SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Ingegneria

Dipartimento di Ingegneria Informatica, Automatica e Gestionale

"Antonio Ruberti"

Master Thesis in Engineering in Computer Science

ArchiGPT
An LLM-based software for the generation of Software Architecture

Author:

Giovanni Nicola DELLA PELLE

Eugenio FACCIOLO

Supervisor:

Prof. Massimo MECELLA

Academic Year MMXXIII-MMXXIV

Contents

Chapter 1

Introduction

The evolution of Large Language Models (LLMs) in recent times has profoundly changed the realm of Artificial Intelligence, particularly in natural language processing. Models like BERT, GPT-3, and GPT-4 represent significant milestones in this field, with their revolutionary scale and capability to understand and generate human-like text. The impact of LLMs extends beyond theoretical research, affecting practical applications across various industries. In Software Engineering, LLMs have introduced new paradigms, transforming traditional development practices and introducing innovative tools that enhance efficiency and productivity.

LLMs are being utilized in contemporary Software Engineering to automate and improve a range of tasks throughout the software development lifecycle. They aid in code completion by predicting code snippets based on context, thereby reducing typing effort and minimizing syntactic errors. Developers benefit from intelligent code suggestions that are contextually relevant and can adapt to different programming languages and frameworks. LLMs also facilitate automated documentation generation, translating code into human-readable descriptions, which is invaluable for maintenance and collaboration. In debugging, LLMs can identify potential issues and suggest fixes by recognizing patterns associated with common errors. Additionally, advancements can be seen in requirement analysis, with their ability to interprete natural language specifications and convert them into formal representations.

Over the past several decades, system architectures have changed significantly to address the demands of software applications. Initially, monolithic architectures was the most used paradigm, where application's components, user interface, business logic, and data access layers were tightly coupled and deployed as a single and inseparable unit. This approach simplified development and deployment, but introduced challenges in scalability, maintainability, and agility.

Service-Oriented Architecture (SOA) represented the next step in system design. Its principles describe loosely coupled services that work together by communicating over a network. SOA encapsulates business logic within services, accessible to each other and to the user through standardized interfaces, with the aim to increase reusability and flexibility. However, SOA implementations often faced large and complex challenges and constraints due to heavy middleware requirements, governance difficulties and intricate boilerplates.

In response to these challenges, microservices architecture emerged as a the new industry paradigm. It decomposes applications into small, independent services communicating over lightweight protocols (like REST). Each microservice is responsible for a specific functionality, feature and/or

business logic slice and can be developed, deployed, and scaled independently from other services. Microservices architecture streamlines continuous integration and deployment practices, giving advantages in deploying modifications and new features within a short period of time.

Key characteristics of microservices architecture are respresented by decentralization, autonomy, and scalability. Decentralization refers to the ability to distribute services across different environments, allowing them to be deployed on different servers and locations, and to use different technologies for each service, allowing architects to choose the stack that best suits service's requirements. Autonomy means that microservices operate independently, with the goal of minimizing dependencies and reducing the risk of failures that invovlve the whole system. Services can scale horizontally based on demand, optimizing resource utilization and allowing for economical and technical advantages.

Microservices also embrace a culture of continuous delivery and DevOps practices. The architecture gives the possibility of frequent and seamless updates, prioritizing in this way the responsiveness to changing business needs. Additionally, microservices facilitate fault isolation: the loosely coupled architecture has the positive effect that failures in one service do not affect the entire system, enhancing overall system resilience.

Dockerization plays a crucial role in enhancing the positive features of microservices architecture. Docker is a containerization platform that packages applications and their dependencies into containers, ensuring consistent environments across development, testing, and production. Encapsulating microservices within Docker containers ensures the portability and flexibility needed by modern practices.

The central premise of this thesis is to adapt the methodologies employed in code generation and evaluation using large language models (LLMs) to the architectural design of systems adhering to microservices architecture principles.

In the domain of code generation, models have been developed that transform textual descriptions of functions into executable code that meets specified requirements. To evaluate these models, researchers have assembled comprehensive datasets of functions from various sources. Each function in these datasets comprises a textual description outlining the function's purpose and unit tests designed to assess the function's output. These unit tests serve as benchmarks to validate the correctness and functionality of the generated code, ensuring that it fulfills the intended requirements.

These models generate source code for the functions based on their textual descriptions, and the generated code is then evaluated using the corresponding unit tests. Metrics have been established to summarize the unit test results across all functions in the dataset and multiple execution runs. These metrics provide meaningful indices that enable researchers to compare different code generation models under consistent parameters, thereby highlighting the strengths and weaknesses of each model.

Applying this concept to the architectural design of systems necessitated the creation of a specific output and dataset format. While the natural output format for functions is the function's source code, the architectural design of systems requires a structured and standardized format to capture the complexities inherent in microservices architecture. To this end, the Project Standard format has been defined to standardize the output. This format structures the system design in accordance with microservice architecture guidelines, specifying parameters to be described for the containers and microservices within the system, including their interactions, dependencies, and configurations.

The input to the model consists of a combination of textual descriptions and detailed user stories that encapsulate the functional requirements and use cases of the system.

Subsequently, a model named ArchiGPT has been developed to undertake the architecture design task. ArchiGPT accepts as input a brief description of the system to be designed along with its user stories. Utilizing a series of prompts that employ advanced prompting techniques—such as context preservation, role assignment, and iterative refinement—ArchiGPT is capable of generating the project's architecture design in the Project Standard format. This approach ensures that the generated architectural designs are coherent, scalable, and aligned with best practices in microservices architecture.

To evaluate the systems designed by ArchiGPT and other models following the same input/output schema, a dataset of projects has been constructed. The Project Dataset originates from projects completed in an academic class at La Sapienza University in Rome. Initially, these projects only provided source code and fragmented information regarding the purpose and architecture of the developed systems. The projects were meticulously selected and cleaned to ensure data quality and relevance. Technical documentation was generated through a thorough analysis of the source code, including architectural diagrams, component specifications, and interaction flows. In their finalized form, each project includes the description and user stories that serve as input for models like ArchiGPT, the source code developed by the students, a comprehensive technical analysis, and an attached JSON file formatted according to the Project Standard for evaluation purposes.

To assess the performance of ArchiGPT and future models adhering to the same input/output schema, specific metrics have been defined. These metrics evaluate the defined architecture for each system in terms of how effectively the user stories are fulfilled by microservices, the granularity and modularity of the defined system, and whether the containers possess the necessary backend and database microservices to satisfy the requirements specified by the user stories. The metrics also consider adherence to microservices best practices, such as service isolation, scalability, and fault tolerance, providing a holistic evaluation of the architectural design.

Collectively, this work encompasses all the essential components required to generate and evaluate the architectural design of microservices-based systems using LLMs. It provides a model for generating such architectures, defines the input and output formats for the model, supplies a testing dataset derived from real-world academic projects, and establishes comprehensive metrics for model evaluation. This framework facilitates the advancement of research in automated architectural design and provides a benchmark for future models to be compared and improved upon.

Chapter 2

Containerization and Microservices

2.1 Introduction

The introduction of Containerization and microservice design have triggered a revolution in the world of software engineering by giving fresh assumptions for constructing, deploying, and scaling out applications. This chapter will present a discussion on the change starting with the presentation of containerization. It will cover what containers are, the key notions about them, with emphasis on those aspects that differentiate them from virtual machines, including aspects such as isolation, resource efficiency. Discussions on container images show how the application code and its dependencies are packaged into them. The following comprises a list of advantages arising out of their use: portability, elasticity, and standard environments across development and production. Some of the most used containerization tool sets, such as Docker and Kubernetes, will be reviewed for practical insights.

The chapter then moves on to microservices, explaining how the growth of a monolithic architecture mushroomed into microservice-based systems, emphasizing a greater drive toward modular and flexible designs. Microservices are described as being characterized by independence and loose coupling, allowing teams to autonomously develop, deploy, and scale services. The session also outlined what it means to remain decentralized with data management techniques necessary to realize the full resilience and performance benefits of a microservice approach. The scalability and elasticity principles of microservices are debated, together with some continuous integration and delivery practices that provide integration of development processes. Furthermore, the communication between the microservices is considered in order to understand how they may cooperate within a distributed system. The final phase describes how microservices and containers integrate, and how the process of containerization amplifies the activity of deployment and management within microservice architecture due to an unprecedentedly agile and efficient run of software systems.

2.2 Overview of Containerization

2.2.1 Definition and Basic Concepts of Containers

In modern software engineering, especially within microservices architectures, containers play a critical role. They offer a streamlined, isolated environment for applications, encapsulating both the software and its dependencies in a lightweight, portable, and consistent runtime setting. A

key advantage of containers is their ability to enable platform independence, ensuring applications can run consistently across different stages, such as development, testing, and production, without requiring modifications [57]. This ability is particularly valuable for avoding dependency conflicts and version inconsistencies that often arise when software is deployed in varying environments [25].

Containers can be viewed as standardized units of software that bundle together code, libraries, and configuration files. This guarantees uniform application behavior across any deployment environment. One of the main benefits containers have over traditional virtual machines (VMs) is that they share the host system's kernel, which minimizes the overhead of running separate operating systems for each instance. As a result, containers are more lightweight, faster to initialize, and use fewer resources than VMs [13].

Docker, the leading platform in the containerization market, has widely contributed to the massive adoption of containers by offering a user-friendly interface for creating, deploying, and managing containers [20]. Docker containers package everything necessary for the application—code, runtime, libraries, and configurations—ensuring consistency across different development and production environments. Moreover, Docker's seamless integration with various cloud services has made containers a cornerstone in microservices architecture, accelerating their adoption across the software engineering landscape [59].

In microservices architectures, containers represent a cornerstone technology, offering an isolated and self-sufficient environment for each service to operate independently. This pattern is aligned with the fundamental principles of microservices, where the aim is to develop, deploy, and scale individual services without interdependence. Containers enable developers to ensure that every microservice is equipped with its necessary runtime dependencies, safeguarding it from interference by other services or the host system. Isolation and flexibility are essential for the dynamic scalability that microservices demand.

A key factor that empowers the use of containers in microservices is container orchestration, which automates the deployment, scaling, and management of containerized applications. Tools like Kubernetes have become standard in large-scale production environments, where they can manage a large number of containers efficiently. Kubernetes ensures that applications result resilient and scalable (performance and resource utilization wise) by automating tasks such as load balancing, horizontal scaling, and container recovery. This significantly reduces the operational overhead and simplifies the process of managing complex microservices ecosystems.

Containers have revolutionized the software development and deployment landscape, particularly in the realm of microservices architectures. They provide a consistent, portable, and resource-efficient environment that facilitates seamless scalability and streamlined operations. Using platforms like Docker and Kubernetes, containers have revolutionized software delivery methods, making modern systems agile, scalable, and maintainable in both development and production settings.

2.2.2 Containers and Virtual Machines: Key Differences

Containers and virtual machines (VMs) represent two key technologies in the virtualization landscape, each designed to optimize resource usage and efficiency in modern computing. Despite having a shared goal, they function at different layers of the system architecture, offering unique advantages and trade-offs depending on the use case.

Virtual machines are essentially complete replicas of physical systems, encompassing not only

the full operating system (OS) but also virtualized hardware components. VMs operate under the supervision of a hypervisor, which orchestrates the running of multiple OS instances on a single physical host [34]. This structure allows each VM to function as a standalone system, fully isolated with its own OS, kernel, and system libraries, thus ensuring that applications within one VM do not interfere with those in others. The high degree of isolation VMs provide is crucial for security and resource control, particularly in environments that demand strong separation between applications to avoid conflicts or excessive resource consumption.

In contrast, containers leverage OS-level virtualization, where the host OS kernel is shared among containers, but each container maintains its own isolated user space. This makes containers far more lightweight compared to VMs, as they don't require a full OS for each instance but only the application and its dependencies. Containers run as separate processes within the host OS, which significantly reduces the memory footprint and speeds up startup times, as well as optimizes disk usage when compared to VMs [12]. This efficiency is largely due to containers avoiding the hardware emulation layer that VMs need to operate.

From a performance standpoint, containers often outpace VMs, particularly in terms of application execution speed. The absence of a full OS per instance and reduced resource management overhead allows containers to achieve higher throughput, especially in environments with high-density workloads or microservices-based architectures. However, this performance edge decreases when more stringent isolation and security requirements come into play. In such scenarios, VMs typically offer better performance because of their ability to provide more comprehensive isolation.

A key factor influencing these performance discrepancies is resource allocation. VMs consume more memory and disk space since they require a full OS for each instance. This, in turn, results in slower startup times and increased overhead in CPU and memory usage. Containers, on the other hand, benefit from sharing the host OS, allowing for faster deployment and significantly reduced overhead. This makes containers particularly appealing for environments that prioritize rapid scaling, such as cloud-native applications and microservices architectures.

Despite their numerous advantages, the lightweight nature of containers does present certain trade-offs. A primary concern arises from the shared kernel architecture, which results in less isolation compared to virtual machines (VMs). This raises significant concerns regarding security and fault tolerance. Due to the fact that containers share the host operating system's kernel, all containers running on the system can be compromised by a breach in the host OS or container engine . Opposingly, VMs use independent kernels, which provide higher isolation. A breach in one VM does not inherently impact other VMs running on the same physical host [62].

Another key difference between containers and VMs lies in their orchestration and management. Containers are commonly deployed with platforms like Kubernetes, which automatize the deployment, scaling, and management of containerized applications across multiple machines. On the other hand, VM orchestration, while possible with tools such as OpenStack or VMware vSphere, is generally more complex because it requires handling entire operating system instances [27]. As a result, containers tend to be the preferred choice in DevOps environments that prioritize agility, continuous integration, and rapid deployment.

In conclusion, the trade-offs between containers and VMs are evident. Containers are more suited for environments that demand fast, resource-efficient, and scalable deployment, particularly in the realm of cloud-native or microservices architectures. On the other hand, VMs offer stronger

isolation and security, making them more indicated for traditional monolithic applications or scenarios where workload isolation is a critical requirement. For infrastructure architects, recognizing these differences is essential for optimizing system architecture according to the specific demands of their applications.

2.2.3 Container Images

Container images are essential to the practical implementation of microservices architecture, especially in ecosystems that leverage containerization technologies like Docker. At their core, container images are self-sufficient packages that encompass all the necessary elements to run an application: the code itself, runtime environment, libraries, environment variables, and essential system tools. A key distinction from traditional virtual machines is that container images virtualize only the application layer, rather than the entire operating system, resulting in significant gains in efficiency and resource utilization. This lightweight, focused approach is what makes container images particularly advantageous for microservices, where scalability, isolation, and streamlined deployment are critical [68].

Building a container image requires bundling the application and its dependencies into a single image file. Dockerfiles play a crucial role in this process, serving as a set of instructions that outline how the image should be constructed. These scripts typically specify the base image, commands to install dependencies, and configurations for executing the application. The base image provides the initial layer of the container, which can be tailored through additional layers defined in the Dockerfile. This layered architecture not only supports modularity but also enhances efficiency in both storage and network transmission, as only changes between layers need to be updated, improving overall performance [28].

Once constructed, container images are uploaded to registries like Docker Hub or Google Container Registry. These registries act as centralized repositories from which developers can pull images or push newly built ones. During the deployment stage, the relevant image is pulled and instantiated as a running container. In a microservices architecture, each service is packaged within its own container image, ensuring the isolation and modularity needed to manage services independently. This independence is key for scaling, updating, and deploying individual services without affecting others—a cornerstone of microservices' appeal [15].

A further advantage of container images is their inherent reproducibility. Since container images encapsulate the full runtime environment, they ensure that the application behaves consistently across different environments—whether it's development, testing, or production. This principle of "build once, run anywhere" helps mitigate the discrepancies that often arise between environments in traditional monolithic architectures. Additionally, the isolation provided by containers means that services can operate without conflict, even if they require different versions of the same libraries or runtimes. This isolation is fundamental to maintaining the stability and scalability of microservices systems.

2.2.4 Benefits of Containerization

In modern software engineering, containerization has emerged as a foundational element, particularly in the context of microservices architecture, due to its transformative impact on application

portability across a wide range of environments. By encapsulating an application along with all its dependencies in a lightweight container, developers ensure consistent performance, irrespective of the underlying infrastructure. This abstraction from the host environment allows seamless deployment across on-premises systems, private or public clouds, and hybrid environments without necessitating modifications to the application or its configuration [53].

A key advantage of containerization is the uniform runtime environment it provides. Containers bundle the application code, along with all necessary libraries and dependencies, into a singular, portable unit that can be deployed in any environment without deviation in performance. This contrasts with traditional deployment models, where variances in library versions or system settings often introduce bugs or unexpected behavior when transitioning from development to production environments. Containers mitigate these risks by ensuring that the application operates consistently, regardless of the host system's specific configuration. This operational reliability makes containers an integral component of modern CI/CD pipelines, streamlining the software delivery process.

When integrated with a microservices architecture, the portability benefits of containerization are further magnified. Microservices enable the decomposition of applications into smaller, independently deployable services, each of which can be scaled and managed autonomously. Containerizing these services not only ensures their portability but also simplifies their migration between environments with minimal reconfiguration. Each microservice is packaged with its unique dependencies, guaranteeing uniform behavior across diverse platforms. This modularity and consistency are particularly advantageous in cloud-native environments, where independent scaling of services is critical to managing dynamic workloads efficiently.

Cloud environments, in particular, leverage containerized microservices to facilitate seamless migration and scalability. Containers, being lightweight, can be rapidly instantiated, terminated, and transferred between environments, ensuring that organizations can flexibly shift workloads between cloud providers or between cloud and on-premises systems. This flexibility significantly reduces vendor lock-in and enhances operational efficiency. Additionally, container orchestration platforms like Kubernetes further streamline the management of containerized applications, automating their deployment, scaling, and maintenance across different infrastructures, thereby maximizing portability and resource optimization.

One of the primary advantages of containerization lies in its remarkable resource efficiency and utilization. By leveraging containers, organizations can achieve much higher application deployment densities on the same physical infrastructure. This is largely because containers, being inherently lightweight compared to traditional virtual machines (VMs), enable a greater packing density of applications. This ultimately allows for more efficient use of the available hardware resources, significantly improving overall resource utilization [56].

Additionally, containers support dynamic resource allocation, which means they can adjust resource consumption based on workload fluctuations. This adaptability ensures optimal resource usage, contributing not only to enhanced efficiency but also to notable cost reductions, particularly in cloud computing environments where resource consumption directly affects billing.

Furthermore, when combined with Continuous Integration and Continuous Deployment (CI/CD) practices, containerization provides considerable benefits for deploying applications, particularly in microservices-based architectures. Microservices, which rely on modular services that operate independently, benefit greatly from the encapsulation offered by containers. This approach ensures

that each service can be deployed in a lightweight, portable container, streamlining the deployment process. The uniformity provided by containerization is crucial, as it guarantees that applications run consistently across various stages, from development to production. By addressing issues like configuration drift and dependency management discrepancies, containerization enhances the stability and reliability of deployments, making it an essential tool in modern software engineering practices.

In the world of CI/CD, containerization is a key factor in streamlining and accelerating the integration and deployment of microservices. Using a CI/CD pipeline, code changes are continuously tested, integrated, and deployed. This allows to have much faster release cycles. Containerization allows these pipelines to be scaled efficiently and on-demand, as containers can be orchestrated efficiently using tools like Kubernetes.

Within CI/CD frameworks, containers offer critical isolation, enabling parallel development and testing. Each microservice can be developed and deployed independently, without impacting others. This independence boosts scalability and flexibility in the software development process. The modularity it provides allows for more frequent updates—each service can be updated, tested, and redeployed separately without disrupting the whole system. By automating these processes, CI/CD pipelines combined with containerization permit continous innovation and upgrades without repercussions on system stability.

Integrated into CI/CD workflows, containerization simplifies deployment, improves scalability, and strengthens system security. Containers ensure consistency across environments isolating microservices, with positive effects on the deployment process. The automation capabilities of CI/CD make software delivery faster and more reliable, and together with containerization, they provide the flexibility and resilience required to effectively manage complex distributed systems.

2.2.5 Popular Containerization Tools

Docker has become synonymous with containerization, forming the standard implementation for modern container-based architectures. As a containerization platform, Docker allows developers to bundle applications along with their dependencies into compact, portable containers that can run consistently in different computing environments. Docker's introduction marked the starting point of the transition from traditional virtual machines to containers, offering a more efficient method to isolate applications without the heavy requirements and constraints associated with virtual machines. Unlike VMs, which require entire guest operating systems, Docker containers share the host operating system's kernel while still keeping each container isolated. This shift significantly reduced the computational resources needed for deployment, enabling applications to scale more efficiently, an essential feature for the rise of microservice architectures.

Continuous integration and continuous deployment (CI/CD) pipelines can largely benefit from Docker's containerization model. The integration of Docker with CI/CD tools like Jenkins and Kubernetes ensures that software updates can be delivered with minimal downtime, with positive effects on development processes [41].

One of Docker's more appreciated features is its ecosystem. Docker Hub represents a repository for pre-built container images. This accelerates application development and deployment by enabling developers to reuse existing images or easily share their own. Docker Compose, a key component of this ecosystem, manages the orchestration of multi-container applications. It allows

developers to define and manage environments consisting of multiple, interconnected services [1], a critical advantage in microservices architecture where the seamless coordination of various services is paramount.

The growing adoption of Docker can largely be attributed to its seamless integration with orchestration platforms like Kubernetes, which facilitates the deployment, scaling, and management of containerized applications across distributed machine clusters. Docker offers its own orchestration solution, Docker Swarm, that presents a direct and straightforward approach, very appreciated in the industry, ideal for smaller or less intricate environments [17]. However, for large-scale deployments, Kubernetes represents the industry standard. Utilizing Docker containers as core components, it excels in advanced scheduling, scaling and self-healing capabilities, making it more suitable for complex workloads.

Another fundamental aspect of Docker's architecture is security. Docker containers are isolated from each other and from the underlying host system, minimizing the risks related to security breaches. Docker Security Scanning provides tools to inspect container images for potential vulnerabilities, a critical feature in enterprise environments. Additionally, Docker's integration with secrets management ensures that sensitive data (API keys or credentials) can be securely transmitted to containers without being exposed in code repositories or logs.

Docker's lightweight container model, together with its expansive ecosystem, robust security measures, and orchestration integrations, reprents itself the contemporary view on containerization.

2.3 Introduction to Microservices

2.3.1 The evolution of systems' architecture: from monolithic architecture to microservices

Over the last few decades, software architecture has undergone a profound architectural shift to answer the industry requests for scalability, ease of maintenance and faster deployment cycles. Initially, monolithic architectures were the prevailing model. These systems were characterized by a single, cohesive codebase: all components were tightly interconnected and deployed together. This approach offered simplicity, particularly in the initial stages of development and deployment. However, as systems'complexity increased, the limitations of monolithic systems became evident, in terms of flexibility, scaling possibilities and in managing sprawling codebases.

In response to these requests, the software industry gradually moved towards microservices architecture. This newer approach breaks down applications into smaller, autonomous services that communicate through well-defined interfaces. Decoupling individual components, is now possible to develop, deploy, and scale services independently.

Monolithic architecture has represented for a long time the foundation of software development,with its unified, self-contained structure. Within this model, the user interface, business logic, and data access layers are bundled into a single deployable unit. For smaller projects or teams with limited resources, this streamlined approach simplifies the development process, making testing and deployment more straightforward.

With the evolution in technologies and the increasing in dimensions for systems, monolithic architecture has revealed several drawbacks. A major concern is its lack of maintainability. Since monolithic systems are tightly coupled, changes made in one section of the codebase can have

dangerous consequences in other components, creating a fragile environment that is difficult to manage [19]. This interdependecy poses serious challenges in the introduction of new features or bug ficing activities without potentially affecting the entire system, leading to longer development cycles and an increased likelihood of errors.

Scalability represents another critical concern. Monolithic applications can be scaled vertically using more powerful hardware, with physical and economic limitations related to this approach. Horizontal scaling (the distribution the load across multiple servers) is inefficient in monolithic systems because it implies the replication of the entire application, rather than scaling only the components that need more resources. This inefficiency leads to resource usage inefficiency and increased operational costs. Deployment bottlenecks represent another critical aspect about monolithic architectures. Since all components are packaged together, any update necessitates redeploying the entire application. This process increases downtime and the potential for deployment-related failures. The rigidity of monolithic systems also impedes technological diversity. Adopting new technologies or languages for specific components is challenging because the entire application must remain consistent [24]. This constraint limits innovation and the ability to optimize parts of the system with the most suitable tools. Security considerations also are amplified in monolithic applications. A vulnerability in one component can compromise the entire system due to the lack of isolation between modules. Implementing robust security measures is more complex, as protections must be applied holistically rather than tailored to individual components.

To overcome the monolithic architecture's weaks, Service-Oriented Architecture (SOA) was theorized and introducted as the new system design standard in the early 2000s. Service-Oriented Architecture represents a software design paradigm where services are the fundamental building blocks that communicate over a network via defined protocols and standards. SOA enables the development of distributed systems where different services represent specific business functions. They are loosely coupled but interoperable. The core idea of SOA is the encapsulation of functionalities within services, each service being a self-contained unit that performs a specific task and can be accessed independently. This architectural approach is the standard schema for modern enterprise systems, where businesses require flexibility, scalability, and agility to adapt to evolving market demands [65].

A key characteristic of SOA is the adoption of loose coupling between services: services can be developed and deployed independently of one another. This allows enhanced flexibility in system maintenance and upgrades: individual services can be modified or replaced without affecting the entire system. Furthermore, SOA encourages standardized coomunication languages through the use of open protocols like SOAP (Simple Object Access Protocol) and REST (Representational State Transfer), to allow communication between heterogeneous systems. These standards become relevant in environments where different systems or platforms need to interact seamlessly.

Service-Oriented Architecture (SOA) is commonly implemented through web services, though its application isn't limited to them. Web services allow services to communicate across different platforms, which makes SOA a strong fit for both cloud computing and large-scale enterprise systems.

From a business standpoint, SOA offers a framework that aligns IT services with business processes-driven enterprise. In this setup, services are more than just technical tools. They are considered business assets that can be orchestrated to deliver strategic value. One of the standout

benefits of SOA is the ability to assemble individual services into higher-level business processes, a concept known as service orchestration. This directly links technology with business objectives, enabling organizations to adapt swiftly to changes in the market.

SOA is inherently different from traditional monolithic software systems, adopting a more flexible, reusable, and interoperable service-based architecture. The advantages of this approach are numerous, including greater system flexibility, lower development costs, and better alignment between IT and business needs. However, successful SOA implementation requires careful planning and strong governance to ensure that services remain reusable, loosely coupled, and adhere to industry standards [29].

Microservices architecture (MSA) is based upon SOA's core ideas of modularity and service autonomy, with the impovements given offering more granular services that can be independently deployed and scaled [31]. Unlike SOA, which relies on a central Enterprise Service Bus (ESB) for orchestration and communication, microservices adopt a decentralized model where services communicate directly, often using lightweight protocols like HTTP or messaging queues. This approach reduces bottlenecks and risks of having a single point of failure, common issues in traditional SOA setups that rely on a centralized ESB.

