi.json". Of course those defaults can change, and that's why we need this info.
- *rateLimiterEnabled* (Boolean - Optional): It indicates if rate limiting should be applied. Default is true.
- *rateLimiterScope* (String - Optional): Defines the rate limiting scope. Accepted values are "global" or "user", default value is "global". Global means that rate limiting will be applied for all requests, while user means that rate limiting will be applied per user (requires an authentication/authorization mechanism to be in place).
- *rateLimiterReplenishRate* (Integer - Optional): The number of requests that are being replenished every second. Default is 100.
- *rateLimiterBurstCapacity* (Integer - Optional): The total number of requests that can be accepted per second. Default is 100.
- *rateLimiterRequestedTokens* (Integer - Optional): The cost of each request. Default is 1.
- *retryAttempts* (Integer - Optional): The number of times the request will be retried in case of failure. Default is 3.
- *retryFirstBackoff* (String - Optional): The amount of time after which the first retry attempt will take place. Default is 10ms.
- *retryMaxBackoff* (String - Optional): The maximum backoff to apply. Default is 50ms.
- *retryBackoffFactor* (Integer - Optional): The backoff retry factor. Default is 2.
- *retryBackoffBasedOnPreviousValue* (Boolean – Optional): If false, retries are performed after a backoff interval of  $\max(\text{retryFirstBackoff} * (\text{retryBackoffFactor} ^ n), \text{retryMaxBackoff})$ , where n is the iteration. If true, the backoff is calculated by using  $\max(\text{prevBackoff} * \text{retryBackoffFactor}, \text{retryMaxBackoff})$ . Default is false.
- *Unique service name*: During registration, the name of the microservice under registration should be unique, otherwise the problem of one microservice overriding another microservice may occur or multiple microservices may be considered as multiple instances of the same microservice which will inevitably lead to an unwanted load balancing.
- *Microservices wrappers*: As per the design specifications of the INFINITECH Open API Gateway, only microservices can be registered. Hence, in the case of ML/DL algorithms each algorithm should be wrapped as a microservice using a corresponding framework, such as Spring Boot for Java, FastAPI for Python, and all the algorithm's functionalities (e.g. train, test, etc) should be exposed as endpoints.
- *Open API specification*: The INFINITECH Open API Gateway embraces Open APIs. While the publishing of the documentation of the APIs of the microservice is optional, if the specific feature is exploited each microservice should document their endpoints using Open API specification.
- *Invoking via the Swagger UI*: In the case where the triggering of the microservice's endpoints is allowed also through the Swagger User Interface then the microservice should be configured properly, as if it was to be behind a reverse proxy, in order to handle X-Forwarded-For and X-Forwarded-Prefix headers.

Upon successful registration to the service registry, each microservice can be accessed in the following path:

```
{GATEWAY_IP:GATEWAY_PORT | GATEWAY_DOMAIN}/{microservice-name}/{endpoint}
```

It should be noted that the context-path is not required as the Gateway module adds it automatically based on the provided metadata.

Service Registry holds a key role in the INFINITECH Open API Gateway solution as it maintains and provides the crucial information of the updated network location of each registered microservices as well as the latest

health status of each microservice. To this end, it is imperative that this information that constitutes the single point of truth for the underlying microservices operates with high availability and flexibility. In the fully functional version of the Service Registry, the ability of Consul to operate in a distributed and fault tolerant manner by the use of a Consul cluster is leveraged. When operating in cluster mode, Consul provides the required high availability without the need of a load balancer ensuring that the service registration, deregistration and discovery operations will not be interrupted in the case of a cluster node failure. The particular offering of Consul has been also leveraged by the INFINITECH Open API Gateway by applying the appropriate configuration on Consul during the service start-up and several optimisations in the exposed services of the Service Registry.

