PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Eduardo Dantas Luna

Towards Building Digital Twins Based in Systems of Systems

Proposta de Projeto Final de Graduação

Thesis presented to the Graduate Program in Informatics, of the department of Informática in PUC-Rio in partial fulfillment of the requirements for the degree of Bachelor's in Computer Science.

Advisor: Prof. Vitor Pinheiro de Almeida

Rio de Janeiro
November 2024

Eduardo Dantas Luna

Towards Building Digital Twins Based in Systems of Systems

Rio de Janeiro, November 25th, 2024

Eduardo Dantas Luna

Bibliographic data

Luna, Eduardo

Towards Building Digital Twins Based in Systems of Systems / Eduardo Dantas Luna; advisor: Vitor Pinheiro de Almeida. – 2024.

53 f: il. color. ; 30 cm

Projeto Final (graduação) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2024.

Inclui bibliografia

1. Informática – Teses. 2. Sistemas de Sistemas. 3. Gerenciamento de APIs. 4. GraphQL. 5. Gêmeo Digitais. 6. Gerador automatico de código. I. Pinheiro, Vitor. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents, to my grandfather, to my sister and Ana Clara,
for their unwavering support and encouragement throughout this journey.

Acknowledgments

I would like to express my sincere gratitude to my advisor, Professor Vitor Pinheiro, for his invaluable guidance, patience, and encouragement in developing not only in this work, but also as a student, a monitor and at work.

I am also grateful to Tecgraf Institute for providing the resources and support necessary for this research.

Abstract

Luna, Eduardo; Pinheiro, Vitor (Advisor). **Towards Building Digital Twins Based in Systems of Systems**. Rio de Janeiro, 2024. 53p. Proposta de Projeto Final de Graduação – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

This work focuses on the interoperability of systems for creating Digital Twins (DTs), focusing on the complex challenges of uniting data from heterogeneous systems that use diverse API paradigms and data structures. Digital Twins, as digital replicas of real-world systems. For that reason, their development poses significant challenges, particularly in systems of systems (SoS) architectures, where data originates from numerous, often heterogeneous and incompatible sources.

This work proposes a method to integrate systems to achieve what was done: a thorough literature review evaluates the state-of-the-art approaches, two prototypes implemented using GraphQL, and three basic performance tests to evaluate the benefits and limitations of the proposed approach. The first prototype employs a fixed integrations approach, while the second leverages an automated system composition algorithm using a JSON description file.

The solution is designed to address challenges such as varying data structures, taxonomy differences, and different paradigms, like REST, gRPC, SOAP, etc, all of which traditionally complicate system integration. By employing GraphQL, the study achieves a more flexible data integration mechanism that can handle different paradigms, enabling seamless querying and aggregation across systems.

The research concludes by discussing potential optimizations and future directions, such as incorporating other system descriptions like knowledge graphs, ontologies or specific languages for enhanced data modelling and improving query composition techniques.

Keywords

System of Systems; API Management; GraphQL; Digital Twins; Automatic Code Generator.

Resumo

Luna, Eduardo; Pinheiro, Vitor. **Em Direção a Construção de Gêmeos Digitais Baseados em Sistemas de Sistemas**. Rio de Janeiro, 2024. 53p. Proposta de Projeto Final de Graduação – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Este trabalho foca em interoperabilidade de sistemas para a criação de Gêmeos Digitais (DTs), com foco nos desafio de unificar dados provenientes de diferentes sistemas que utilizam diferentes paradigmas de APIs e estruturas de dados. Os Gêmeos Digitais são réplicas digitais de sistemas reais. No entanto, seu desenvolvimento apresenta desafios significativos, especialmente em arquiteturas de sistemas de sistemas (SoS), onde os dados originam-se de inúmeras fontes frequentemente heterogêneas e incompatíveis.

Este trabalho propõe um metodo para integrar sistemas, para isso foi feito: uma revisão aprofundada da literatura avalia as abordagens mais avançadas do estado da arte; dois protótipos implementados utilizando GraphQL; três testes básicos de desempenho para avaliar os benefícios e limitações da abordagem proposta. O primeiro protótipo adota uma abordagem de integrações fixas, enquanto o segundo utiliza um algoritmo automatizado de composição de sistemas baseado em um arquivo de descrição em JSON.

A solução é projetada para lidar com desafios como variações nas estruturas de dados, diferenças de taxonomia e diferentes paradigmas, como REST, gRPC, SOAP, etc, que tradicionalmente complicam a integração de sistemas. Ao empregar o GraphQL, o estudo alcança um mecanismo de integração de dados que consiga lidar com os diferentes paradigmas, permitindo consultas e agregações entre sistemas com tecnologias diferentes.

A pesquisa conclui com uma discussão sobre possíveis otimizações e direções futuras, como a incorporação de outros tipos de descrições de sistemas como grafos de conhecimento, ontologias ou linguagens especificas para aprimorar o modelagem de dados e o aprimoramento de técnicas de composição de consultas.

Palavras-chave

Sistemas de Sistemas; Gerenciamento de APIs; GraphQL; Gêmeo Digitais; Gerador automatico de código.

Table of contents

List of figures

List of Abreviations

SoS – System of Systems

DT – Digital Twins

GraphQL – Graph Query Language

API – Application Programming Interface

KG - Knowledge Graph

SOA - Service Oriented Architecture

IoT - Internet of Things

CGS - Code Generation Service

1 Introduction

Data in the modern world emerges from various sources, often disconnected and asynchronous. This phenomenon is evident across a broad spectrum of environments, ranging from large-scale data centres to small micro-controllers, making data omnipresent. Many devices are now interconnected via web technologies, and the rapid evolution of internet infrastructure has significantly enhanced this connectivity. A compelling concept that arises in this context is the integration of data from diverse sources.

One domain that stands to gain significantly from such integration is smart cities. By aggregating data from systems across an entire city, a multitude of innovative solutions can be developed. These range from improving day-to-day activities, such as transportation and dining, to addressing large-scale challenges, such as urban planning. For instance, a citizen in a smart city could identify the optimal method of transportation, or even a combination of options, to travel from point A to point B without needing to understand the origin of the data. Furthermore, the individual could seamlessly modify and add new parameters to the query, such as arranging for a delivery to coincide with their arrival at the destination.

The potential applications of data integration are not limited to smart cities. In fact, it could be used in a broad spectrum of domains. For example, in the oil and gas industry, a digital simulation of an oil platform could be created by combining data from disparate systems managing various platform components. For instance, a service that exposes images of equipment of an oil platform could be used by another system to create a new system that could calculate the corrosion of such equipment.

While integrating data from diverse sources at scale unlocks numerous possibilities for innovation, achieving this in a reliable and straightforward manner poses significant challenges.

1.1 Challanges of Data Integration

A key issue in data integration is that systems often model the same element differently or provide only partial descriptions. Variations in taxonomy, data formats, or even currency usage can introduce complexities. Also, the data of those systems may be scattered or unavailable in some cases. An example is two airlines, one using the dollar and another using the euro. Integrating both airline systems would require knowing what currency each system uses and how to convert from one currency to another.

Another significant challenge is the diversity of data transfer protocols and paradigms employed by different systems. Each paradigm serves specific use cases, hindering the development of a framework that could integrate various heterogeneous systems. An instance of that can be an old system that uses legacy protocols trying to communicate with newer systems that use newer and incompatible protocols.

Moreover, the creation of data exchange interfaces requires substantial effort. Developers must learn various programming languages and technologies to establish connections between systems. Without standardization, different developers may independently create similar interfaces, leading to redundancy and inefficiency. Although creating a unified standard for all systems is possible, that solution could lead to restructuring pre-existing systems.

Furthermore, these interfaces often involve more than a few systems, adding to the complexity and time required for their development. To illustrate, a system that compares the price of computers would need to retrieve data from various store systems to be able to create such comparisons.

1.2 Objectives and Approach

The main objective of this work is to develop a flexible layer that can integrate multiple heterogeneous web systems or data sources seamlessly and without restructuring the system. To achieve that, this work addresses the challenges of integrating systems through a comprehensive approach consisting of two steps: Implementation and Evaluation.

In the implementation step, the goal is to generate a server to integrate the mock-up systems of real-world applications from the oil and gas industry. It was done employing two prototypes; the first prototype was coded to showcase possible integrations between systems that were hard-coded. Hard-coded codes are codes that do not change if any of the underlying system interfaces change. The second goal is to generate the hard-coded code of the first prototype, using a smaller example to minimize the complexity of the implementation by employing a JSON file which describes the systems.

In the Evaluation step, the difference between hard-coded GraphQL and the REST implementation against the created algorithms using a series of tests to analyse the performance difference between generated, hard-coded, and original APIs by calling the endpoints and comparing their response time.

There are two important facts to mention before continuing. The first fact is that the prototypes for this work focus on developing a method for system integration rather than exploring the optimal approaches to describe the systems and determine the best strategies for combining them. The only artefact defined for describing how to combine systems in this work is a JSON file, which is the input of the server generation algorithm. Chapter 7 discusses what future work could be created about the description of systems to complement this work.

The second fact is that this work does not propose creating a Digital Twin but rather just a layer for communication between systems that can be used to help implement Digital Twins.

Chapter 2 introduces key concepts essential for understanding the research conducted in this study. Chapter 3 describes the methodology used to find and analyse related papers, providing a comparative analysis of their findings. Chapter 4 specifies the conceptual model for the infrastructure and how conceptually the algorithm works. Chapter 5 delves into implementing the prototypes and the algorithm. Chapter 6 describes the tests to compare the fixed code, the generated and the original APIs. Chapter 7 discusses problems encountered during implementation and future works. Finally, Chapter 8 summarizes the key conclusions drawn from this research.

2
Theoretical Foundation

An interface is essential for accessing data from a system. On the web, these interfaces are primarily implemented as Web Application Programming Interfaces (APIs). An API comprises a set of interfaces, commonly referred to as endpoints. To effectively manage APIs, a system must oversee various aspects such as authorization and rate limiting. This management framework is known as API Management (BONDEL; LANDGRAF; MATTHES, 2021).

API Management consists of two primary components: the API Gateway and the API Portal. The API Gateway acts as a bridge between services and the API, effectively decoupling the client interface. It intercepts all incoming requests and routes them to the appropriate service while offering caching, scaling, and load-balancing features. In contrast, the API Portal is a frontend for both API and consumer system developers. It provides documentation, developer guides, and a centralized endpoint catalogue for the services available across multiple APIs (BONDEL; LANDGRAF; MATTHES, 2021).

API Management is frequently utilized in microservice architectures. Unlike monolithic systems, microservice architectures consist of multiple decoupled systems. In this context, API Management orchestrates interactions between these systems. By facilitating connections among various systems, API Management also enables the creation of Systems of Systems (SoS)—complex systems formed by integrating multiple interconnected systems.

SoS architectures are commonly applied in scenarios such as a specific type of Digital Twins called Systems of System Digital Twins. A Digital Twin (DT) is a digital representation of a physical system designed to replicate real-world behaviours by integrating and presenting up-to-date information from the diverse technologies that constitute it (SHI et al., 2016; ANACKER et al., 2022; OLSSON; AXELSSON, 2023).

There are six levels of Digital Twin implementation, with each higher level encompassing all the properties of the levels less then it. The levels are described as follows:

1. At this foundational level, the digital twin represents only the underlying constituent systems, offering a static representation without real-time data integration.

2. Digital twins at this level integrate live data, like sensors and other systems, enabling the querying of historical and real-time information from them.

3. At this stage, digital twins gain the ability to adapt and evolve, often employing Artificial Intelligence models, advanced analytics, and statistical methods to optimize system behaviour dynamically.

4. This level focuses on the digital twin's ability to predict both short-term and long-term system states, leveraging predictive modelling and simulation techniques.

5. At this advanced level, the digital twin can recommend actions and simulate various scenarios. It may incorporate knowledge bases and decision-making assistants to aid in evaluating complex options.

6. The highest level represents autonomous digital twins capable of replacing human operators or experts by making decisions and executing actions independently.

From level 2 onward, the implementation of the DT can be supported by a unified communication bus that interconnects all constituent systems since level 2 relies on systems being able to interact with all other systems. This approach would allow seamless access to data and functionality from the underlying systems (ALTAMIRANDA; COLINA, 2019).

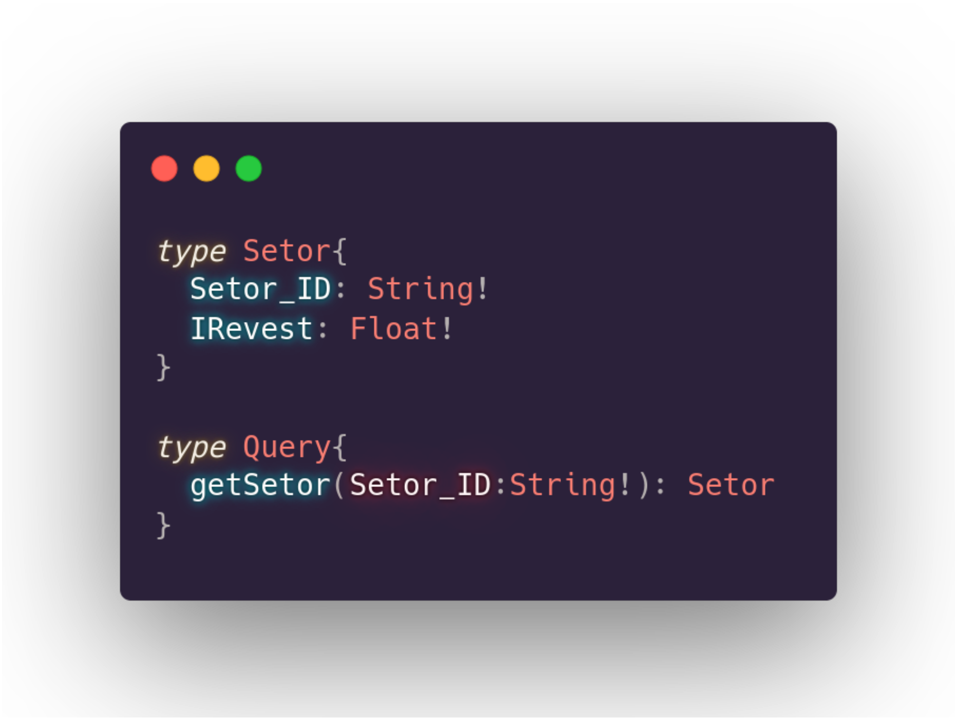
An API can use various paradigms, including REST, SOAP, and gRPC. This work, however, focuses on the GraphQL paradigm. GraphQL was developed by Facebook in 2012 and is now maintained by the GraphQL Foundation under the Linux Foundation. It offers a structured approach to querying and managing data. Its key advantages include efficient data retrieval by allowing precise data requests, preventing over-fetching or under-fetching, a single endpoint architecture compared to multiple endpoints in other paradigms, strong typing for data, and the ability to aggregate data from multiple sources (GraphQL Foundation, 2024).