The transition from Service-Oriented Architecture (SOA) to microservices represents a major evolution step in system design, particularly on how services are structured, deployed, and managed. Microservices architecture emphasizes modularity: each service is tailored to a specific business function, operating independently from others. This independence facilitates continuous deployment, enabling more frequent updates without necessitating changes across the entire system [71]. This stands in contrast to SOA, where services tend to be more monolithic and often carry interdependencies that can complicate the deployment and scaling processes. Consequently, microservices present enhanced agility and responsiveness, particularly valuable in cloud-native and containerized environments where rapid iteration is crucial.

Nevertheless, while microservices alleviate many of the scalability and flexibility constraints present in SOA, they also introduce their own set of complexities. Chief among these are challenges surrounding service discovery, inter-service communication, and maintaining data consistency. In a distributed microservices architecture, ensuring reliable communication between services requires sophisticated service discovery mechanisms that can dynamically manage and locate services as they are deployed or scaled [58]. Moreover, the decentralized nature of microservices, where each service may maintain its own database, introduces data consistency issues that necessitate advanced patterns such as event sourcing and eventual consistency, adding to the overall system complexity.

Despite these challenges, both SOA and microservices follow service-oriented principles. However, microservices offer a more modern approach to system architecture, addressing many of the limitations of traditional SOA, particularly around scalability, deployment flexibility, and system resilience. The shift towards microservices is part of a more general industry movement towards distributed and decentralized systems, which prioritize agility, fault tolerance, and operational efficiency.

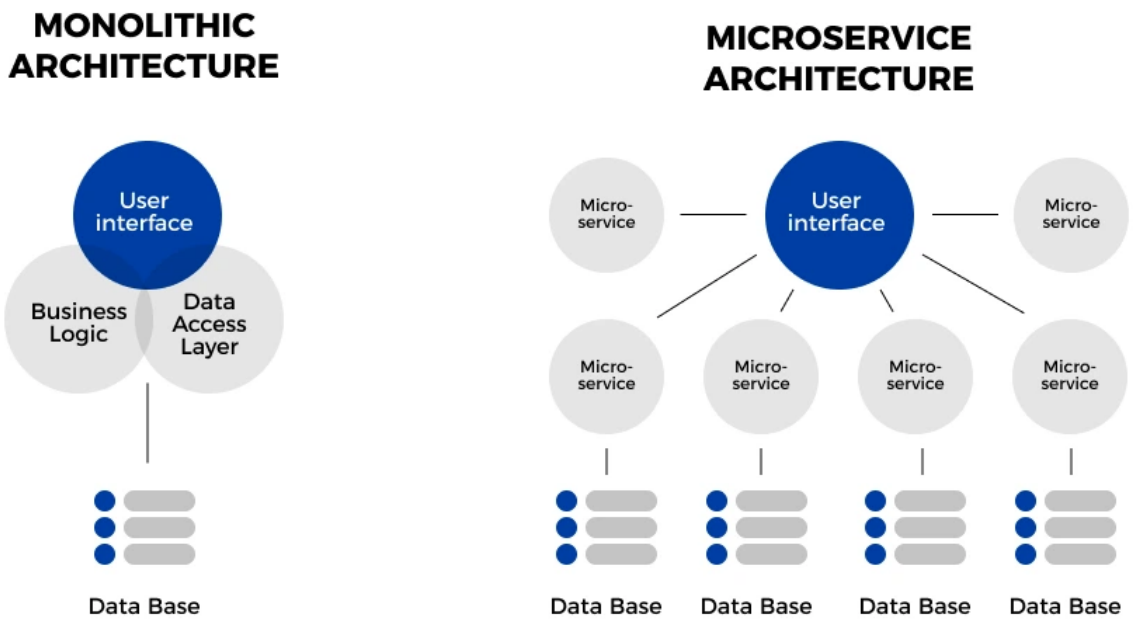

Figure 2.1: Comparison between Monolithic Architecture and Microservices Architecture

2.3.2 Key Characteristics of Microservices

Service independence and loose coupling

Service independence and loose coupling are essential principles in microservices architecture, and they greatly contribute to improving both scalability and maintainability in software systems. In traditional monolithic architectures, different parts of an application are strictly interconnected and interdependent, creating complex dependencies that make it easier for issues to spread across the system. Microservices focus on small, independently deployable services that communicate via network protocols. These services are designed with a specific business function in mind, promoting high cohesion internally while ensuring minimal dependency—loose coupling—between them.

Loose coupling is key in ensuring that changes to one service don't ripple across the system, modifying the behaviour and functionality of other services. This autonomy is achieved by encapsulating each service's business logic and allowing them to communicate only through defined interfaces. This structure builds resilience: even if a service fails or needs updates, the system can continue running smoothly, minimizing downtime and enhancing overall robustness.

Service independence also bolsters fault tolerance. Since each service operates in isolation, failures are easier to manage and contain. This makes microservices particularly well-suited for applications requiring high availability, as issues in one service are less likely to affect others.

Another significant advantage of loose coupling is the flexibility it provides in choosing technologies. Teams can select the best tools and programming languages for each service without being tied to decisions made for other parts of the system. This freedom encourages scalability and innovation, enabling teams to adopt new technologies or optimize services without causing widespread disruption.

Decentralized data management strategies

In a microservices architecture, decentralized data management strategies are fundamental to reach the agility, scalability, and fault tolerance requirements of distributed systems. Unlike traditional monolithic architectures, where data is centrally managed with all components relying on a single database, microservices adopt a decentralized approach. Each service maintains its own data, resulting in enhanced autonomy and reduced dependencies. This separation not only mitigates the risk of single points of failure but also significantly improves system performance and scalability [18]. However, while decentralization offers these advantages, it introduces its own set of challenges, particularly in relation to data consistency, governance, and communication between services [30].

A core advantage of decentralized data management is that it follows perfectly the autonomy principle in microservices. Each service has the freedom to choose the data storage technology that best meets its specific requirements, a relational database, a NoSQL solution, or an in-memory store. This flexibility encourages technological heterogeneity, promoting service-level optimization. Additionally, this model supports horizontal scalability, allowing individual services to scale independently, without affecting other services in the system. However, this autonomy adds complexity, particularly in managing distributed transactions and ensuring consistency when multiple services need to coordinate updates across their respective data stores.

Event-driven architectures and patterns like CQRS (Command Query Responsibility Segregation) are usually emplyed to deal with these challenges. In event-driven systems, services publish events to signal changes in their data, enabling other services to react without creating tight couplings. This decoupling enhances both scalability and resilience, as services can continue to operate asynchronously, meaning that the failure of one service does not necessarily cascade to the entire system [11]. Despite these benefits, achieving strong data consistency remains a challenge. Systems typically adopt eventual consistency models, where updates propagate gradually throughout the system. Over time, all services will converge to the same state, but this does not occur instantaneously, which necessitates careful handling of consistency expectations across the system.

Another crucial aspect of decentralized data management is data governance. In a microservices architecture, each service owns its data, but this autonomy makes enforcing global data governance policies more complex. It is important to establish frameworks for security, access control, and compliance across all microservices. Data partitioning strategies, such as sharding, are often implemented to optimize performance. By partitioning data across different nodes based on specific keys (e.g., user ID), microservices can efficiently manage large datasets without centralized bottlenecks [61]. However, sharding introduces the complexity of ensuring distributed data is managed consistently and reliably. Cross-shard transactions, where operations span multiple partitions, require coordination mechanisms as two-phase commit or distributed consensus algorithms, increasing the complexity of the system.

Achieving a balance between decentralization and effective coordination remains a significant challenge. Microservices architectures rely on message brokers or service meshes to handle inter-service communication and coordination. These mechanisms provide the much needed resilience, load balancing, and failover capabilities but also introduce latency and potential points of contention if not properly designed.

Scalability and Elasticity

Microservices architecture naturally promotes scalability by breaking down an application into smaller, loosely connected services that can be independently scaled across multiple nodes. Each of these services operates in its own container, with orchestration tools like Kubernetes that handle resource allocation based on current demand. This approach increases fault tolerance for the syetm and helps lowering latency, resulting in quicker response times and a more fluid user experience.

Elasticity plays a crucial role by dynamically adjusting resource allocation, making sure you're not over-provisioning or under-provisioning. In cloud setups, platforms like Amazon Web Services (AWS) use auto-scaling features to ensure microservices get the right amount of resources, based on factors like CPU usage, memory needs, and the volume of incoming requests [2]. Elastic Load Balancing (ELB) complements this by distributing traffic across multiple instances, preventing any single instance from being overloaded. Thanks to the combination of elasticity and horizontal scaling, systems can handle unpredictable traffic surges without sacrificing performance or uptime.

One of the key benefits of elasticity in a microservices architecture is cost efficiency. In traditional monolithic systems, scaling the entire application would be both resource-intensive and costly, as all components would need to scale simultaneously, even if only a few services experienced increased demand. In contrast, microservices architectures allow organizations to scale specific services independently, optimizing resource use and minimizing operational expenses by avoiding unnecessary scaling of low-demand components.

However, the combination of scalability and elasticity in microservices introduces challenges, particularly in maintaining efficient communication between services as the system grows. When scaling horizontally, technologies such as Apache Kafka and Pulsar are commonly used to manage high-throughput messaging between services, ensuring scalable and reliable communication across distributed systems. Additionally, managing state across multiple instances of a microservice presents another significant challenge. Stateless services, which do not retain data between requests, are easier to scale since they don't require synchronization. Conversely, stateful services, which must preserve data such as user sessions or transaction logs, require more sophisticated approaches like distributed caching or database partitioning to maintain data consistency and availability during scaling operations.

Continuous Integration and Continous Deployment Practices

Continuous integration (CI) and continuous deployment (CD) represent a critical component in the development and maintenance of microservices architectures. These practices are designed to simplify and speed up the software delivery process by automating testing, integration, and deployment. In a microservices environment, where each service functions independently, CI/CD pipelines play a vital role in ensuring the overall system's stability and scalability.

The primary advantage of CI in microservices is the ability to integrate code changes into a shared repository, with each service being properly tested in an isolated environment and in conjunction with other services. Tools like Jenkins and GitLab CI are often employed for this purpose, offering automated build and test processes that reduce manual intervention and minimize the risk of integration issues. These pipelines are particularly beneficial in microservices environments where different services may be developed by separate teams and deployed on varying timelines, necessi-

tating a high degree of coordination and integration. The decoupling of services allows CI processes to focus on small, incremental changes, reducing the scope of potential errors during integration.

In addition to CI, continuous deployment plays a significant role in ensuring that microservices can be rapidly updated and scaled. CD automates the process of deploying services to production environments after passing predefined tests, ensuring that any code that meets quality standards is automatically released without human intervention. This practice is critical in microservices architectures due to the distributed nature of services, where maintaining consistency and functionality across multiple components is challenging. By using CD tools, organizations can automate container orchestration and ensure that microservices are deployed consistently across environments [64].

One of the main advantages of the combination of Continuous Integration (CI) and Continuous Deployment (CD) in microservices is the ability to support rapid development cycles. Microservices deployment practices focus on incremental progress and frequent releases. CI/CD pipelines make it easier to implement changes quickly, allowing faster responses to user feedback and shifting business needs. This kind of agility is vital in large-scale systems where services need constant updates to maintain performance, scalability, and reliability.

However, implementing CI/CD in a microservices setup poses various challenges. A key issue is managing dependencies between different services. Since microservices are generally loosely coupled, a change in one service can impact others. Managing dependencies effectively is crucial to avoid deployment-related issues, with advanced CI/CD pipelines playing and important role in coordinating testing across multiple services. Techniques like canary deployments, where new service versions are rolled out gradually to a small subset of users, help reduce the risks of deploying updates across multiple microservices at once. Another challenge is scaling the CI/CD infrastructure as the number of services grows, adding complexity to pipeline management and maintenance.

2.3.3 Communication between Microservices

In a microservices architecture, seamless communication between services is necessary to preserve the system's scalability and independence. The choice of communication protocols adopted (REST, gRPC, AMQP, and WebSockets) directly influences key system attributes like latency, throughput, and ease of integration. Each of these protocols comes with advantages and trade-offs, with the right choice depending on the requirements of the specific system. Furthermore, the communication models used, whether synchronous or asynchronous, impact factors like service coupling, resilience, and overall complexity.

Commonly used protocols for inter-service communication include REST (Representational State Transfer) and gRPC (gRPC Remote Procedure Call), with newer options like AMQP (Advanced Message Queuing Protocol) and WebSockets also gaining popularity.

REST, built on HTTP, is widely used due to its simplicity and cross-platform integration capabilities. It uses stateless interactions and HTTP methods. REST's human-readable format, typically JSON (JavaScript Object Notation), further enhances its usability. However, research has shown that when microservices use REST, they tend to experience higher response times and diminished performance under heavy loads compared to more efficient binary protocols like gRPC.

gRPC, on the other hand, is designed for high-performance, low-latency communication between services. It utilizes HTTP/2 as its underlying transport protocol and supports bidirectional

streaming, multiplexing, and full-duplex communication, making it the perfect choice for scenarios requiring real-time data transmission or large-scale distributed systems. Unlike REST, which uses text-based formats, gRPC employs Protocol Buffers (protobuf), a binary serialization format more efficient in terms of both size and processing speed. Studies have shown that gRPC can significantly reduce the bandwidth required for inter-service communication while offering lower latency compared to REST. Additionally, gRPC provides built-in support for defining services using interface definition language (IDL), which promotes strong typing and helps in the generation of client and server code in various programming languages, further enhancing productivity in polyglot environments. Despite these advantages, gRPC's binary nature makes it harder to debug compared to REST, and it can require more effort to integrate with services that are not natively built to handle Protocol Buffers.

Another protocol used in microservices communication is AMQP, a messaging protocol often used in event-driven architectures. AMQP allows services to communicate asynchronously via message brokers such as RabbitMQ. This approach decouples the communication flow, making it resilient to service failures or delays by enabling message queuing. Asynchronous communication is especially beneficial in systems where services may experience variable loads, as it prevents services from being blocked while waiting for responses. AMQP is particularly effective in maintaining system availability and performance in the face of service downtime, as messages can be persisted and reprocessed later. However, AMQP introduces additional complexity in managing message brokers, and the overhead of message serialization and deserialization may introduce latency compared to direct protocols like REST or gRPC.

The choice of communication protocol in microservices is a balance between ease of development, performance, and specific system requirements. While REST remains the default choice for many due to its simplicity and flexibility, protocols like gRPC, AMQP, and WebSockets offer compelling alternatives in environments where performance and real-time capabilities are crucial.

To design microservices architectures, two communication models can be chosen: synchronous and asynchronous communication. Each model presents distinct characteristics, benefits, and drawbacks, making them suitable for different operational contexts in distributed systems.

Synchronous communication involves real-time, direct interaction between services. In this model, one service sends a request and waits for a response before proceeding. It is typically implemented using protocols like HTTP/REST or gRPC. The primary advantage of synchronous communication is its simplicity and ease of tracing interactions between services, which can be beneficial when debugging or monitoring service flows. However, this real-time interaction creates a tight coupling between services, which can lead to increased system latency. If one service becomes slow or unresponsive, it can propagate delays or even failures across the entire system. This tight coupling also impacts system resilience, especially in large-scale distributed systems, where performance bottlenecks become more likely [60].

Asynchronous communication decouples the interaction between services, allowing the sending service to continue with other operations without waiting for a response. This model is commonly implemented using message brokers like Kafka or RabbitMQ, which queue messages until the recipient service is ready to process them. Asynchronous communication, given its princicples, can improve system scalability and resilience. By decoupling services, this model reduces inter-service dependencies, preventing system-wide delays or failures when a service is slow or unavailable. It

also allows services to handle traffic surges more efficiently, enabling them to process requests in bulk or at a time when system resources are more readily available.

In the realm of distributed systems, asynchronous communication introduces significant complexity, particularly around message handling and ensuring eventual consistency. Since services operate independently without real-time interaction, developers must design and implement robust mechanisms to manage message retries, acknowledgments, and potential failures. One key challenge is maintaining data consistency across distributed services, as operations might not be completed in a predefined sequence. To address this, developers frequently apply patterns like eventual consistency, which ensures that systems will converge to a consistent state over time rather than requiring immediate synchronization.

The decision between synchronous and asynchronous communication methods is heavily influenced by the specific requirements of the application. In scenarios where real-time interaction and immediate feedback are essential, such as user-facing components, synchronous communication proves to be a better fit. Conversely, for systems that must manage high throughput or continue functioning during service disruptions, asynchronous communication tends to provide superior resilience and scalability. Many modern microservices architectures leverage a hybrid communication model, utilizing synchronous methods for time-sensitive operations and asynchronous methods for more flexible, resilient processes, depending on the nature of each interaction.

2.4 Integration of Containers and Microservices

The combination of containerization and microservices architecture has profundly changed software development and deployment. Containers offer a streamlined, portable, and uniform environment for microservices, vastly increasing scalability, operational efficiency, and overall management.

Microservices architecture breaks down systems into small, autonomous services that interact through clearly defined interfaces. Containers bundle each microservice with its dependencies, ensuring that it operates consistently across any environment. This encapsulation solves the common issue of applications working on one developer's machine but failing in other environments.

Because containers isolate microservices from one another, multiple services can run on a single host without conflicts. Each container s its own file system, network settings, and process space, preventing dependency clashes and enhancing security. Additionally, containers are more efficient than traditional virtual machines, as they share the host system's kernel, which significantly reduces resource overhead.

Managing numerous containerized microservices requires powerful orchestration tools. Platforms such as Kubernetes, Docker Swarm, and Apache Mesos automate many aspects of deployment, scaling, and management. They schedule container placement across clusters, handle service discovery, load balancing, and ensure that configurations stay aligned with the desired state.

Orchestration platforms also bring advanced features like rolling updates, enabling new versions of microservices to be deployed without any downtime. They continuously monitor the health of services, automatically restarting or replacing failed containers. This level of automation minimizes the need for manual intervention, reducing errors and increasing the reliability of applications.

Chapter 3

LLM

3.1 Introduction

Large Language Models (LLMs) are becoming a transformative force in the field of NLP and AI, wherein machines understand and generate human-like text with remarkable coherence and accuracy. The proliferation of LLMs has so far advanced not only academic research but has also innovatively altered industries such as healthcare, finance, and education through the automation of complex language tasks and presentation of sophisticated conversational agents.

This chapter provides a comprehensive exploration of LLMs, beginning with a general overview that introduces their fundamental principles and significance in modern technology. It delves into key concepts integral to LLMs, including tokenization, embeddings, attention mechanisms, and transfer learning. Understanding these concepts is essential for appreciating how LLMs process and generate language, and how they differ from traditional NLP techniques.

Subsequently, the chapter examines the main types of LLM architectures, highlighting their unique features and the advancements they bring to language modeling. This includes a discussion on transformer architectures, which have become the backbone of most state-of-the-art LLMs due to their ability to handle long-range dependencies and parallelize training processes efficiently.

The evolution section traces the historical development of language models, starting from early NLP techniques that relied heavily on handcrafted rules and statistical methods. It explores the introduction of word embeddings, which marked a significant shift towards distributed representations of words, and paved the way for the development of more complex models. The growth of LLMs is chronicled, illustrating how incremental innovations have led to exponential improvements in language understanding and generation capabilities.

A focused review of the OpenAI Generative Pre-trained Transformers-GPT series is later in the last part of this chapter. The GPT models set new benchmarks in language modeling by demonstrating unprecedented capabilities for machine translation, summarization, and conversational AI. The discussion of the architectural and functional details here clearly presents how they work. Equally appealing is a deep overview of the ChatGPT API: how its architecture and the use of LLMs to expose these unprecedented, powerful language capabilities through such a user-friendly interface are possible.

3.2 General Overview

3.2.1 Introduction to Large Language Models (LLMs)

Concept and Importance of Large Language Models

Large Language Models represent a quantum leap in the research into natural language processing; using vast datasets and neural networks of considerable complexity to understand, generate, and perform operations on human language. The typical foundation upon which LLMs are built is transformer architectures, a class of neural network that revolutionized NLP by parallelizing the training and greatly improving model scalability relative to the more standard recurrent neural networks or long short-term memory networks.

The ability of LLMs to generalize across a wide array of tasks without requiring task-specific training is their strong ability and defining characteristic. For instance, large language models generate text that is similar to human generated by learning statistical relationships and linguistic patterns between words, phrases, and sentences from extended corpora of text.

Unlike typical NLP models, which often require domain-specific datasets and perform poorly across different domains, LLMs leverage self-supervised pre-training on enormous amounts of data with only unlabeled text. This reduces the need for task-specific labeled data, allowing generalization to a wide range of NLP tasks. For example, GPT-3, part of the Generative Pre-trained Transformers family, has obtained exceptional few-shot learning results on various tasks using limited to no task-specific data [4]. Those are some very fundamental models that showcase the role of LLMs by being efficient in tasks such as translation, summarizing, and even creative writing.

Role of Large Language Models in AI

Speaking about Artificial Intelligence, the large language models nowadays started to find their places in the modern artificial intelligence systems. They come into play for NLU and NLG. Because of their being prone to understand human-like texts, they are very much crucial in handling tasks such as language translation, answering questions, and also chatbots. Another contribution that LLMs make in AI is to facilitate smooth and natural interaction between humans and computers.

LLMs play a significant role in AI because they are not only able to generate fluent text but also apply to a wide range of domains. For example, GPT-3 has been tried in code generation, diagnosis in medical treatments, and analysis of legal documents. Such wide applicability is underlain by the model architecture, which captures in language not only syntactic but also semantic and contextual relationships [51]. These advances enable the AI systems to make complex decision-making, develop better performance due to a deeper understanding of the context, and have improved performance in multi-modal systems where integration includes text as well as image data.

Structural Components of LLMs: Parameters, Layers, and Training Processes

To begin with the understanding of the capability of LLM, the following are the structural components that are necessary to include:

- Parameters, including weights and biases. These are the substantial parts in the performance of the model, while the number of parameters is considered a key measure for the capacity

of a model to store linguistic knowledge. For example, the 175 billion parameters of GPT-3 make it one of the largest models ever developed [4] and able to handle very complex tasks with a minimum of fine tuning.

- Layers, the hierarchical structure of transformer blocks, that stack to form deep networks, forms the backbone of LLMs [23]. Each block consists of a self-attention mechanism with feed-forward layers that give the model the ability to weigh the importance of a certain word or token with respect to all the others in a sequence. This architecture provides the LLMs with capabilities for handling long-range dependencies in language, which was a limitation with previous architectures, such as RNNs.

- Training processes, they usually consist of two steps:

  1. Pre-training: during this stage the model learns the structure of the language from large sets of unlabeled texts by means of techniques like masked language modeling-as seen in BERT [5] or through auto-regressive methods-as used in GPT models [4].
  2. Fine-tuning: on a smaller set of labeled data, the model specializes itself for a particular task.

  Pre-training is computationally intensive, usually requiring specialized hardware like GPUs or TPUs, while fine tuning is lightweight and task-specific [50].

This architecture indeed has shown the flexibility and power of LLMs for NLP applications and is opening paths for more advanced uses in the near future.

Initial Use Cases of LLMs in Natural Language Processing

The most relevant use cases will be represented below:

- Text completion and generation: These models generate coherent and contextually relevant text based on the given input prompt. This finds applications in content creation and automated report generation [4].

- Language translation: LLMs which are trained on multilingual datasets can perform the translation of languages with little to no task specific fine-tuning, which is comparable to traditional machine translation systems [5].

- Question answering: LLMs are being utilized in chatbots and virtual assistants to respond like humans to user's queries by synthesizing information from large data to come up with accurate answers [7].

3.2.2 Key Concepts in LLMs

In this section, key concepts of LLMs will be described and analysed in detail to gain a deeper understanding of its mechanisms.

Tokenization and Context Window Concepts

Tokenization is a very basic process consisting of simplifying text into small units, called tokens, which further becomes the input for LLMs. The token can be anything, from a single character to a full word or sub-word, relating to the approach towards tokenization. Its objective is to turn input text into a format that is workable and understandable by the model while conserving the meaning and structure of the original text.

The most usual tokenization method used in LLMs is the sub-word segmentation-based approach called *Byte Pair Encoding*. BPE merges the most frequent character sequences into a sub-word, hence highly efficient in handling out-of-vocabulary rare words by means of decomposing these into more frequent units. For instance, the word "unhappiness" can be tokenized as ["un", "happiness"], where the model learns how much each sub-word contributes toward the meaning of the whole word [4]. Due to its balance between word-level semantics capture and generalization across many languages, BPE tokenization has become widely adopted in models such as GPT-3, BERT, and T5 [5].

Context window basically defines the maximum number of tokens a model can process at one time. The size of the context window influences the long-range capability of the model to capture dependencies within a certain text. For instance, GPT-3 uses a context window of 2048 tokens, which enables it to process and respond to a large portion of input data in one go [4]. On the other hand, if the input is longer than the context window of the model, important information may get truncated, causing loss of context in longer documents. The larger context window is fitting for tasks that require the processing of longer inputs, such as document summarization or generating text in longer forms.

The size of the context window also ensures coherence in generated texts. If the context window size is too small, in some tasks the model's capacity to remember earlier parts when generating text is reduced. On the other hand, extending the context window requires more computational resources and memory, posing a trade-off between processing capability and efficiency [23]. Techniques in the handling of better contexts include memory-efficient transformers and continue to be under exploration in order to enhance the performance of the model on longer inputs.

Transfer Learning and Its Importance in Large Language Models

Transfer learning has become a cornerstone of modern NLP models, especially for LLMs. The process involves first training the model on a general dataset-a large one in size, what is called pre-training and then adapting it to the particular task with a smaller dataset of the task being modeled, which is called fine-tuning. In pre-training, the model is able to grasp the general features of language, syntax, and semantics. Fine-tuning allows the model to be specialized to meet the specific requirements the task requires.

The most important benefit of using transfer learning is that it drastically reduces the need for labeled training data. Conventional supervised learning methods rely on large annotated datasets to train models from scratch for each new task. Contrarily, LLMs can be pre-trained, through transfer learning, on massive corpora such as Common Crawl or Wikipedia, which have billions of words. For example, masked language modeling, as in BERT, masks some tokens in a sentence and asks the model to predict those tokens given the context of the other tokens. In the same way,

models such as GPT make use of an auto-regressive approach where the model predicts the next word given a sequence using previously seen words. These models have demonstrated how transfer learning allows quick adaptation across tasks from machine translation to sentiment analysis with very limited task-specific data [23][5].

Another advantage of transfer learning is that it decreases the training time and computational cost for new tasks. It is possible to fine-tune a pre-trained LLM to high performance in just a few epochs, rather than from scratch. This has enabled the rise of few-shot learning where models such as GPT-3 achieve good results on tasks where there are only a small number of task-specific examples [4]. It has made transfer learning the backbone of just about every state-of-the-art LLM, and in effect, has accelerated progress on many axes within natural language processing.

Role of Attention Mechanisms in LLM Performance

The attention mechanism is an integral part of the performance of LLMs, especially those that adopt the transformer architecture. Attention mechanisms, as first introduced in the seminal paper "Attention is All You Need", allow LLMs to weigh the importance of different tokens in a sequence based on their contextual relevance [23]. This capability enables transformers to capture long-range dependencies within text data, something previous architectures, such as RNNs and LSTMs, had difficulties doing due to their nature of processing inputs one after the other or in fixed-size windows.