### 4.3 The INFINITECH Open API Gateway Solution

The current deliverable constitutes an accompanying report documenting the delivery of the final fully functional version of the INFINITECH Open API Gateway Solution. As explained in the previous subsections, the implementation of the solution has been driven by the design specifications documented in section 3 towards the delivery of a sophisticated solution with TRL level 6. The delivered solution is available in the INFINITECH Marketplace in order to be leveraged by all financial and insurance sectors stakeholders. In addition to this, the INFINITECH Open API Gateway is part of the INFINITECH end-to-end framework for the deployment of ML/DL-based microservices which is an end-to-end framework developed, in collaboration with Task 5.4 of WP5, for the definition and instrumentation of ML/DL pipelines in a standardized manner in order to produce ready-to-use ML and DL models as well as to deploy and publish these models as microservices. In this framework, the INFINITECH Open API Gateway offerings are leveraged for the easy and efficient discovery and usage of the ML/DL-based microservices. The complete documentation of the aforementioned framework is documented in the deliverables of Task 5.4, namely deliverables D5.8 and D5.9.

In the following paragraphs, a walkthrough of the provided solution from the user's perspective is presented along with a guideline for the smooth and successful integration of the delivered solution within financial services and solutions.

In order to leverage the delivered solution, the financial and insurance sectors stakeholders should access the INFINITECH Marketplace and select the delivered solution<sup>16</sup>. Upon reading the available documentation of the INFINITECH Open API Gateway Solution, they can download the fully functional version in the form of two Docker images (API Gateway and API Gateway UI). Since the delivered solution is provided through Docker images, the latest version of Docker and docker-compose services should be available and installed on the system that the INFINITECH Open API Gateway will run. Once these prerequisites are met, the docker-compose file which is documented in the following table is utilised in order to install and start the INFINITECH Open API Gateway Solution.

```
version: "3.8"

services:

  consul:
    image: consul:1.11.4
    container_name: api_gateway_consul
    ports:
      - "8400:8400"
      - "8500:8500"
      - "8600:8600"
      - "8600:8600/udp"
    volumes:
```

<sup>16</sup> INFINITECH Open API Gateway in the Marketplace, <https://marketplace.infinitech-h2020.eu/assets/infinitech-open-api-gateway>

```

- api-gateway-consul-data:/consul/data
networks:
- api-gateway-network
command: "agent -server -ui -node server-1 -bootstrap-expect 1 -
client 0.0.0.0"
healthcheck:
  test: [ "CMD", "curl", "--fail",
"http://127.0.0.1:8500/v1/health/node/${NODE_NAME}" ]
  interval: 10s
  timeout: 5s
  retries: 5
  start_period: 10s
logging:
  options:
    max-size: "5MB"
    max-file: "5"

redis:
  image: redis:6.2.6-alpine3.15
  container_name: api_gateway_redis
  ports:
    - "6379:6379"
  volumes:
    - api-gateway-redis-data:/data
  networks:
    - api-gateway-network
  command: redis-server --appendonly yes
  healthcheck:
    test: [ "CMD", "redis-cli", "ping" ]
    interval: 30s
    timeout: 20s
    retries: 3
    start_period: 10s
  logging:
    options:
      max-size: "5MB"
      max-file: "5"

api-gateway:
  image: harbor.infinitech-h2020.eu/interface/api-gateway:1.1.1
  container_name: api_gateway
  ports:
    - "8080:8080"
  volumes:
    - api-gateway-logs-data:/logs
  networks:
    - api-gateway-network
  environment:
    REDIS_HOST: redis
    REDIS_PORT: 6379
    CONSUL_HOST: consul
    CONSUL_PORT: 8500
    GATEWAY_FORWARDED_ENABLED: false
    CIRCUIT_BREAKER_SLIDING_WINDOW_TYPE: COUNT_BASED
    CIRCUIT_BREAKER_SLIDING_WINDOW_SIZE: 10
    CIRCUIT_BREAKER_MINIMUM_NUMBER_OF_CALLS: 10
    CIRCUIT_BREAKER_PERMITTED_NUMBER_CALLS_IN_HALF_OPEN_STATE: 10