The architecture of GraphQL is built around two parts: the GraphQL Server and the query language. The server is responsible for processing and executing queries using three key concepts: a type definition language, a query definition language and resolvers. The type definition language defines the structures of outputs and inputs of queries by defining a schema. This schema is loaded before the server is executed, and the GraphQL Specification does not provide any tool to mutate it during execution time. This means every query in GraphQL has its type defined statically. An example of the type definition language can be visualized in Figure 2.1. The picture defines a query for getting the Setor object given an ID.

The query definition language is very similar to the type definition but is used to query the server. An example of a query for the Setor is illustrated by 2.2, in which a query is defined for fetching the Setor using the query getSetor with the parameter S01 as Setor_ID.

The resolver functions are blocks of code responsible for retrieving data related to certain queries. In the case of the getSetor, Figure 2.3 is a Javascript implementation of such.

Like GraphQL, which has its own specifications, REST APIs can follow a standard called OpenAPI. OpenAPI files act as descriptor documents, widely used by REST API tools and libraries to provide comprehensive details about



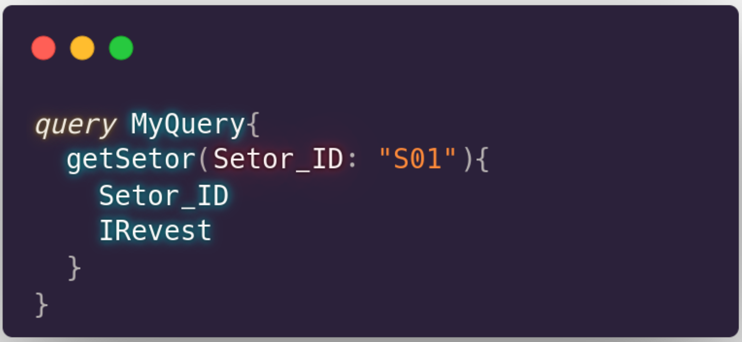
```
type Setor{
  Setor_ID: String!
  IRevest: Float!
}

type Query{
  getSetor(Setor_ID:String!): Setor
}
```

Figure 2.1: Schema definition

the API. These files include: endpoint paths, response codes, input and output schemas, and parameter requirements, among other specifics. It's important to note that different API paradigms utilize distinct types of description files (OpenAPI Initiative, 2024).

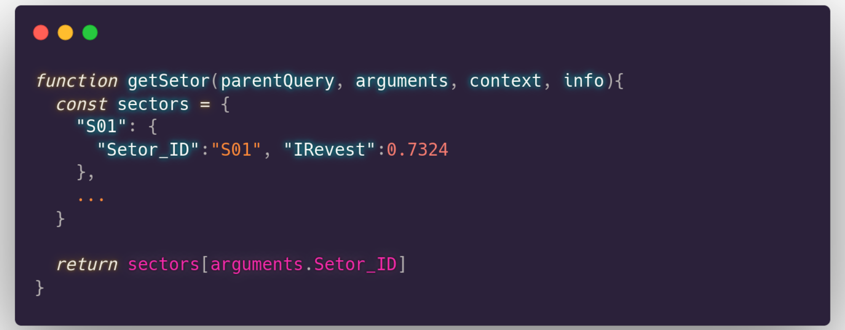
A knowledge graph (KG) is a sophisticated data structure frequently employed to model, organize, manage, and analyse heterogeneous and intricate datasets. Owing to its graph-based architecture, it encapsulates a complex abstraction of knowledge on a particular domain and delineates the interrelationships among various data entities or data models (RAMONELL; CHACóN; POSADA, 2023).

Ontology, originally a concept from philosophy, refers to the study of the kinds and structures of objects, properties, and relationships in a domain of interest. Like a Knowledge Graph, an ontology formally represents knowledge within a domain, defining terms, their relationships, and rules that combine them. It includes components such as concepts (entities), instances, relations, and axioms, which are domain-specific truths or restrictions (LI et al., 2024).



```
query MyQuery{
  getSetor(Setor_ID: "S01"){
    Setor_ID
    IRevest
  }
}
```

Figure 2.2: Query calling for getSetor



```
function getSetor(parentQuery, arguments, context, info){
  const sectors = {
    "S01": {
      "Setor_ID":"S01", "IRevest":0.7324
    },
    ...
  }

  return sectors[arguments.Setor_ID]
}
```

Figure 2.3: Resolver function for getSetor

3 Related Work

This work embarks on a systematic search for the most relevant academic papers to be reviewed. This systematic search is illustrated by Figure 3.1. The search process was conducted using the Findpapers app (GROSMAN, 2024), which allows users to create queries based on specific keywords to retrieve academic papers. The data sources utilized by this library include ACM, arXiv, bioRxiv, IEEE, medRxiv, PubMed, and Scopus.

3.1 Research Pipeline

Since the focus of this research is on System of Systems (SoS) and related aspects of Interoperability, a query was created using two sets of keywords. The first group included: Integration, Interoperability, Digital Twins, API Management, Representational State Transfer (REST), GraphQL and Federated Systems. To address the SoS focus, a second group of keywords related to SoS was created. A conditional AND was applied between these two parts, and only papers published after 2019 were used.

The initial query returned a considerable number of papers. An additional filter was applied to refine the results using a third group of keywords with AND NOT conditions. This last group is comprised of authentication, security, authorization, and cybersecurity. This filtering process reduced the number of papers to 111.

Due to the inclusion of papers lacking DOIs or those restricted behind paywalls, the pool of accessible papers was reduced to 48. A word count analysis was conducted on the papers to ensure their primary focus was on Systems of Systems and Interoperability. Articles were selected based on the frequency of the terms "Interoperability" and "System of Systems" appearing in the body text with similar prominence. Applying these criteria, the selection was narrowed down to seven papers, specifically: Pickering, Duke e Lim (2020), Mittal et al. (2020), Mohsin e Janjua (2018), Weinert e Uslar (2020), Neureiter et al. (2020), Cândea, Cândea e Staicu (2023), Anacker et al. (2022).

Since this systematic selection process was executed to find articles focusing on SoS and Interoperability, it was also added two papers related to the SoS, Interoperability and GraphQL (BORGES; ROCHA; MAIA, 2022; LI et al., 2024), and a Survey about Digital Twins and System of Systems (OLSSON; AXELSSON, 2023).

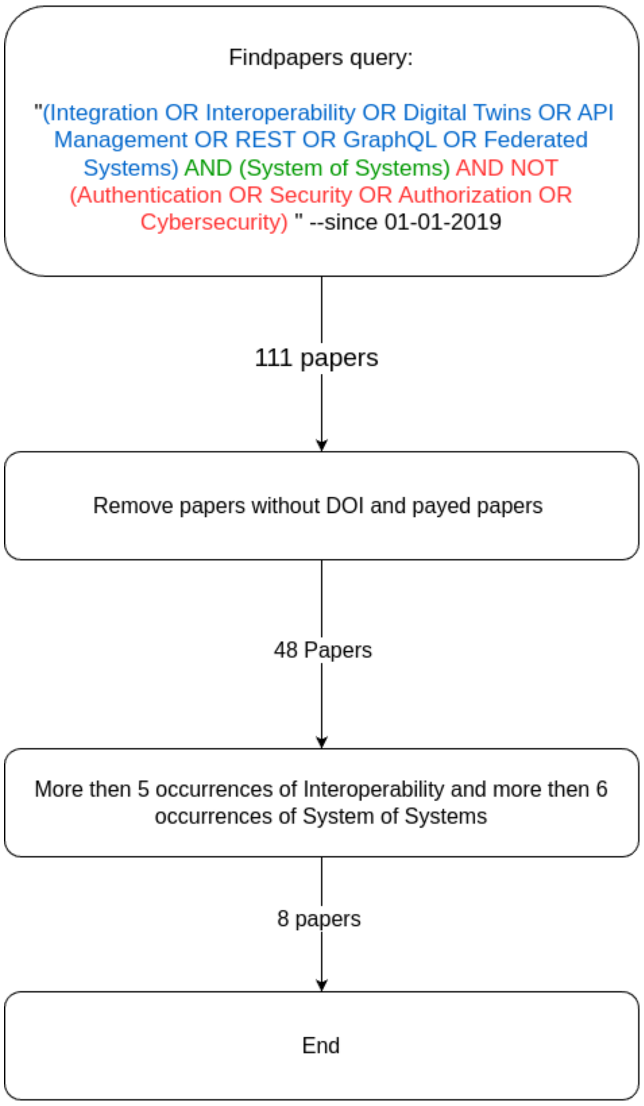

Findpapers query:
"(Integration OR Interoperability OR Digital Twins OR API Management OR REST OR GraphQL OR Federated Systems) AND (System of Systems) AND NOT (Authentication OR Security OR Authorization OR Cybersecurity) " --since 01-01-2019
111 papers
Remove papers without DOI and payed papers
48 Papers
More then 5 occurrences of Interoperability and more then 6 occurrences of System of Systems
8 papers
End

Figure 3.1: Systematic Research Diagram

3.2 Paper Analyses

The paper Anacker et al. (2022) is a review of SoS. It helped create the query for finding papers by improving definitions and keywords to find the following research papers.

The study in Mittal et al. (2020) explores the development of SoS of autonomous systems and simulation environments. There, it defines the framework for simulation, experimentation, analytics and testing to integrate diverse domains into a unified, multi-domain virtual environment. That integration is done by using standardized data models and distributed simulation standards. Unfortunately, those agreed-upon data models fix a vocabulary, meaning that pre-existing systems might have to change their structure.

The work presented at Borges, Rocha e Maia (2022) introduces a solution called MicroGraphQL for establishing API interfaces between systems using GraphQL. Their approach uses three microservices: a GraphQL API Gateway, a service for syntactically analyzing API description files called OpenAPI Files, and a service that uses syntactical analyses for generating GraphQL code. The objective of the code generated by MicroGraphQL is to identify similarities with the OpenAPI to create services composed of multiple APIs. While MicroGraphQL focuses on the syntactical analysis of the OpenAPI files is a helpful solution, it still has many problems. Since anyone can define the names of properties in different APIs, with completely different names, it will not be able to identify such properties or combine them.

The paper Cândea, Cândea e Staicu (2023) focuses on integrating Internet of Things (IoT) technologies to create an SoS, and it also highlights important use cases of SoS. It discusses two frameworks, RoboFuse and Arrowhead. Robofuse implements the Web of Things architecture, and it has most of the API Mangement functionalities and some IoT specific. Arrowhead is a system that ensures interoperability even in an automated environment by treating every device as a service using Service Oriented Architecture (SOA). Although this paper defines a unified API for getting the data from IoT devices, it differs from the current study as it focuses on IoT interoperability. In contrast, the current research is interested in any service and systems, encompassing a broader scope beyond just IoT.

The paper Mohsin e Janjua (2018) reviews SOA-based architectures for modelling SoS. They are similar to Cândea, Cândea e Staicu (2023) since they both discuss SOA architecture to facilitate interoperability. They argue that SOA-based approaches are not enough to address the unique demands of SoS, such as managing emergent behaviours, ensuring runtime adaptability, and maintaining quality attributes like performance and security. They also highlight that the need for more robust tools and formal models is evident, as these would allow architects to better design and simulate the intricate dynamics of SoS since this process can become extremely complicated.

A paper that defines such formal models for APIs is Li et al. (2024). They propose an ontology-driven GraphQL Server generation to enhance data access and integration across heterogeneous sources and structures. That is done by using a *generic resolver* to fetch data from different types of data sources and by using its Ontology to describe entities, relationships and attributes of data sources. It differs from Borges, Rocha e Maia (2022) since it does not rely upon syntactical analyses. It is a very robust solution, however, it has a few drawbacks, like limited query features, optimization, and model maintenance.

Just like Mohsin e Janjua (2018) the papers: Neureiter et al. (2020), Weinert e Uslar (2020) and Pickering, Duke e Lim (2020) focus in modeling. Pickering, Duke e Lim (2020) and Weinert e Uslar (2020) are also from the domain of agriculture. Pickering, Duke e Lim (2020) identifies the challenges of siloed and closed systems, highlighting how these issues hinder effective collaboration and scalability. Through a structured, five-stage SoS discovery process, the paper emphasizes practical solutions to foster interoperability. This process uncovers opportunities for capability reuse, data integration, and hardware standardization, which can improve system adaptability and scalability. The paper also outlines forward-looking recommendations, such as leveraging digital twins and augmented reality to enhance data sharing and decision-making.

The paper Weinert e Uslar (2020) proposes a conceptual model inspired by thriving frameworks such as the Smart Grid Architecture Model, aiming to harmonize information exchange across diverse and heterogeneous agricultural systems. It emphasizes the creation of an open vendor-neutral communication framework that allows various stakeholders to interact seamlessly through standard interfaces and service specifications. It also proposes the components of Service Registry and Identity Registry to manage services and trust relationships effectively.