Self-attention is one of the underlying key deep learning mechanisms for modern LLMs, including transformers. As such, self-attention will compute attention scores for each token relative to all others in the sequence, thus enabling the model to capture contextual relationships between words regardless of how far apart they are in the input [52]. Put another way, considering the sentence "The cat, which was hungry, chased the mouse," self-attention will allow the model to understand that "cat" and "chased" have a strong relation, even though "which was hungry" intervenes.

Especially, it has proved to be particularly effective in tasks requiring understanding context across long documents, such as document summarizing, translation, and question answering. The multi-headed attention mechanism extends this further in that it allows the model to attend to different aspects of the input simultaneously. Each different attention head is focused on a distinct part of the input, capturing everything from syntactic structure to semantic meaning. This multi-headed approach further enhances its generalization capability over various tasks [23].

Successes of different types with attention mechanisms within LLMs are due to their flexibility and scaling. Therefore, self-attention can handle dependencies along inputs of varying lengths and shows great potential in a wide range of NLP applications. It has given the possibility for models such as GPT-3 and BERT to outperform the previous architectures that existed in benchmarks over tasks related to text classification, generation, and machine translation.

Importance of Pre-training Data Size and Diversity in LLMs

Large size and diversity are two of the most important features in the pre-training data that determine the effectiveness of LLMs. For instance, GPT-3 was pre-trained on hundreds of billions of tokens from varied sources, including books, websites, and Wikipedia [4]. In fact, it is this large dataset that made it perform on a variety of tasks with optimal results. Diversity in the training data makes the model not biased much towards any particular domain, thus it can be used for

different applications.

While large, diverse datasets build better models, they also introduce a host of challenges, such as biases in the data. Large language models accidentally learn and express societal bias found in the training corpus. This, of course, is a fact with grave ethical implications when deploying these models in sensitive domains such as health care, law, and hiring. Therefore, a lot of researchers are developing ways to ameliorate the mentioned biases either by better cleaning of data or with appropriate post-hoc model adjustments so the model outputs are fair.

Another important dimension of diversity in data is that it enables the model to encode and process languages other than English. Multilingual LLMs such as mBERT and XLM-R are trained on corpora in many languages and can perform cross-lingual tasks like translation and language modeling. In such cases, both the size and the diversity of the pre-training data are determinants of meaningful relationships between the languages that can be learnt by the model.

3.2.3 Main Types of LLM Architectures

Large language models can be differentiated based on the underlying architectures, the two major ones are encoder-decoder and decoder-only models. While both use transformers at their center, their processing of input and generating output are done quite differently.

Encoder-Decoder Architectures

An encoder-decoder architecture takes in two components: an encoder that encapsulates input into some sort of fixed-size representation, and a decoder that generates an output sequence given this representation. Such models are best applied to tasks like sequence-to-sequence, where the goal is to map an input sequence (such as a sentence in one language) to an output sequence (such as the translation of that sentence).

The encoder first embeds the input tokens into a context-sensitive representation, encoding semantic meaning through a successive number of self-attention layers, while the decoder takes these representations and produces the target sequence, paying attention to both the encoded input and the previously generated tokens. Arguably, though, the biggest strength of encoder-decoder models is their capability to handle tasks that require deep understanding of the input-output relationship-such applications include machine translation and summarization [23].

A well-known example is BERT, which stands for Bidirectional Encoder Representations from Transformers. It processes input bidirectionally to understand context from both directions [5]. However, full encoder-decoder models like T5 (Text-to-Text Transfer Transformer) employ the use of both encoder and decoder for such tasks believed to be of a more complex nature, framing every NLP task as a text-to-text problem, making it highly versatile across tasks such as translation, summarizing, and question answering [52].

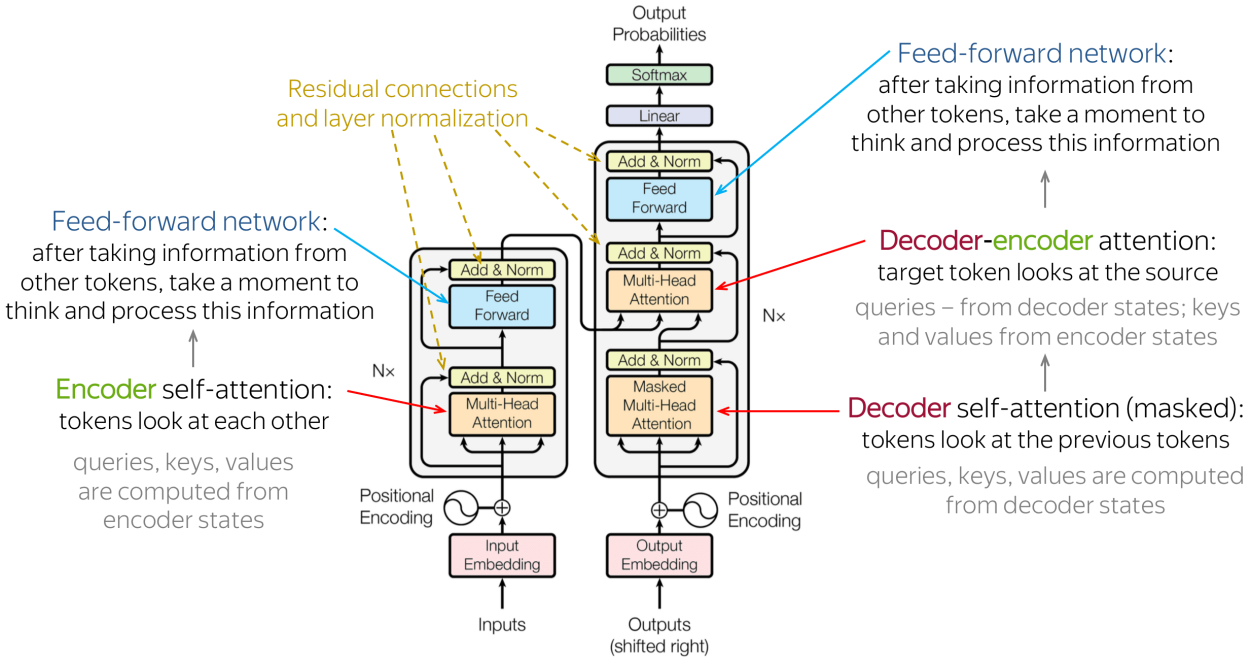

Figure 3.1: Encoder and Decoder architecture

Decoder-Only Models

On the other hand, decoder-only models involve a single transformer that auto-regressively generates outputs by predicting the next token in a sequence based on the previous tokens. In such models, text is processed unidirectional, it means that it generates text from left to right, hence pretty effective for tasks such as text generation, dialogue systems, and language modeling.

Another well-known decoder-only model is GPT-3, which stands for Generative Pre-trained Transformer; this model is very effective in generating coherent and contextually relevant text because it is continuously trained to predict the next word in the sequence based on the context given by the earlier words [4]. They achieved state-of-the-art results in few-shot and zero-shot learning, where, given a new task, they could perform well with little to no task-specific training but instead relied on general language patterns learned during pre-training.

Basic Variants related to Scalability and Efficiency

Scalability is one of the main focus points when comparing LLM architectures because the ability to scale with more extensive data and parameter sizes normally comes hand in hand with better performance of wide ranges of NLP tasks. However, scaling LLM models brings along several burdens such as increased computational costs, longer training times, and greater memory consumption.

This encoder-decoder architecture has been proven to scale effectively by several models, such as T5 and BART. These models are scaled up with an increase in the number of layers, attention heads, and model parameters. Model sizes range from small with 220 million parameters to T5-11B with 11 billion parameters, proving that T5 is effective across the board on different scales and tasks [51]. However, as the model grows, the computational complexity also increases due to the need to maintain separate encoder and decoder components, both of which must process the input and output sequences.

This scaling requires many computational resources, as both the encoder and decoder layers are performing complex self-attention. It is this bidirectional attention that really enables applications like machine translation, although it increases memory and slows down training times. Moreover, because of the duality maintained in this architecture, encoder-decoder models are always more expensive to scale compared to a decoder-only model.

Decoder-only models, such as GPT-3, generally tend to be more scalable and efficient because of their simple architecture. GPT-3 set another milestone in the scalability of LLMs, with 175 billion parameters, which showed that a decoder-only model can outperform the rest of the architectures by far in many tasks, using enormous amounts of data and computer power [37]. It is an auto-regressive model, which is very efficient for text generation, as it predicts one token at a time and avoids complicated bidirectional attention mechanisms.

This is also helped by ease of parallelization during training: since the model does not need to consider both an encoder and a decoder, training can more easily be distributed across multiple GPUs or TPUs, allowing for faster training times even as the model grows in size [63]. This is the immediate advantage that can be seen in training models like GPT-3 and their variants on supercomputers with thousands of GPUs fired simultaneously.

However, while decoder-only models scale more easily, they may be less effective for tasks that really require deep contextual understanding of both input and output, such as translation. Intrinsic in their unidirectional nature, decoder-only models generate text based on left-to-right progression and may thereby be more limited in tasks which will need to consider context from the entire sequence at once.

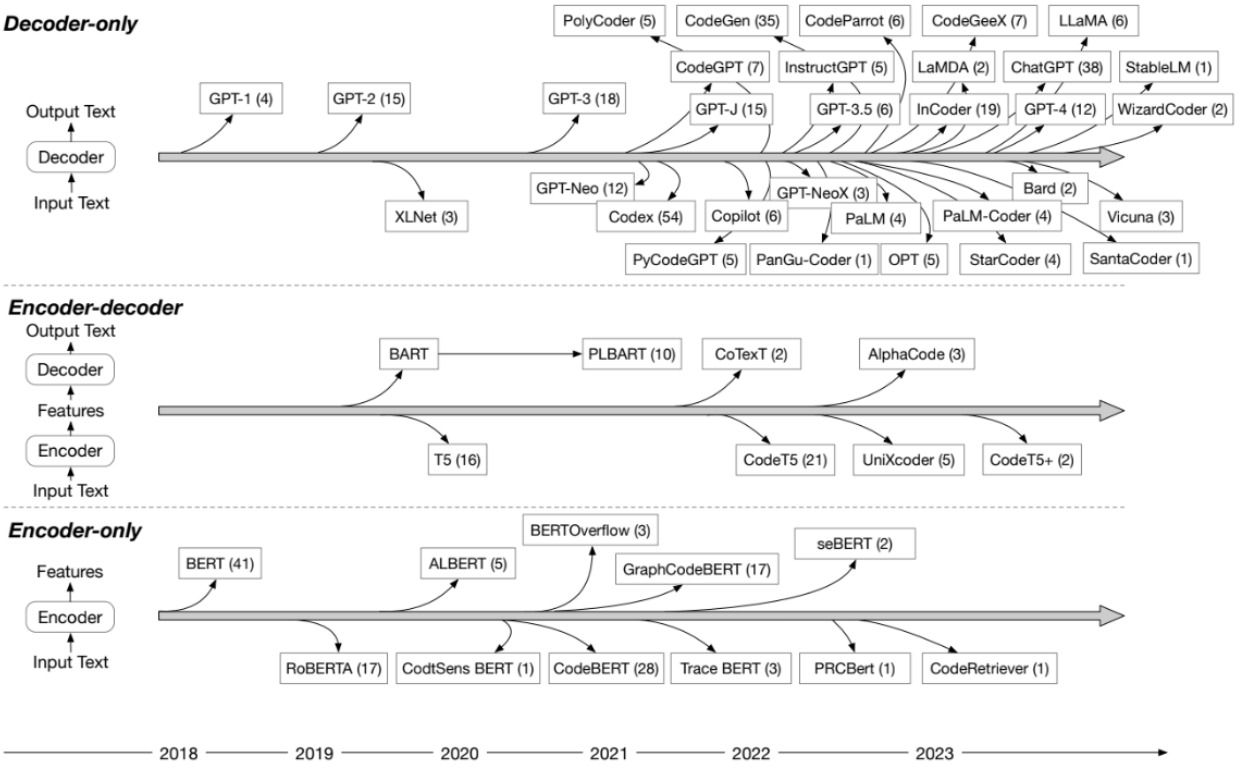

Figure 3.2: Tree Evolution of each type of Model

Efficiency and Performance Trade-offs

The efficiency and performance trade-offs will be closely related to the scalability of both encoder-decoder and decoder-only models. Encoder-decoder models are relatively more difficult to scale but usually perform better than decoder-only models on tasks requiring full contextual understanding. For instance, in machine translation, the encoder has a bidirectional nature that lets the model capture the input context fully before it proceeds to generate a translation. Therefore, this results in better performance [14].

On the other hand, decoder-only models perform well when generation and completion of text are to the fore. These are also remarkably efficient, thus easily scaling up to hundreds of billions of parameters, yielding state-of-the-art performance in text generation tasks such as creative writing, dialogue systems, and summarizing [4].

Overall, the decision on the applicability of an encoder-decoder or a decoder-only model comes from what is needed for the particular task at hand and from the resources one has for training. While encoder-decoder models are sufficient when full comprehension of the sequence is needed, decoder-only models provide an efficient and scalable solution for open-ended text generation tasks.

3.3 Evolution

3.3.1 Early NLP Techniques

The integration of machine learning (ML) techniques into NLP first represented a significant movement from rule-based systems to data-driven approaches. Early AI systems in NLP relied on handcrafted rules and symbolic methods, which required domain expertise and were limited in scalability. These rule-based models often suffered from issues of robustness and generalization across diverse domains and languages.

Machine learning has given a new paradigm by providing the facility to let the model learn from data instead of explicit programming. In particular, supervised learning methods, which include decision trees, support vector machines (SVMs), and logistic regression, were among the very first applications to several NLP tasks like text classification, sentiment analysis, and information retrieval. These models relied on feature engineering, where linguistic features such as word frequencies, part-of-speech tags, and syntactic structures are extracted manually from the text [43].

Statistical NLP Approaches: The Role of the N-Gram Model

Perhaps the earliest, yet most influential statistical approach to NLP, was the n-gram model. The N-gram models assign a probability to sequences of words. In other words, they are probabilistic language models in which the occurrence of a word is predicted given the preceding n -1 words. Although quite straightforward, this type of approach furnished a very effective way of modeling word sequences and has been seminal in work related to speech recognition, machine translation, and spelling correction.

N-gram models implement the Markov assumption when calculating the given word probability based on previous words in the stream, which is expressed by the assumption that from a limited-size window of preceding words-a typical size is one to three words-the probability of a word can be obtained. For instance, a bigram model (n=2) estimates the probability of a word based on the

immediately preceding word, while a trigram model (n=3) looks at the two preceding words. The general formulation of an n-gram model is expressed as:

$$P(w_1, w_2, \ldots, w_n) \approx \prod_{i=1}^{n} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) \tag{3.1}$$

Despite their simplicity, the weaknesses of the n-gram models can be outlined by the fact that they do not capture the meaning of sentences or texts. They are usually computationally very expensive for large n, as it keeps huge tables of probabilities for all possible combinations of words. Besides, n-gram models suffer from a data sparsity problem: even with large corpora, most of the word sequences may never appear in the training data, hence leading to zero probabilities for unseen sequences. To overcome this, smoothing techniques were introduced, such as Laplace Smoothing or Good-Turing discounting, in order to estimate the probabilities of the unseen n-grams [21].

However, the key limitation with the n-gram model is the inability to model long-range dependencies of language because of reliance on fixed-length context windows. For instance, the relationship between a subject and a verb separated by several words cannot be properly related using an n-gram model. This inability to model long-range sequences was one of the driving reasons for developing more sophisticated models such as neural networks and transformers.

Hidden Markov Models and Their Impact on Speech and Text Processing

HMMs represent one of the oldest statistical approaches applied to NLP. They first provided some of the major breakthroughs in speech recognition and text processing. Generally speaking, HMMs are probabilistic models representing sequences of observable events given a hidden process. For NLP, these may be words, phonemes, or part-of-speech for the visible part and syntactic or semantic categories for the latent states.

HMMs started to reach their high impact in speech recognition and part-of-speech tagging. Speech recognition is typically characterized by some sort of input, a sequence of acoustic signals, while trying to map those signals into a sequence of words. In the case of HMMs, speech is probabilistic modeled as a sequence of phonemes, where hidden states are representative of phonemes while the observable variables are the acoustic signals. Transition probabilities between the phoneme and the likelihood that a given phoneme generates an acoustic signal are learnt from large speech corpora [49].

The HMMs model in POS tagging signify the sequence of words where the hidden states are grammatical categories like nouns, verbs, adjectives, etc., and the observable events are the actual words. The model learns transition probabilities between the POS tags and the likelihood that a word is associated with a particular tag. This approach proved to be effective in tasks requiring the analysis of word order and grammatical structure.

While HMMs proved to be very useful for processing sequences, hence building a platform for probabilistic models in NLP, they still struggled with long-range dependencies in text and speech data due to the first-order Markov assumptions that constrained them to a small context between neighboring states. They substantially developed the area by formalizing the usage of probabilistic methods in NLP and were considered a standard approach until the emergence of more robust machine learning methods [21].

3.3.2 Introduction of Word Embeddings

Conceptual Foundations of Word Embeddings: From One-Hot Encoding to Dense Vectors

Word embeddings represent one of the biggest milestones in NLP, giving models capabilities to capture the meaning of words in a continuous vector space. Simple early methods of word representation, such as one-hot encoding, are quite ineffective at capturing relationships between words. In one-hot encoding, each word is a sparse vector with all values being zero except for one in the place corresponding to the word position in the vocabulary. So, in the case of 10,000 words in the vocabulary, every word would be a 10,000-dimensional vector where just one element is "1" and the rest are "0."

The disadvantages of one-hot encoding are two-fold: First, all the words are treated as independent entities with no scope for capturing the relationship between the words such as synonyms or similar contexts in which a word is used; Second, since the number of vectors is equal to the size of the vocabulary, the dimensions of the vectors grow with the size of the vocabulary, creating inefficiency both in computation and memory. Vectors resulting from one-hot encoding thus cannot be used for tasks needing semantic similarity understanding or relationships between words [3].

Word embeddings were the solution that mapped words into dense, low-dimensional vectors, where semantically similar words are located close to each other in the vector space. These embeddings learn from large corpora and encode both syntactic and semantic information. For example, in a good embedding space, vectors for words "king" and "queen" would be close to each other, and the vector difference between "king" and "man" would be similar to that between "queen" and "woman." Word embeddings can thus allow models to capture more context and meaning in language tasks like machine translation, question answering, and sentiment analysis [38].

Word2Vec: Continuous Bag-of-Words (CBOW) and Skip-gram Models

One of the most influential advancements in learning word embeddings came with the introduction of Word2Vec by Mikolov et al. in 2013 [39]. Word2Vec offered an efficient method for learning word embeddings using two primary model architectures:

- CBOW Model: Continuous Bag-of-Words model tries to project the algorithm for the current word based on its surrounding context words. More formally, given a set of context words surrounding a target word, the model attempts to predict the target word. Whenever the surrounding context provides some reliable clue about the target word, CBOW works pretty well, hence good for tasks where context words strongly predict the central word.

- Skip-gram Model: The Skip-gram model does the opposite; it predicts context words given a target word. Target words will be used to predict the words that surround them within the text. Skip-gram handles a big corpus in an effective way. It works well with sparse data since, given each target word, it focuses on predicting many context words. It turns out that this model is particularly powerful for capturing semantic relationships between words, as it learns to represent the target word in a way that reflects its relationship with multiple neighboring words.

Both CBOW and Skip-gram learn word vectors by maximizing the likelihood of correctly predicting context words. The key innovation of Word2Vec is the transformation of words into dense vectors that capture not only the identity of the words but also their relationships with other words. The efficiency of Word2Vec's training, using techniques like negative sampling and hierarchical softmax, allows it to scale to large corpora, enabling it to produce high-quality embeddings with limited computational resources [26]. Success in Word2Vec lies in the representation of words in a continuous space by keeping semantic similarities and linguistic regularities intact. The word embeddings learned from Word2Vec show a linear relationship; for example, the famous example of vector arithmetic, where:

$$vec(king) - vec(man) + vec(woman) = vec(queen) vec(king) - vec(man) + vec(woman) = vec(queen) \tag{3.2}$$

These kinds of vector arithmetics reveal that Word2Vec captures, in addition to the meaning of single words, higher-order features such as gender and royalty, that are encoded in vector differences.

From Word Embeddings to Sentence Embeddings

While Word2Vec and its variants are doing an excellent job at word embeddings themselves, this is where their limitation begins to manifest themselves in how they fail to encode longer elements of text such as sentences or paragraphs.

The representations of words are context-independent, meaning that a word has the same vector representation regardless of the sentence it occurs in. This leads to problems when the meaning of a word strongly depends on the context, such as homonyms or words with several meanings, for example, "bank" as in the bank, where one goes to deposit money, but also as a riverbank. To address this limitation, research in NLP moved toward sentence embeddings: the entirety of sentences or larger units of text were to be projected into vectors, carrying the general meaning of a sequence. A seminal model in this respect is Doc2Vec, which further developed Word2Vec by adding paragraph vectors representing not only the individual words but also the entire text document. Yet, Doc2Vec still retains some of the weaknesses of Word2Vec, like the lack of dynamic contextualization [32].

The next important embedding techniques came with the development of contextual word embeddings in models like ELMo and BERT.

Unlike the static embeddings resulting from Word2Vec, these models produce context dependent word representations, the embedding for a word would vary with the surrounding text. The word "bank" would be represented by different vectors when it has to do with finance or geography. This ability of generating dynamic embeddings allows BERT and its siblings to perform much better compared to older models on a wide range of NLP tasks like question answering, natural language inference, and sentiment analysis. Further research into sentence embeddings, as well as building upon models like BERT, has resulted in models such as Sentence-BERT (SBERT). The latter fine-tunes BERT in order to produce fixed-size sentence embeddings using Siamese and triplet networks. These sentence embeddings allow better performances on tasks like semantic textual similarity and information retrieval where the meaning of an entire sentence or document is more important compared to the representation of single words [55].

3.3.3 Growth of LLMs

Role of Computational Power in the Growth of LLMs

The more significant factor in the exponential rise of Large Language Models, or LLMs, has to do with computational power and the utilization of special hardware to train more complex models. Initial systems in NLP, like n-gram models and hidden Markov models, were vastly dependent on computation, often limited, thereby regulating the scale and depth of the models being trained. As a result, migrating to modern LLMs has required giant leaps in computational power, since GPT-3 and BERT needed bigger architectures and utilized bigger datasets.

With the introduction of GPUs, and a little later TPUs, faster matrix operations were enabled, which constitute the heart of deep learning. Due to parallel processing, GPUs have now become the chief cornerstone in the training of LLMs. They grant one the ability to train models with billions of parameters within a relatively short time frame. For instance, models like BERT contain 340 million parameters. They were trained for 4 days on 16 TPUs [5]. GPT-3 has gained 175 billion parameters in its scale, while developing LLAMs has been made possible. According to reported figures, the training of GPT-3 consumed approximately 3640 petaflop/s-days of computation, which is equivalent to the running of a state-of-the-art GPU for at least over 350 years [4].

More recently, this growth has been accelerated by training models on multiple devices in parallel using distributed computing techniques. Model parallelism splits the model across several GPUs or TPUs, parallelizing the execution of different parts of the model. In addition, data parallelism aims to scatter the training data across a range of devices, effectively accelerating computation. Innovations here have made it possible to scale LLMs up to billions of parameters without linear increases in training time or computational costs [42].

Moreover, other algorithmic optimizations, such as gradient check-pointing, which reduces memory consumption in the back-propagation process, and mixed precision training, which utilizes lower-precision calculations to accelerate training without sacrificing accuracy, further improve the scalability of LLMs. These developments create conditions for modern LLMs to more effectively exploit available computational resources, thus saving time and energy, while even greater models can be built.

Societal Impacts and the Growing Use of LLMs in Commercial Products

As LLMs continued to scale up and improve, their impacts on society started being felt more strongly, especially in the large-scale use of these models in commercial products. Several large companies, including Google, Microsoft, and OpenAI, have already deployed LLMs across many consumer-facing services, including chatbots, virtual assistants, and automated content generation tools.

By far, some of the most famous applications of LLMs involve automated customer service: chatbots powered by models such as OpenAI's GPT and Google's LaMDA can handle complex queries without resorting to human intervention. These chatbots can answer customer questions, help with troubleshooting, and even hold more natural-sounding conversational dialogues. The deployment of LLMs in these systems has saved money for companies, provided better experiences for clients, and increased the numbers regarding consumer queries handled [67].

Apart from customer service, other applications of LLMs are in content generation and automation tools. Copy-writing, email writing, and code generation are some areas that could be automated using models like GPT-3, which generates high-quality human-like text with simple prompts. Microsoft Word and Outlook make use of LLM in their Copilot feature, aimed at providing assistance to the user in writing emails and generating content by predicting text based on the given context by the user.

LLMs continue to creep into creative fields applying the technology to write marketing copy, help with story writing, and even create works of art from textual descriptions. It has created new opportunities for creators but at the same time raised red flags of job losses and potential over-reliance on AI-generated content for industries reliant on human creativity and critical thinking.

Applications for LLMs in the healthcare sector include summarizing medical records, symptom checking, and even assisting with preliminary diagnoses. These models are not intended to supplant the doctor but can be handy tools whereby the professional may better direct his time and energy while dealing with high volumes of data about his/her patients. This also raises ethical concerns in critical areas of healthcare related to accountability, bias issues, and possibly incorrect medical advice generated through AI.

Despite the many positive contributions of LLMs in commercial applications, their pervasive use has also raised concerns over bias, equity, and misinformation. Because they are normally trained on open and sometimes unfiltered datasets, LLMs might learn harmful biases about gender, race, and other sensitive attributes. Several studies show that even GPT-3 generates partial or harmful content upon receiving biased language stimulation. Addressing these issues, AI researchers have presented several bias mitigation techniques and ethical AI frameworks with the goal of ensuring that deployment of LLMs will be responsible and equitable [70].

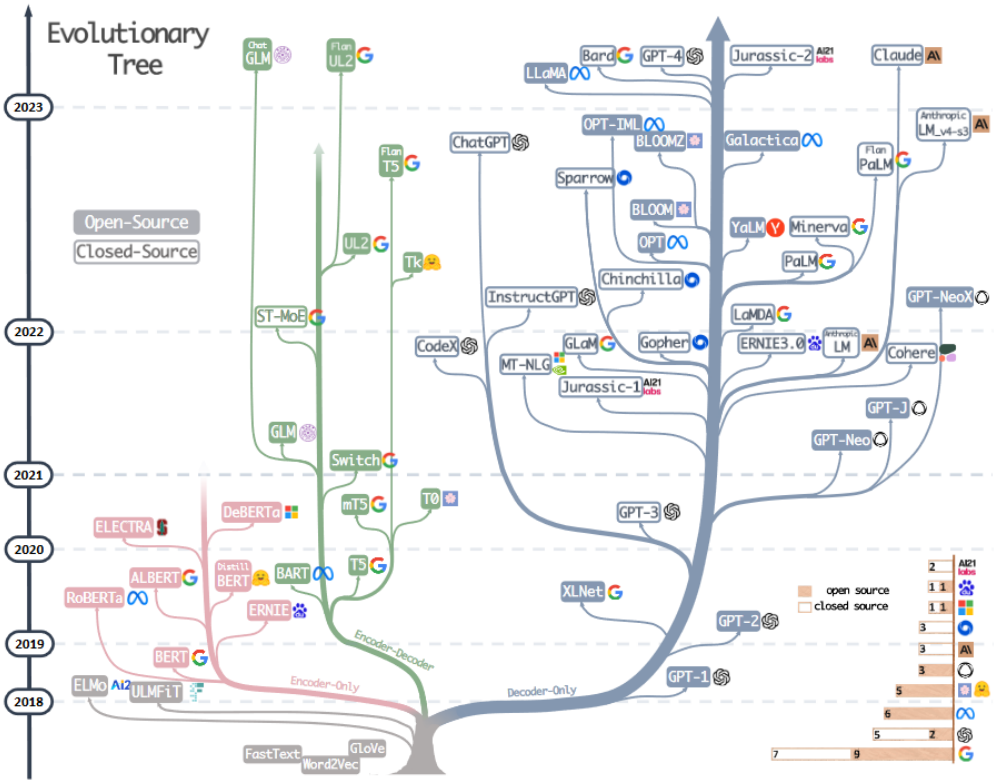

Figure 3.3: Evolutionary Tree of LLMs in business environment

Challenges in Building and Scaling LLMs

While the rise of LLMs has really pushed the performance bar for NLP applications, some key challenges in developing and scaling such models remain, considering highly increasing computational needs, energy consumption, and explainability of models.