```

```

CIRCUIT_BREAKER_FAILURE_RATE_THRESHOLD: 50
CIRCUIT_BREAKER_WAIT_DURATION_IN_OPEN_STATE: 30s
CIRCUIT_BREAKER_SLOW_CALL_DURATION_THRESHOLD: 5m
CIRCUIT_BREAKER_SLOW_CALL_RATE_THRESHOLD: 50
TIMELIMITER_TIMEOUT_DURATION: 10m
TZ: Etc/UTC
SPRING_PROFILES_ACTIVE: prod
command: bash -c "java -Xms256M -Xmx2048M -jar api-gateway.jar"
healthcheck:
  test: [ "CMD", "curl", "--fail",
"http://127.0.0.1:8080/actuator/health" ]
  interval: 10s
  timeout: 5s
  retries: 5
  start_period: 15s
logging:
  options:
    max-size: "5MB"
    max-file: "5"
depends_on:
  consul:
    condition: service_healthy
  redis:
    condition: service_healthy

api-gateway-ui:
  image: harbor.infinitech-h2020.eu/interface/api-gateway-ui:1.1.0
  container_name: api_gateway_ui
  ports:
    - "3000:80"
  environment:
    GATEWAY_URL: http://192.168.1.19:8080 # TODO: Your IP goes there
  logging:
    options:
      max-size: "5MB"
      max-file: "5"

volumes:
  api-gateway-consul-data:
    name: api_gateway_consul_data
    driver: local
  api-gateway-redis-data:
    name: api_gateway_redis_data
    driver: local
  api-gateway-logs-data:
    name: api_gateway_logs_data
    driver: local

networks:
  api-gateway-network:
    name: api_gateway_network
    driver: bridge

```

It should be noted that the only change required on the provided docker-compose file is the editing of the value of the GATEWAY\_URL variable with the one that will actually host the provided solution.

Once the docker-compose file is available locally as a YAML file along with the two aforementioned docker images, the INFINITTECH Open API Gateway can be easily started via the following simple command:

**docker-compose up -d**

Within a few seconds, the users can navigate to their browser at the address: <http://localhost:3000> where they will be able to access the user interface of the INFINITECH Open API Gateway.

Once the installation is successfully performed, the users are able to register their microservices in order to be discoverable via the INFINITECH Open API Gateway by the potential stakeholders of these microservices. The registration of the available microservices is adhering to a specific set of rules and requirements that should be met, as explained thoroughly on section 4.2. Once all these rules and requirements are met, the last step of the registration includes the integration of the candidate microservice with Consul that constitutes the core service of the Service Registry of the INFINITECH Open API Gateway. Although there are many options in terms of libraries for the integration of microservices with Consul (most of them provided by the Consul community), the most suitable and easy to use is the one provided by Spring Boot, namely the Spring Cloud Consul<sup>17</sup>.

In order to use Spring Cloud Consul, since it is a Java-based library the first step that should be performed is to include the dependency of the Spring Cloud Consul on the microservice's pom.xml as listed below.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>2021.0.1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

```
<!--Spring Cloud Consul →
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

In addition to the pom.xml, in the properties file, namely the application.yml, the following information should be included:

```
spring:
  application:
    name: my-microservice-name
  cloud:
    consul:
      discovery:
        enabled: true
        register: true
        instance-id: ${spring.application.name}:${random.value}
        health-check-path: /my-health-check-path
        health-check-interval: 10s
        hostname: my-local-ip (e.g. 192.168.1.2)
      metadata:
        authentication: false
```

<sup>17</sup> Spring Cloud Consul, <https://spring.io/projects/spring-cloud-consul>



```

authorization: false
contextPath: /my-context-path
swaggerPath: /v3/api-docs
enabled: true
host: 127.0.0.1
port: 8500