Lastly, the paper Neureiter et al. (2020) tackles the challenges of integrating diverse systems like Smart Grids, Automotive, and Smart Cities by defining and extending the concept of Domain Specific Systems Engineering to SoS. It highlights improvements in enabling systems from different domains to work together by addressing interoperability and compatibility issues. The key advancement is creating a unified framework that aligns domain-specific models, processes, and tools, making managing complex interactions and dependencies across systems easier.

3.3 Comparison

Before completing this analysis, a comparison table was devised to compare the papers, and the following 6 questions were formed.

1. **Q1**: Does it convey about joining data from multiple sources?
2. **Q2**: Does it propose any model describing the systems and how to integrate their data?
3. **Q3**: Does the paper uses GraphQL?

4. **Q4**: Could existing systems need to be modified to use this approach effectively?

5. **Q5**: Does it provide a language to query multiple data sources simultaneously?

6. **Q6**: Does the paper focus on modelling or implementation?

Q1 outlines the importance of combining data from multiple sources and not only exposing them in a unified way. Since the systematic research found papers about modelling, question **Q2** tries to identify papers which define a way to model the SoS as a whole, going from abstract modelling to a highly specific way to model the systems, its data, and how to operate together.

Q3 tries to identify which papers use GraphQL since some of the papers use them.**Q4** tries to define which approaches can be easily decoupled from integrating those systems. **Q5** defines our final goal, that is, to be able to create a single query that adds data from multiple sources.

Table 3.1: Comparison of researched papers.

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
Anacker et al. (2022)	NA	No	NA	NA	NA	NA
Borges, Rocha e Maia (2022)	Yes	No	Yes	Yes	Yes	Implementation
Cândea, Cândea e Staicu (2023)	No	No	No	Yes	No	Implementation
Li et al. (2024)	Yes	Ontology	Yes	No	Yes	Both
Mittal et al. (2020)	Yes	No	No	Yes	No	Implementation
Mohsin e Janjua (2018)	Yes	No	No	Yes	No	Modelling
Neureiter et al. (2020)	Yes	DSSE	No	Yes	No	Modelling
Pickering, Duke e Lim (2020)	Yes	SysML	No	Yes	No	Modelling
Weinert e Uslar (2020)	Yes	No	No	Yes	No	Modelling

After analyzing Table 3.1, some interesting conclusions can be made. Most papers about modelling do not go as deep as talking about the data itself of each system. Even with the remaining modelling papers that talk about the data, they are highly heterogeneous in their modelling types.

But the main takeaway from this table is that besides Cândea, Cândea e Staicu (2023), which is a model-focused paper, the only two GraphQL papers, Li et al. (2024) and Borges, Rocha e Maia (2022) explicit defines a language to query multiple data sources. The only paper that defines a system that would not need to be alter any other system to integrate them is Li et al. (2024).

4
Conceptual Model

The conceptual infrastructure architecture is illustrated in Figure 4.1. This architecture is structured into three main layers: the Micro-Services Layer, the API Management Layer, and the External Layer.

- **Micro-Services Layer**: This layer includes all applications and services managed by the organization. These micro-services provide the foundational data and functionality that other systems rely on.
- **External Layer**: The outer layer consists of applications or systems that consume data provided by the organization's services through the API Management Layer.
- **API Management Layer**: This layer is what this work proposes to generate. It serves as the unified communication interface. It dynamically creates GraphQL servers to facilitate seamless data integration and querying across multiple APIs. Both the **External Layer** and the **Micro-Services Layer** can access the **API Management**.

The primary function of this work is to generate the API Management Layer, as detailed in Figure 4.2. This approach is inspired by the methods proposed in Borges, Rocha e Maia (2022), Li et al. (2024), aiming to create a unified communication layer capable of querying data composed from multiple APIs and assisting the implementation of the level 2 of a Digital Twin. To achieve this, the system dynamically generates a GraphQL server that serves as a centralized interface for managing and unifying data queries across various APIs.

The reason for using a GraphQL implementation as the API Management is that it has the capability to be a unified access point for all underlying systems. Additionally, it can be potentially used to compose new queries using the existing query, which would assist interoperability.

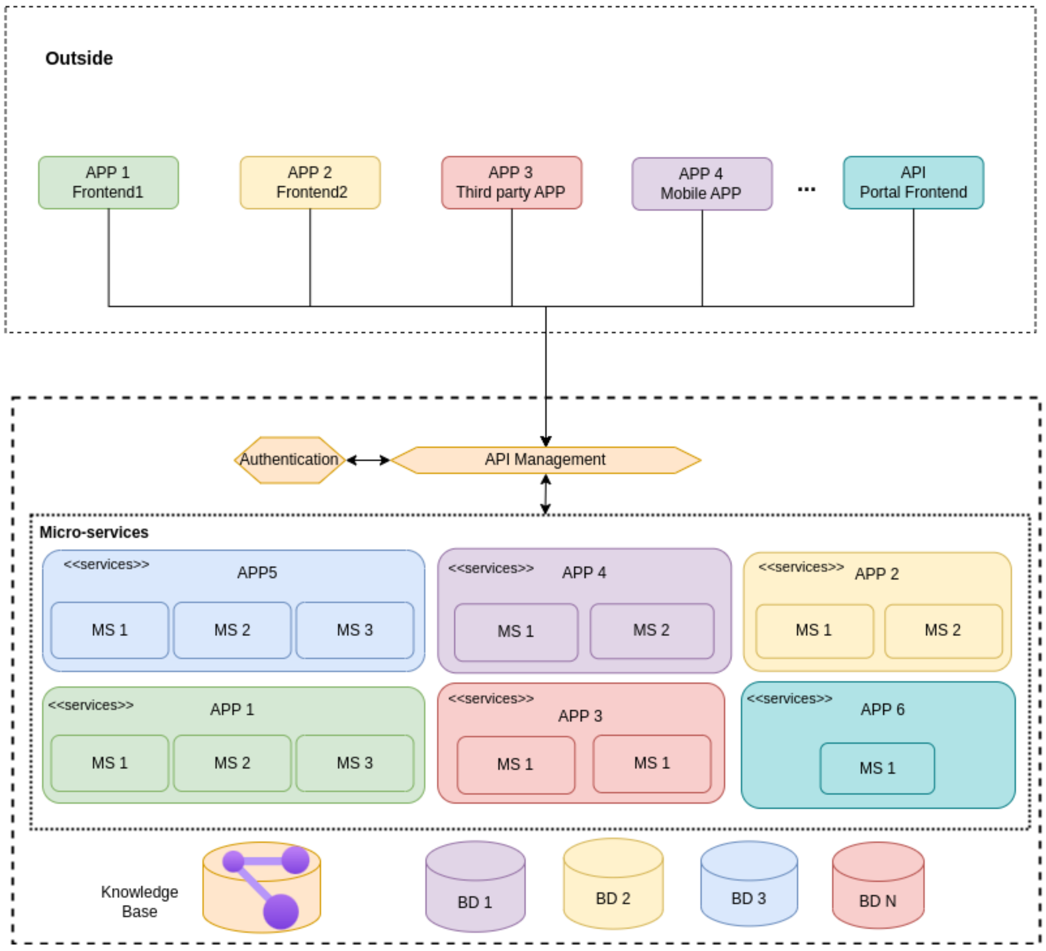


Figure 4.1: Conceptual Model

Similar to the approach in Borges, Rocha e Maia (2022), the system relies on service description files to automatically generate the GraphQL types definitions and resolver function for each API. The GraphQL code for querying individual APIs using the GraphQL API Management is referred to as **Atomic Services**.

Additionally, the code generator would also need a description file for composing new queries using the **Atomic Service**. The Code Generator uses a JSON description file as its input. An example of such a JSON file is in the Appendix A.1.

In future research, a knowledge graph could be used to create more complex queries that span multiple systems. This is achieved by mapping which queries and attributes from one API are linked to queries and attributes in other APIs, facilitating a higher level of data integration. These integrated services are referred to as **Composed Services**.

The **Semantic Description Parser** is not implemented in this work but, conceptually, would play a role in translating a KG or any other description of composition between systems, into the input for the **Code Generator Service**. Therefore, it will be discussed as part of future work in Chapter 7. Instead, this work will concentrate on defining the **Code Generator Service** (CGS) and its implementation, as detailed in the following Chapters.

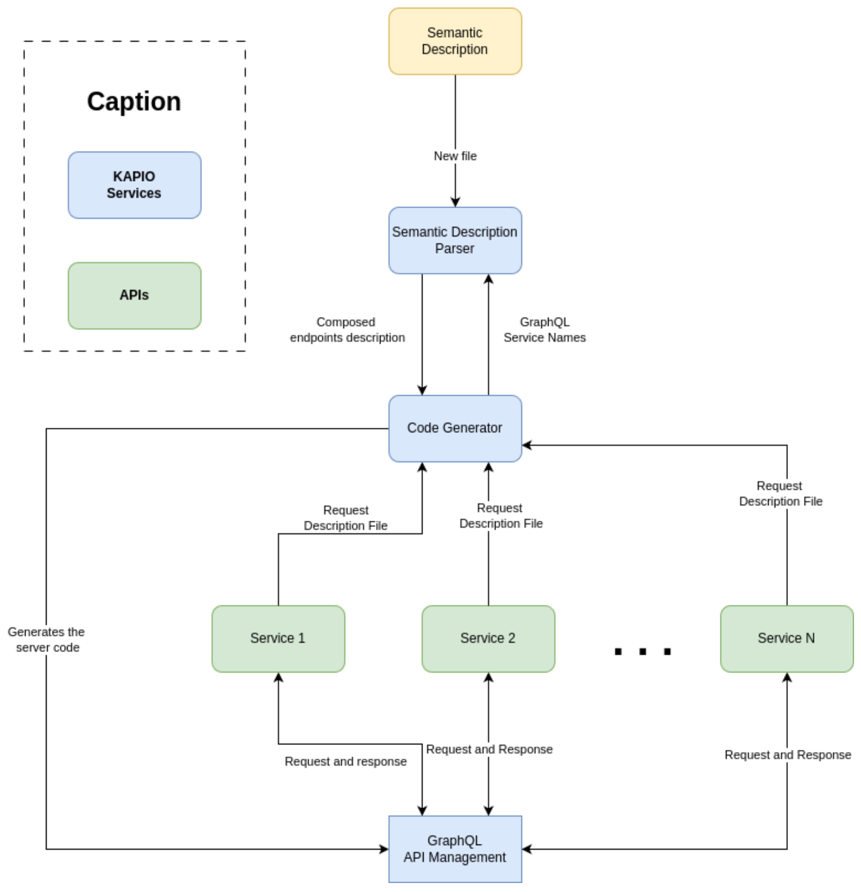
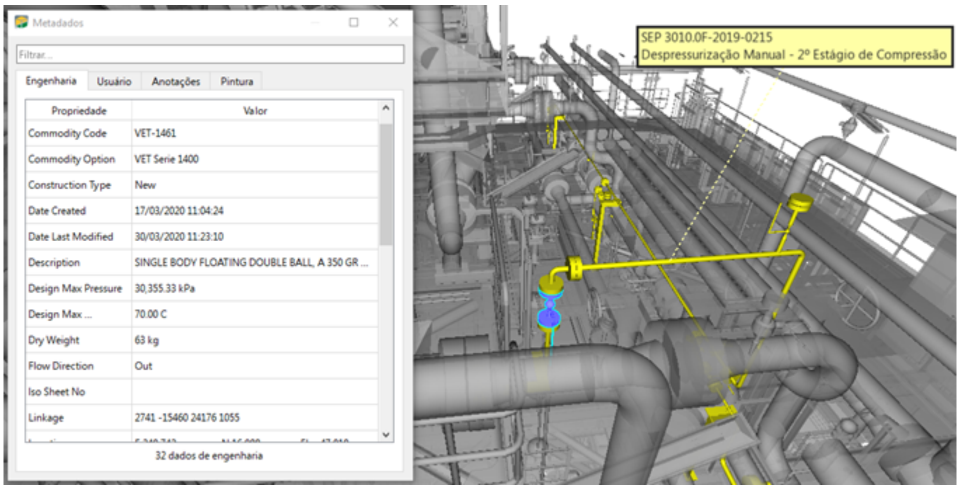


Figure 4.2: Code Generation Process for the API Management Layer

5
Implementation

This chapter describes the implementation of the conceptual model described in Chapter 4. A series of mock-up services were developed to address the main use case in the oil and gas industry. These five Atomic Services are as follows: Environ, Plan360, Busca360, Cronos, and Algo360. They were derived from five real-world applications from the oil and gas domain.

Environ is a system for integrating and visualization of multi-domain engineering data for building and analysing scenarios of industrial plants using 3D models. It is integrated into multiple data sources to support scenario analyses and decision-making. Figure 5.1 illustrate the visual interface of Environ.



SEP 3010.0F-2019-0215
Despressurização Manual - 2º Estágio de Compressão

Propriedade	Valor
Commodity Code	VET-1461
Commodity Option	VET Serie 1400
Construction Type	New
Date Created	17/03/2020 11:04:24
Date Last Modified	30/03/2020 11:23:10
Description	SINGLE BODY FLOATING DOUBLE BALL, A 350 GR ...
Design Max Pressure	30,355.33 kPa
Design Max ...	70.00 C
Dry Weight	63 kg
Flow Direction	Out
Iso Sheet No	
Linkage	2741 -15460 24176 1055

Figure 5.1: Environ frontend

Plan360 is a system for digitization and remote visualization of assets through 360º images, integrated with point clouds and 3D models. Its main goal is to assist in planning maintenance on industrial plants, visual inspection, and other visual tasks. The platform enables immersive navigation, georeferenced annotations, and automated comparisons over time, allowing for better monitoring of asset conditions. With AI-driven insights and telemetry, Plan360 enhances operational efficiency, optimizes inspection processes, and supports data-driven decision-making for industrial asset management. Its dashboard can be visualized through Figure 5.2.