- **Computational and Energy Costs**: The most important issue seems to be the huge computational cost for training LLMs. This increases model sizes, which is a need for more and more GPU or TPU clusters to handle such a huge number of parameters-for example, hundreds of petaflop/s-days of computation with OpenAI GPT-3, thereby resulting in considerable energy consumption [4]. The environmental impact of such training has gradually become a concern, with some estimates suggesting that the training of giant LLMs can cause carbon emissions equivalent to the lifetime output of five cars [66]. There are active efforts towards reducing the carbon footprint using AI. This includes making hardware more energy-efficient and developing algorithms which demand less computational power.

- **Memory and Hardware Constraints**: In turn, the size of LLMs brings their use to challenges concerning memory consumption and to hardware constraints. For example, the 175 billion parameters in GPT-3 necessitate very high memory for storage and processing. This makes it very hard to deploy LLM into real-world applications that require real-time processing, especially on less computationally powerful devices such as smartphones or edge devices. Techniques like pruning, quantization, and knowledge distillation are hence developed to make these models smaller in size with a lesser memory footprint while retaining performance. These methods form an important feature in furthering the ease of access and scalability for LLMs in general.

- **Interpretability and Explainability**: Another important challenge about LLM development has something to do with its black-box nature; hence, it is hard to interpret how such models make certain predictions or outputs. Basically, the absence of interpretability is a problem in applications like legal or medical ones, where transparency with accountability is in order. The inherent reasoning process for any given output is hard to trace through LLMs, since these models are deep with billions of parameters. There is, therefore, an urgent need for techniques that enhance model explainability without sacrificing performance. The current methods investigated for these issues include attention visualization, notability mapping, and local interpretability methods.

- **Data Quality and Bias**: The quality of data used in training LLMs is a very critical challenge that still needs resolution. LLMs rely on substantial, uncleaned datasets that include misinformation and bias. Such biases may manifest in the outputs of these tools, sometimes reinforcing troublesome stereotypes or even spreading false information. This requires training LLMs on high-quality and diverse data, which is a challenging task, so research is focused on developing techniques for data filtering and data augmentation that increase robustness in the data used for training. Further, fairness constraints and post-hoc methods for bias correction are being integrated into the LLM pipelines with an objective to reduce bias in models' predictions.

Challenges concerning computational scalability, energy efficiency, interpretability, and data quality remain, nevertheless, active areas of research, despite the remarkable advance made by LLMs. Only by overcoming these challenges can LLMs be assured of being deployed responsibly and sustainably across a wide range of applications.

3.4 OpenAI's GPTs

3.4.1 GPTs Models and their Capabilities

GPT's Ability to Understand and Generate Human-Like Text

One of the intriguing features of GPT models involves generating human-like text that is fluent and coherent. The credit goes to the transformer architecture, which applies self-attention mechanisms in order to capture the long-range word dependencies according to their realization of the wider context of a sentence or paragraph [23]. The pre-training on enormous-sized datasets enables GPT models to internalize not just grammatical rules but also semantic relationships between concepts, facilitating their human reasoning capability in text generation.

A key aspect of GPT's language generation ability is its skill in handling zero-shot, one-shot, and few-shot learning.

- Zero-shot learning refers to the model's capability to carry out some task with no examples present within the prompt. A good example is that when GPT-3 or GPT-4 are asked to summarize some document, it will do an excellent job, even though no explicit summaries were given to it as input. This is possible because through training, GPT would have seen enough and a wide variety of texts and thereby generalize across different kinds of tasks without their explicit training for each type of task [10].

- One-shot learning: GPT models can perform tasks with only one example. Suppose a certain sentence was given as an example to translate it into another language, GPT models will create the proper translation for the next sentences. Making GPT models highly adaptable in different languages, domains, context, with very less task-oriented data.

- Few-shot learning: where they are capable of doing some sort of task by just looking at a few examples. As for coding or solving math problems as some examples, GPT-4 needs but a few input-output pairs before it can generalize quickly and produce results often comparable or better than what humans can write. The ability of the model to understand and synthesize information from a few inputs is due to its pre-training on an extensive range of linguistic data, which equipped it to make inferences of certain patterns even from a few examples [42].

Also, GPT is an auto-regressive text generator, therefore it produces coherent and contextually appropriate text even in longer passages. By predicting the next word in sequence, given prior context, GPT models can create creative writing, draft legal documents, develop software code, or even perform customer service with little intervention by humans. Particular to GPT-4 is that it improves upon earlier versions in increased factual consistency and logical coherence and, therefore, reduces the amount of hallucination and error found within them. It would be enough to say that GPT models-most of all, their latest form, GPT-4-can understand and generate text almost indistinguishable from that written by a human.

GPT Models: From GPT-1 to GPT-4

The line of Generative Pretrained Transformers developed by OpenAI had graduated from GPT-1 to GPT-4 in a development marred by major architectural and capability improvements. Successive models increased their parameter count, training data, and ability to perform complex NLP tasks significantly.

- **GPT-1 (2018)**: GPT-1 was the first in the series of GPT models. It was built using 117 million parameters. It utilized the transformer's decoder-only architecture from the original model [23] pre-trained on the BooksCorpus dataset comprised of 7000 unpublished books. GPT-1 is a model containing an architecture of 12 transformer layers that employ self-attention mechanisms to generate coherent text given previous tokens. Although it was relatively small in number, GPT-1 demonstrated the power of unsupervised pretraining with task-specific fine-tuning, doing very well on a range of benchmarks such as natural language inference and question answering.

- **GPT-2 (2019)**: GPT-2 was much larger, at as many as 1.5B parameters, and was trained on a much larger and more varied dataset-around 40GB of web text. GPT-2 did a great job in generating text, being relevant over large portions of it, and also few-shot learning, or the tendency of the model to perform tasks without explicit training for them. GPT-2 was a first-of-its-kind model where very valid concerns about misuse of generative models came into being since it was able to produce misleading or harmful material [4].

- **GPT-3 (2020)**: GPT-3 made a quantum leap to 175 billion parameters, making it one of the largest models in the world at the time it was developed. GPT-3 was trained on 570GB of text from books, websites, and academic papers. Unlike previous models, GPT-3 outperformed any other model in zero-shot and few-shot learning, where scant examples or no explicit task-specific data were available [10]. GPT-3 attracted wide attention because it was able to carry out a variety of NLP tasks, such as translation, summarizing, and conversational tasks, using a single general-purpose model.

- **GPT-4 (2023)**: GPT-4 further improved the depth and precision of GPT-3, extending multimodal capabilities to include image processing. However, the exact parameter count for GPT-4 has not been shared publicly by OpenAI; it is larger in scale and much more powerful in performance than GPT-3. GPT-4 set new records on factual accuracy, preservation of context, and solving complex problems with better reasoning skills and less bias. In the real world, it has found applications in medical diagnosis, reviewing legal documents, automated tutoring systems, and many others.

Difference between GPT-4 Turbo, GPT-4o, and GPT-4o Mini

The GPT-4 architecture introduced several variants tailored for different applications, balancing performance, cost, and efficiency. Notably, GPT-4 Turbo, GPT-4o and GPT-4o Mini have emerged as streamlined versions of GPT-4, each designed for specific use cases.

Figure 3.4: GPTs History given by OpenAI

- **GPT-4 Turbo**: it is a variant optimized for high efficiency and reduced computational cost while maintaining the same architecture as GPT-4. It is often used in large-scale applications, such as interactive AI systems and real-time services where low latency is critical. GPT-4 Turbo provides comparable language generation quality to GPT-4 but is more cost-effective in terms of deployment. OpenAI has implemented GPT-4 Turbo in products like ChatGPT Plus, making it accessible for users who require both speed and accuracy [10]. This version sacrifices some of the model's depth and precision for faster response times but remains highly capable for general-purpose NLP tasks.

- **GPT-4o**: it is a specialized variant of GPT-4 designed to offer an optimal balance between model size and performance in constrained environments. GPT-4o typically targets enterprise-level applications where full-scale GPT-4 may be overkill, but GPT-4 Turbo's speed may be insufficient for complex tasks. The model is used in industries where computational resources are limited but accuracy and reliability are essential, such as financial forecasting and business process automation. Although GPT-4o may have fewer layers or parameters than GPT-4, it retains key features like multimodal input processing and advanced reasoning capabilities [44].

- **GPT-4o Mini**: it is the smallest variant of GPT-4 designed specifically for mobile and edge devices where hardware constraints are significant. GPT-4o Mini sacrifices more depth and precision compared to GPT-4 and GPT-4 Turbo but is engineered for applications requiring minimal latency and energy consumption, such as real-time translation on smartphones or embedded systems in autonomous vehicles. By drastically reducing the model's size, GPT-4o Mini achieves fast inference speeds with acceptable levels of accuracy for less demanding tasks. While it lacks the full capabilities of its larger counterparts, GPT-4o Mini remains useful in edge-AI scenarios where lightweight models are needed [47].

3.4.2 ChatGPT API Architecture

Key Features and Capabilities

OpenAI's ChatGPT API is one of the interfaces that allow developers to embed the capabilities of OpenAI's GPT models into their various applications. A part of the release of OpenAI's product suite, this API allows developers programmatically to access GPT models, such as GPT-3.5 and GPT-4, and their powerful language generation capabilities for a variety of purposes, from customer service automation to content creation to conversational AI.

One salient feature of the ChatGPT API is the capability for natural language understanding

and natural language generation. It takes input in the forms of prompts or instructions and gives contextually appropriate responses in a manner that would show humanness. The transformer architecture of the underlying GPT models lets the API generate coherent text based on the input, maintaining contextual awareness over long conversations [23].

The other major feature is that this API supports contextual dialogue. ChatGPT keeps the context of previous interactions in a given thread of conversation, therefore making that thread interactive and more dynamic. That enables applications, such as chatbots, to maintain fluid, multi-turn conversations simulating human interaction with users. It is very simple because the memory window for context retention may change with versions of the model and settings, so development can manage the balance between context depth and the computational cost.

Fine-tuning is also allowed by the API, which enables developers to train the model on more specific datasets so that domain-specific knowledge aligns better, or the model understands brand-specific language. While this feature is used more in models like GPT-3, it does remain integral to applications that need to customize the model. This fine-tuning ensures the model will make its output fit the specialized use case at hand, be it legal document drafting, technical support, or even creative writing in a certain style.

Finally, the API has rate limits and token-based pricing that enable flexible use in various patterns. OpenAI charges for the use of APIs in tokens processed, where both input and output tokens count. A rough understanding of a token is a word or part of a word, with the models able to process a few thousand tokens at a time in a single request [48]. This makes the API more accessible by offering scalability and flexibility in pricing, from individual developers to enterprises for large-scale deployment.

Application Development Integrating ChatGPT API

The integration of the ChatGPT API will be done by connecting to the cloud-based OpenAI service through RESTful API endpoints. It is, therefore, quite easy for any developer to include GPT capabilities within their software platforms. Support for various development environments and languages such as Python, JavaScript, and many others are supported in requests.

The core steps in integrating the API involve the following:

- **Authentication**: OpenAI provides developers with an API key, which is required for all requests. This key allows the application to authenticate against OpenAI's servers and track API usage. Ensuring that the key is securely stored is a critical security consideration during integration.

- **API Requests**: A basic API request to ChatGPT involves specifying a prompt that the model uses to generate a response. Developers can tailor these prompts based on the application's needs, whether it's to answer a user query, complete a text, or simulate a conversation. The prompt is sent to the API in JSON format, with parameters such as model version (e.g., gpt-4 or gpt-3.5-turbo), max tokens, temperature (which controls randomness in the output), and top-p (which adjusts output diversity) [46].

- **Handling Responses**: The API responds with a JSON object containing the generated text, which the application can then render or use for further processing. For example, in

a chatbot, the response can be directly relayed to the user in the chat interface. In more complex applications, the response may undergo additional steps, such as sentiment analysis, to determine the next interaction.

- **Error Handling**: Developers must ensure their applications can handle various types of errors, including rate limit exceedances, invalid API keys, or network issues. OpenAI provides clear error messages, and best practices include implementing retry logic or backoff strategies to mitigate temporary failures or throttling.

The integration of the ChatGPT API in applications can enable various functionalities, including:

- **Conversational Agents**: ChatGPT is commonly used in customer service applications to respond to customer queries, generate automated responses, and handle routine interactions, significantly reducing the need for human agents.
- **Content Creation**: The API can be used to automate content generation, from blog posts and news summaries to product descriptions. The ability to fine-tune the model ensures content aligns with the desired tone or subject matter.
- **Code Assistants**: Developers can use ChatGPT to assist with code generation, documentation, and debugging. The API can provide code snippets or explain coding concepts, making it useful for educational platforms or integrated development environments (IDEs).

The modularity and ease of integration make the ChatGPT API highly versatile for application development across multiple domains, from e-commerce and education to healthcare and finance.

Technical Architecture of the ChatGPT API: Tokenization, Request, and Response Cycle

The technical architecture of the ChatGPT API is one of the most efficient designs to handle the entire request-response cycle while executing large-scale text processing tasks. Key components include tokenization, API request processing, and the response generation pipeline.

Tokenization Before the model actually begins processing any input text, it needs to convert that input into something the model will know how to use-tokens.

In this context, tokenization refers to the breaking down of the input text into smaller units or tokens representatives of the whole words, sub-words, or even characters. The GPT models use Byte Pair Encoding as the tokenization method that well balances the efficiency of breaking rare words into sub-word units while keeping common words as single tokens. Each of them has a limit in token numbers, such as GPT-3, which is limited to 4096 tokens, while GPT-4 may have as many as 8192 or even more, which means both the input and generated output must not exceed this limit [4]. For example, a sentence like "The quick brown fox jumps over the lazy dog" would be divided into tokens, each represented as a sequence of integers that the model can process. Then, the model processes these tokens sequentially with self-attention mechanisms in order to determine word relationships and produce coherent responses.

Request Cycle Once the input is tokenized, the application sends an API request to OpenAI's servers. Each request contains several key parameters:

- **Model**: Specifies which version of the GPT model to use (e.g., gpt-4, gpt-3.5-turbo).
- **Prompt**: The input text that the model uses as a basis for generating the output.
- **Max tokens**: Limits the number of tokens in the generated output.
- **Temperature**: Controls the randomness of the output (higher values make the output more creative, lower values make it more deterministic).
- **Top-p**: Regulates the diversity of the output by sampling from a subset of probable outputs.

The request is forwarded as a POST request in JSON format. At the back end, this request fires up the ChatGPT API. This then processes it on the cloud using model weights and generating a response to the prompt.

Response Generation This model is intended to predict, one at a time, the next token in sequence, given an input prompt and previous tokens. The answer is tokenized and returned to the application in JSON format, with the generated tokens decoded into human-readable text. In addition to the response, the answer can include metadata, such as the number of tokens used, that helps developers track usage and manage their costs.

This is because, within the limit of tokens, the developers have to make sure the input and the expected output do not exceed the token maximum capacity of the model. If the input is more than the token limit, then the developer always can do the truncation, summarize, or even slice the input into smaller chunks.

Most whole request-response cycles finish within milliseconds to seconds, depending on the length of the prompt and computational complexity of the model version in use. The OpenAI infrastructure is optimized for high-throughput API requests so that applications can scale to meet user demand without performance degradation.

3.4.3 ChatGPT API Overview

The ChatGPT Assistant API by OpenAI offers a wide range of endpoints [45], organized into four key categories: Assistant, Thread, Message, and Run. These APIs enable developers to create and manage assistants, maintain context across conversation threads, exchange messages, and execute code or other operations through the assistant. Precisely for this, they represent a crucial part for describing the implementation of the component API-Handler built for ArchiGPT described in 6.4.1. Below is a comprehensive breakdown of the API categories, including all key endpoints, their functions, and usage.

1. Assistant APIs

The Assistant API enables developers to create, modify, retrieve, and delete conversational assistants. These assistants are customized with specific models and instructions, making them adaptable

for various use cases such as customer support, educational tutoring, and more.

1.1 Create Assistant

- Endpoint: POST *https://api.openai.com/v1/assistants*
- Description: Create a new assistant by specifying the model and the instructions for its behavior.
- Request Body:
  - model (string, Required): The model version to be used (e.g., gpt-4o).
  - instructions (string, Required): Defines the assistant's role and how it interacts with users.
  - name (string, Optional): A custom name for the assistant.
  - tools (array, Optional): Specifies tools the assistant can use (e.g., code interpreter).
- Example Code:

```
from openai import OpenAI
client = OpenAI()

my_assistant = client.beta.assistants.create(
instructions="You are a personal math tutor. When asked a question, write and ru
name="Math Tutor",
tools=[{"type": "code_interpreter"}],
model="gpt-4o",
)
```

1.2 Modify Assistant

- Endpoint: PATCH *https://api.openai.com/v1/assistants/{assistant_id}*
- Description: Update an assistant's configuration, including its model and instructions.
- Request Body:
  - model (string, Optional): New model version for the assistant.
  - instructions (string, Optional): Updated instructions defining assistant behavior.
  - tools (array, Optional): Updated list of tools for the assistant.
- Example Code:

```
client.beta.assistants.update(
assistant_id="assistant-id",
instructions="You are a physics tutor, offering detailed explanations.", model="
)
```

1.3 Retrieve Assistant

- Endpoint: GET *https://api.openai.com/v1/assistants/{assistant_id}*
- Description: Retrieve detailed information about a specific assistant, including its model, instructions, and tools.
- Example Code:

```
assistant_details = client.beta.assistants.get(
assistant_id="assistant-id"
)
```

1.4 Retrieve Assistant List

- Endpoint: GET *https://api.openai.com/v1/assistants*
- Description: Retrieve a list of all assistants associated with the user's account.
- Example Code:

```
assistants_list = client.beta.assistants.list()
print(assistants_list)
```

1.5 Delete Assistant

- Endpoint: DELETE *https://api.openai.com/v1/assistants/{assistant_id}*
- Description: Delete an existing assistant, removing it from the system permanently.
- Example Code:

```
client.beta.assistants.delete(assistant_id="assistant-id")
```

2. Thread APIs

The Thread API enables the creation and management of conversation threads, allowing the assistant to maintain context across multiple user inputs. Each thread represents a dialogue session where the assistant tracks previous messages for coherence and relevance.

2.1 Create Thread

- Endpoint: POST *https://api.openai.com/v1/assistants/{assistant_id}/threads*
- Description: Create a new conversation thread for a given assistant. This thread allows the assistant to maintain context over multiple interactions.

- Request Body:
  - initial message (string, Required): The first message that starts the conversation in the thread.
- Example Code:

```
thread = client.beta.threads.create(
assistant_id="assistant-id",
initial_message="What is the capital of France?"
)
print(thread)
```

2.2 Retrieve Thread

- Endpoint: GET *https://api.openai.com/v1/threads/{thread_ id}*
- Description: Retrieve details about a specific thread, including all messages and the conversation's state.
- Example Code:

```
thread_info = client.beta.threads.get(thread_id="thread-id")
print(thread_info)
```

2.3 Delete Thread

- Endpoint: DELETE *https://api.openai.com/v1/threads/{thread_ id}*
- Description: Delete a specific conversation thread, removing all associated messages.
- Example Code:

```
client.beta.threads.delete(thread_id="thread-id")
```

3. Message APIs

The Message API manages the exchange of individual messages within a conversation thread. This API enables users to send input to the assistant and receive contextually relevant responses.

3.1 Create message

- Endpoint: POST *https://api.openai.com/v1/threads/{thread_ id}/messages*

- Description: Send a message to an assistant within an ongoing conversation thread. The assistant will respond based on the provided input and the context maintained in the thread.
- Request Body:
  - content (string, Required): The input or query to be sent to the assistant.
- Example Code:

```
response = client.beta.messages.create(
thread_id="thread-id",
message="Explain how gravity works."
)
print(response)
```

3.2 List messages

- Endpoint: GET *https://api.openai.com/v1/threads/{thread_id}/messages*
- Description: Retrieve the entire message history from a specific thread, including user inputs and assistant responses.
- Example Code:

```
message_history = client.beta.messages.list(thread_id="thread-id")
print(message_history)
```

4. Run APIs

The Run API handles the execution of specific tasks or tools within a conversation. This is useful for assistants that can run code or perform operations such as calculations, data analysis, or language translation.

4.1 Create Run

- Endpoint: POST *https://api.openai.com/v1/threads/{thread_id}/runs*
- Description: Execute a specific task or code within a conversation thread. The assistant can perform operations such as running code or calculations.
- Request Body:
  - assistant id (string, Required): The id of assistant to be called
- Example Code:

```
run = client.beta.runs.create(
thread_id="thread-id",
assistant_id="asst_abc123",
)
```

4.2 Retrieve Run

- Endpoint: GET *https://api.openai.com/threads/{thread_id}/runs/{run_id}*
- Description: Retrieve the results of a completed run, including any output or error logs generated during the execution.
- Example Code:

```
run = client.beta.threads.runs.retrieve(
  thread_id="thread_abc123",
  run_id="run_abc123"
)
```

Chapter 4

State of the art

4.1 Introduction

This chapter will analyze the current state of the research about LLMs and their use for designing Software Architecture. Since few studies focus primarily on Software Architecture, this report extends its scope to all the works that refer to Software Engineering tasks, focusing on studies that address similar problems, propose new techniques or utilize existing LLM techniques for Software Engineering tasks.

We can note that there are some stages in the realm of research related to Software Development Lifecycle and LLMs. Some tasks are in an embryonal phase, in which theoric frameworks are proposed and the problem is defined and subdivided into smaller tasks. Then, some basic models are presented, along with new, hand-crafted datasets for benchmarking and "ad hoc" features and capabilities in LLMs themselves. This stage poses a milestone after which newer researches utilize heavily these established models and datasets as the basis for further work. After that, LLMs generic techniques are adopted to upgrade the models and generative AI is used to perfect training datasets and benchmarks. The Code Generation task represents the field, in the SE-related tasks, that poses itself in the latest stage of this research scale. Most other SE-related fields find themselves in the embryonic stage, with newer concepts and researches being introduced but without a real, existing base to expand.

At first, a brief description of LLMs is presented, describing recent evolution and trends. Then, there is a description of recent studies that address Software Engineering tasks similar to Architectural Design. Full System Realization with the usage of LLMs represents a task that approaches a similar problem. Requirements Classification poses categorization tasks similar to challenges encountered in Architectural Design. Finally, Code Evaluation related papers face the challenge of proposing benchmarks in the realm of Software Engineering tasks and LLM.

The next section proposes a selection of LLM techniques. At first, some general LLM prompting techniques are presented. Then, two of them (Chain of Thoughts and Multiple Agents) are described, along with related papers that apply them to the Code Generation task. These techniques are applied only to the Code Generation task, as this is the only field in which research on LLMs has reached this enrichment stage.

The last two sections analyze how these researches have sourced, processed and used data for training purposes and how the same data has been crafted to build benchmarks in the Code

Generation task.

4.2 SE Tasks

4.2.1 General Overview

Research about the use of LLMs to address Software Engineering tasks focused on various aspects of the Software Development Life Cycle (SDLC) (i.e., requirements engineering, software design, software development, software quality assurance, software maintenance, and software management).

The analysis of research volume in the field of Large Language Models (LLMs) for Software Engineering (SE) tasks reveals a concentrated focus on software development, which represents the majority of the studies. This highlights the emphasis on leveraging LLMs to augment coding and development workflows. Software maintenance emerges as the second most significant area, underlining the importance of LLMs in facilitating software updates and enhancements. The area of software quality assurance, while smaller, showcases an increasing interest in automating testing procedures, suggesting a notable area for LLM application.

Conversely, the domains of requirements engineering and software design exhibit a relatively minor share of research activity, indicating these areas have yet to be extensively explored within the context of LLM applications. The software management domain is identified as the least researched area, suggesting it remains largely untapped potential for LLM integration.

Furthermore, when categorizing studies based on the types of problems addressed by LLMs in SE tasks, the majority of research focuses on generation tasks, such as code or text production. This is followed by classification tasks, which involve categorizing software elements, highlighting LLMs' relevance in these contexts. Recommendation tasks also represent a significant portion of the studies, illustrating LLMs' utility in suggesting solutions. However, regression tasks, related to predictive modeling, account for a minimal fraction of the research, indicating this area has seen limited exploration.

The following is an analysis of the papers that have objectives in common with this work, posing the basis for this research. The papers that aim to improve LLM code generation capabilities are presented in the "Techniques" section: this work does not have code generation as its main goal, but those work are also presenting new interesting techniques that heavily influenced this research.

4.2.2 Full System realization

[40] investigates the application of Large Language Models (LLMs) for the end-to-end construction of software systems, specifically using ChatGPT. The paper reports an experiment in which three developers with varying levels of experience were tasked with implementing a web-based application solely using ChatGPT. The developers were required to follow the same set of user stories, using technologies such as TypeScript, Vue.js, ExpressJS, and SQLite. The system they implemented was a Q&A forum application, and their work was evaluated against a manually created reference implementation.

The developers utilized different prompting strategies, highlighting two key approaches: top-down and bottom-up. In the top-down approach, all requirements are presented in the initial prompt, while in the bottom-up approach, prompts are structured incrementally, addressing one

feature at a time. Both approaches had strengths and weaknesses, with the top-down approach allowing for a comprehensive initial scope but leading to more bug-fixing prompts. On the other hand, the bottom-up approach allowed for more focused, feature-specific prompts, which resulted in fewer bugs related to individual features, but challenges arose when integrating different parts of the system.

A notable finding from the experiment is the importance of prior software development knowledge in crafting effective prompts for ChatGPT. More experienced developers were better at guiding ChatGPT through complex issues, especially during bug-fixing phases. For instance, one developer, who had previously implemented the application manually, provided more precise prompts to fix issues, even suggesting potential causes for bugs, such as missing tokens in HTTP headers. This suggests that while LLMs like ChatGPT can significantly assist in software development, they still require experienced developers to navigate the intricacies of system implementation, particularly in error handling and debugging.

The study also revealed the limitations of ChatGPT in producing fully functional code in more complex scenarios, especially when faced with integration challenges between frontend and backend components. In one case, a developer struggled to resolve a communication issue between the frontend and backend after 18 attempts to fix the bug using ChatGPT. This led to the conclusion that while ChatGPT can provide usable code snippets and suggestions, it may fall short in scenarios that require deep integration or nuanced debugging.