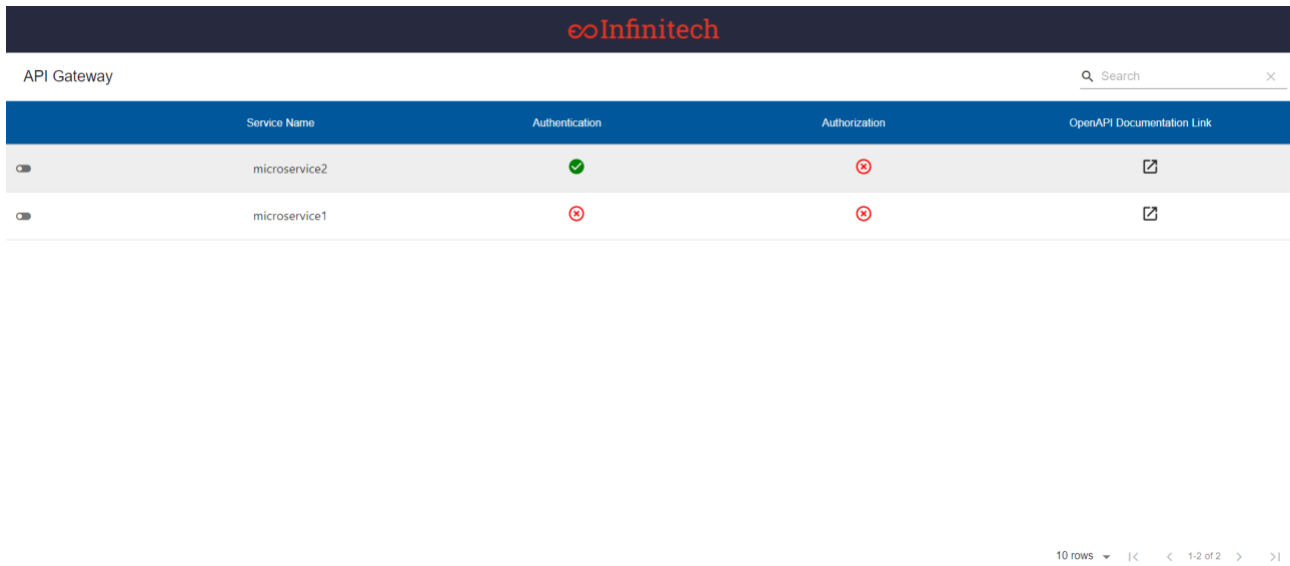
```

Finally, a RestController should be created which exposes the endpoint “/my-health-check-path” which just returns a status 200 and will be used for health-checking. Once all steps are performed, the microservice should be ready to self-register into the Service Registry of the INFINITECH Open API Gateway.

While Java constitutes the recommended solution for the microservice implementation, several other programming languages can be leveraged, such as the dominant Python programming language. In this case, the equivalent library should be used for the integration, for example the `python-consul` library<sup>18</sup>, for which rich documentation and a complete tutorial is available.

The delivered INFINITECH Open API Gateway solution constitutes mainly a backend-oriented solution as explained in the previous sections. Nevertheless, several aspects of its offerings are also provided through an easy-to-use user interface that enables the clients to leverage the documentation of the Open APIs of the registered microservices. In this sense, the INFINITECH Open API Gateway provides the means to the clients to perform a more effortless discovery of the underlying microservices by integrating and displaying the provided by each microservice during registration Open API documentation.

Once the user navigates the INFINITECH Open API Gateway’s home page, the user is presented with the list of registered microservices (Figure 4-5). As explained in the previous section, the list contains only the registered microservices that have successfully passed the health-check operations performed by the INFINITECH Open API Gateway. For each registered and healthy microservice, the user is presented with the information related with the name of the service, the requirement for authorisation and authentication in order to properly access and leverage the microservice and finally the link to respective Open API documentation of each specific microservice.



Service Name	Authentication	Authorization	OpenAPI Documentation Link
microservice2	✓	✗	<a href="#">🔗</a>
microservice1	✗	✗	<a href="#">🔗</a>

Figure 4-5: List of registered microservices

Once the user is able to select the desired microservice’s Open API documentation, the selected list item is expanded in order to display the complete Open API documentation in an easy and user-friendly manner (Figure 4-6). The user can navigate through the list of offered API endpoints and prepare the invocation of the desired microservices from the user’s client.

<sup>18</sup> Python-consul library, <https://python-consul.readthedocs.io/en/v1.1.0/>

As described above, the INFINITECH Open API Gateway is part of the INFINITECH end-to-end framework for the deployment of ML/DL-based microservices which is developed as a collaboration between Task 5.4 and Task 5.5 towards the easy definition and execution ML/DL pipelines. Within this framework, the produced via the framework ML and DL models are deployed and published as microservices. These ML/DL-based microservices are then integrated with the Service Registry of the INFINITECH Open API Gateway with the use of the respective Consul client library (as presented above). Hence, the deployed and published ML/DL models can be discovered and invoked by any potential client in order to exploit their results easily through the INFINITECH Open API Gateway. An example of usage of this end-to-end framework is the ML/DL-driven recommendation algorithms implemented in BetaRecsys, a novel open-source framework, developed by the GLA that is currently used in Pilot 6. The details of the implemented framework are documented in the deliverables of Task 5.4, namely deliverables D5.8 and D5.9.