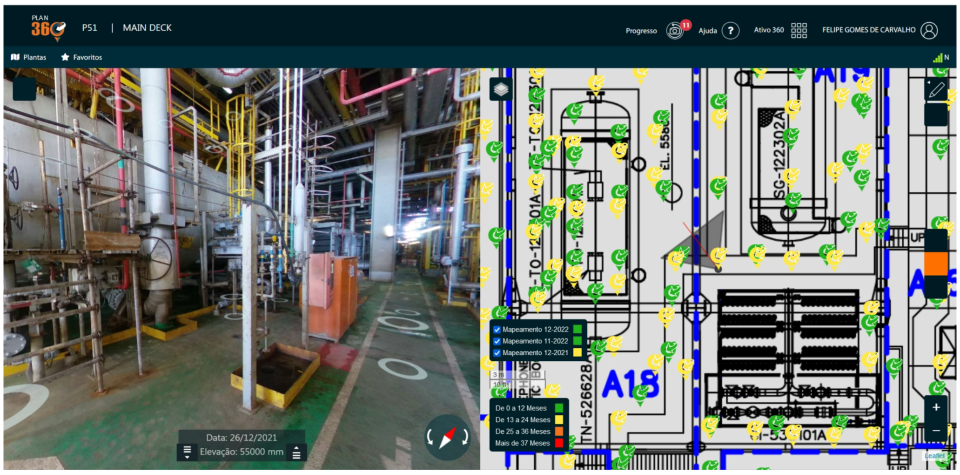


Figure 5.2: Plan360 frontend

Busca360 is an intelligent search tool for inspection and maintenance records of equipment, compliance, and equipment safety, utilizing Big Data and Natural Language Processing (NLP) solutions. The system provides a database that connects data from multiple sources and aims to answer complex queries and correlate information across different sources (IZQUIERDO et al., 2024).

Cronos and Algo360 are systems for dealing with corrosion. Algo360 is a system capable of distinguishing intact surfaces from those with coating degradation to estimate the corrosion index. Cronos is a tool designed to optimize inspection and painting plans while simulating equipment and pipeline corrosion. Cronos uses machine learning techniques to predict the criticality of corrosion and how it will evolve. By doing that, Cronos optimizes the power plant painting team.

Based on those systems, the following real-world use case was devised: "Which sectors and equipment (TAGs) need the painting on platform P77 to complete the painting plans and all painting RTIs to maximize efficiency?". To simplify, building a proof of concept of the conceptual model, the systems were reduced to mock-up systems with the following objectives:

1. **Environ**: Exposes engineering data and TAG data of each Sector.
2. **Plan360**: Exposes images of each TAG.
3. **Busca360**: Exposes RTIs of each Sector.
4. **Cronos**: Exposes inspection and painting plans of each Sector.
5. **Algo360**: Exposes IRevest data of each Sector.

In Figure 5.3, the output of each system and its properties can be observed. Each service with the **PK** is the primary key of the system. These primary keys can appear as foreign keys in other systems, enabling the integration of data across systems. An example of such an approach is the Environ and Cronos APIs. The Sector ID serves as the primary key in Environ and as a property in Cronos. Notably, in the Cronos API, the property is a list of Sector IDs, and in the Environ API, each instance has only one identifier Sector ID.

Two prototypes based on these services were developed. The first prototype was devised to show the multiple transformations that can be made to each service's data structure to merge data seamlessly. For its complexity, it uses a fixed server code across five systems.

The second prototype has two implementations. The first implementation was created using a fixed code, and the second used the algorithm shown in Chapter 4 to generate the GraphQL code dynamically. To minimize the complexity of generating the GraphQL code, since there are many different ways to join data, only two types of transformation were created based on only three of the five systems: Environ, Algo360 and Cronos. Another reason for the creation of those different prototypes was to create joins between multiple paradigms, which are not in the first prototype. The repository with the codes for the services and the server is in Luna (2025).

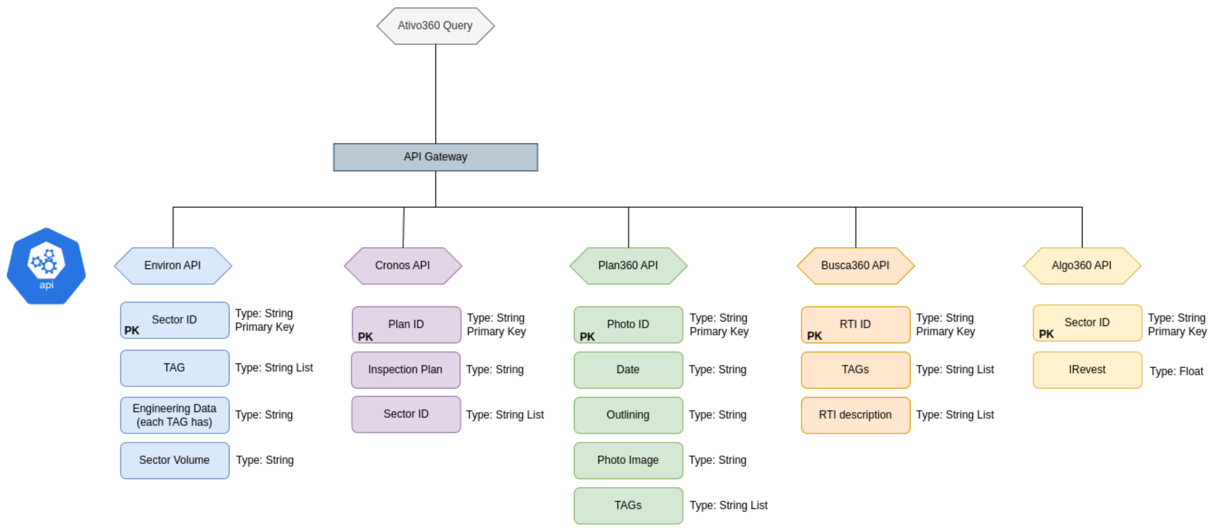


Figure 5.3: Complete use Case

5.1 First Prototype

For the implementation of the use case, the API management was developed using a NodeJS library called Apollo GraphQL. The Environ and the Cronos also use NodeJS with the ExpressJS library, and its API exposes a JSON file. The Algo360 and Busca360 also expose a JSON file but are implemented in Python with the FastAPI library. Plan360 is the only one that differs in data persistence. It uses SQL Lite with JavaScript's ExpressJS library. Docker will be used to encapsulate each micro-service. The complete infrastructure is shown in Figure 5.4 (Express.js, 2024; Apollo Graph Inc., 2024; RAMíREZ, 2024; Docker, Inc., 2024).

Also, for this use case, each application has two types of queries. One query retrieves a single instance by primary key, while the other retrieves a list of instances by a list of keys or all instances if no keys are provided. If a list of keys is provided or if the parameter is null, the query returns all instances of the API. An exception is the Plan360 API, where TAGs are used as foreign keys rather than primary keys. Instead of those two queries, Plan360 exposes a single endpoint that runs an SQL query.

To solve the problem of combining Plan360 data with other services, a new Atomic Service is defined inside GraphQL. Treating the TAGs as primary

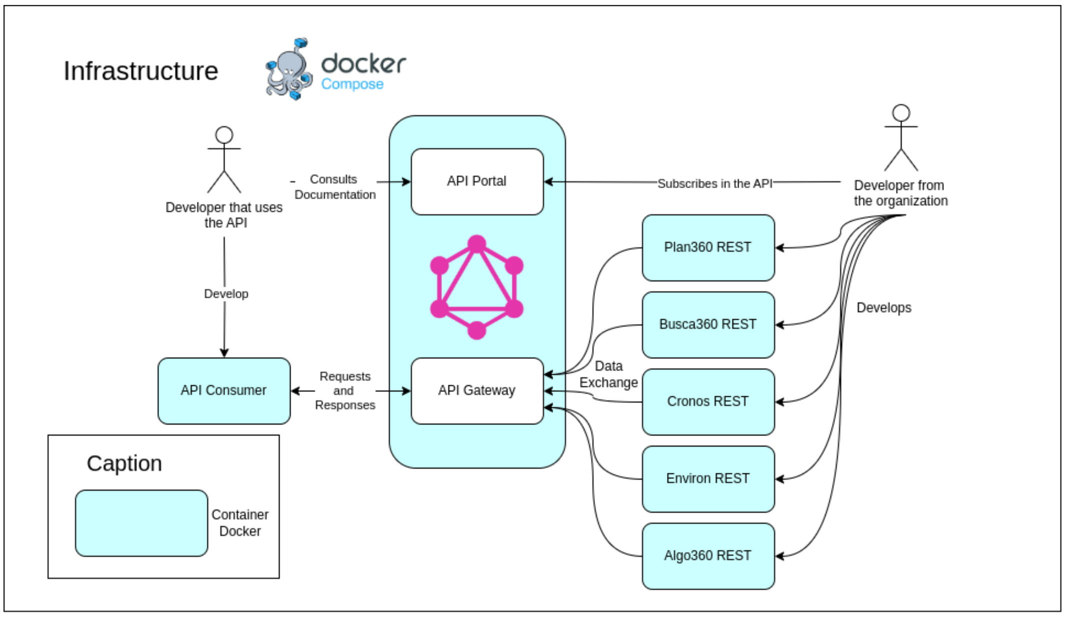


Figure 5.4: Prototype 1 Infrastructure

keys can create those two queries that were previously mentioned. Environ and Busca360 also have a similar problem. To join the Tag properties, a new Atomic Service is created to make TAG the Primary Key. All other APIs also have an Atomic Service for each of them, but in their case, their endpoints do not need to be changed to be created inside GraphQL.

With all the Atomic Services created, their data can be combined by creating new services. Seven new services were created within GraphQL for this prototype: Environ Equipment, Plan Photo, Busca Equipment, Plan Equipment, Ativo Equipment, Ativo Sector, and Ativo Painting Plan, as shown in Figure 5.5. Plan Photo, Environ Equipament, Busca Equipament and Plan Equipament are the Atomic Services mentioned earlier. By uniting them, Ativo Equipment is created, and in the same way, Ativo Sector combines Environ, Algo360, and Ativo Equipment. Finally, combining Ativo Sector with Cronos results in the Ativo Painting Plan, directly addressing the use case question.

A query call is illustrated by Figure 5.6. That query was run using the API Portal of Apollo GraphQL, but another application can also call it. An example of such using NodeJS was created with the service API Consumer.

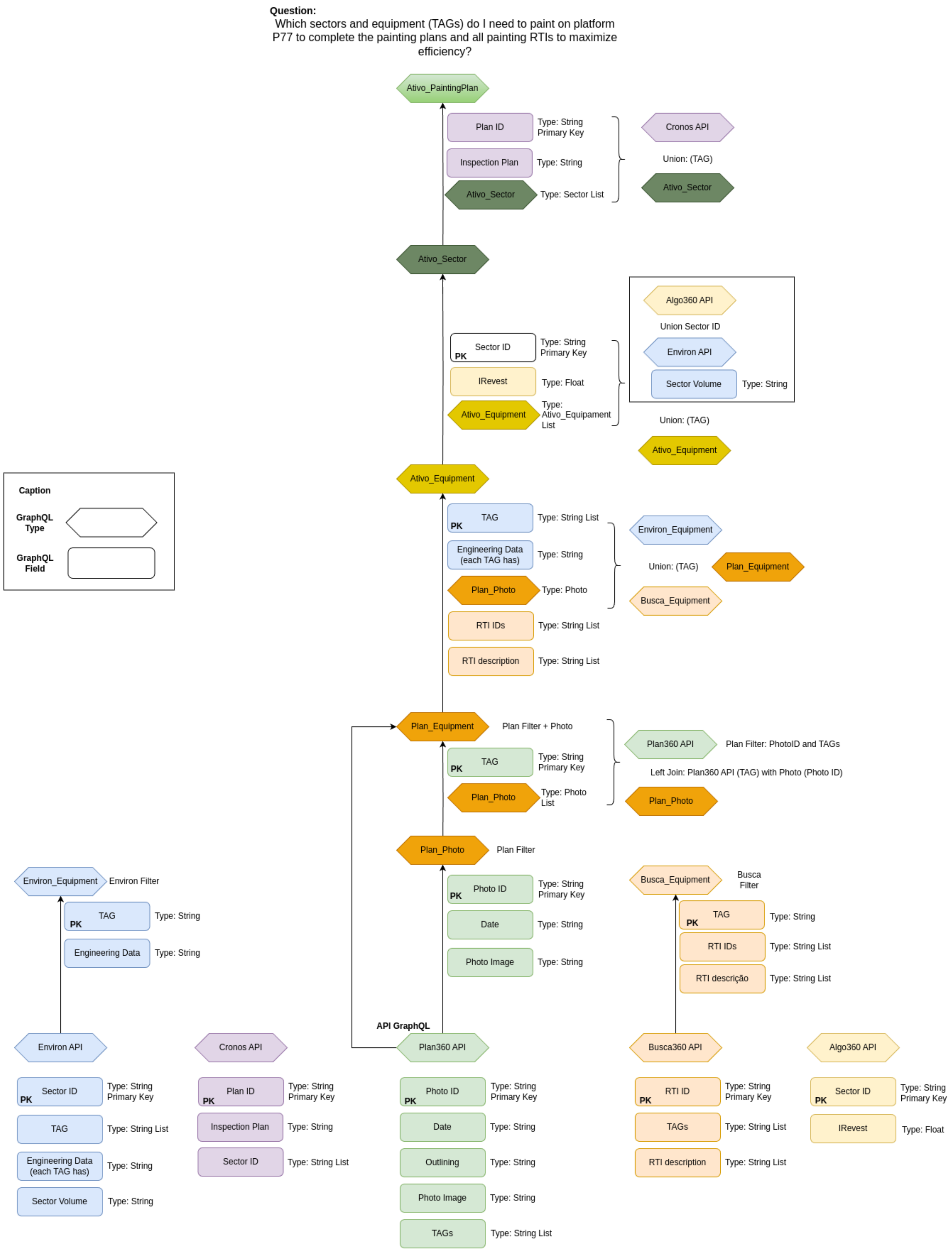
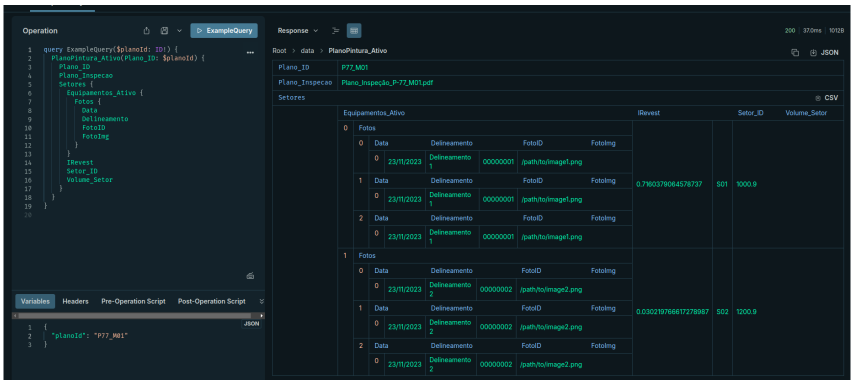