Moreover, the developers noted inconsistencies in the quality and style of the generated code. For instance, ChatGPT generated code using both the setup() and data() methods in Vue.js, which led to variations in the reactivity of the user interface components. This inconsistency higlights how developers need to have a solid understanding of the underlying technologies to ensure that the code generated by ChatGPT adheres to best practices and is maintainable.

[69], instead, proposes a series of prompt patterns to enhance LLM responses' quality in various stages of SW development. Some of these proposed patterns fall within the scope of this thesis, providing a way to generate API specifications of a system, a Domain-Specific Language and a generalized system architecture design.

The API Generator pattern enables the automatic generation of API specifications from system descriptions or user requirements. By utilizing LLMs, developers can rapidly create API specifications in structured formats, such as OpenAPI, without manual coding effort. This pattern not only accelerates API development but also facilitates early design phase explorations by providing multiple potential API configurations. The API Generator simplifies the creation of modular, maintainable APIs, allowing developers to iterate over different designs quickly and reducing the risk of tightly coupling services to specific implementations. The ability to simulate these generated APIs can further refine the design process, ensuring that developers can interact with potential API structures before implementation begins. This capacity for rapid iteration over API designs leads to more flexible systems that are easier to modify or scale over time.

The Domain-Specific Language (DSL) Creation pattern highlights the potential of LLMs to generate customized languages tailored to specific software domains. By defining the syntax and semantics of a DSL, developers can use more concise and domain-relevant terms to describe system behavior, often resulting in more compact, token-efficient formats that mitigate the limitations of token-based models. This pattern is particularly useful in scenarios where a highly specific, domain-

focused abstraction can simplify the description of complex systems or processes. By automating the creation of DSLs, LLMs reduce the cognitive load on developers, enabling them to focus on higher-level design tasks. The DSLs created can be reused and refined across various software projects, enhancing productivity and consistency in development workflows. The use of few-shot learning in conjunction with DSLs ensures that the LLM can interpret and apply the language correctly, creating an adaptive, scalable solution for complex software problems.

The Architectural Possibilities pattern offers an innovative way to explore different system architectures during the design phase. By prompting LLMs to generate multiple architectural options based on system requirements and constraints, developers can evaluate a broader range of design strategies with minimal effort. This pattern encourages the consideration of diverse architectural models, such as monolithic versus microservices architectures or different data storage approaches. The LLM can also suggest the implications of each architectural choice on performance, scalability, and maintainability, allowing developers to make more informed decisions early in the project lifecycle. This pattern supports rapid prototyping and experimentation, which is critical in large-scale, distributed systems where architectural decisions can have significant long-term impacts. Combining this pattern with other design-focused patterns, such as the API Generator, can produce a cohesive, flexible architecture that adapts to evolving business and technical requirements.

4.2.3 Requirements Classification

The two papers presented in this section aim to categorize requirements into Functional and Non-Functional Requirements (FR/NFR) and to subcategorize them according to SE standards.

[16] addresses the challenge of classifying software requirements, particularly functional (F) and non-functional requirements (NFR), in projects where natural language processing (NLP) of textual requirements documents plays a central role. The authors propose a model named NoRBERT, which fine-tunes BERT, a powerful pre-trained language model, for the specific task of classifying requirements. One of the main contributions of the paper is the demonstration that NoRBERT improves generalization, which is a common limitation of existing approaches. Traditional methods often fail to maintain performance when applied to unseen projects due to the variation in wording and sentence structures between different software requirements documents. NoRBERT addresses this issue through transfer learning, allowing it to achieve superior generalizability and performance with less training data compared to other deep learning models. The authors present a comprehensive evaluation of NoRBERT on the PROMISE NFR dataset, commonly used in the field, achieving F1-scores as high as 94%, surpassing existing methods.

In addition to classifying functional and non-functional requirements, NoRBERT demonstrates effectiveness in classifying NFR subclasses, such as security, usability, and performance requirements, which are critical in ensuring software quality. The model also performs well when distinguishing between functional requirements that describe different concerns (functionality, data, or behavior).

By comparing NoRBERT with other methods, including those that rely on lexical and syntactical features, the authors show that NoRBERT not only achieves better results but also eliminates the need for manual preprocessing, making it more applicable in real-world scenarios. The paper highlights that NoRBERT's transfer learning capability is key to overcoming the data scarcity problem in requirements engineering, where labeled datasets are often small and imbalanced. The

research also emphasizes the importance of applying classification models to unseen projects, which is often overlooked in prior studies but is crucial for practical applicability. The success of NoRBERT in project-specific evaluations, such as leave-one-project-out cross-validation, further validates its robustness.

[36] has the same objective of [16], proposing flexible prompting templates by converting one multi-class classification problem into K binary classification problems.

The proposed model, PRCBERT, addresses the challenges of requirements classicifaction by utilizing flexible prompt templates, transforming the multi-class classification problem into a series of binary classification tasks. This approach allows for more accurate classification of software requirements, particularly in the context of zero-shot and few-shot learning. The model demonstrates superior performance over existing approaches like NoRBERT and BERT-MLM by applying transformer-based architecture in a more nuanced manner, enhancing both the classification accuracy and the ability to generalize to unseen datasets.

The authors highlight the importance of creating large-scale, real-world datasets to improve the relevance of requirement classification models. They introduce NFR-SO, a dataset compiled from StackOverflow, which contains over 17,000 labeled NFR samples. This dataset provides a more up-to-date and representative sample of software requirements, addressing the shortcomings of the PROMISE dataset. PRCBERT is tested on both the PROMISE and NFR-SO datasets, showing that it outperforms state-of-the-art models in both accuracy and transferability.

The paper also introduces a self-learning strategy to enhance the model's performance on unseen requirements. This method iteratively refines the classification results by selecting samples with high confidence and updating the model accordingly. Experiments also show that PRCBERT achieves significant performance improvements over previous models.

4.2.4 Code Evaluation

The papers presented in this section pose the problem of evaluating code generated by LLMs.

[6] presents a comprehensive analysis of Codex, a language model fine-tuned on publicly available code from GitHub. Codex demonstrates notable advancements in code generation capabilities, particularly in its ability to synthesize standalone Python functions from natural language docstrings. Along with the code generation model, they presented the HumanEval dataset.

The HumanEval dataset is a collection of 164 hand-written programming problems designed to assess functional correctness in code synthesis tasks. Each problem consists of a function signature, docstring, and a set of unit tests, providing a robust framework for evaluating how well a model can translate natural language descriptions (in the form of docstrings) into executable Python code. Unlike previous code evaluation methodologies that often relied on metrics such as BLEU scores or exact string matching, HumanEval focuses on functional correctness, which is a more suitable criterion for code generation tasks. The dataset's unit tests serve as the primary mechanism for this evaluation, aligning the evaluation approach with real-world programming practices like test-driven development.

The pass@k metric, developed to evaluate models on this dataset, measures the fraction of problems that a model can solve correctly after generating k different solutions per problem. If any one of the k generated solutions passes the problem's unit tests, the model is credited with solving that problem. To compute pass@k, the researchers generate multiple code samples per problem

(n = 200 in this case) and determine the unbiased estimator based on combinatorial calculations of correct solutions among the generated samples. This method, while computationally intensive, provides a stable and reliable measure of a model's problem-solving capabilities.

The use of HumanEval also provides insights into the limitations of match-based evaluation metrics, which fail to capture the nuances of program behavior. Programs that differ syntactically can be semantically equivalent, a reality that match-based metrics like BLEU are not designed to handle. Functional correctness, evaluated through unit tests, addresses this limitation by focusing on whether the generated code performs the task correctly, regardless of its structural differences from a reference solution. This approach more closely mirrors the expectations of human developers, who prioritize working solutions over syntactical exactness.

The evaluation process itself centers on the automatic execution of generated code within a secure sandbox environment, which ensures that malicious or erroneous code does not pose a security risk. This sandbox runs unit tests for each sample generated, determining whether the code passes or fails based on its output. Importantly, this evaluation method focuses entirely on the ability of the code to function correctly, which is a standard commonly used in real-world software development practices like test-driven development (TDD). By adhering to this metric, the HumanEval dataset provides a more robust and reliable means of assessing the coding capabilities of LLMs.

In [35], the authors propose a new framework, EvalPlus, which extends the popular HUMANEVAL benchmark with significantly more rigorous test cases, resulting in the enhanced dataset HUMANEVAL+. The focus of the paper is on identifying the inadequacies of current testing practices for evaluating LLM-synthesized code and proposing ways to overcome these limitations.

The primary innovation discussed is the augmentation of the HUMANEVAL dataset with automatically generated test cases using both LLM-based and mutation-based test generation methods. While HUMANEVAL consists of approximately 9.6 manually curated test cases per programming task, the HUMANEVAL+ dataset expands this to over 764 test cases per task, an 80-fold increase in test coverage. This increase is intended to address the issue of insufficient testing in the original benchmark, which often results in semantically incorrect code being incorrectly marked as functionally correct. By providing significantly more test cases, HUMANEVAL+ captures a wider variety of edge cases and complex scenarios, thereby ensuring that the synthesized code is more thoroughly vetted for functional correctness.

In addition to the increased test volume, HUMANEVAL+ also incorporates a series of automated methods to generate new test cases. These methods begin by leveraging ChatGPT to generate a set of seed inputs designed to challenge the synthesized code more rigorously. These seed inputs are then mutated to create a large variety of additional test inputs, ensuring that a broader range of inputs are tested against the generated code. This approach effectively reveals flaws that were previously undetected in LLM-generated code due to the limitations of the HUMANEVAL dataset.

Another key contribution of the paper is the revision of evaluation metrics to better reflect the robustness of LLM-synthesized code. While HUMANEVAL employed traditional pass@k metrics (where k represents the number of generated solutions evaluated), HUMANEVAL+ refines this by integrating a more detailed analysis of the generated test results. For example, differential testing is used to cross-check the correctness of the code outputs against ground-truth solutions, and type-aware mutation is applied to generate additional tests that more comprehensively cover different edge cases.

The modifications in evaluation methods between HUMANEVAL and HUMANEVAL+ are significant. HUMANEVAL relied on a relatively small set of manually designed test cases, which led to many LLM-synthesized codes being incorrectly classified as correct. By contrast, HUMANEVAL+ not only increases the quantity of test cases but also applies automated test generation techniques that allow for more exhaustive coverage of potential inputs. This results in more accurate performance assessments of the LLMs in terms of functional correctness, as HUMANEVAL+ identifies errors that were previously overlooked.

To manage the large volume of generated test cases and maintain evaluation efficiency, HUMANEVAL+ includes a test-suite reduction technique, which minimizes the number of test cases while preserving the same level of testing effectiveness. This reduction is achieved through methods such as branch coverage and mutation analysis, ensuring that the essential functional properties of the code are still rigorously tested without overburdening the evaluation process.

[9] addresses a significant gap in current code generation benchmarks by focusing on class-level code generation, rather than the more common function-level or statement-level tasks. Previous benchmarks like HumanEval primarily assess large language models (LLMs) on simple tasks where models are asked to generate isolated code units such as individual functions. However, this approach limits the understanding of how LLMs perform on more complex software development tasks that involve interdependent methods and classes. The authors highlight that only about 30% of methods in open-source projects are independent of other code contexts, underscoring the necessity of evaluating LLMs's ability to generate code that accurately reflects real-world development scenarios.

The ClassEval dataset is a manually crafted benchmark, comprising 100 Python-based class-level code generation tasks. These tasks cover a broad spectrum of real-world software development topics, such as management systems, game development, and file handling. Each task in the dataset was designed to reflect practical development scenarios by involving multiple methods with diverse interdependencies, including field, method, and library dependencies.

The evaluation metrics used in ClassEval are designed to provide a thorough assessment of both the correctness and contextual dependencies of the generated code. A prominent metric is Pass@k, commonly used in code generation tasks. Pass@k measures the percentage of successfully generated code snippets from k samples. It evaluates the correctness of the generated code by running predefined test suites that come with each task. The benchmark utilizes both method-level and class-level Pass@k metrics, where method-level Pass@k checks correctness at the individual method level, and class-level Pass@k requires all methods in a class to be correct. This dual-level evaluation ensures that LLMs are not only capable of generating individual methods correctly but also handle the interdependencies between methods within a class context.

Moreover, ClassEval includes a dependency metric, designed to assess the model's ability to generate method-dependent code. This metric, denoted as DEP, evaluates the proportion of correctly generated dependencies—either method or field dependencies—relative to the expected number in the canonical solution.

In addition to these evaluation metrics, the dataset includes high-coverage test suites to validate the correctness of the generated code. Each class-level task in the dataset is equipped with extensive unit tests, averaging 33.1 tests per class, which is significantly more comprehensive than previous benchmarks.

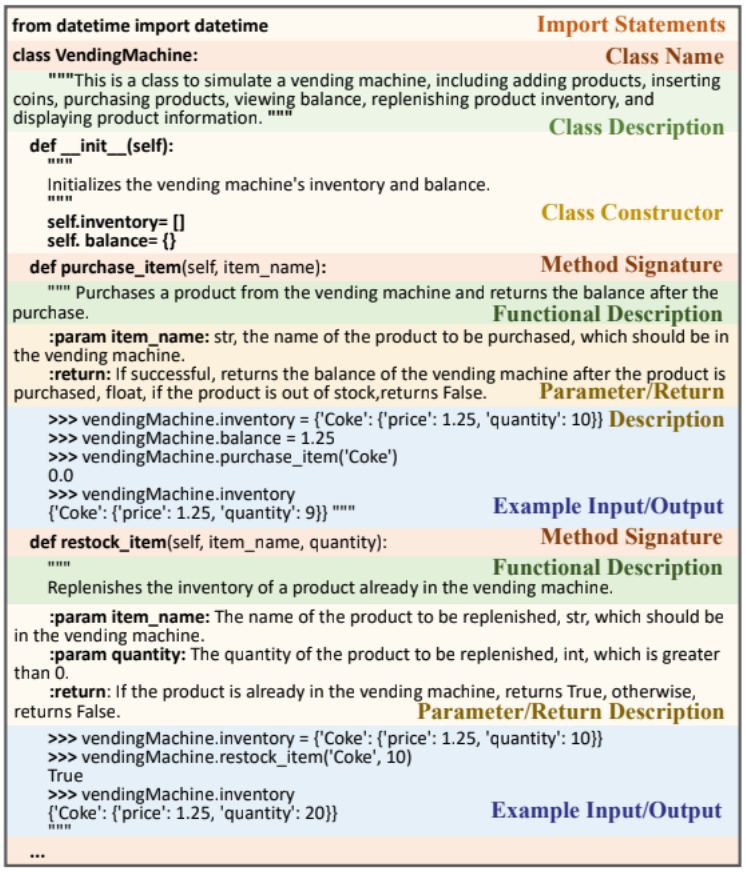

Figure 4.1: Example of Class Skeleton in ClassEval

4.3 Techniques

This section presents multiple techniques used for Software Engineering tasks. Most of them have been previously proposed as general techniques to improve generic LLMs capabilities. However, their use in the following studies shows how these techniques can be adapted to be used in tasks related to the Software Development Life Cycle. Most of the research shown here has the goal of improving LLMs' code generation capabilities, but the methods used can be used with positive results in other goals related to SE tasks.

4.3.1 Basic Prompting techniques

Here two basic techniques are presented. Knowledge of these general LLM prompting techniques represent the basis on which more advanced techniques are built on.

Zero-Shot Prompting

Zero-shot prompting refers to the interaction with a language model where the model is given a task description or query without any additional examples or context. The model relies entirely on its pre-trained knowledge to generate a response. This technique exploits the model's ability to generalize from vast amounts of training data to perform tasks it has not been explicitly trained on.

In zero-shot prompting, the prompt includes only the task instruction or question, without any demonstrations of the desired output format. The language model interprets the prompt based on

patterns it has learned during pre-training. This approach leverages the model's comprehensive understanding of language and concepts to infer the appropriate response.

One of the primary advantages of zero-shot prompting is its simplicity and efficiency. It eliminates the need for crafting detailed prompts with examples, saving time and effort. This technique is particularly useful when examples are unavailable or when quick, generalized responses are acceptable. Zero-shot prompting demonstrates the model's capacity to handle a wide range of tasks, showcasing its versatility.

Despite its convenience, zero-shot prompting has limitations in accuracy and reliability. Without examples, the model may misinterpret the task or produce responses that deviate from the desired format. The lack of context can lead to ambiguous or irrelevant outputs, especially for complex or specialized tasks. Furthermore, zero-shot prompting may not capture nuanced requirements or specific domain knowledge necessary for certain applications. The model's reliance on general patterns from pre-training can result in superficial understanding, making it less effective for tasks that demand precision and detailed comprehension.

Few-Shot Prompting

Few-shot prompting involves providing the language model with a small number of examples within the prompt to guide its response. These examples demonstrate the desired input-output behavior, enabling the model to infer patterns and generate outputs that align more closely with the user's expectations.

In this technique, the prompt includes a brief instruction followed by a few example pairs that illustrate the task. The examples serve as in-context training data, allowing the model to adapt its responses based on the demonstrated patterns. Few-shot prompting harnesses the model's ability to perform in-context learning without modifying its underlying parameters.

Few-shot prompting significantly enhances the model's performance on specific tasks by reducing ambiguity. The inclusion of examples provides clear guidance, resulting in more accurate and consistent outputs. This technique is particularly effective for tasks with complex formats or specialized content, where zero-shot prompting may fall short. By tailoring the prompt with relevant examples, users can influence the model's behavior to meet specific requirements. Few-shot prompting bridges the gap between zero-shot prompting and full fine-tuning, offering a balance between customization and efficiency without the need for extensive additional data or computational resources.

Crafting effective few-shot prompts requires careful selection of examples that accurately represent the task. This process can be time-consuming and may necessitate domain expertise to ensure the examples are appropriate and instructive. Poorly chosen examples can mislead the model, resulting in suboptimal or erroneous outputs.

A limitation to this technique comes from the constraint on prompt length. Language models have a maximum input size, and including multiple examples can consume a significant portion of this limit. This restricts the number of examples that can be provided, which may be insufficient for highly complex tasks. Additionally, few-shot prompting does not guarantee perfect generalization, and the model may still produce unexpected responses outside the demonstrated patterns.

Comparison and Considerations

Selecting between zero-shot and few-shot prompting depends on the task's complexity, the required accuracy, and the resources available. Zero-shot prompting is advantageous for simple tasks, rapid prototyping, or when minimal input is preferred. It leverages the model's general knowledge but may lack specificity.

Few-shot prompting offers improved performance for tasks that benefit from explicit guidance. By providing examples, users can direct the model toward desired behaviors, enhancing reliability and precision. However, this comes at the cost of increased effort in prompt construction and potential limitations due to input size constraints.

In practice, few-shot prompting often yields better results for specialized or complex tasks, while zero-shot prompting suffices for general inquiries. Understanding the trade-offs between these techniques is crucial for effectively leveraging language models in various applications.

4.3.2 Chain of Thoughts

Chain of Thought (CoT) prompting is a technique developed to enhance the reasoning capabilities of Large Language Models (LLMs) by guiding them to generate intermediate reasoning steps when responding to prompts. Unlike traditional prompting methods that elicit direct answers, CoT prompting encourages models to articulate the sequential thought processes leading to a conclusion. This approach leverages the inherent capacity of LLMs to handle complex, multi-step reasoning tasks, thereby improving both the accuracy and interpretability of their outputs.

The fundamental concept behind CoT prompting involves presenting the model with exemplars that include not only the question but also a detailed breakdown of the reasoning steps required to reach the answer. By incorporating these examples into the prompt, the model learns to mimic the structure of logical reasoning demonstrated. This method is particularly effective in few-shot learning scenarios, where a limited number of examples can significantly influence the model's response patterns.

Empirical studies have demonstrated that CoT prompting substantially improves performance on tasks requiring logical deduction, mathematical computations, and commonsense reasoning. For instance, when faced with complex arithmetic problems, models utilizing CoT prompting are better equipped to perform intermediate calculations, reducing errors associated with attempting to solve the problem in a single step. This iterative reasoning process allows the model to handle sub-tasks sequentially, ensuring that each step is validated before proceeding to the next.

Moreover, CoT prompting enhances the transparency of the model's decision-making process. By explicitly outlining the reasoning steps, users can trace how the model arrived at a particular conclusion. This transparency is crucial in applications where understanding the rationale behind an answer is as important as the answer itself, such as in educational tools or decision support systems. It fosters trust and allows for the identification and correction of any flawed reasoning within the model's output.

However, the effectiveness of CoT prompting is contingent upon the quality of the reasoning examples provided. Poorly constructed exemples can deviate the model from the correct behavior, causing it to generate irrelevant or incorrect reasoning steps. Careful consideration must be given to balance the depth of reasoning with practical constraints on response length and processing time.

The following researches apply CoT concept to code generation task, expanding this technique by building articulate prompts aimed at guiding the LLM through subsequent steps in order to reach their goal.

[22] highlights how previous works' generated CoT goes from the input problem to a real detailed sequence of instructions, very similar to code itself. This research proposes instead an intermediate planning phase that produces a generic step-by-step plan, dividing the act of generating code in subsequent phases that resemble the process of generating code instead of code itself.

[33] approaches the expansion of CoT in regard to code generation task in the opposite way. They subdivided the code to be produced into three main structures: "loop" structure, resembling a for/while cycle, "branch" structure, resembling an if/else structure, and "sequence" structure, used for plain instructions. The LLM is instructed to generate at first pseudo-code using only these structures, paired with detailed instructions about the variables to manipulate. The result of this intermediate step is then used to generate code. Has been shown that both these techniques improve code quality largely with respect to vanilla prompting and basic CoT technique.

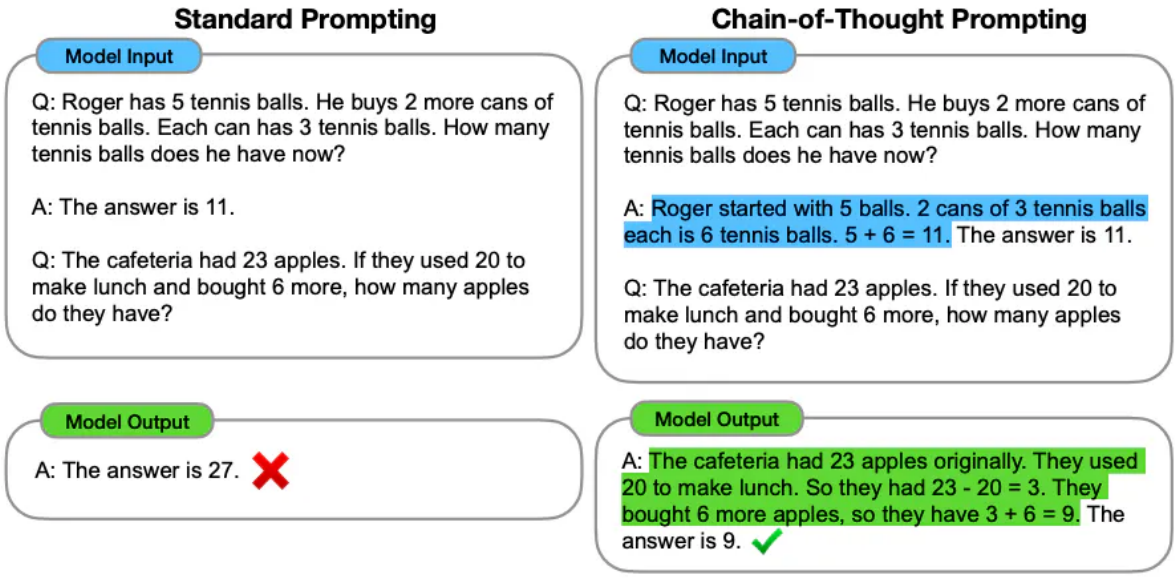

Figure 4.2: An example of Chain of Thought Prompting

4.3.3 Multi-agent prompting

Multi-agent prompting is an advanced technique in the domain of Large Language Models (LLMs) that involves the orchestration of multiple AI agents to collaboratively solve complex tasks or simulate interactive dialogues. Unlike traditional single-agent prompting, where a solitary model responds to user inputs, multi-agent prompting leverages the synergistic capabilities of multiple models or instances of models to enhance problem-solving efficiency and output quality.

At its core, multi-agent prompting relies on the interaction between agents that can assume various roles, such as a questioner and an answerer, or specialized agents focusing on different aspects of a problem. This interaction can be structured to encourage agents to critique, refine, or build upon each other's responses. For instance, one agent might generate an initial solution, while another evaluates its correctness or optimizes it further. Such collaborative dynamics can lead

to more comprehensive and accurate results, especially in complex tasks that benefit from diverse perspectives or iterative refinement.

One of the primary applications of multi-agent prompting is in scenarios requiring creative brainstorming or critical analysis. By simulating a dialogue between multiple agents, LLMs can generate a spectrum of ideas or arguments, facilitating a more thorough exploration of a topic. Additionally, in fields like programming, multi-agent prompting can be employed to debug code, where one agent writes code and another reviews it for errors or inefficiencies.

The implementation of multi-agent prompting often involves careful prompt engineering to define the roles and interactions of the agents effectively. Prompts must be crafted to guide each agent's behavior and ensure productive collaboration. This may include specifying the context, objectives, and communication protocols between agents. Moreover, considerations around the alignment of agents are crucial to prevent the propagation of errors or the generation of unproductive dialogue.

Despite its advantages, multi-agent prompting also presents challenges. The increased complexity in prompt design can lead to unintended behaviors if not meticulously managed. There is also a computational overhead associated with running multiple agents, which may impact the feasibility of deploying such techniques in resource-constrained environments. Furthermore, ensuring consistency and avoiding redundancy in the agents' outputs requires sophisticated coordination mechanisms.

Two researches expanded this concept to Software Engineering tasks.

[54] explores the application of multiple GPT-based agents in automating various tasks within the software development lifecycle. The framework proposed by the authors employs multiple autonomous agents, each specializing in distinct phases of the software development process. For instance, one agent focuses on project planning by defining goals and creating a detailed execution plan, while another ensures that the plan aligns with the project's objectives. Similarly, other agents handle requirements extraction, system design, and quality analysis. The model emphasizes collaboration between agents, where each agent performs a specific task in tandem with others to produce high-quality code, test scripts, and deployment plans. Early experiments using this framework have shown impressive results in terms of efficiency. In one test, the agents successfully generated a simple game (Snake) within a few minutes, producing requirements specifications, code, and documentation autonomously.

The development process follows a waterfall approach, so each role represents a stage in delivering the final product.

[8], instead, focuses on code generation instead of full product realization. It presents an innovative framework aimed at enhancing code generation tasks by leveraging multiple large language models (LLMs) in a collaborative setup. The approach addresses the limitations of direct code generation by employing a framework where distinct roles—namely, analyst, coder, and tester—are assigned to LLMs, simulating a virtual software development team. Each LLM is responsible for a specific stage of software development, thereby mimicking the collaborative nature of human teams in real-world development environments.