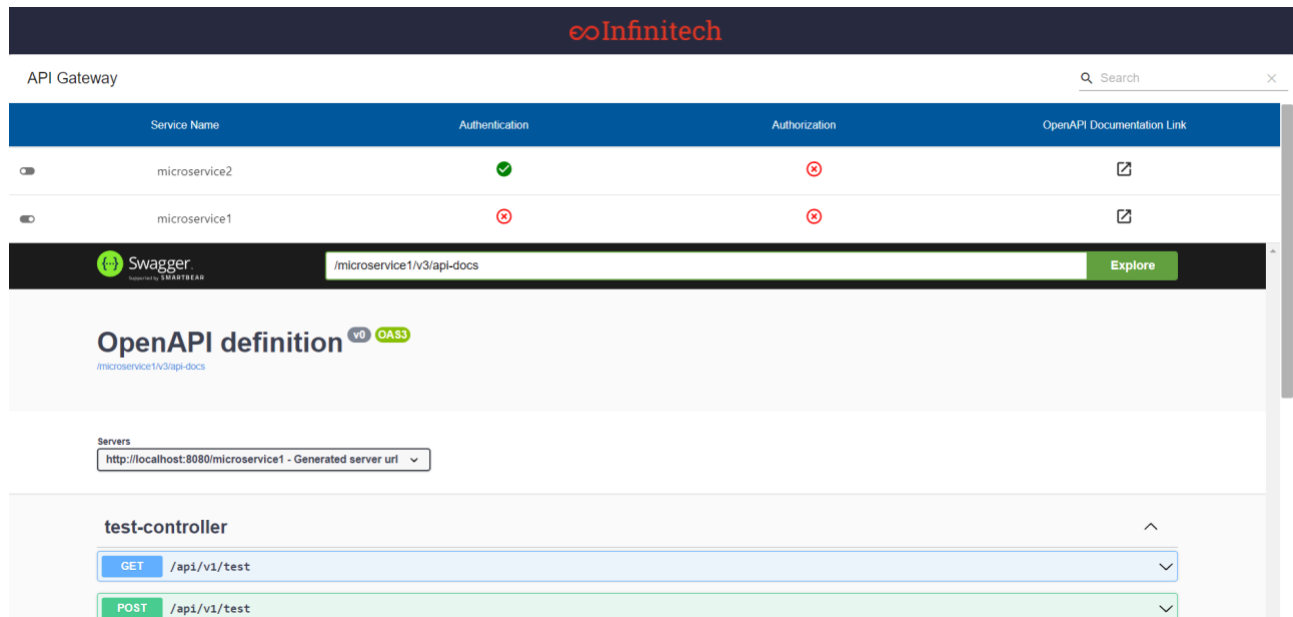


Figure 4-6: Open API Documentation of a registered microservice

## 5 Baseline technologies and tools

### Updates from D5.11:

*This particular section remained unchanged from the previous version. It presents the documentation of the usage of technologies and tools which have been leveraged for the implementation of the final fully functional version of the INFINITECH Open API Gateway.*

During the design phase of the INFINITECH Open API Gateway, several high-potential technologies and tools were investigated and an evaluation of multiple aspects of each candidate technology and tool was conducted. The basic criteria during the selection process were as follows:

- The offered functionalities of each technology and tool, as well as their relevance to the aspired functionalities of the INFINITECH Open API Gateway.
- The applicability of the offered functionalities to the designed use cases for the INFINITECH Open API Gateway.
- The level of maturity of each technology and tool.
- The effort for the implementation of the various components with the specific technologies and tools.
- The integration options and the level of compatibility between them.

For each of the two modules incorporated in the INFINITECH Open API Gateway, different technologies, tools and frameworks were selected as a result of this selection process. During the implementation phase of the final fully functional version of the INFINITECH Open API Gateway, the list of technologies, tools and frameworks has been updated and further extended with additional ones that were evaluated as the most appropriate ones for the realisation of the INFINITECH Open API Gateway's features.

In detail, the Gateway module that constitutes the core module of the component's architecture is based on the Java programming language and specifically Java version 11. To facilitate the implementation of the module, the dominant open-source Java-based Spring Boot Framework<sup>4</sup> is utilised. Spring Boot is a powerful lightweight application framework that facilitates the implementation of Java-based enterprise applications in an efficient and modular manner. Spring Boot offers out-of-the-box functionalities capable of supporting the application development such as the embedded Netty Server and easy integration with a large variety of Java-based libraries and technologies. The already embedded and well-established open-source Netty Server<sup>19</sup> is leveraged in the implementation of the module. On top of the Spring Boot Framework, the Spring Cloud Gateway<sup>3</sup> library is leveraged. Spring Cloud Gateway provides the means to implement an API Gateway utilising the Spring Boot Framework, offering several functionalities such as flexible routing and request handling, as well as cross-cutting concerns such as security, resiliency and monitoring. With regard to the Circuit Breaker pattern implementation, the Spring Cloud Circuit Breaker<sup>9</sup> is exploited in order to provide the required abstraction layer and out-of-the-box integration with multiple open-source implementations such as the Resilience4j<sup>10</sup> which is also exploited for the fault-tolerance aspects and the forced timeout functionality. Additionally, there is a compatibility a number of complementary libraries such as the Spring Data Redis Reactive<sup>20</sup> that enables the integration with the Redis key-value store and SpringDoc Open API<sup>21</sup> which enables the automatic generation of an Open API documentation based on the provided YAML or JSON source. The user interface of the Gateway module is built on top of Node.js<sup>22</sup>, the dominant JavaScript framework, React<sup>13</sup> which is one of the most commonly used Single Page Applications frameworks and the complementary libraries Material UI<sup>14</sup> and Material Table<sup>23</sup>.