Figure 5.5: GraphQL Service Composition Tree



```
query ExampleQuery($planoId: ID!) {
  PlanoPintura_Ativo(Plano_ID: $planoId) {
    Plano_ID
    Plano_Inspecao
    Setores {
      Equipamentos_Ativo {
        Fotos {
          Data
          Delineamento
          FotoID
          FotoImg
        }
      }
      IRevest
      Setor_ID
      Volume_Setor
    }
  }
}
```

```
{
  "planoId": "P77_M01"
}
```

Plano_ID	P77_M01
Plano_Inspecao	Plano_Inspeção_P-77_M01.pdf
Setores	

Equipamentos_Ativo	Fotos		Data	Delineamento	FotoID	FotoImg	IRevest	Setor_ID	Volume_Setor
0	0	0	23/11/2023	Delineamento 1	00000001	/path/to/image1.png	0.7160379064578737	S01	1000.9
	1	0	23/11/2023	Delineamento 1	00000001	/path/to/image1.png			
	2	0	23/11/2023	Delineamento 1	00000001	/path/to/image1.png			
1	0	0	23/11/2023	Delineamento 2	00000002	/path/to/image2.png	0.030219766617278987	S02	1200.9
	1	0	23/11/2023	Delineamento 2	00000002	/path/to/image2.png			
	2	0	23/11/2023	Delineamento 2	00000002	/path/to/image2.png			

Figure 5.6: Query for the first prototype

5.2 Second Prototype

This second prototype differs slightly from the one in Section 5.1 for a few reasons. This prototype aims to replicate the data structure that the automatically generated server will use and compose.

Plan360 posed a challenge in the first prototype due to its direct use of SQL queries. Automatically creating queries and types for this setup is complex. GraphQL's reliance on fixed types makes it challenging to handle SQL-generated tables, which can vary in structure through joins and unions. It is important to note that it is possible to solve that problem.

Another point to consider is that it is interesting to keep the goal of answering the use case question defined in Section 5.1. However, a problem emerges: An important part of the algorithm for generating the GraphQL Server relies on description files, and the library ExpressJS does not generate that file natively, but the FastAPI does. Since Cronos is of foremost importance in developing the final response, it was rewritten in Python for this second prototype.

Since Algo360 was already written using the FastAPI library, it was kept for the second prototype. Ultimately, our second prototype has Algo360 and Cronos APIs as its Atomic Services. The system composition in this prototype is much simpler than in the first. It only generates a single Composed Service called Plano Ativo, which uses the Setor ID list from Cronos to integrate with

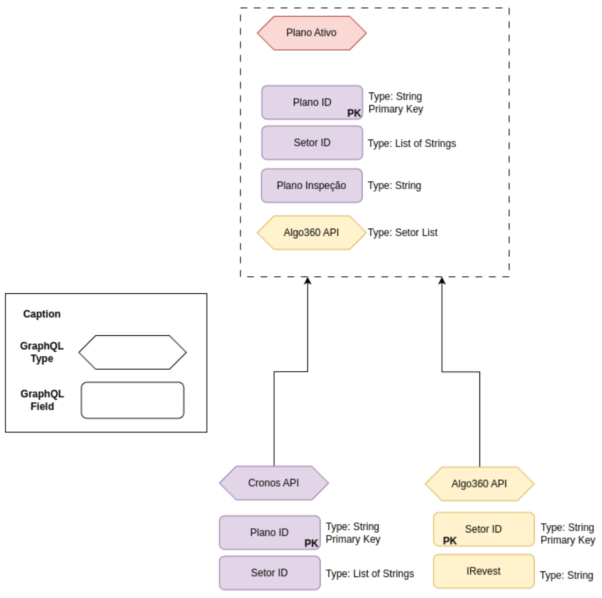


Figure 5.7: GraphQL Service Composition Tree for the second prototype

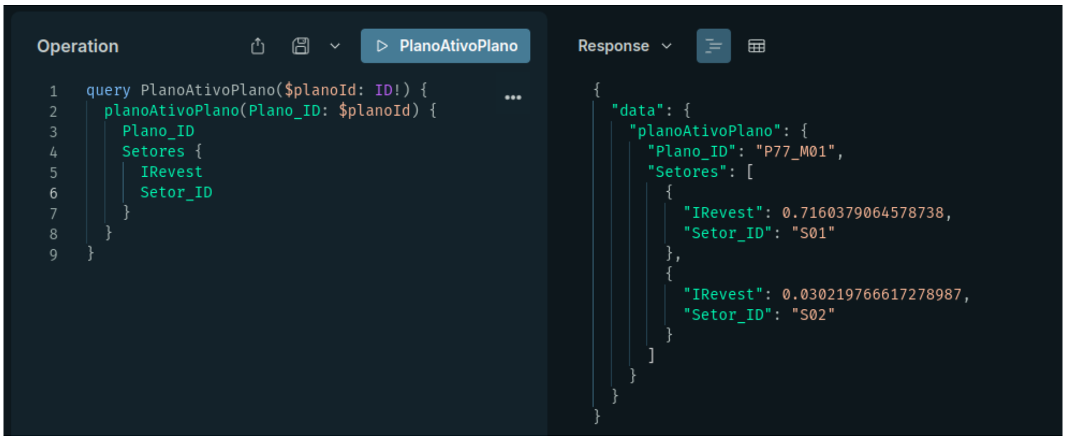
the Setor ID primary key of Plano Ativo. That is illustrated by the 5.7.

Another difference is in the queries. This setup employs a unified query to retrieve data, which functions similarly to the two-query setup. If you send a single key as input of the query, it returns the object of that key. If you send null, it gathers all instances of the object, and if you send a list of keys, it returns all elements of those keys. From that, an interesting problem emerged. That problem is discussed in Section 7.4. For now, it is necessary to know that GraphQL can not express exactly that query, but it can express an equivalent query, which returns a list with only one object in the specific case of getting only one object and all other cases just the same as before.

Even though a single query in GraphQL could be implemented expressing the equivalent of this unified query, it was decided to create two queries the same way the first prototype did. A return of such a query can be visualized in Figure 5.8.

5.3 Code Generator Service

Before describing how the Code Generator Service (CGS) operates in the prototype from Section 5.2, a modification was made to demonstrate



```
query PlanoAtivoPlano($planoId: ID!) {
  planoAtivoPlano(Plano_ID: $planoId) {
    Plano_ID
    Setores {
      IRevest
      Setor_ID
    }
  }
}
```

```
{
  "data": {
    "planoAtivoPlano": {
      "Plano_ID": "P77_M01",
      "Setores": [
        {
          "IRevest": 0.7160379064578738,
          "Setor_ID": "S01"
        },
        {
          "IRevest": 0.030219766617278987,
          "Setor_ID": "S02"
        }
      ]
    }
  }
}
```

Figure 5.8: Query return of second prototype

CGS's ability to integrate data across different API paradigms. The Environ service, previously using ExpressJS, now utilizes Apollo GraphQL to expose its data directly since ExpressJS does not generate OpenAPIs files natively. To illustrate CGS's ability to combine data from different paradigms, a new query, Setor Ativo, was created. This query is illustrated in Figure 5.9.

As mentioned in Chapter 4, the CGS relies on the description files and a JSON file for gathering information to generate the server. The FastAPI generates a type of description file called the OpenAPI file, which describes the endpoints and types. GraphQL, on the other hand, does not require an explicit description file because every GraphQL server supports introspection. GraphQL introspection is a special type of query that returns information about queries, schema, and other information.

To make it a bit easier to implement, the API Management of this prototype does not use Apollo GraphQL. The justification for that is the use of two frameworks, the GraphQL Hive and GraphQL Mesh. Chapter 7.2 examines the reasons for that change, yet for now, all that is needed to know is that GraphQL Mesh can generate GraphQL Resolvers, Queries and Types automatically using some specific types of data source description files, a few examples are: OpenAPI, GraphQL, gRPC, and others, assisting in the creation of Atomic Services (The Guild, 2024).

For Composed Services, as discussed in Chapter 4, it relies on using a

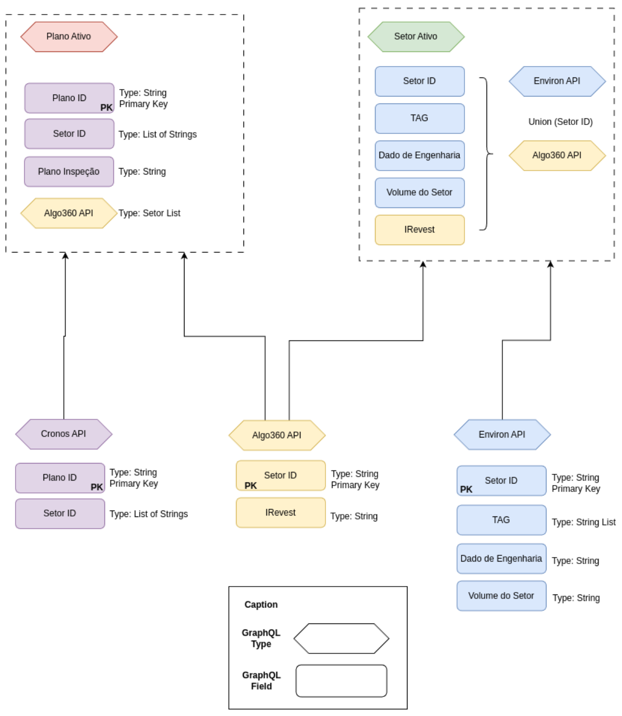


Figure 5.9: Second prototype Composition tree variation

JSON file. However, the input of the CGS is a specific JSON file. However, future research could get from the KG to the input of the CGS, and a parse could be created to translate the Knowledge Graph to the JSON.

5.3.1 The input file

The JSON input for the GraphQL Code Generator Service contains three main properties: **atomic**, **composed**, and **rename**. Each property serves a specific purpose in configuring the API management layer. The JSON file can be viewed in the Appendix A.1

- **atomic**: This property defines a list of Atomic Services. Each service requires: **API type** that specifies the API type, which currently supports `OpenAPI` or `GraphQL`; **endpoint** the endpoint URL for accessing the API; **Name**: The identifier for the API; and source which is a reference for the description file.
- **rename**: Specifies a string replacement rule that changes all substrings with a specified name in the GraphQL type and queries to a designated

string. This feature standardizes naming across APIs.

– **composed**: This property defines the operations to compose multiple services. Currently, it supports two types of operations, the **List Unit Union** that requires a list of foreign keys on the left side to generate a new list of items that includes objects from the right side. And the **Unit Unit Union** that performs a set union of properties from two objects into a single object, such as combining data from `Environ` and `Algo360` for `Setor Ativo`.

Each composition operation under **composed** includes the following required sub-properties:

- **query name**: Defines the name of the new query created by the composition.

- **query signature**: The GraphQL schema string defines the input and output of the new query.

- **new type**: An object specifying the structure and name of the new type to be associated in GraphQL.

- **parameters**: A string that defines how the query's parameters are structured.

Both the left and right operands in a composition operation include additional sub-properties:

- **return list**: A boolean indicating whether the query should return a list or a single item.

- **name**: The name of the query being referenced.

- **params**: Parameters for the query.

- **attributes**: The specific attributes within the query, as GraphQL requires explicit definition of returned fields.