The division of labor (DOL) is at the core of this self-collaboration framework. By assigning specialized roles through detailed role instructions, the framework transforms LLMs into domain-specific "experts." For instance, the analyst LLM is tasked with breaking down the problem and generating high-level plans, while the coder LLM implements code based on these plans, and the

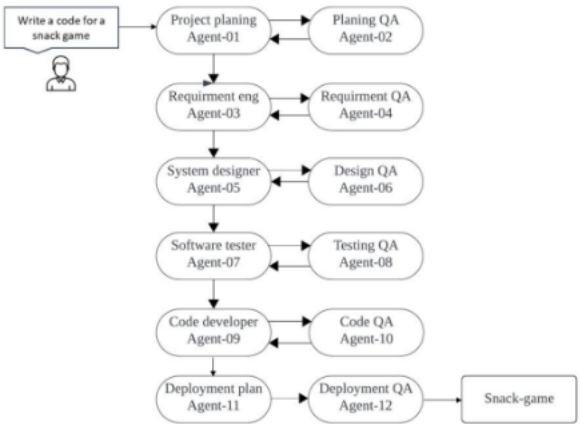

Figure 4.3: Autonomous code generation for snake-game

tester LLM reviews the code, generating feedback on its correctness and potential issues. This workflow mirrors traditional software development methodologies, particularly agile and waterfall models, thus aligning with well-established practices in software engineering.

Empirical results demonstrate the efficacy of this framework, yielding significant improvements in Pass@1 scores, with a relative increase of 29.9% to 47.1% compared to direct code generation approaches. The structured interaction among LLMs allows for refined outputs that address complex coding requirements, which direct code generation often struggles with. Furthermore, the framework's ability to consider a broader set of test cases, including edge cases, highlights its potential to improve code reliability.

Another key contribution of the paper is its adaptability. The framework is designed to be flexible, allowing for various software development methodologies to be incorporated depending on the task. The paper shows how a simplified version of the waterfall model can be instantiated within this framework to optimize team collaboration. The introduction of role-playing within LLMs is critical to the success of this approach, as it enables each model to function within its specific expertise area, enhancing the overall quality of the generated code.

4.4 Training Data

4.4.1 General Overview

This section will analyze data used for training LLMs for Software Engineering tasks. Since LLM benefit largely from accurate crafting of the data used for training, this section will analyze what type of data is used in recent researches and how they are processed before feeding the LLM.

4.4.2 Data Sources

Data sources can be categorized into four distinct types: open-source, collected, constructed, and industrial datasets, each with its unique method of compilation and intended purpose.

- Open-source datasets are characterized by their public availability and are typically hosted on open-source platforms, providing a credible and community-vetted resource for academic research. These datasets, such as HumanEval with its collection of Python problems, are

essential for enabling LLMs to learn from real-world data, thereby enhancing their applicability and performance in practical scenarios.

- Collected datasets, on the other hand, are amassed directly from a wide array of sources like websites and social media, tailored to specific research needs. This approach allows for the creation of datasets that closely align with the research objectives, offering a customized foundation for model training.
- Constructed datasets represent a further refinement, where existing datasets are modified or augmented to meet specific research criteria. This process often involves manual or semi-automatic methods to produce datasets that are highly specialized, such as annotated code snippets for studying automated program repair techniques.
- Industrial datasets are sourced from commercial entities and contain proprietary data, presenting a valuable but challenging avenue for research due to the sensitive nature of the information and the legal and ethical considerations involved.

Various analysis highlight a predominant reliance on open-source datasets in academic research, attributed to their authenticity, credibility, and representation of real-world scenarios. However, a notable gap is identified in the utilization of industrial datasets, which are underrepresented in current research. This discrepancy points to a potential misalignment between academic research and real-world industrial needs, underscoring the importance of integrating industrial datasets into future research to ensure the development of LLMs that are both robust and applicable across diverse contexts.

4.4.3 Data Types

SE-related datasets can be categorized into five distinct types: code-based, text-based, graph-based, software repository-based, and combined data types, with a pronounced prevalence of text-based datasets. The predominance of text-based data underscores the LLMs' adeptness at natural language processing, vital for tasks such as code comprehension, bug fixing, and code generation. These models leverage extensive textual data to provide insights and solutions across various software engineering applications, demonstrated by the frequent use of text-based datasets with numerous prompts in training to guide model behavior effectively.

Great importance is given to programming problems and source code in enriching models' understanding and performance in SE tasks. Programming challenges and source code datasets facilitate a broad knowledge base and skills generalization, essential for a wide range of software engineering challenges. Moreover, the inclusion of specific data types like Stack Overflow posts, bug reports, and programming tasks within text-based datasets enriches the training material, further enhancing model capabilities.

Graph-based datasets, while less common, offer unique advantages by representing code through graphical structures, illustrating the relationships and dependencies among code components. This approach is particularly beneficial for tasks that require understanding complex code interactions, such as rapid prototyping.

Software repository-based datasets tap into the wealth of information available in version control and issue tracking systems, providing comprehensive insights into the software development process.

This data type supports empirical analysis and model training aimed at improving software quality and development efficiency.

Lastly, the value of combined datasets is remarkable, integrating multiple data types to enrich model training further. Examples include datasets combining source code with comments, annotations, error messages, Q&A pairs, and other relevant information. These combined datasets demonstrate enhanced model performance in specific tasks, such as automating code review, by leveraging the synergistic effects of diverse data types.

4.5 Evaluation Data and Benchmarks

4.5.1 General Overview and Trends

Since research about LLM usage for solving Software Engineering tasks has arised very recently, widely accepted benchmark datasets are rare. Their presence is related to the popularity of the task within the research community. A bright example of this is the code generation task, where newer, more precise benchmarking tools have been continuously proposed through the last 3 years, with newer models being continuously developed and tested on these benchmarks. On the other end, research about LLM usage for software design has seen way less popularity, with fewer studies proposing much more generic models and, in many cases, proposing techniques or architectural frameworks without real implementation. This lack of presence in the research scenario causes the absence of benchmark datasets in this field.

However, analysis of the evolution of benchmark datasets with respect to code generation tasks, with numerous researches and proposals in the last three years, can give an idea of the evolution path of benchmarks with respect to a specific SE task.

The first milestone is represented by [6]. Along with a model for code generation, this paper proposed HumanEVAL, a new manual-crafted benchmark for code generation. HumanEVAL consists of 164 hand-written programming problems, along with unit tests and an algorithm to evaluate results. This research presents for the first time a dedicated tool to evaluate code generation models.

On top of HumanEval, [35] constructed EvalPLUS, refining numerous flaws of the initial work. EvalPLUS expands the unit test dataset largely and proposes parameters changes to the evaluation algorithm, representing, as today, the state of the art in this field. While these researches aimed at evaluating code functions, [10] proposed a benchmark to analyze AI-generated classes, expanding the concept to more abstract code paradigms.

Two main topics can be discussed about this evolution. The first one regards the language used in those datasets. Python has emerged as the standard language in code generation models and subsequently benchmark tools, particularly for its simplicity and LLM-friendly structure. While having a standard “de facto” simplifies the comparison between models and benchmarks, it raises problems regarding the generalization of those models and poses threats in expanding the research to other languages and frameworks. Moreover, the lack of complex architectural abstractions of Python represents an obstacle in expanding the complexity of the tasks the models will be asked to solve.

The second topic regards the evolution path of those benchmarks. The first stage, represented by HumanEVAL, can be seen as the initial stage of the creation of a benchmarking datasets. Problems in HumanEVAL had to be manually crafted in order to have the characteristics to be consistently

analyzed by LLMs. Previous open-source datasets couldn't be used for this purpose, since they were written for other purposes or for the evaluation of code written by humans. In the second stage, represented by EVALPlus, the dataset has been expanded to be more precise, adding unit tests and changing the evaluation algorithm. To do that, AI tools could be used, since the expansion of the unit tests doesn't require human effort to be carried on. Lastly, ClassEval expands the concept by proposing a new benchmark to evaluate classes. Again, problems had to be manually crafted along with evaluation metrics.

This analysis shows how the evolution of benchmarks follows two phases for each stage. At first, a totally new benchmark is proposed, raising the complexity of the analyzed problems. Then, the same benchmark is refined and expanded, also using AI tools. When the models have reached the capability to address more abstract problems (like classes compared to functions), a new stage starts and a new benchmark has to be proposed from scratch.

Chapter 5

ArchiGPT Overview

5.1 Idea Overview

The idea behind this whole thesis is to follow the path that research has traced for code generation and evaluation with LLMs and translate it to the architectural design of system following microservices architecture principles.

For code generation, models have been developed to start from the textual description of a function and output actual code that fulfills the requirements. To evaluate those models, as said in 4.2.4, researchers have produced datasets of functions, from various sources, where each function is composed of the following elements:

- Textual description of the function purposes
- Unit tests to evaluate the functions output

Models produce the source code of said functions, based on the textual description, and that source code is evaluated with the uni tests. Finally, metrics have been developed to summarize the results of unit tests over all the functions of the dataset and multiple runs with meaningful indexes that allow researchers to compare different code generation models on the same parameters, highlightning strenghts and weaknesses of the models.

Applying the same concept to architectural design of systems, at first the need for a specific output/dataset format has been satisfied. While for functions the output format naturally is the function's source code, for architectural design of systems the Project Standard format has been defined, in order to structure the output in a standard way. The Project Standard format structures the system design following microservice architecture guidelines, defining parameters to be described for the containers and microservices of the system. The model's input is given by a combination of textual descriptions and user stories.

Then, a model, ArchiGPT, has been developed to carry on the architecture design task. ArchiGPT takes as input a brief description of the system to be designed and its user stories and, through a series of prompts that use advanced prompting techniques, is able to generate the projects architecture design in Project Standard format.

To evaluate the systems designed by ArchiGPT and models that have the same input/output format, a dataset of projects has been structured. The Project Dataset derives from projects from an academic class in La Sapienza University in Rome. The projects presented only source code

and scattered informations about the purpose and architecture of the developed system. They have been carefully selected, cleaned, and technical documentation has been generated, resulting from an analysis of the source code. In their final form, each project comprises the description and user stories that serve as input for systems like ArchiGPT, the source code developed by the students, a technical analysis and an attached Json file for evaluation purposes.

To evaluate ArchiGPT (and all future models following the same input/output schema), metrics have been defined to evaluate the performances of the model. The metrics evaluate the defined architecture for each system in terms of how the user stories are fulfilled by microservices, the granularity of the defined system and if the containers have the necessary backend and database microservices to satisfy the requirements defined by the user stories.

Overall, our work covers all the aspects needed to generate and evaluate the architecture design of microservices-based systems. It provides a model to generate said architectures, defines the format for input and output of said model, provides a testing dataset and metrics for model evaluation.

5.2 ArchiGPT Overview

This section will describe how ArchiGPT works. The idea behind ArchiGPT is to develop a software that leverages the prompt chaining technique and LLMs capabilities to build the architecture design of a system from scratch. ArchiGPT takes as input a brief description of the system to be designed and its user stories. From there, it uses the prompt chaining technique by calling a series of LLM assistants in sequence to build the system's microservice architecture. The system designed by ArchiGPT follows microservice architecture guidelines and best practices and mainly employes REST as its communication architectural style.

At first, the containers' structure of the system is designed by ArchiGPT, grouping the user stories based on their real world scope and assigning each one of these groups to a newly designated container. After that, the software describes the characteristics of each container, defining in a general way their need to have a database and to connect to external services to fulfill its previously defined purposes. Then, ArchiGPT designs the microservice structure of each container, diving deeply into the microservices characteristics by defining the technologies employed by each microservice and their structure and patterns. Lastly, it describes the backend services' endpoints and the pages of the frontend microservices.

ArchiGPT uses the prompt chaining technique by employing the responses by the previous assistants, filtered and carefully structured, to craft the user prompts for further assistants down the chain. To accomplish this task, it employs various string manipulation algorithms and leverages MongoDB capabilities in managing objects that follow varying data structures, given the constantly changing environment proper of the designing task.

The designed systems are meant to employ Docker for containerization: from a wider point of view, the containers characteristics describe the structure and features of the containers' docker-compose files, while the microservices description can be used to craft their Dockerfiles.

ArchiGPT itself is built following microservices architecture directives: there is a clear separation of concerns between the different components of the system. API-handler microservice manages the quirks and features of the usage of OpenAI APIs, providing a simple way to connect to their services. It also stores the assistants' prompts. MongoDB microservice manages the persistance

needs of the system, to store projects' designs in a structured manner. The backend microservice manages user requests, handles business logic and connects to the API-handler for connection to the assistants and to MongoDB for the persistance layer. It exposes direct endpoints to complete each phase of the system design. The frontend microservice provides a user interface to interact with ArchiGPT, visualizing each assistant result and giving the user the possibility to regenerate an assistant's response if the quality is not deemed sufficient. Lately, the metrics microservice provides a way to generate an architecture design for all the projects of the Projects Database and generate a JSON file suitable for the metrics software.

Overall, ArchiGPT provides a direct way to design systems' architecture following microservices architecture principles, based solely on a description of the system to be designed and its user stories. Its realization is self-contained and dockerized, and it can be used as a stand-alone design generator or as a tool to help software architects in designing their systems.

5.3 Dataset Overview

The dataset chapter describes the development process of the Archi Dataset, which is responsible for validating ArchiGPT work and his relative generative outputs. The final dataset is composed by 8 different projects, each characterized by the following files:

- A brief description of the system to be developed
- User stories of the system
- Source code produced by the students
- Technical documentation of the students' developed system in Project Standard format
- Datametrics JSON file

The chapter covers all the dataset creation phases, starting from a detailed description of the academic context in which the projects were developed and their general characteristics. Then, the selection phase is described, analyzing the selection parameters, the projects' status and the existing documentation for them. The next section describes the selected projects' source code from a technical point of view, analyzing how the microservices architecture is applied and their design choices. The following sections descibe the processes of cleaning the projects'user stories and description and the crafting of new technical documentation in Project Standard. Finally, the Datametrics format is described.

Chapter 6

ArchiGPT

6.1 Overview

This section will describe how ArchiGPT works. The idea behind ArchiGPT is to develop a software that leverages the prompt chaining technique and LLMs capabilities to build the architecture design of a system from scratch. ArchiGPT takes as input a brief description of the system to be designed and its user stories. From there, it uses the prompt chaining technique by calling a series of LLM assistants in sequence to build the system's microservice architecture. The system designed by ArchiGPT follows microservice architecture guidelines and best practices and mainly employes REST as its communication architectural style.

At first, the containers' structure of the system is designed by ArchiGPT, grouping the user stories based on their real world scope and assigning each one of these groups to a newly designated container. After that, the software describes the characteristics of each container, defining in a general way their need to have a database and to connect to external services to fulfill its previously defined purposes. Then, ArchiGPT designs the microservice structure of each container, diving deeply into the microservices characteristics by defining the technologies employed by each microservice and their structure and patterns. Lastly, it describes the backend services' endpoints and the pages of the frontend microservices.

ArchiGPT uses the prompt chaining technique by employing the responses by the previous assistants, filtered and carefully structured, to craft the user prompts for further assistants down the chain. To accomplish this task, it employs various string manipulation algorithms and leverages MongoDB capabilities in managing objects that follow varying data structures, given the constantly changing environment proper of the designing task.

The designed systems are meant to employ Docker for containerization: from a wider point of view, the containers characteristics describe the structure and features of the containers' docker-compose files, while the microservices description can be used to craft their Dockerfiles.

ArchiGPT itself is built following microservices architecture directives: there is a clear separation of concerns between the different components of the system. API-handler microservice manages the quirks and features of the usage of OpenAI APIs, providing a simple way to connect to their services. It also stores the assistants' prompts. MongoDB microservice manages the persistance needs of the system, to store projects' designs in a structured manner. The backend microservice manages user requests, handles business logic and connects to the API-handler for connection to

the assistants and to MongoDB for the persistance layer. It exposes direct endpoints to complete each phase of the system design. The frontend microservice provides a user interface to interact with ArchiGPT, visualizing each assistant result and giving the user the possibility to regenerate an assistant's response if the quality is not deemed sufficient. Latly, the metrics microservice provides a way to generate an architecture design for all the projects of the Projects Database and generate a JSON file suitable for the metrics software.

Overall, ArchiGPT provides a direct way to design systems' architecture following microservices architecture principles, based solely on a description of the system to be designed and its user stories. Its realization is self-contained and dockerized, and it can be used as a stand-alone design generator or as a tool to help software architects in designing their systems.

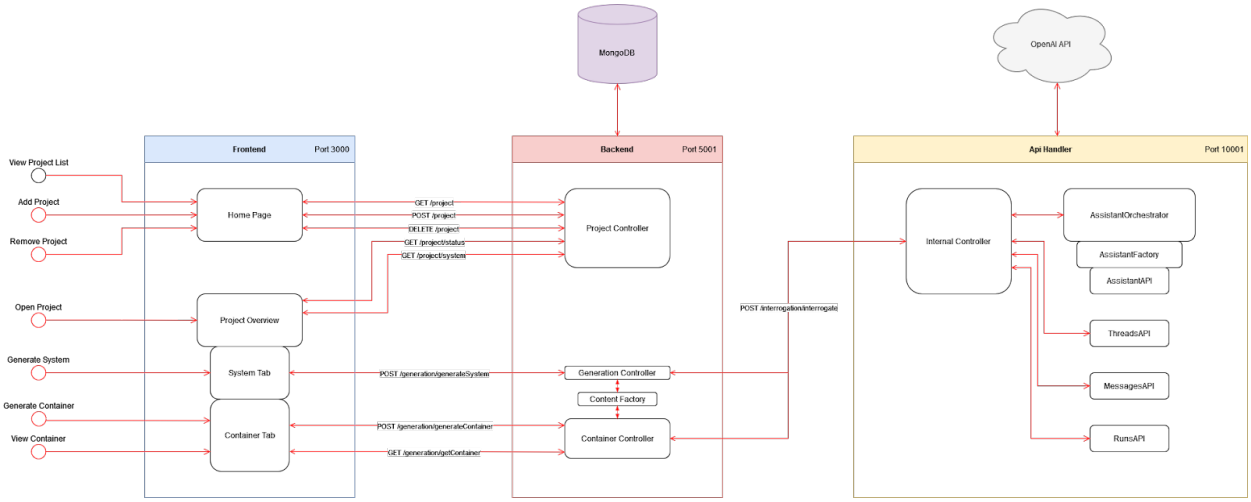

Figure 6.1: ArchiGPT Workflow

6.2 Prompt Chaining

6.2.1 Definition

Prompt chaining is a strategic technique for working with Large Language Models (LLMs) by breaking down complex tasks into a series of connected prompts. Each prompt builds upon the output of the previous one, allowing the model to handle tasks that require multiple steps or advanced reasoning. This approach makes it easier for the model to perform tasks like planning, problem-solving, and decision-making, which can be tough to achieve through a single prompt.

The main advantage of prompt chaining is that it overcomes some limitations of one-off interactions with LLMs. While these models are quite adept at generating coherent, relevant responses, they can struggle when it comes to solving problems that involve deeper reasoning or require pulling together information from different sources. By laying out a series of prompts that guide the model step by step, prompt chaining helps navigate these challenges, making it more effective in tackling complex problems.

The process kicks off with an initial prompt that introduces the main task or question. Then, subsequent prompts focus on specific aspects of the problem, asking the model to expand on certain points or refine earlier answers. For example, if the task is to develop a comprehensive plan, the first prompt might ask for an outline, with later prompts digging deeper into the details of each

section. This structured, layered approach mimics how humans tend to break down complicated tasks into more manageable steps.

Moreover, prompt chaining enables the incorporation of external knowledge and constraints at various stages of the interaction. By introducing new information or modifying the context through intermediate prompts, users can guide the model more effectively toward the desired outcome. This flexibility is particularly beneficial in domains where precision and adherence to specific guidelines are critical.

From a theoretical standpoint, prompt chaining aligns with concepts in cognitive science and artificial intelligence related to hierarchical planning and decomposition. It reflects an understanding that complex cognitive tasks are often more effectively addressed through a series of incremental steps rather than attempting a comprehensive solution in a single attempt.

6.2.2 Types of Prompt Chaining techniques

Prompt chaining techniques provide structured ways to tackle complex problems by leveraging different approaches in large language models (LLMs). The four primary methods are Sequential, Conditional, Iterative, and Recursive Prompt Chaining, each offering a unique path to decompose tasks.

- Sequential Prompt Chaining involves breaking down a complicated task into a linear series of simpler prompts. In this case, the result of one prompt naturally leads into the next, allowing for a step-by-step progression toward solving the problem. It's ideal for tasks with a clear sequence of actions, as it simplifies each stage, making it easier for the model to focus and reduce the overall complexity.

- Conditional Prompt Chaining takes it a step further by introducing decision-making into the process. Here, what happens next depends on the previous outcome, similar to "if-then" logic in programming. This approach works well for scenarios that require adaptability, such as troubleshooting or diagnosing issues, where the model's next step depends on prior answers.

- Iterative Prompt Chaining involves repeating prompts to gradually refine the outcome. It's similar to how iterative algorithms work in computing, refining a solution through multiple rounds. This is particularly helpful for tasks that need improvement over time, like editing or optimizing content, as the model builds upon earlier responses to deliver more accurate results.

- Recursive Prompt Chaining is a more advanced technique where a prompt is designed to invoke itself, either directly or indirectly, with modified parameters. This self-referential approach allows the model to handle tasks that involve nested or hierarchical structures, similar to recursive functions in programming languages. Recursive Prompt Chaining is effective for processing data that naturally forms a hierarchy or for solving problems that require the same solution approach at different levels of granularity. By utilizing recursion, the model can manage complexity efficiently, breaking down tasks into ever-smaller components until they are simple enough to solve directly.

The differences among these prompt chaining techniques lie in their structural design and applicability to various problem types. Sequential Prompt Chaining is linear and deterministic, best suited

for tasks with a predefined sequence of steps. It emphasizes clarity and simplicity, ensuring that each stage is completed before moving on to the next. Conditional Prompt Chaining introduces flexibility and adaptability, allowing the prompt sequence to change in response to the model's outputs. This makes it ideal for interactive applications where the path to the solution is not fixed and depends on intermediate findings.

Iterative Prompt Chaining focuses on refinement through repetition, enabling the model to enhance its output incrementally. It is particularly useful when the initial output is expected to be imperfect and can benefit from successive improvements. Recursive Prompt Chaining, on the other hand, addresses tasks with inherently recursive structures. It leverages the model's ability to apply the same logic at multiple levels, facilitating the handling of complex, nested information.

In selecting a prompt chaining technique, it is crucial to consider the nature of the task at hand. Sequential Prompt Chaining is appropriate when the task can be neatly divided into ordered steps. Conditional Prompt Chaining is preferable when the task requires adaptability and decision-making based on previous outputs. Iterative Prompt Chaining is suitable for tasks that benefit from gradual refinement, while Recursive Prompt Chaining excels in managing hierarchical or self-similar data structures.

6.2.3 Strategies for Prompt Chaining

A prevalent strategy is hierarchical decomposition, where an overarching task is broken down into subordinate tasks arranged in a logical sequence. Each prompt addresses a specific subtask, and the cumulative outputs contribute to the final goal. For example, in a research assistance scenario, the first prompt might instruct the model to gather relevant literature on a topic, the second to summarize key findings, and the third to identify gaps in the research. This structured approach leverages the model's strengths in handling sequential information processing.

Another strategy is iterative refinement, which involves repeatedly improving the model's output through successive prompts. In this approach, the initial prompt elicits a baseline response, and subsequent prompts focus on enhancing specific aspects such as accuracy, depth, or style. This method is particularly useful in creative tasks like writing or design, where the model can be guided to develop ideas progressively. By requesting revisions or elaborations, users can steer the model toward the desired outcome.

Context preservation is a critical consideration in prompt chaining due to the limited context window of LLMs. Strategies such as context summarization involve condensing previous interactions into a concise form that still conveys the necessary information. This helps maintain continuity without overloading the model's context capacity. Another technique is context anchoring, which entails repeatedly including key pieces of information in subsequent prompts to ensure they remain within the model's attention.

Conditional branching is an advanced strategy that allows the prompt sequence to diverge based on the model's outputs, effectively enabling decision-making processes. By setting up conditions that check for certain responses or patterns, the chain can adapt dynamically, providing a more interactive and responsive experience. For instance, if the model's output meets a specific criterion, the next prompt might proceed to a new topic; otherwise, it might request further clarification or correction.

Integrating external knowledge sources or tools within the prompt chain can also enhance the

model's capabilities. Combining LLM outputs with data retrieval systems or computational tools allows for tasks that require up-to-date information or complex calculations, which the model alone might not handle effectively. For example, prompts can instruct the model to draft code snippets that are then tested and refined in subsequent prompts based on execution results.

Implementing these strategies requires careful monitoring of the cumulative effects on the model's performance. Overly long or complex chains can lead to degradation in output quality due to context window limitations and the potential for error propagation. Therefore, iterative testing and optimization of the prompt chain are crucial. Techniques such as prompt paraphrasing, adjusting prompt lengths, and refining instructions can help in fine-tuning the chain for optimal results.

6.2.4 Prompt Chaining implementation in ArchiGPT

In ArchiGPT, the Sequential Prompt Chaining technique is used, in order to generate precisely each step of the system design. The prompt chain is divided in 3 phases, each one having a different context extract after the completion of the previous phase.

In the first phase (system phase), the informations given by the user (system description and user stories) provide the context for the first assistant. It generates the initial system subdivision into containers, then this subdivision is added to the context of the following assistants as user prompt, to give them the current status of the system design to be expanded.

In the second phase (container phase), each container is analyzed singularly: the assistants gradually build the specific container technical specifications. The context of these assistants, provided in the user prompt, consists of:

- Initial informations on the system to be built (Description and user stories)
- Container structure provided by the assistants of the first phase

Informations from the previous assistants of this phase are dinamically added to the context in the user prompt.

In the third phase (service phase), the assistants gradually build the specifications of a single microservice inside a container. The context of these assistants consists of the specifications of the father container generated by the assistants of the container phase. While in the previous phases the context given to the assistants regards the entire system, in this phase the context is reduced to not waste tokens and to give the assistants a more specific view of the problem, allowing them to generate a tailored response. As in the previous phases, the reponses given by the previous assistants of this phase are dynamically added to the user prompt.

Assistant prompts are designed with the explained context limitations in mind: the analysis executed in the container phase (persistance evaluation and connection to external services) can benefit from their system-wide context to give each container a generic description of the container's characteristics, with a point of view of the architecture of the whole system. This already conducted analysis allows the assistants in the service phase to focus only on the technical specifications and features of the microservices.

6.3 Technological Choices

In order to build a scalable and modular solution for ArchiGPT, we took conscious decisions on what technologies to use in order to fulfill both our architectural and functional requirements. The microservice architecture we implemented is designed to facilitate the addition and removal of features without significant overhead, ensuring flexibility and maintainability throughout the development process. The technologies chosen to satisfy these needs are the following:

- **Docker**: Docker was selected for containerization and modularity. It allows us to package each service as an independent, isolated unit, enabling services to be started, stopped, or restarted individually without affecting other components. Access to the container logs through Docker provides for some important debugging, monitoring and system performance oversight.
- **Postman**: This tool was applied during the APIs testing and debugging processes. Its comprehensive set of tools for sending requests, analyzing responses, and managing API collections made it essential for verifying the correct behavior of our services, particularly during the integration of various microservices.
- **Visual Studio Code (VS Code)**: As our primary integrated development environment (IDE), Visual Studio Code offers extensive support for plug-ins, which streamline coding, testing, and version control. It also facilitates collaboration through built-in Git support and enables us to work efficiently with the various languages and frameworks used in the project.
- **GitHub**: GitHub was chosen as our version control system, providing a centralized repository for tracking changes and collaborating. It ensures that both developers have synchronized access to the latest code versions, promoting effective collaboration and avoiding potential code conflicts.
- **MongoDB**: MongoDB, a NoSQL database, was chosen for its simplicity in managing JSON-like documents, which serve as the primary data format for storing project information generated by ArchiGPT. Its flexibility in handling semi-structured data aligns well with the dynamic nature of our dataset.
- **Python Flask**: Flask, a lightweight Python framework, was used to build the backend microservices. Flask offers the basic functionality needed to handle HTTP requests, and we selected it based on our prior experience with the framework, which allowed for faster development and ease of integration.
- **React**: For the frontend, we utilized React due to its component-based architecture, which promotes reusable UI elements and seamless data handling. React's synergy with MongoDB, combined with its ability to efficiently render dynamic data, made it an ideal choice for building the user interface.