<sup>19</sup> Netty, <https://netty.io/>

<sup>20</sup> Spring Data Redis Reactive, <https://spring.io/projects/spring-data-redis>

<sup>21</sup> SpringDoc. Open API, <https://github.com/springdoc/springdoc-openapi>

<sup>22</sup> Node.js, <https://nodejs.org/en/>

<sup>23</sup> Material Table, <https://material-table.com/#/>

As explained in previous sections, the Service Registry module is based on Consul<sup>15</sup> which is the well-established open-source solution for service discovery and health checking. Consul provides a sophisticated service registry that enables the self-registration and self-deregistration of microservices, maintaining their dynamic network location and providing simplified service discovery via an open and extensible API and an HTTP interface. Furthermore, it provides real-time health monitoring in order to track the availability and operational status of the registered microservices. Additionally, Consul provides the registration client in multiple programming languages that is integrated within the implementation of each microservice. The registration client undertakes the responsibility of self-registering the respective microservice during the microservice start-up process and the self-deregistration of the microservice during the microservice shutdown process.

In addition to this, the integration of the API Gateway with the Service Registry is enabled with the Spring Cloud Consul<sup>24</sup> that covers the integration aspects of Spring Boot applications with Consul.

The following table summarises the main technologies and tools which are utilised in the implementation of the INFINITECH Open API Gateway:

Table 5-1: INFINITECH Open API Gateway list of technologies

Module Name	Software Artefact Name
<i>Gateway</i>	<ul style="list-style-type: none"> <li>• Java version 11</li> <li>• Spring Boot Framework</li> <li>• Netty server</li> <li>• Spring Cloud Gateway</li> <li>• Spring Cloud Circuit Breaker</li> <li>• Spring Cloud Consul</li> <li>• Resilience4J</li> <li>• Spring Data Redis Reactive</li> <li>• SpringDoc Open API</li> <li>• Node.js</li> <li>• React</li> <li>• Material UI</li> <li>• Material Table</li> </ul>
<i>Service Registry</i>	<ul style="list-style-type: none"> <li>• Consul</li> </ul>

<sup>24</sup> Spring Cloud Consul, <https://cloud.spring.io/spring-cloud-consul/reference/html/>

## 6 Conclusions

The purpose of the deliverable at hand, entitled “D5.12 - “Data Management Workbench and Open APIs - III”, was to provide the final documentation of the efforts undertaken and the work performed within the context of Task 5.5 “OpenAPI for Analytics and Integrated BigData/AI WorkBench” of WP5. To this end, the deliverable provided the complete documentation, updating the information included in the previous iteration, namely deliverable D5.11, with the comprehensive details of the design specifications of the INFINITECH Open API Gateway component and the technical details of its final fully functional version that effectively addresses these challenges with a sophisticated (high-TRL) solution that has potential for novel extensibility.

In particular, the deliverable presented the outcomes of the performed thorough analysis of the proposed solutions for the effective and efficient access of microservice-based added-value functionalities from the clients. In detail, the two most common approaches, namely the direct client-to-microservices pattern and the API Gateway pattern, were analysed and evaluated by presenting their list of advantages and disadvantages. In addition to this, the rationale behind the decision to adopt the API Gateway pattern as the basis for the design of the INFINITECH Open API Gateway was presented. It should be noted that the results remained unchanged from the previous iteration of the deliverable and they were reported here for coherency reasons.