- **keys**: The properties used to join objects. This is defined as a path array, where each item represents a level in the object hierarchy, with the final element being the property used for integration. For example, in the

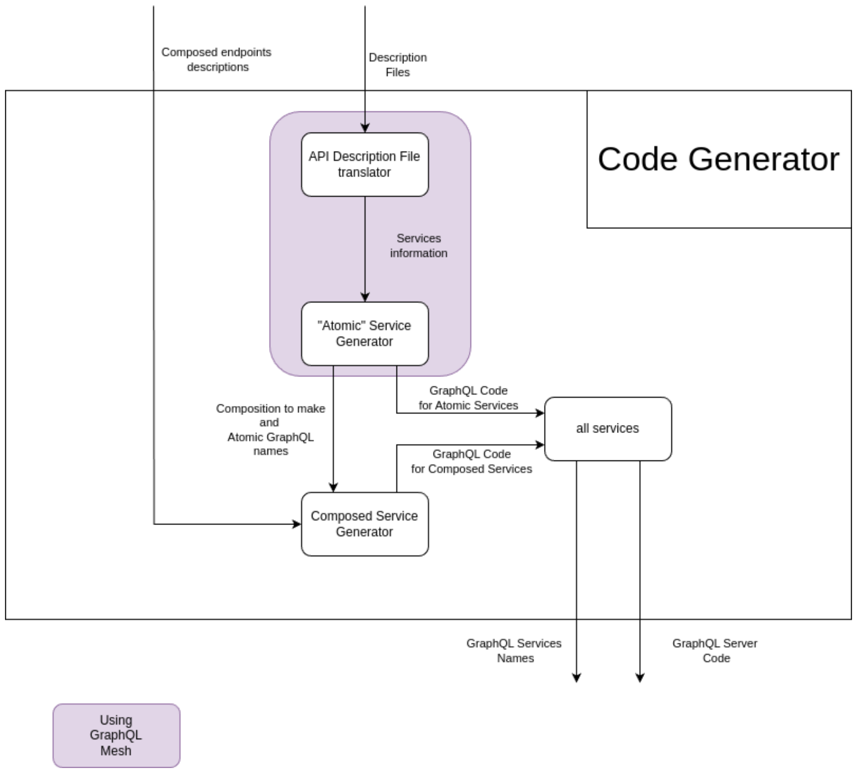

Composed endpoints descriptions
Description Files
Code Generator
API Description File translator
Services information
"Atomic" Service Generator
GraphQL Code for Atomic Services
all services
Composition to make and Atomic GraphQL names
GraphQL Code for Composed Services
Composed Service Generator
GraphQL Services Names
GraphQL Server Code
Using GraphQL Mesh

Figure 5.10: Code Generator Service

`Setor Ativo` composition, `Setor ID` serves as the key connecting `Algo360` and `Environ`.

Figure 5.10 illustrates the process of generating Composed and Atomic services using the described JSON file.

5.3.2 Creating Atomic Services

With the input file defined, the code generator service passes to the GraphQL Mesh library to generate all Atomic Services. Each atomic item in the JSON list of atomic attributes generates what the GraphQL Mesh calls a subgraph. However, two key challenges arise during this process.

The first comes from the FastAPI library, which creates some validation objects in their OpenAPI files that GraphQL Mesh cannot understand, to solve that, it is necessary to remove them. Regardless, since GraphQL Mesh cannot load it directly from memory, the server must save the OpenAPI file locally and then give it to GraphQL Mesh, and that is the reason why, in the Appendix

A.1, there is a **source** property only in the Rest Services, that property serves only to GraphQL Mesh know where the OpenAPI of each Service is stored.

The second problem is the problem raised in Section 5.2, that queries that return a list of items or items cannot be generated. To address this, the OpenAPI file must be updated to explicitly define return types as lists wherever a list or unit conflict arises. In contrast, Apollo GraphQL does not face these limitations and, therefore, does not require the inclusion of a **source** attribute.

After resolving these challenges, GraphQL Mesh generates a unified GraphQL Schema file containing types and queries for all APIs registered. For GraphQL Mesh, the resolver functions do not need to be created. The reason for that is discussed in Section 7.2.

To incorporate a new **Atomic Service**, the following elements are required: a valid API, its corresponding endpoint, a designated name for the API, and a valid API description file. This information should then be included within the **atomic** property of the JSON file.

5.3.3
Creating Composed Services

To create the Composed Services, GraphQL types, queries, and resolvers need to be implemented. The types and queries are defined by updating the GraphQL Schema file with the necessary information. That information is available in the JSON file, specifically in the **new type**, **query signature** and **parameters** within the **composed** property.

To create new resolver functions for the Composed Services, adding them to the GraphQL Hive resolvers list is necessary. A Javascript function with the same signature as the GraphQL Resolver function must be created for each new query. To solve this problem, a Javascript function that returns a resolver function was created for each type of operation.

Each resolver function created by the Composed Service was designed to use the inner GraphQL query executor of GraphQL Hive's server to interact with each API. This approach avoids defining multiple API call definitions regardless of the API paradigm. To invoke a GraphQL query, it is necessary to know the shape of the return type of that query. It is important to note that GraphQL can figure that out for you, which is discussed in Section 8.1, but that was not used for this implementation. The return type shape is defined in the **attributes** property within the **left** or **right** properties.

Another property that is necessary to know to call the query is the name of the query, which can be found inside either **left or right**, in the **name**. The last item necessary to run GraphQL queries is the parameters, which are defined in the **params** and also in the **left** or **right** property.

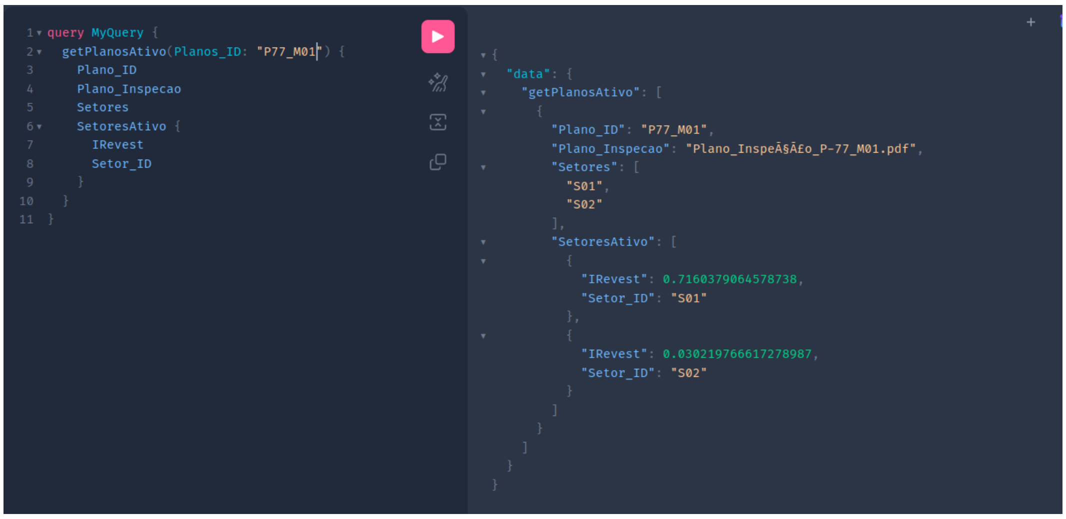
Calling both **left** and **right** queries alone is insufficient; their data must be combined. To achieve this, the **keys** property is required, specifying a reference to the equivalent property in both queries.

The only property left is the **return list**, which is used to mitigate the problem of a single query that either returns a list or a single object. When composing the final response, the function cannot understand if the query returns a single object or if it just returned a list with a single object. This property guides the composition function in treating the query's return value as a list or a single object, ensuring accurate response formatting.

To add a new **Composed Service**, provide the composed properties—such as queryName, querySignature, operationType, items, and others—in the JSON file. These details should be included within the **compose** property of the JSON file.

5.3.4
Querying the Server

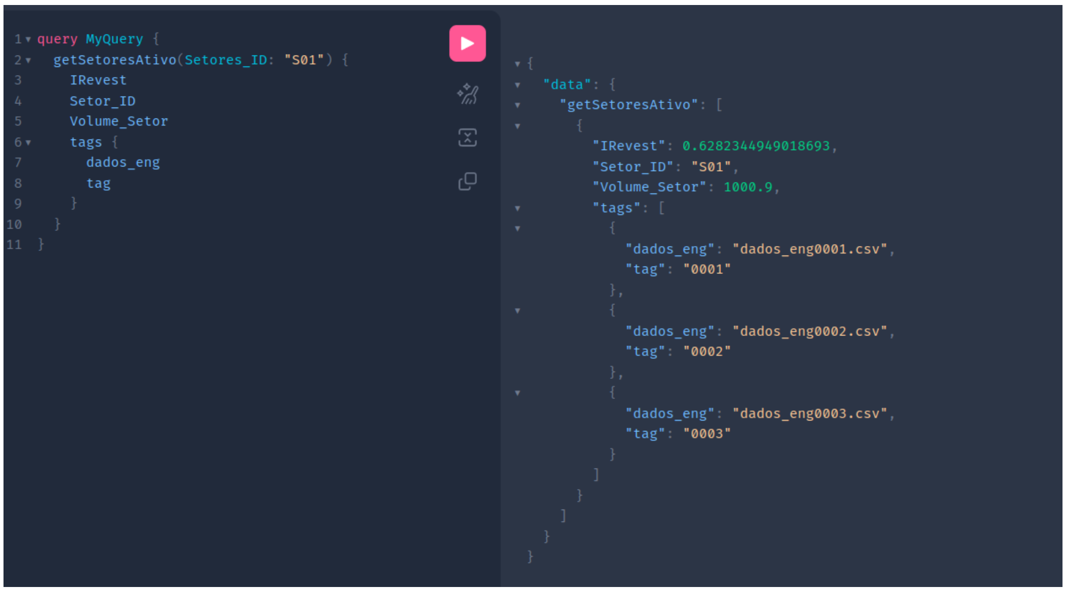
After implementing everything described by Subsections 5.3.2 and 5.3.3, a GraphQL server can be generated. Figure 5.11 illustrates a query of type *LIST UNIT UNION*, specifically the *getPlanosAtivo* query. In this example, the query returns a list with a single item instead of an object, despite only



```
query MyQuery {
  getPlanosAtivo(Planos_ID: "P77_M01") {
    Plano_ID
    Plano_Inspecao
    Setores
    SetoresAtivo {
      IRevest
      Setor_ID
    }
  }
}
```

```
{
  "data": {
    "getPlanosAtivo": [
      {
        "Plano_ID": "P77_M01",
        "Plano_Inspecao": "Plano_InspeÃ§Ã£o_P-77_M01.pdf",
        "Setores": [
          "S01",
          "S02"
        ],
        "SetoresAtivo": [
          {
            "IRevest": 0.7160379064578738,
            "Setor_ID": "S01"
          },
          {
            "IRevest": 0.030219766617278987,
            "Setor_ID": "S02"
          }
        ]
      }
    ]
  }
}
```

Figure 5.11: Query getPlanosAtivo



```
query MyQuery {
  getSetoresAtivo(Setores_ID: "S01") {
    IRevest
    Setor_ID
    Volume_Setor
    tags {
      dados_eng
      tag
    }
  }
}
```

```
{
  "data": {
    "getSetoresAtivo": [
      {
        "IRevest": 0.6282344949018693,
        "Setor_ID": "S01",
        "Volume_Setor": 1000.9,
        "tags": [
          {
            "dados_eng": "dados_eng0001.csv",
            "tag": "0001"
          },
          {
            "dados_eng": "dados_eng0002.csv",
            "tag": "0002"
          },
          {
            "dados_eng": "dados_eng0003.csv",
            "tag": "0003"
          }
        ]
      }
    ]
  }
}
```

Figure 5.12: Query getSetoresAtivo

providing one parameter, as expected. Again, in this second example of query, illustrated by Figure 5.12, with the *getSetoresAtivo*, the same can be observed as the Figure 5.11.

6 Evaluation

6.1 Technical Specifications

A series of tests were designed to evaluate the difference in the performance of those two implementations. Each test uses the mean of 1000 queries for each individual part of the test.

All tests were done using docker to virtualize each application (REST and GraphQL). The computer specifications used to run the tests used an Intel CPU i7-8750H 4.100GHz, with 16GB of RAM, using Ubuntu 22.04.5.

6.2 Tests

The first test compares the response time of the generated code with the hard-coded query for retrieving all Ativo Plans, which are composed of Cronos and Algo360, using the APIs described in Section 5.2. This test was designed

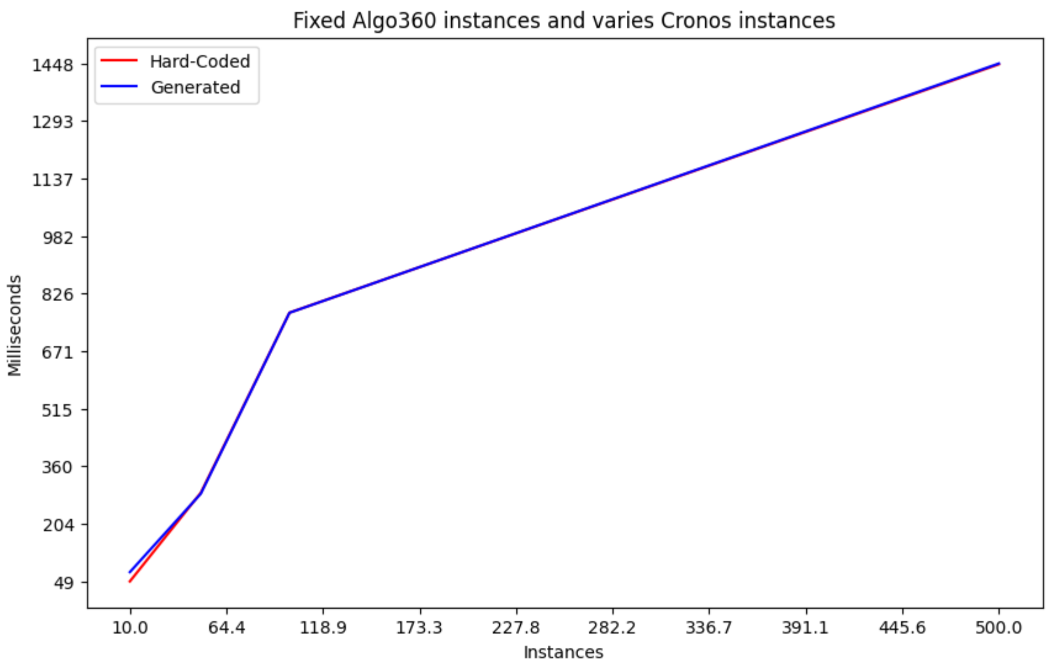


Figure 6.1: First performance test

to assess whether there is a performance difference between generated and non-generated code. For simplicity, this test uses twice the number of instances in Plan360 as in Cronos. The test ran with 10, 50, 100, and 500 Cronos instances. As illustrated in Figure 6.1, the difference in response time between the two implementations for a smaller number of instances is negligible.