6.4 Architectural Design

This section describes the workflow of the entire architecture, analyzing in detail the major components that form the core of ArchiGPT and their functionalities.

6.4.1 API-Handler

The API-Handler serves as the critical bridge that facilitates communication between ArchiGPT and the OpenAI API. It hosts a Flask server on port 10001, providing the following routes:

- */assistant*: Provides several APIs for managing assistants, as described in subchapter 6.5, including their creation, deletion, and retrieval. For each of these APIs, an *AssistantOrchestrator* is instantiated, which prepares the request to be sent to OpenAI, attaching all necessary information. For instance, when creating a new assistant, the *AssistantOrchestrator* attaches the *instructions.txt* file, which contains the prompt for the new assistant. The following APIs offered by API-Handler call the OpenAI API, as described in Chapter 3.4.3 (**all references refer to this chapter**):
  - *creationHandler*: Handles the creation of an assistant using OpenAI API *[1.1]* as shown below:

```
assistant = current_app.config['CLIENT'].beta.assistants.create(
    instructions=instructions_text,
    name=assistantObj.name,
    tools=assistantObj.tools,
    model=assistantObj.model,
    tool_resources=tool_resources,
)
```

  - *getAssistant*: Retrieves an assistant using OpenAI API *[1.3]*.
  - *getAssistantList*: Retrieves a list of all assistants using OpenAI API *[1.4]*.
  - *assistantDelete*: Deletes an assistant using OpenAI API *[1.5]*.
- */thread*: Provides several APIs for managing threads, messages, and runs, as described in subchapter 6.5, including their creation, deletion, and retrieval. The following APIs offered by API-Handler call the OpenAI API, as described in Chapter 3.4.3:
  - *createThread*: Handles the creation of a thread using OpenAI API *[2.1]*.
  - *getLastMessage*: Retrieves the last message in a thread using OpenAI API *[3.2]*.
  - *createMessage*: Creates a message with content using OpenAI API *[3.1]*.
  - *createRun*: Creates a run to initiate a conversation with an assistant using OpenAI API *[4.1]*.
- */interrogation*: Provides the core API that manages the entire assistant lifecycle, from creation to the retrieval of generated messages, using the previously mentioned APIs. This API

is essential for generating any document and is repeatedly used throughout the ArchiGPT workflow. It deletes the existing assistant and creates a new one, including a new thread, message, and run. Once the assistant generates a response, this API creates a JSON object from the response and sends it back to the requester.

External OpenAI Communication To communicate with the OpenAI API, we use an interface that builds a client with the appropriate API keys, connecting to our OpenAI account, which handles billing and permissions. Additionally, the OpenAI user interface provides an overview of the created assistants, attached documents, request history, usage information, costs, and API key management.

```
OPENAI_API_KEY = os.environ["OPENAI_API_KEY"]
CLIENT = OpenAI(api_key = OPENAI_API_KEY)
```

6.4.2 Backend

The Backend microservice serves as the core of ArchiGPT, mediating between frontend requests 6.4.3 and API-Handler responses 6.4.1. It hosts a Flask server on port 5001, offering the following routes:

- */db*: Provides various APIs for managing and testing connections with MongoDB.
- */project*: Offers several APIs for managing projects and their associated documents, including their creation, deletion, and the retrieval of two key documents:
  - **Status**: This document tracks the generation phase of the entire system, including the containers and microservice assistants involved in the project's generation. The generation phase is categorized into three types:
    - *"OK"*: The assistant has been successfully called and has generated the message, completing its assigned task.
    - *"NEXT"*: This assistant is next in line to be called in order to continue the project's generation process.
    - *"NO"*: This assistant cannot be called yet, as certain prerequisites involving previous assistants must be satisfied before proceeding.

    This document is primarily used by the frontend to visualize the generation status updates of a project for the user.

```
{
    _id: ObjectId('67069691cb4ca255269c381f'),
    type: 'status',
    data: {
        system: [
            {
                name: 'Container Design',
                status: 'OK'
            },
            {
```

```
            name: 'User Interaction Analysis',
            status: 'OK'
        }
    ],
    containers: [
        {
            name: 'User_Authentication',
            ContainerDescriptionGenerator: 'OK',
            ContainerSpecificationGenerator: 'OK',
            MicroServices: 'OK',
            services: [
                {
                    name: 'auth',
                    description: 'OK',
                    ServiceSpecificationGenerator: 'NEXT',
                    ServiceEndpointGenerator: 'NO'
                }, ...
```

– **System**: This document contains the project's user stories, descriptions, and the initial messages from the first two system assistants.

```
{
    _id: ObjectId('67069691cb4ca255269c381e'),
    type: 'system',
    data: [
        {
            name: 'userstories',
            message: '1) As a User, I want to be able ...'
        },
        {
            name: 'description',
            message: 'OneSport is dedicated to offering up-to-date and...'
        },
        {
            name: 'Container Design',
            message: 'CONTAINERS:\n- CONTAINER NAME: User_Authenticat...'
        },
        {
            name: 'User Interaction Analysis',
            message: 'CONTAINERS:\n- CONTAINER NAME: User_Authenticat...'
        }
    ]
}
```

This document is primarily used during the second phase (container phase) of the generation process, as described in subchapter 6.2.4.

- */generation*: Provides the main generation APIs that handle the prompt chaining implementation, as previously described in subchapter 6.2.4:
  - *generateSystem*: Manages the first phase (system phase) of prompt chaining, handling and updating documents in MongoDB. The following figures illustrate the complete workflow of function calls between the Backend and API-Handler during the execution of the first two assistants, as described in 6.5.1.

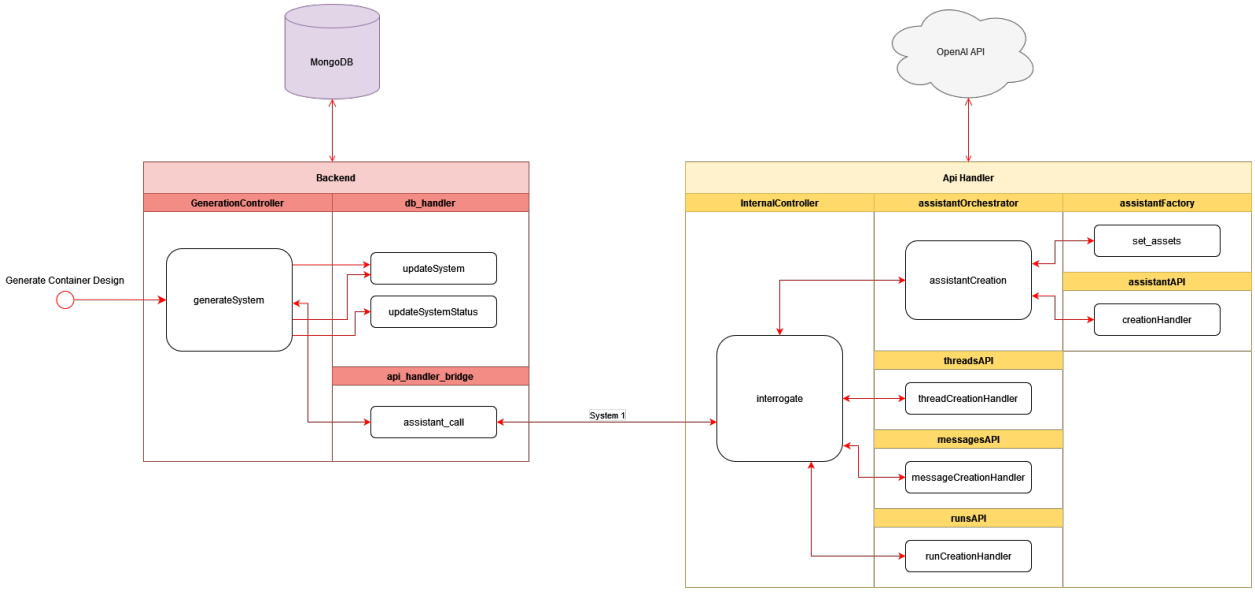

Figure 6.2: System 1 Assistant Workflow

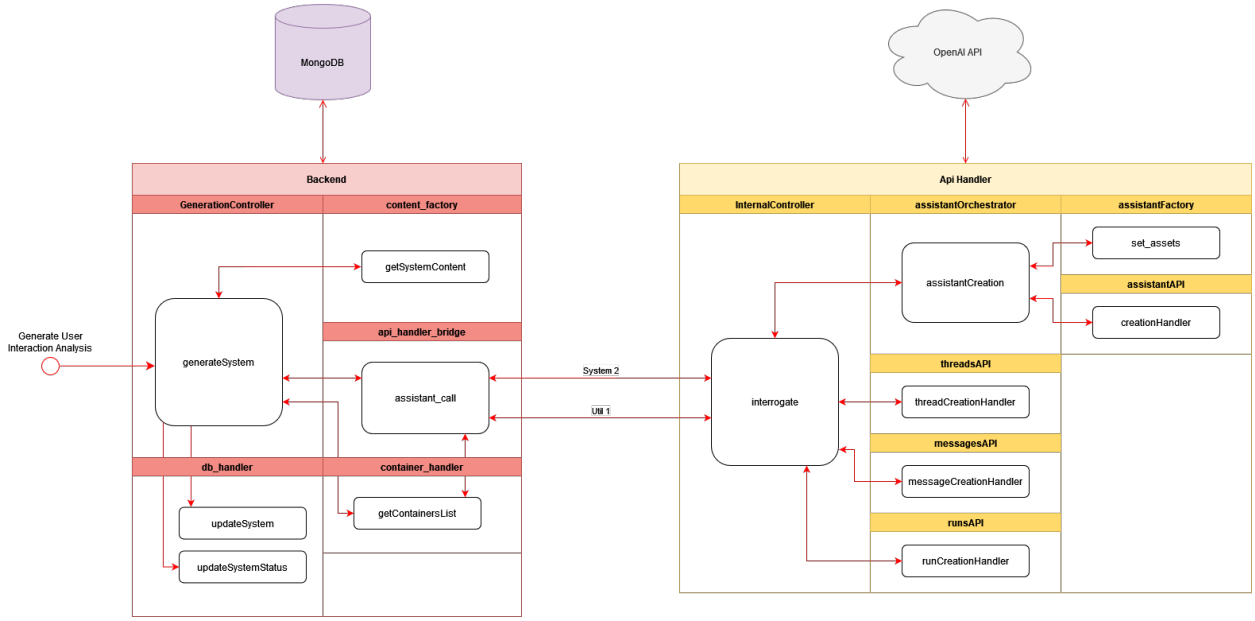

Figure 6.3: System 2 Assistant Workflow

– *generateContainer* : Manages the second phase (container phase) of prompt chaining, handling and updating documents in MongoDB. The following figures illustrate the complete workflow of function calls between the Backend and API-Handler during the execution of the three container assistants, as described in 6.5.1.

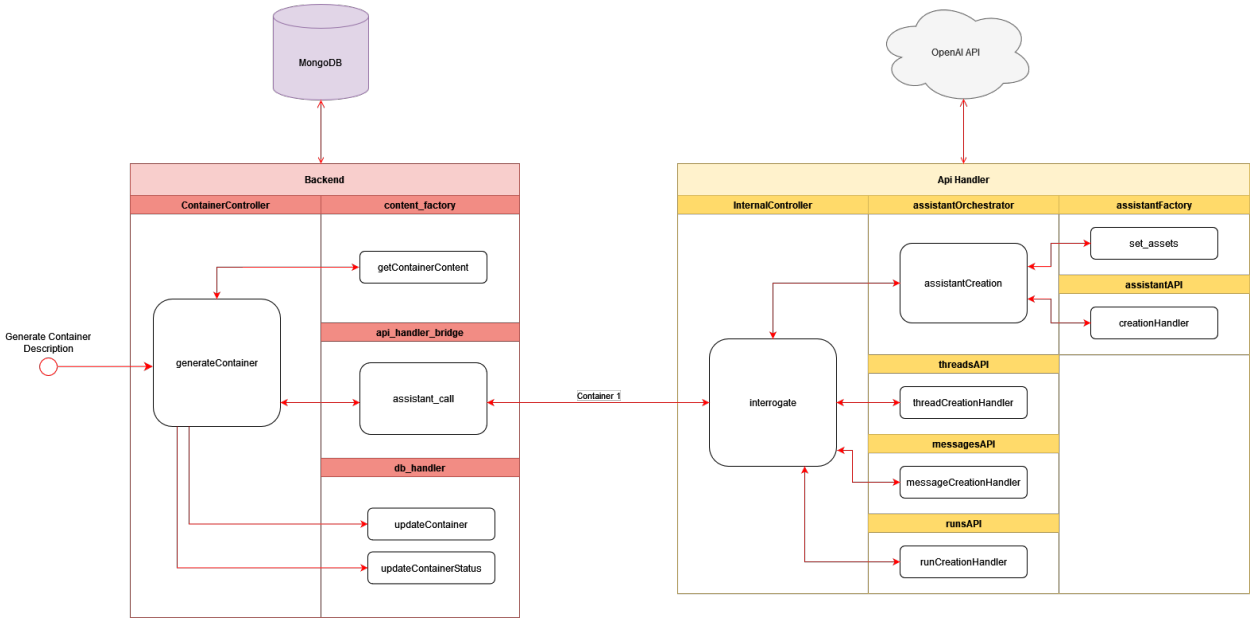

Figure 6.4: Container 1 Assistant Workflow

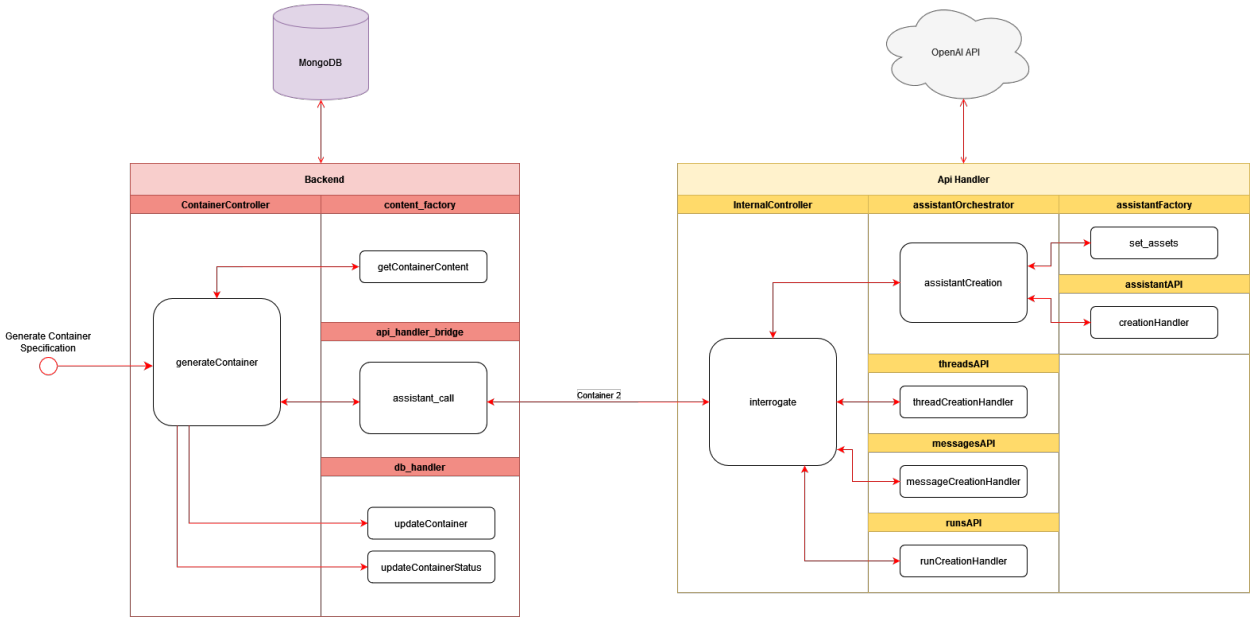

Figure 6.5: Container 2 Assistant Workflow

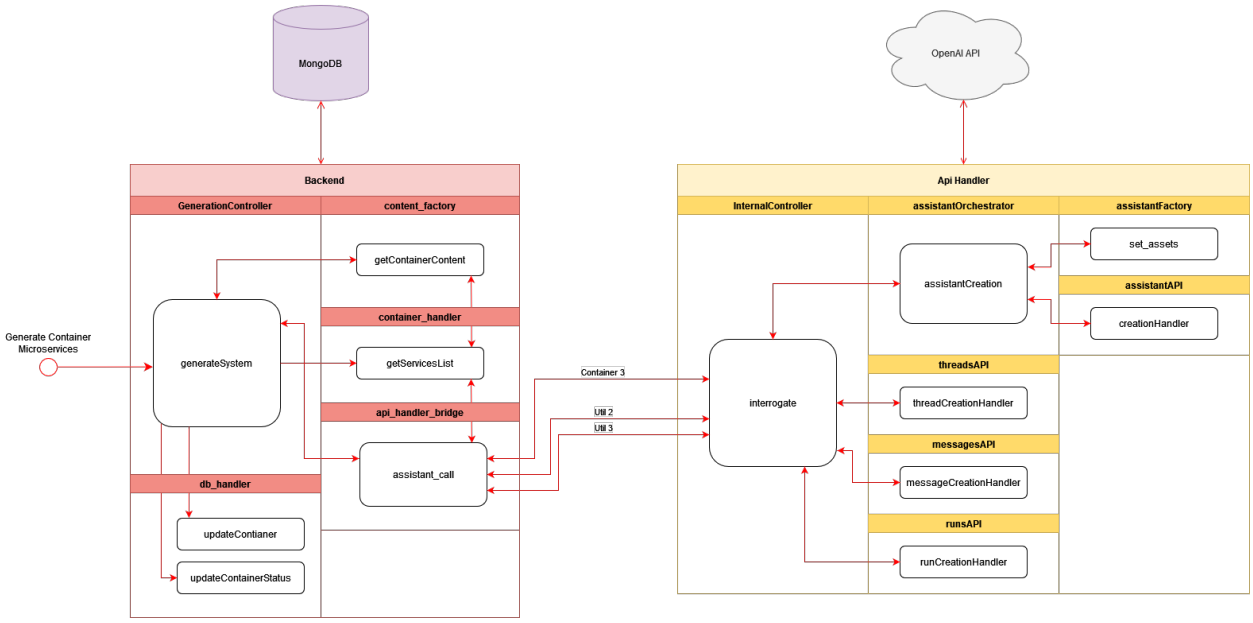

Figure 6.6: Container 3 Assistant Workflow

– *generateService* : Manages the third and last phase (service phase) of prompt chaining, handling and updating documents in MongoDB. The following figures illustrate the complete workflow of function calls between the Backend and API-Handler during the execution of the three service assistants, as described in 6.5.1.

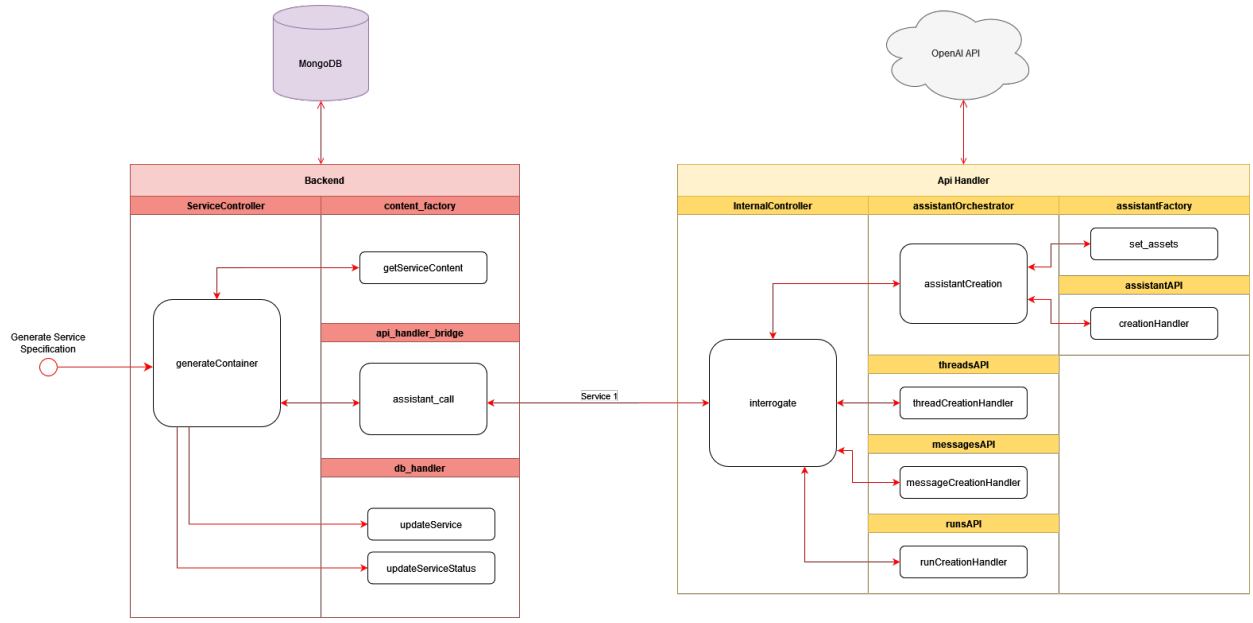

Figure 6.7: Service 1 Assistant Workflow

– *getContainer* : Provides the retrieval of the document that holds all information generated for a container (each container has his own document). An example is shown below.

```
{
_id: ObjectId('670696becb4ca255269c3820'),
type: 'container',
data: {
    name: 'User_Authentication',
    ContainerDescriptionGenerator: 'DESCRIPTION: \nThe User_Authentication...',
    ports: '20000:20100',
    userstories: '1) As a User, I want to be able to signup in order to ha...',
    services: [
    {
        name: 'auth',
        type: 'backend',
        description: 'The microservice manages user signup, login, and...',
        port: '20000',
        ServiceSpecificationGenerator: '# Output: #\nTECHNOLOGICAL SPECI...',
        ServiceEndpointGenerator: 'ENDPOINTS: [{"Method":"POST","URL":"/auth...'
    ...
```

– *regenerateContainer*: Provides an additional feature for re-generating a container assistant document, primarily used by the user through the frontend.

– *regenerateService*: Provides an additional feature for re-generating a service assistant document, primarily used by the user through the frontend.

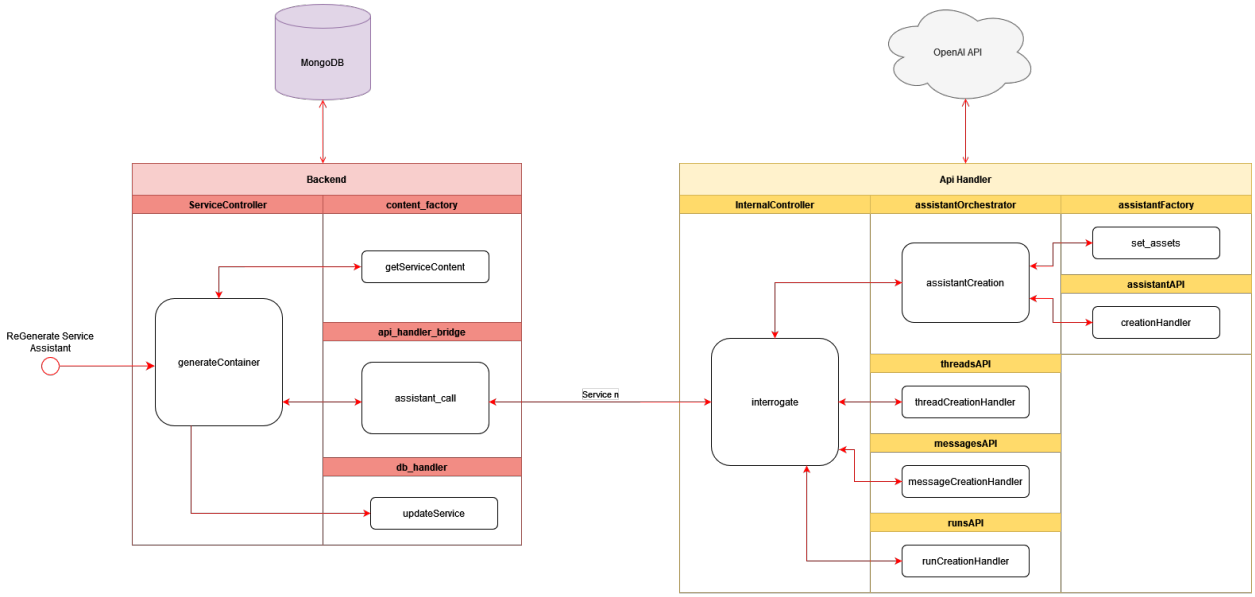

Figure 6.8: Regeneration feature Workflow

6.4.3 Frontend

The frontend microservice, developed using the React framework, primarily serves the user by providing visualization and interaction capabilities with ArchiGPT through two main pages:

- **Home.js**: The home page allows basic functionalities for managing projects, such as creation, deletion, and a link to redirect the user to the project's generation view. Moreover, using the green button at the bottom right corner of a project, the user can download the project in a JSON format for the Metric Evaluation described in 8.2. Instead, in the left side of the page, the user can fully generate a pre-configured *Ready-To-Generate Project* in one click. The "Generate All Projects" option is mainly used for generating Run.json files for the evaluation step described in details in subchapter 8.

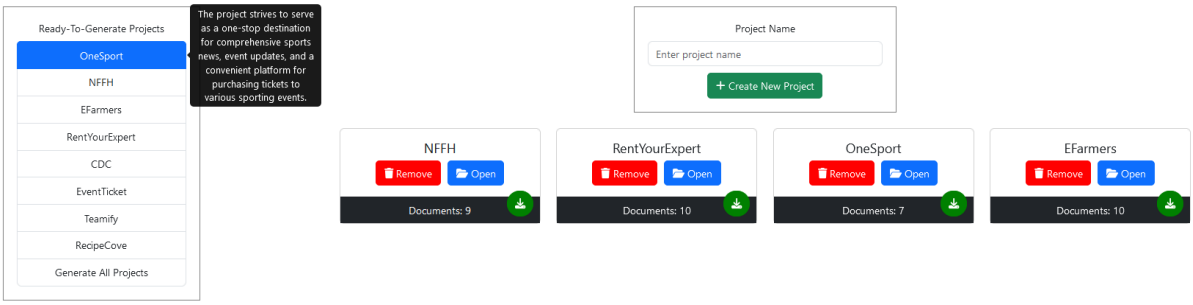

Figure 6.9: HomePage.js View

- **ProjectOverview.js**: The project overview page is the core interface of ArchiGPT, displaying the project's generation steps. It enables the user to initiate the process by submitting user stories, which trigger the creation of the first assistant, followed by a sequential waterfall generation of other assistants.