Furthermore, the deliverable presented the detailed documentation of the design specifications of the INFINITECH Open API Gateway component. At first, the detailed documentation was extended in order to include the details of the addressed business need and the rationale for the implementation of a sophisticated solution. Moreover, the core design decisions that were taken during the early steps of the design phase and that have formulated the core aspects of the design specifications were presented. In addition to this, the final design specifications and the main functionalities that are offered by the INFINITECH Open API Gateway, as well as the modular architecture of the designed solution were presented. At its core, these design specification remain unchanged and include two core modules, namely the Gateway and the Service Registry. For each module, a solid and distinct role has been assigned along with a subset of functionalities from the overall list of functionalities of the INFINITECH Open API Gateway. Based on these functionalities, a set of detailed use cases were documented and for each use case a detailed sequence diagram was elaborated and presented. Each sequence diagram was designed with the purpose of depicting the interactions between the modules and the clients of the INFINITECH Open API Gateway. The design specifications remained also unchanged from the previous iteration of the deliverable and they were reported here for coherency reasons.

The deliverable provided the updated detailed technical specifications of the final and fully functional version of the INFINITECH Open API Gateway that is delivered on M30 as per the INFINITECH Description of Action. The deliverable documents and highlights the updates for each of the two modules, describing the implemented functionalities and the corresponding functions that were implemented to realise them, as well as the details of the integration of both modules towards the delivery of this final version. In particular, both the Gateway Backend and the Gateway Frontend received a series of updates and enhancements with new functionalities and features, while also the Service Registry module was optimised towards the support of high availability and flexibility.

Finally, the deliverable presented the final complete list of baseline technologies and tools which were exploited and combined for the implementation of the final version of the INFINITECH Open API Gateway component. In the course of development of the INFINITECH Open API Gateway multiple mature and well-established technologies and tools were leveraged in order to deliver the fully functional version.

The deliverable constitutes the final report of the work performed within the context of T5.5 of WP5 in accordance with the INFINITECH Description of Action and concludes the activities of the specific task. The produced artefact, namely the INFINITECH Open API Gateway, is available in the INFINITECH Marketplace in order to be leveraged by all financial and insurance sectors stakeholders.

Table 6-1: Conclusions (TASK Objectives with Deliverable achievements)

Objectives	Comment
<i>Exploit the API Gateway pattern offerings.</i>	The proposed solution successfully exploits the offerings of the API Gateway pattern related to the effective and efficient service discovery and straight-forward access to the dynamically deployed microservices, the elimination of multiple server round trips and the hiding of the microservice’s architecture details and complexity from the clients.