The second test evaluates whether the performance difference between the two API management implementations persists with an increasing number of Algo360 instances. In this case, the number of Cronos units is fixed at 10, while the number of Algo360 instances increases by 5, 10, 25, 33 and 50 for each Cronos instance. As shown in Figure 6.2, queries with a higher number of compositions take longer to complete. This is because, for queries of type *LIST UNIT UNION*, a query to Algo360 is made for each item in each list of Cronos. It was also demonstrated that the generated code is slower as the number of instances for composition grows.

The third and final test measures the difference in performance between calling the endpoint directly, using the hard-coded API management, and using the generated API management. In this test, Cronos was queried with 50, 100,

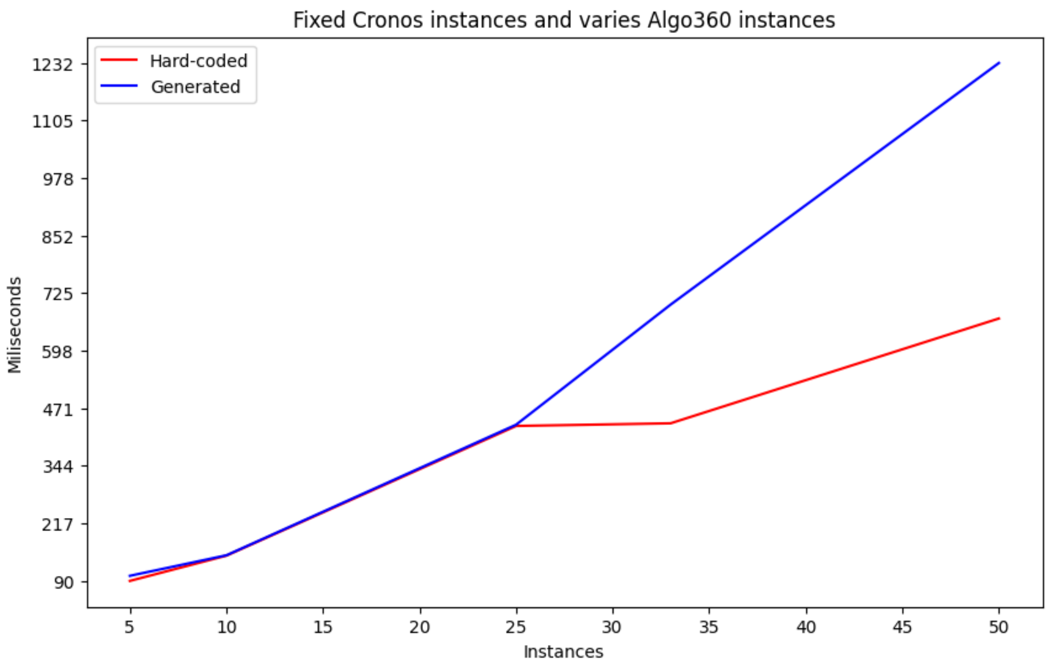


Figure 6.2: Second performance test

500, and 1000 instances. As expected, the REST API is faster than using GraphQL to fetch REST data because GraphQL introduces an additional processing layer. This difference is evident in Figure 6.3, which shows that the REST API is 1.7 times faster than the API management for 1000 instances.

This evaluation demonstrates that GraphQL introduces at least some latency in response times. However, it is noteworthy that the generated server performs comparably to the hard-coded server when handling a small amount of data. The most significant limitation of this version of GQS is its performance when joining large datasets. Potential optimizations to mitigate this issue are discussed in Section 7.5.

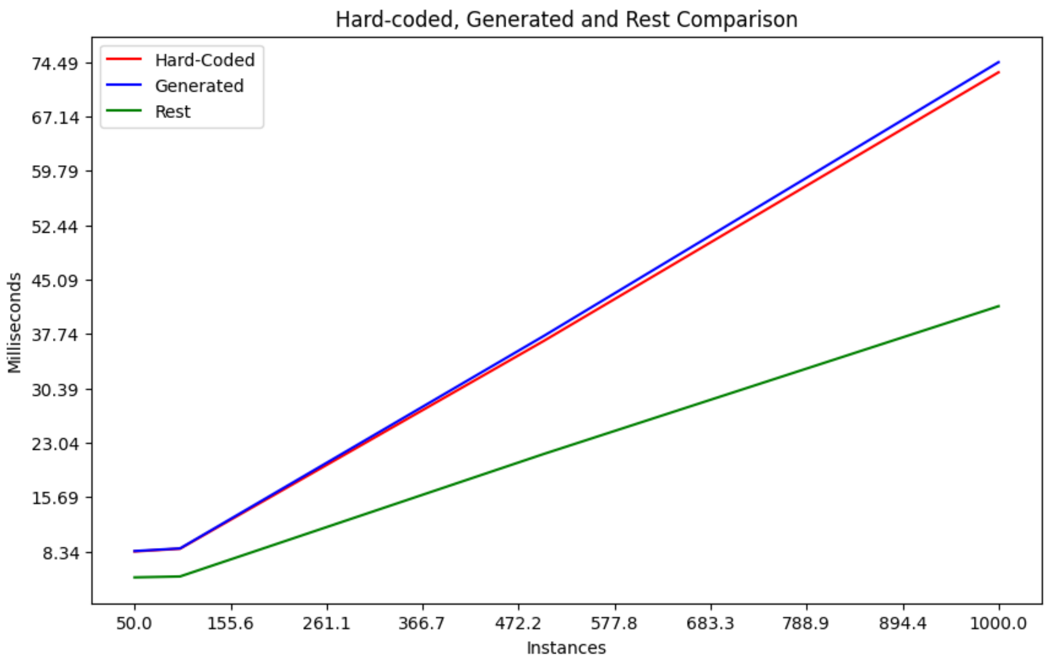


Figure 6.3: Third performance test

7 Discussion

As detailed in Chapter 5, CGS presents numerous opportunities for improvement. This chapter discusses potential enhancements, limitations, and avenues for future development to extend and refine CGS.

7.1 The input file of Code Generator Service

As shown in Section 5.3, the JSON file that works as the input of the Code Generator is quite complex and extensive. However, some of its complexity and extensiveness are due to a simplification of coding the GraphQL Code Generator. For instance, the query signature can be represented as a concatenation of a query name and a return type. Although the return type is not defined, it should be simpler than building the whole signature.

A further challenge lies in handling items within the GraphQL Code Generator. Firstly, the *UNIT UNIT UNION* could be adapted to support an arbitrary number of objects. Secondly, the query object theoretically could be programmed only to need the key property and the name property. Parameters, attributes, and return lists could be inferred automatically through introspective queries that extract all requisite data directly from the server. Nevertheless, future work could improve the GraphQL Code Generator as a whole, from minimizing the input file to adding the feature to compose queries with multiple inputs.

7.2 GraphQL Implementations

From the hard code to the generated server from prototype 2, a change was made to use GraphQL Hive instead of Apollo GraphQL. There are a vast amount of GraphQL implementations in multiple languages, each one with its strengths and its problems. Initially, Apollo GraphQL was chosen since it seemed to be the staple for JavaScript implementations of GraphQL. It has many interesting features, like API Federations, a tool for composing GraphQL Servers. The main problem with that was discovered later: that feature is not open source and needs to be paid for.

Other competing libraries, such as WunderGraph GraphQL and Strawberry GraphQL, are entirely open-source. In my research for GraphQL Server, I found a set of GraphQL helping libraries from a company called The Guild. They focus on creating open-source GraphQL solutions and are backed by the GraphQL Foundation. One of those libraries is GraphQL Mesh.

GraphQL Mesh implements parsers from a list of data sources description files (OpenAPI, gRPC, mySQL, Neo4J) to GraphQL code. Since this aligns with the requirements of Atomic Services, GraphQL Mesh was selected, as it simplifies the integration of additional data sources.

Although it generates all the necessary code for executing the GraphQL code, it is not server-agnostic. GraphQL lets all its implementations define directives, which can be used to execute custom functions other than just the resolver. Unfortunately, those directives are often tailored to work within the server implementation, so they do not work in all GraphQL servers. Currently, GraphQL Hive Server is the only GraphQL implementation that can execute such directives.

Future research should focus on designing a generic framework for the Atomic Service Generator, enhancing compatibility across diverse GraphQL server implementations. Since most GraphQL Server implementations need to have queries, types and resolvers, a solution could be devised to generate types and queries for any GraphQL server and a solution for generating resolvers for a specific programming language.

GraphQL Mesh also has a transformer feature that lets you transform the data. Since the Atomic Service Generator is already implemented in a non-server agnostic way, it was decided to be developed without using

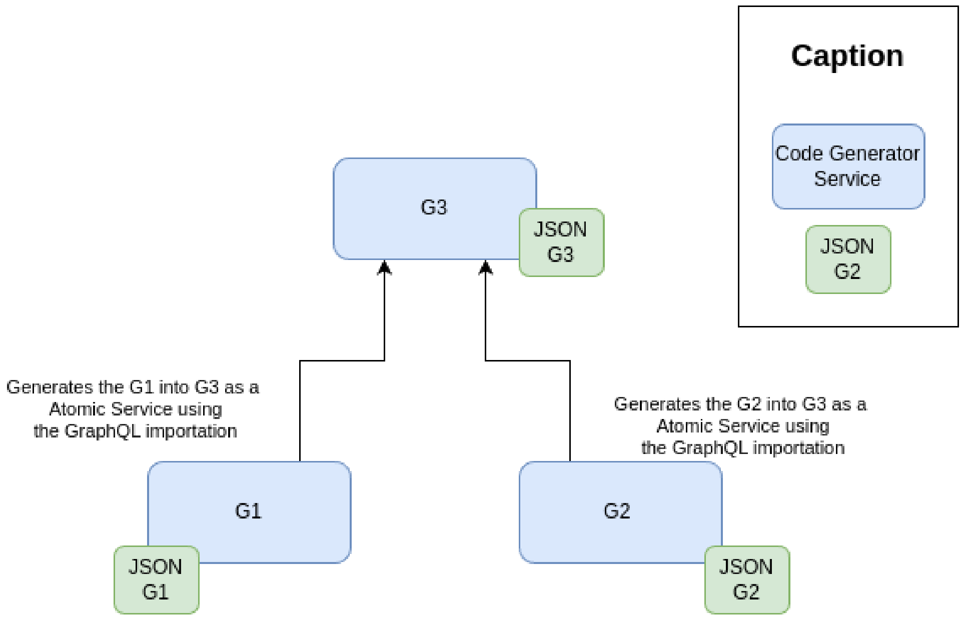

Caption
Code Generator Service
JSON G2
G3
JSON G3
Generates the G1 into G3 as a Atomic Service using the GraphQL importation
Generates the G2 into G3 as a Atomic Service using the GraphQL importation
G1
JSON G1
G2
JSON G2

Figure 7.1: Recursive CGS

the transformer feature of GraphQL Mesh. That way, the Compose service generator could be developed with any Atomic Service generator.

Future research could delve deeper into comparing GraphQL implementations. At the time of this study, no comprehensive review of GraphQL implementations was identified. Existing literature primarily focuses on comparing various aspects of GraphQL itself but does not specifically address the differences or performance of its implementations. The main example of that is the paper Mera et al. (2023).

7.3 Scalability for Encapsulation

An idea that could arise from Section 5.3 is since CGS can generate a GraphQL Server using pre-existing GraphQL Servers, that in theory, you could connect a CGS instance to another CGS instance, this process is illustrated by Figure 7.1. Let *G1* and *G2* be two instances of CGS, with two separated composition descriptions; if you want to connect *G1* to *G2*, it is possible to create a third CGS instance called *G3* that has its own separated composition description that connects *G1* and *G2*. This approach enables *G1* and *G2* to be developed independently. Future research should assess the scalability and performance impact of such recursive configurations.

7.4 GraphQL Union and Type Grammar

Previously, in Section 5.2, the issue of creating a unified query in GraphQL to retrieve all, some, or a single instance was highlighted. To understand the root cause of this limitation, an analysis of the GraphQL Specification was conducted, and the relevant grammar rules were extracted from the current specification and are shown as follows:

$$
\begin{array}{lcl}
\textbf{UnionTypeDefinition} & \rightarrow & \textit{Description}_{opt}\ \texttt{union}\ \textit{Name} \\
 & & \textit{Directives}_{[Const]\ opt}\ \textit{UnionMemberTypes}_{opt} \\
\textbf{UnionMemberTypes} & \rightarrow & \textit{UnionMemberTypes}\ \texttt{|}\ \textit{NamedType} \\
\textbf{UnionMemberTypes} & \rightarrow & \texttt{=}\ |_{opt}\ \textit{NamedType} \\
\textbf{Type} & \rightarrow & \textit{NamedType} \\
\textbf{Type} & \rightarrow & \textit{ListType} \\
\textbf{NamedType} & \rightarrow & \textit{Name} \\
\textbf{ListType} & \rightarrow & \texttt{[}\ \textit{Type}\ \texttt{]}
\end{array}
$$

The rule **Type** is used to refer to any GraphQL type possible. Therefore, $[Type]$ is a valid type for **Type**. Regardless, **NamedType** can only generated by the usage of the *Name* token, which represents basic types such as Strings, Float, or non-nullable objects. Consequently, it does not allow the definition of a GraphQL type that combines a list and an object.

7.5 Optimizations

As shown in Chapter 6, the response time increases as the number of compositions grows. This is because the query for the left item must be completed before initiating the query for the right item. Additionally, in the case of the *LIST UNIT UNION*, a new query is generated for each element in the array. While this issue likely cannot be entirely eliminated, it can be mitigated through optimizations. For example, the query for the right item could be initiated concurrently while receiving the left item. Another potential optimization is to replace individual queries for each item with a single query that retrieves all elements of the left composition at once, thereby reducing the number of separate queries.