<i>Provide a single-point-entry for the added-value offerings functionalities of INFINITECH.</i>	The INFINITECH Open API Gateway delivers the required single entry-point for all the ML/DL microservices and their exposed Open APIs that will be available in INFINITECH. Furthermore, it provides the basis for the support of additional microservice-based added-value functionalities in INFINITECH.
<i>Design and deliver the required solution that enables the discovery and consumption of the ML/DL microservices of INFINITECH and their exposed Open APIs.</i>	The detailed design specifications of the INFINITECH Open API Gateway are delivered with the current deliverable; its offered functionalities successfully cover both the discovery and the consumption of the underlying ML/DL microservices of INFINITECH as well as of their exposed Open APIs. The final fully functional version of the INFINITECH Open API Gateway is delivered with the current deliverable on M30.

Table 6-2: (map TASK KPI with Deliverable achievements)

KPI	Comment
<i>API Gateways available for enabling the access to INFINITECH ML/DL microservices.</i>	<b>Target Value = 1</b> The specific KPI is successfully achieved with the presented solution, namely the INFINITECH Open API Gateway, whose detailed design specifications and first prototype implementation were documented in the current deliverable.
<i>Coverage of ML/DL microservices exposing Open APIs.</i>	<b>Target Value = 100%</b> Per the design specifications documented in the current deliverable, the INFINITECH Open API Gateway is capable of covering all ML/DL microservices that are exposing Open APIs within the context of INFINITECH, hence the coverage is 100%.
<i>Number of services/features exposed by the API Gateway for its clients.</i>	<b>Target Value &gt;= 4</b> The INFINITECH Open API Gateway exposes five main services: <ul style="list-style-type: none"> <li>a) the request handling service,</li> <li>b) the discovery service of the Open APIs of the ML/DL microservices,</li> <li>c) the ML/DL microservices service registry,</li> <li>d) the self-registration service, and</li> <li>e) the self-deregistration service.</li> </ul>

## Appendix A: Literature

- [1] “Microservices.io,” [Online]. Available: <https://microservices.io/>. [Accessed 20 October 2020].
- [2] M. Dudjak and G. Martinović, “An API-first methodology for designing a microservice-based Backend as a Service platform.,” *Information Technology and Control*, vol. 49, no. 2, pp. 206-223, 2020.
- [3] F. Montesi and J. Weber, “Circuit breakers, discovery, and API gateways in microservices,” 2016.
- [4] E. Commission, “PSD2 Directive,” 2015. [Online]. Available: [https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366\\_en](https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en). [Accessed 20 January 2022].
- [5] “Self-Registration,” 2020. [Online]. Available: <https://microservices.io/patterns/self-registration.html>. [Accessed 10 October 2020].
- [6] “3rd-Party Registration,” 2020. [Online]. Available: <https://microservices.io/patterns/3rd-party-registration.html>. [Accessed 15 October 2020].
- [7] D. Taibi, V. Lenarduzzi and C. Pahl, “Architectural patterns for microservices: a systematic mapping study,” 2018.
- [8] “Open API specification,” 2020. [Online]. Available: [https://swagger.io/specification/#:~:text=The%20OpenAPI%20Specification%20\(OAS\)%20defines,or%20through%20network%20traffic%20inspection..](https://swagger.io/specification/#:~:text=The%20OpenAPI%20Specification%20(OAS)%20defines,or%20through%20network%20traffic%20inspection..) [Accessed 20 October 2020].
- [9] “Circuit Breaker Pattern,” 2020. [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker>. [Accessed 10 October 2020].