7.6 Benefits and drawbacks

After the development, it is clear that the process to add new **Atomic Services** and **Composed Services** is straightforward after improving maintainability and usability compared to the Li et al. (2024), even when considering the refinement that can be done for the JSON file, described at Section 7.1.

Unfortunately, only two types of composed queries can be created for this work. Further research could investigate the creation of more compositions, and further data transformations can extend the current implementation. An example of a new type of query that can be developed is the one used in Section 5.1 to serve Photos from Plan360.

Another limitation of this implementation is its inability to compose multi-parameter queries, as all queries are limited to a single parameter. This is because parameters must be mapped from one query item to another.

While it is not possible to generate all types of compositions using only those two types of compositions, the server can be extended in future versions to support additional compositions types and enable the creation of custom compositions. It is important to highlight that implementing complex Level 2 DT cannot rely solely on these composition methods but rather just be a tool for building them.

A remarkable benefit of this approach is that it can combine heterogeneous systems using different paradigms without the need to change the internal structure, even if some API Gateway implementation don't generate their description file, which could be created without changing the internal structure, thus generating the entire server seamlessly.

Another notable benefit of this approach is its capacity to serve as a unified API for all defined systems. Consolidating all endpoints into a single API management framework eliminates the redundancy of duplicated interfaces, streamlining system interactions.

8 Conclusion

This research proposed a tool for building level 2 or higher Digital Twins by addressing the challenge of interoperability in Systems of Systems (SoS), offering new perspectives on composing and utilizing data without compromising existing systems. This was achieved by combining the capabilities of GraphQL and the description of the integration of APIs, which presented a novel framework for defining a unified API that integrates all endpoints of an SoS and facilitates the composition of new queries derived from those endpoints.

The literature review demonstrated that this research was built on a solid foundation, with the works of (LI et al., 2024; BORGES; ROCHA; MAIA, 2022) serving as key inspirations. The ideas from these studies contributed to the development of a generic solution to address some challenges of their research, like the lack of usage of a better system than just syntactical analyses or a method that could eventually simplify the description of their systems.

8.1 Future work

Chapter 4 defined the **Semantic Description Parser**, even though its development using any type of parser to JSON was set aside, the implications of such must be discussed. Firstly, the goal of the parser would be to map how to connect distinct APIs that could be modelled in various ways, as long as its parser could generate the JSON input of the GraphQL Code Generator Service. Since there was no **Semantic Description Parser** other types of Composed Service Descriptions could be employed. In Borges, Rocha e Maia (2022), they compose new queries using syntactical analyses as an example, but the powerfulness of this implementation is that it could be achieved not relying on syntax but rather in a description.

For those descriptions, ontologies, like the one outlined in Li et al. (2024), offer a promising approach. Another potential solution involves developing a domain-specific language (DSL) tailored to describing and defining service integrations. Additionally, creating a Knowledge Graph to represent the relationships and integration pathways between services could provide a robust and flexible alternative.

A potential solution to further simplify the creation of composition description files is to partially generate them using the existing API description files. That would facilitate the creation of certain sections of the composition description, leaving only the composition itself to be defined manually. This eliminates the need to repeatedly specify detailed type structures, reducing complexity and effort.

A few other future research were defined in Chapter 7 as direct improvements of the current work, like the improvement of the input file, the creation of more types of composed queries and further optimizations.

8.2
Closing Remarks

The prototypes made for this work exemplified how uncomplicated it would be to connect two APIs using this approach and how, in future research, that could even be more simplified. However, there are still many challenges, like the actual best implementation of GraphQL to be used or optimizations.

Even though the performance results varied—showing improvements in some scenarios and limitations in others it is significant that the initial version of CGS managed to achieve comparable performance to hard-coded solutions in two of the tests. That demonstrates the promise of the proposed framework while highlighting areas for potential optimization and improvement.

Overall, this study has significantly enhanced my understanding of APIs, particularly GraphQL, as well as concepts such as Systems of Systems and Digital Twins. Moreover, the research opens avenues for further exploration, as discussed previously, and hopefully, this contribution marks a notable step towards the interoperability of Systems of Systems.

9 Bibliography

ALTAMIRANDA, E.; COLINA, E. A system of systems digital twin to support life time management and life extension of subsea production systems. In: **OCEANS 2019 - Marseille**. [S.l.: s.n.], 2019. p. 1–9. Cited in page 6.

ANACKER, H. et al. Pattern based engineering of system of systems - a systematic literature review. In: **17th Annual System of Systems Engineering Conference (SOSE)**. [S.l.: s.n.], 2022. p. 178–183. Cited 4 times in pages 5, 11, 12, and 15.

Apollo Graph Inc. **Apollo GraphQL Documentation**. 2024. <https://www.apollographql.com/docs/>. Accessed: 2024-05-22. Cited in page 23.

BONDEL, G.; LANDGRAF, A.; MATTHES, F. Api management patterns for public, partner, and group web api initiatives with a focus on collaboration. **Proceedings of the ACM on Programming Languages**, ACM, v. 5, n. OOPSLA, p. 1–28, 2021. Disponível em: <https://doi.org/10.1145/3489449.3490012>. Cited in page 5.

BORGES, M. V. D. F.; ROCHA, L. S.; MAIA, P. H. M. Micrographql: a unified communication approach for systems of systems using microservices and graphql. In: **2022 IEEE/ACM 10th International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems (SESoS)**. [S.l.: s.n.], 2022. p. 33–40. Cited 8 times in pages 11, 12, 13, 15, 16, 17, 18, and 44.

CÂNDEA, C.; CÂNDEA, G.; STAICU, M. Impact of iot and sos in enabling smart applications: A study on interconnectivity, interoperability and quality of service. **Procedia Computer Science**, v. 221, p. 1226–1234, 2023. Disponível em: <https://doi.org/10.1016/j.procs.2023.08.110>. Cited 4 times in pages 11, 12, 13, and 15.

Docker, Inc. **Docker Official Website**. 2024. <https://www.docker.com/>. Accessed: 2024-05-22. Cited in page 23.

Express.js. **Express Documentation**. [S.l.], 2024. Accessed: 2024-05-23. Disponível em: <https://expressjs.com/en/4x/api.html>. Cited in page 23.

GraphQL Foundation. **Introduction to GraphQL**. 2024. <https://graphql.org/learn/>. Accessed: 2024-05-22. Cited in page 7.

GROSMAN, J. **Findpapers: A tool for helping researchers who are looking for related works**. 2024. <https://github.com/jonatasgrosman/findpapers>. Accessed: 2024-05-22. Cited in page 10.

IZQUIERDO, Y. T. et al. Busca360: A search application in the context of top-side asset integrity management in the oil gas industry. In: **Anais do XXXIX Simpósio Brasileiro de Bancos de Dados**. Porto Alegre, RS, Brasil: SBC, 2024. p. 104–116. ISSN 2763-8979. Disponível em: <https://sol.sbc.org.br/index.php/sbbd/article/view/30686>. Cited in page 21.

LI, H. et al. Ontology-based graphql server generation for data access and data integration. **Semantic Web**, IOS Press, 2024. Disponível em: <https://doi.org/10.3233/SW-233550>. Cited 8 times in pages 8, 11, 13, 15, 16, 17, 42, and 44.

LUNA, E. D. **GraphQL Code Generator**. GitHub, 2025. Accessed: 2025-01-29. Disponível em: <https://github.com/Luna-v0/gql-code-gen>. Cited in page 22.

MERA, A. Q. na et al. Graphql: A systematic mapping study. **ACM Comput. Surv.**, v. 55, n. 10, p. 202:1–202:35, 2023. Cited in page 40.

MITTAL, S. et al. Autonomous and composable m&s system of systems with the simulation, experimentation, analytics and testing (seat) framework. In: IEEE. **Proceedings of the 2020 Winter Simulation Conference**. [S.l.], 2020. p. 2305–2316. Cited 3 times in pages 11, 12, and 15.

MOHSIN, A.; JANJUA, N. K. A review and future directions of soa-based software architecture modeling approaches for system of systems. **Service Oriented Computing and Applications**, Springer-Verlag London Ltd., part of Springer Nature, v. 12, n. 3, p. 183–200, 2018. Disponível em: <https://doi.org/10.1007/s11761-018-0245-1>. Cited 3 times in pages 11, 13, and 15.

NEUREITER, C. et al. Extending the concept of domain specific systems engineering to system-of-systems. In: IEEE. **2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)**. 2020. p. 391–396. Disponível em: <https://ieeexplore.ieee.org/document/9130484>. Cited 4 times in pages 11, 13, 14, and 15.

OLSSON, T.; AXELSSON, J. Systems-of-systems and digital twins: A survey and analysis of the current knowledge. In: IEEE. **2023 18th Annual System of Systems Engineering Conference (SoSE)**. 2023. Disponível em: <https://doi.org/10.1109/SOSE59841.2023.10178527>. Cited 2 times in pages 5 and 11.

OpenAPI Initiative. **OpenAPI Specification v3.1.0**. 2024. <https://spec.openapis.org/oas/latest.html>. Accessed: 2024-05-22. Cited in page 8.

PICKERING, N.; DUKE, M.; LIM, S. H. A time constrained system of systems discovery process and canvas - a case study in agriculture technology focusing on an automated asparagus harvester. In: IEEE. **2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)**. [S.l.], 2020. p. 67–74. Cited 3 times in pages 11, 13, and 15.

RAMONELL, C.; CHACóN, R.; POSADA, H. Knowledge graph-based data integration system for digital twins of built assets. **Automation in Construction**, v. 156, p. 105109, 2023. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0926580523003692>. Cited in page 8.

RAMíREZ, S. **FastAPI Documentation**. 2024. <https://fastapi.tiangolo.com/>. Accessed: 2024-05-22. Cited in page 23.

SHI, W. et al. A survey on edge computing for the internet of things. **IEEE Internet of Things Journal**, IEEE, v. 3, n. 5, p. 637–646, 2016. Disponível em: <https://ieeexplore.ieee.org/document/7030212>. Cited in page 5.

The Guild. **GraphQL Mesh**. 2024. Accessed on 13 November 2024. Disponível em: <https://the-guild.dev/graphql/mesh>. Cited in page 28.

WEINERT, B.; USLAR, M. Challenges for system of systems in the agriculture application domain. In: IEEE. **2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)**. 2020. p. 355–360. Disponível em: <https://ieeexplore.ieee.org/document/9130484>. Cited 4 times in pages 11, 13, 14, and 15.

A
Appendix

A.1
JSON Input of the GraphQL Code Generator Service

Code 1: Sample JSON File

```
{
    "atomic": [
        {
            "type": "openapi",
            "name": "Algo360",
            "source": "openapi/Algo360.json",
            "endpoint": "http://localhost:8001"
        },
        {
            "type": "openapi",
            "name": "Cronos",
            "source": "openapi/Cronos.json",
            "endpoint": "http://localhost:8002"
        },
        {
            "type": "graphql",
            "name": "Environ",
            "endpoint": "http://localhost:8003",
            "source": ""
        }
    ],
    "rename": {
        "getSetor": "Setor"
    },
    "composed": [
        {
            "operationType": "UNIT UNIT UNION",
            "queryName": "getSetoresAtivo",
            "querySignature": "getSetoresAtivo(Setores_ID:[String])
                :[SetorAtivo]",
            "newType": {
                "name": "SetorAtivo",
                "type": {
                    "Setor_ID": "String",
                    "IRevest": "Float",
                    "Volume_Setor": "Float",
                    "tags": "[Tag]"
                }
            },
            "parameters":"(Setor_ID:[String])",
            "items":{
                "left": {
                    "query": {
                        "returnsList": true,
                        "name": "getSetor",
                        "params":{
                            "Setor_ID": "String"
                        },
                        "attributes": [
                            "Setor_ID",
                            "Volume_Setor",
                            "tags{dados_eng\ntag}"

                        ],
                        "key": {
                            "path": [
                                "Setor_ID"
                            ]
                        }
                    }
                },
                "right": {
                    "query": {
                        "returnsList": false,
                        "name": "getSetores_setores_get",
                        "params":{
                            "Setor_ID": "String"
                        },
                        "attributes": [
                            "Setor_ID",
                            "IRevest"
                        ],
                        "key": {
                            "path": [
                                "Setor_ID"
                            ]
                        }
                    }
                }
            }

        },
        {
            "operationType": "LIST UNIT UNION",
            "attributeName": "SetoresAtivo",
            "queryName": "getPlanosAtivo",
            "querySignature": "getPlanosAtivo(Planos_ID:[String]):[
                PlanoAtivo]",
            "newType": {
                "name": "PlanoAtivo",
                "type": {
                    "Plano_ID": "String",
                    "Plano_Inspecao": "String",
                    "Setores": "[String]",
                    "SetoresAtivo": "[Setor]"
                }
            },
            "parameters": "(Plano_ID:[String])",
            "items": {
                "left": {
                    "query": {
                        "returnsList": true,
                        "params":{
                            "Plano_ID": "String"
                        },
                        "name": "getPlanos_planos_get",
                        "attributes": [
                            "Plano_ID",
                            "Plano_Inspecao",
                            "Setores"
                        ],
                        "key": {
                            "path": [
                                "Setores"
                            ]
                        }
                    }
                },
                "right": {
                    "query": {
                        "returnsList": false,
                        "name": "getSetores_setores_get",
                        "params":{
                            "Setor_ID": "String"
                        },
                        "attributes": [
                            "Setor_ID",
                            "IRevest"
                        ],
                        "key": {
                            "path": [
                                "Setor_ID"
                            ]
                        }
                    }
                }
            }
        }
    ]
}
```