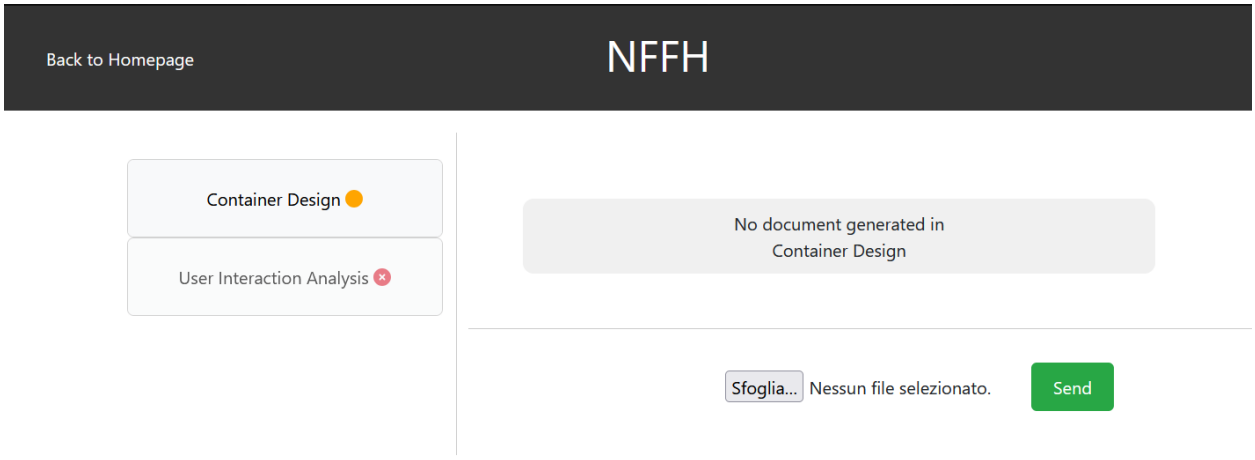

Figure 6.10: ProjectOverview.js View of a new project

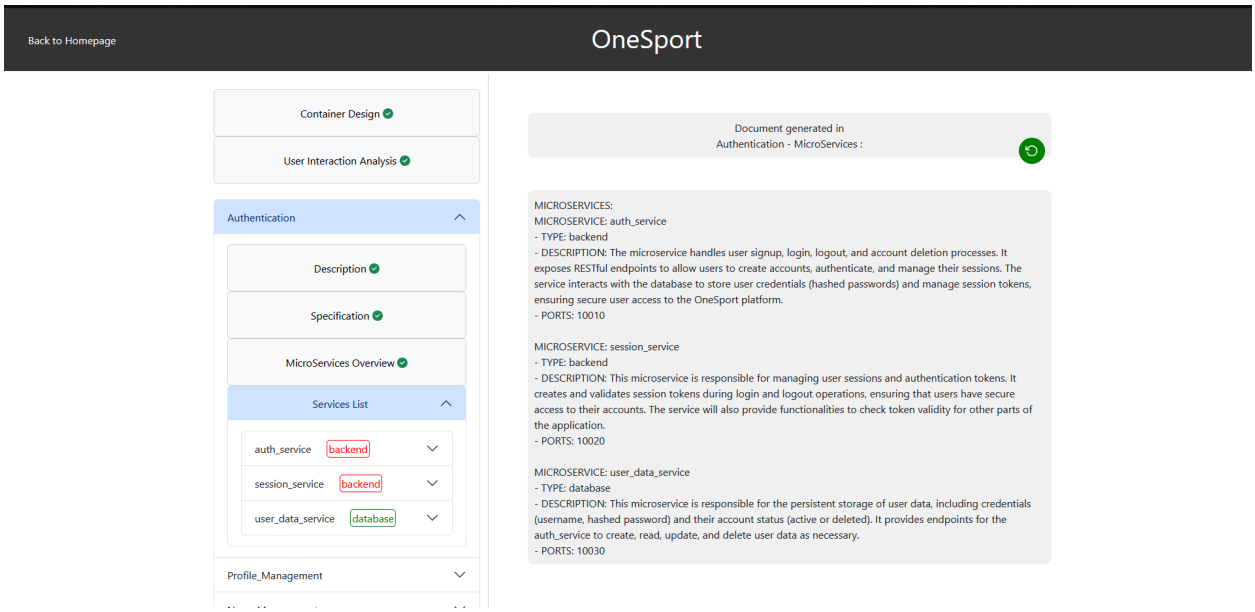

Figure 6.11: ProjectOverview.js View during generation process

6.4.4 Metric-Handler

The Metric-Handler microservice provides the functionality required by Archi Metric for validation, as described in chapter 8. Its primary purpose is to take as input a project, selected from the student projects, that needs to be validated, along with the number of attempts (i.e., the number of projects to be generated) in order to produce a consistent and reliable collection of projects for metric analysis and scoring. The output is a JSON format containing these projects, which is then passed to Archi Metric for further analysis and metric calculation.

After receiving the input, each project is processed according to the generation schedule, invoking the appropriate assistants. The project is generated with all container and microservice features ready for evaluation. Meanwhile, the JSON object is incrementally built, step by step, as the generation process progresses. An example of the JSON output is shown below:

```
{
"containers": [
    {
        "name": "authentication_management",
        "services": [
            {
                "endpoints": [
                    {
                        "method": "POST",
                        "url": "/auth_backend/signup",
                        "userStoryIndex": [1]
                    },
                    {
                        "method": "POST",
```

```
                            "url": "/auth_backend/login",
                            "userStoryIndex": [2]
                        }...
                    ],
                    "name": "auth_backend",
                    "type": "backend"
                },
                {
                    "name": "auth_storage",
                    "type": "database"
                }...
                }
            ],
            "userStories": [1,2,3,4,5,6]
        }...
        {
            "name": "frontend",
            "services": [
                {
                    "name": "web_app",
                    "pages": [
                        {
                            "name": "Homepage",
                            "userStoryIndex": []
                        }...
                    ],
                    "type": "frontend"
                }...
    ],
    "name": "OneSport"
}
```

6.5 Assistants

6.5.1 Description and Overview

Assistants in ArchiGPT are subdivided into 3 main categories, where each category represents a different layer of the system design process.

The system layer is the the first one to be executed. It takes as input the system description and user stories and generates the container architecture of the system. Here the first step from real world requirements to system specifications is taken: the user stories are categorized based on their scope and the container structure is defined. Moreover, the user interface of the system to be designed is taken into consideration: while the user stories may not define precisely how the users should interact with the system, the second assistant of this layer defines if a frontend container is

needed and it is eventually added to the containers' design.

Then the container layer is executed for each container described by the assistants of the first phase. At first each container is defined in terms of purpose and business logic, analyzing what are the features fulfilled by the container and how it will act. After that, a persistance evaluation is performed, analyzing the eventual need of the container to store data, the type of data to be stored and the actions performed on the database. Moreover, the assistant analyzes the eventual need of the container to connect to external services to complete its actions. For example, a container that needs to perform payments should have to connect to financial APIs to complete the transaction. Finally, the container's microservices are defined, based on the container's features, actions and constraints. Each microservice is defined by its type, exposed ports and a brief description of its expected behaviour and purpose.

Lastly, the service layer is executed for each microservice defined in the last step of the previous layer. At first, the technological specifications and the service architecture are defined. The technological specification section describes what technologies the microservice uses, what are the frameworks employed (if any) and eventual libraries needed to perform the microservice's tasks. The service architecture section describes the internal design of the service: it describes the service's layers, directory structure and design patterns to be employed. Then, a deeper analysis on backend and frontend microservices only is performed. For backend services, the analysis concentrates on the service's endpoints, defining for each endpoint the URL, HTTP method, a brief description and the user stories the endpoint fulfills. For frontend services, the eventual pages are analyzed, descibing their related URL and eventual fulfilled user stories.

6.5.2 Prompt Structure

Prompt Frameworks Overview

Prompt frameworks are structured methodologies that guide the formulation of prompts to Large Language Models (LLMs) to elicit desired outputs effectively and efficiently. These frameworks encapsulate best practices and design principles for crafting prompts that align with specific objectives, whether generating creative content, answering factual questions, or performing complex tasks such as code generation or translation. By providing a systematic approach to prompt construction, prompt frameworks bridge the gap between human intentions and the model's interpretative capabilities, enhancing the overall quality and reliability of the model's responses.

The structure of prompts based on prompt frameworks typically comprises several essential components designed to optimize communication with the LLM. One fundamental element is the context, which situates the prompt within a particular scenario or provides background information relevant to the task. This could involve setting up a role-playing scenario where the model adopts a specific persona or expertise level. Another critical component is the explicit instruction or task description, which clearly delineates the action the model is expected to perform. Clarity and specificity in instructions are paramount to avoid ambiguity and misinterpretation. Additionally, including examples or demonstrations can further guide the model by showcasing the desired format, style, or content of the expected response. These examples act as anchors, helping the model align its outputs with the user's expectations. Constraints and specifications, such as length limits, stylistic guidelines, or content boundaries, are also integral to the prompt structure, as they refine

the model's outputs to meet precise requirements.

The advantages of using a prompt framework are multifaceted. Firstly, it enhances the effectiveness of LLM interactions by promoting precision and clarity in communication, leading to more accurate and relevant outputs. A well-designed prompt reduces the likelihood of the model generating off-topic or nonsensical responses, saving time and resources. Secondly, prompt frameworks contribute to consistency and reproducibility in the model's outputs, which is particularly valuable in professional and research settings where predictable performance is essential. By standardizing the prompt construction process, users can achieve more uniform results across different interactions and use cases. Thirdly, prompt frameworks democratize the use of LLMs by providing users, regardless of their technical expertise, with accessible guidelines to harness the full potential of these models. This inclusivity fosters broader adoption and innovation in leveraging LLM technologies. Furthermore, prompt frameworks play a crucial role in mitigating ethical concerns associated with LLM usage. By incorporating constraints and guidelines that address issues such as bias, inappropriate content, or misinformation, prompt frameworks ensure that the model's outputs align with ethical standards and societal values. Lastly, employing prompt frameworks enhances the adaptability and scalability of LLM applications. As new use cases emerge, prompt frameworks can be extended or modified to accommodate evolving requirements, facilitating continuous improvement and customization of LLM deployments.

Prompt Structure in ArchiGPT

The structure of assistants' prompts in ArchiGPT follows a precise framework, composed of 4 different components:

- Context
- Task
- Instructions (optional)
- Output format
- Example

Context The context component of the assistants' prompts follows a precise schema to give the LLM the context it needs to perform the required task.

At first, this section assigns the role of Software Architect/Requirements Engineer to the assistant. Assigning a specific role to a language model within a prompt's context substantially enhances the relevance and quality of its responses. By defining a role, the prompt provides the model with a focused perspective and set of expectations. This guidance enables the model to tailor its language, tone, and content to align closely with the desired output, thereby improving its utility in the given context. Role specification reduces ambiguity by narrowing the scope of the model's vast knowledge base to a particular domain, which facilitates more precise and contextually appropriate information retrieval. Moreover, it helps maintain consistency throughout the interaction, as the model adheres to the established persona, minimizing irrelevant or off-topic responses. This approach leverages

the model's ability to emulate various expertises, enhancing its adaptability and effectiveness across diverse applications.

Then, a generic task is given in the context. The assistant is instructed on the generic designing task it is carrying on (designing a software architecture) and the principles it is following (dockerization and microservice architecture).

```
You are a Software Architect/Requirements Engineer.
You are designing an Software architecture from scratch.
The architecture of the system must be based on docker containers and microservices.
```

After that, the assistant is provided with a list of all the elements provided in the user prompt.

Task The task component of the assistants' prompts gives the assistant the instructions on the task to be performed. For some assistants, the task component is very minimal, giving only a brief description of the task to be executed. Other assistants, like the "Container1" and the "Service1" assistants, have to generate a more complex analysis, divided in 2 or more sections. For these assistants, the task component of the assistant prompt describes precisely how each section must be generated.

This is an example of a simple task component from the "Service2" assistant:

```
Describe the service's endpoints in terms of HTTP Method, URL, smart
description and satisfied user stories
```

This is a more complex task component from the "Service1" assistant:

```
Describe the service in terms of technological specifications and architecture
1) TECHNOLOGICAL SPECIFICATION
    - Evaluate the technological stack
    - Specify the language and framework and the eventual libraries to be used
2) SERVICE ARCHITECTURE
    - Describe the service architecture in terms of patterns adopted
```

Instructions The instructions component of the assistants' prompts is only present in the "System phase" assistants. These assistants need to craft the system containers, with the user stories they fulfill and a brief description of their purpose and structure. These tasks are vague and is difficult to obtain a satisfying outcome based on the generic task description. To tailor the LLM response to our needs, a list of instructions guides the LLM's reasoning to execute a series of tasks in order to reach the desired outcome.

The following is the instructions component of the "System1" assistant, that has to categorize the user stories and generate the system's container structure:

```
1) Propose categories for the user stories.
These categories can be based on users, actions, behaviour etc.
2) Based on the generated categories, design containers that match the categories
and fulfill the user stories
3) For each User Story, define a container that fulfills it (it can be one
```

```
container, more than one container or it can be unassigned)
4) Describe briefly the purpose of each container and the features it accomplishes
5) Define a ports range for each container
```

Output format The output format component of the assistants' prompts describes briefly the expected output format of the assistant. The assistants generate one of the following responses format:

- Nested list
- Text, in some cases with more sections
- Json

The following is the output format of the "System1" assistant, that generates a nested list:

```
List of the containers in the following form:
- CONTAINER NAME: "container"
    - DESCRIPTION: "container description"
    - USER STORIES:
        - "user story 1"
    - PORTS: "ports range start : ports range end"

If there are unassigned User Stories, put them in a separate list
- UNASSIGNED:
    - "user story unassigned"
```

Example The example component of the assistants' prompts provides one-shot learning to the LLM. It comprises an example of an input for the specific assistant and the expected output. One-shot learning implementation in ArchiGPT will be discussed extensively in the following subchapter.

6.5.3 Prompting Techniques

One-shot and Few-shot Prompting Techniques

One-shot prompting One-shot prompting includes a single example of the task within the prompt. This example serves as a reference point, guiding the model towards the expected output format and style. By providing one example, users can significantly improve the model's performance on tasks that may be ambiguous or have multiple valid responses. The single example reduces ambiguity and helps the model align its output with the user's expectations. One-shot prompting strikes a balance between prompt length and performance enhancement, making it suitable for tasks where providing multiple examples is impractical.

Few-shot prompting Few-shot prompting extends the concept of one-shot prompting by including several examples within the prompt. By supplying multiple instances of the input-output relationship, users provide richer contextual information that the model can leverage to produce more accurate and reliable responses. Few-shot prompting is particularly beneficial for complex tasks or when the model needs to generalize from nuanced patterns. The additional examples help the model understand the subtleties of the task, such as stylistic preferences or specific formatting requirements. However, this approach increases the length of the prompt, which may be a limitation due to token restrictions in some models.

In few-shot prompting, the selection of examples is critical. The examples should be representative of the task and cover a range of scenarios the model is expected to handle. By carefully curating these examples, users can guide the model more effectively, improving performance on tasks that would otherwise be challenging with no-shot or one-shot prompting.

One-shot prompting in ArchiGPT

In ArchiGPT prompts, one-shot prompting technique has been selected. Few-shot prompting has been deemed unfeasible due to the nature and length of the prompts: having more than one example would have extended the assistant prompt and, in some situations, could have limited the output quality due to token restrictions of the model. No-shot prompting, on the other hand, would not specify in a precise way the expected output format to the model, leading to imprecisions that would have led to unusable responses in a prompt-chaining context.

Projects from the Projects Dataset are unfeasible to one-shot prompting: the real-world context they describe is too large and the containers of the expected systems should be heavily interconnected and with complex business logic. Their usage as example for ArchiGPT would led to confusionary responses and excessive token usage.

Instead, a simple project has been designed and described to be used as an example for one-shot prompting. This example project describes a system that realizes a simple blog.

The following is the description and user stories of the example project:

```
DESCRIPTION:
This blog platform allows users to register, log in, and log out securely.
Once logged in, users can read articles, view comments on each article, and
contribute their own comments. The platform features a user-friendly interface
that makes navigation and interaction straightforward. Articles are displayed
in a list format on the homepage, with individual article pages showing the full
content and associated comments. The system ensures that all comments are
attributed to registered users, promoting accountability and constructive discussions.

USER STORIES:
1) As a new user, I want to register an account so that I can log in and
interact with the blog.
2) As a registered user, I want to log in to my account so that I can access
and interact with the platform.
3) As a logged-in user, I want to log out of my account so that I can securely
```

```
end my session.
4) As a user, I want to read a list of articles on the homepage so that I can
choose which articles to read in full.
5) As a user, I want to click on an article title to read the full content
so that I can engage with the material.
6) As a user, I want to see all comments associated with an article so
that I can read others' opinions and insights.
7) As a logged-in user, I want to comment on an article so that I can share
my thoughts and engage in discussions.
```

For the example project, a basic container structure has been designed to show the model how the categorization of user stories has to be carried on and the output format of the first assistant.

```
CONTAINERS:
- CONTAINER NAME: Authentication
    - DESCRIPTION: Deals with registration, login, session management, and security
    aspects to prevent unauthorized access
    - USER STORIES:
        1) As a new user, I want to register an account so that I can log in
        and interact with the blog.
        2) As a registered user, I want to log in to my account so that I can
        access and interact with the platform.
        3) As a logged-in user, I want to log out of my account so that I can
        securely end my session.
    - PORTS: 10000:10100
- CONTAINER NAME: Articles
    - DESCRIPTION: Deals with articles search and read
    - USER STORIES:
        4) As a user, I want to read a list of articles on the homepage so that
        I can choose which articles to read in full.
        5) As a user, I want to click on an article title to read the full
        content so that I can engage with the material.
        6) As a user, I want to see all comments associated with an article so
        that I can read others' opinions and insights.
        7) As a logged-in user, I want to comment on an article so that I can
        share my thoughts and engage in discussions.
    - PORTS: 11000:11100
```

The authentication container has been chosen to serve as an example for the container phase assistants. The following is a summary of the examples for the assistants.

```
DESCRIPTION:
The Authentication container is responsible for handling registration of the users,
login and logout. It also manages credentials storage and retrieval, along with
the authentication tokens' management
```

```
PERSISTANCE EVALUATION
The container needs to store credentials in order to manage registration and login.
The credentials are tuples username, passsword.
The container also needs to store authentication tokens.

EXTERNAL SERVICES CONNECTIONS
Based on the container's behaviour and purpose, there is no need for the container
to connect to external services.

MICROSERVICES:
MICROSERVICE: auth
- TYPE: backend
- DESCRIPTION: The microservice handles registration, login and logout operations
for the users. It exposes endpoints to access these operations. For the registration
operation, the microservice interacts with the database to ensure that the user is
not already registered. For the login operation, the microservice interacts with
the database to validate credentials, then produces a token, stores it in the DB
and returns it to the user. For the logout operation, the microservice removes
the token from the database. The microservice also exposes enpoints for other
containers/microservices to validate authentication tokens.
- PORT: 10010

MICROSERVICE: storage
- TYPE: database
- DESCRIPTION: The microservice stores credentials in the form username-password.
It also stores authentication tokens.
- PORT: 10020
```

The auth microservice has been chosen to serve as an example for the SERVICE phase assistants. The following is a summary of the examples for the assistants.

```
TECHNOLOGICAL SPECIFICATION
The microservice uses NodeJS as its main programming language, with express as a
backend web application framework for building RESTful APIs.

SERVICE ARCHITECTURE
The service implements a clear subdivision of roles, with a file describing routes,
a controller to manage HTTP requests and a handler to interact with the database.

ENDPOINTS:
[{ "Method" : "POST", "URL" : "/auth/register", "Description" : "Create a new
user using the inserted information, return a message that confirm the action",
"UserStories" : [1] },
```

{ "Method" : "POST", "URL" : "/auth/login", "Description" : "Handle the login for the current user, generating as response a token", "UserStories" : [2] },
{ "Method" : "POST", "URL" : "/auth/logout", "Description" : "Remove the token for the logged user, and execute the logout from current session, return a confirmation message", "UserStories" : [3] }]

Chapter 7

Dataset

7.1 General Overview

This chapter describes the development process of the Archi Dataset. The final dataset is composed by 8 different projects, each characterized by the following files:

- A brief description of the system to be developed
- User stories of the system
- Source code produced by the students
- Technical documentation of the students' developed system in Project Standard format
- Datametrics JSON file

The chapter covers all the dataset creation phases, starting from a detailed description of the academic context in which the projects were developed and their general characteristics. Then, the selection phase is described, analyzing the selection parameters, the projects' status and the existing documentation for them. The next section describes the selected projects' source code from a technical point of view, analyzing how the microservices architecture is applied and their design choices. The following sections descibe the processes of cleaning the projects'user stories and description and the crafting of new technical documentation in Project Standard. Finally, the Datametrics format is described.

7.2 Student's projects description

The initial projects dataset consisted of 28 projects realized by university students for the Laboratory of Advanced Programming course in the academic years 2021/2022 and 2022/2023.

7.2.1 Course Description

The course "Software Engineering - Laboratory of Advanced Programming" is taught by professor Massimo Mecella in La Sapienza University of Rome for master students in Engineering in Computer Science. The Software Engineering part covers topics that vary from architectural patterns and concepts (like microservices architecture and middleware) to software development metodologies

and estimations (Agile and COCOMOII). The Laboratory of Advanced Programming instead covers more pratical topics: gives to the students an overview of containerization, middlewares application and webservices technologies, along with practical examples and laboratories on those topics.

7.2.2 Projects Structure

For the Laboratory section, students have to design, develop and present a system based on microservices architecture. The project must be composed of:

- A description of the system purposes and functionalities, along with the intended users
- Users stories for the system
- The source code of the system, along with all the docker-related files
- Technical documentation

7.2.3 Projects Overview

A technical analysis was conducted on a collection of student projects from the Software Engineering course to evaluate their application of microservices architecture. This analysis focuses on the architectural choices, implementation technologies, user story development, and the quality of technical documentation within these projects.

Project Scope and Objectives

The majority of the projects aimed to develop websites of varying complexity, utilizing microservices to modularize system functionalities. Each project provided a description of the system's purposes and functionalities, along with the identification of intended users. The systems ranged from simple informational websites to complex platforms offering interactive services.

Use of Docker and Containerization

All analyzed projects employed Docker for containerization, showcasing an understanding of modern deployment practices essential in microservices architecture. Each project included one or more Dockerfiles and one or more docker-compose files. Furthermore, some projects extended their use of Docker to fulfill the requirements of the Dependable Distributed Systems course. These projects deployed replicas of database microservices to ensure availability and integrity in case of failures.

Employed Technologies

Backend Technologies Backend microservices were predominantly developed using Python and Java. Python was frequently chosen due to its widespread familiarity among students, allowing for rapid development and ease of comprehension. Java-based projects often utilized the Spring framework components, which were introduced during laboratory sessions as real-world examples.

Frontend Technologies Frontend development varied across the projects. The majority opted to serve custom HTML files, providing straightforward user interfaces without the complexities of modern frontend frameworks. However, some students chose to implement microservices using frontend frameworks, predominantly React. The adoption of React indicates an effort to create dynamic and responsive user interfaces, aligning with contemporary web development practices.

Database Choices Most projects utilized SQL Database Management Systems (DBMSs) for data persistence, reflecting traditional relational database models commonly taught in academic settings. A few students explored alternative database technologies by implementing MongoDB, a NoSQL database. The use of MongoDB indicates an understanding of different data modeling approaches and their applicability based on system requirements.

User Stories Analysis

User stories were present in all but two projects as a means of capturing system requirements from the user's perspective. However, several issues were identified in their formulation and completeness. Some projects included only a few user stories, resulting in systems with limited features available to the user. This minimalistic approach could hinder the system's ability to meet comprehensive user needs.

Additionally, some projects did not adhere to standard templates or formats for writing user stories, leading to confusion and ambiguity in understanding system requirements. The lack of a consistent structure made it challenging to ascertain the intended functionalities and user interactions. In certain cases, critical user actions were missing from the user stories, indicating incomplete requirement analysis and potential oversights in system design.

Technical Documentation

Technical documentation was found to be insufficient or overly brief in the vast majority of projects. While many students summarized the system's general architecture and described the technologies used, detailed documentation regarding the realization of microservices and the architectural decisions made was often missing. This omission hinders the ability to understand the system's inner workings and the rationale behind specific design choices.

Furthermore, most projects lacked descriptions of the system's endpoints and the database structures. Only two projects presented comprehensive, albeit scattered, technical documentation. These projects demonstrated a stronger grasp of the importance of thorough documentation in software development.

7.3 Projects' selection

To select the projects to be included in the final dataset, we set up standards about user stories, system realization and documentation.

7.3.1 User Stories and System Description

The system description and user stories are the starting point for designing the system. They provide the only source of the system's expected features and behaviour. They also represent the real-world base of the system's architectural design choices.

We decided to discard all the projects whose description and user stories describe a system that have a small number of features. For that kind of system, a classical monolithic architecture would be a more sensible choice, given the simplicity of the tasks to be implemented.

We discarded 5 projects (of 28 total projects) for having not sufficient user stories and description:

- 2 of them were entirely missing user stories, rendering them useless
- 3 of them were describing a small system, not coherent with the scope of this research

7.3.2 System Realization

Our primary aim was to build a dataset that could validate the projects generated by ArchiGPT through the metrics we established. To achieve this, we set guidelines for selecting the student projects that had the greatest potential to demonstrate these metrics.

With 23 projects remaining, we followed a structured schedule for each one:

1. Thoroughly analyzed the source code and related documentation to determine whether the project could meet our objectives and be a candidate for our dataset.
2. Discarded projects that had poor implementation and conceptualization, including monolithic architectures, and those lacking commitment and creativity.
3. Discarded projects that were entirely unsuitable for our purposes, particularly those with low-quality coding, weak architectural designs, or inappropriate technology usage.
4. Discarded projects that were overly focused on building distributed systems, such as those relying heavily on node replication and the extensive use of RabbitMQ and other queue management technologies. This was primarily due to the fact that the project for the "Software Engineering - Laboratory of Advanced Programming" course was intertwined with another university course, "Dependable and Distributed Systems," which led students to design projects with a strong emphasis on distributed systems, lacking a deeper focus on containerization and architectural patterns.
5. Analyzed the implementation of containers and microservices from an architectural perspective, classifying the potential types of microservices for further analysis in the design of ArchiGPT Assistants as follows:
    - **Backend microservice**: A service related to the business logic of the container.
    - **Database microservice**: A service responsible for hosting the database system, managing data across the entire container.
    - **Frontend microservice**: A service responsible for deploying the frontend infrastructure.

- **Middleware microservice**: A service mainly used to facilitate communication between other microservices.
- **Hybrid microservice**: A service that combines both backend and frontend features within a single microservice.

Using this classification, we discarded projects that were entirely or partially composed of middleware or hybrid microservices, as they did not align with our metric-based goals.

After this filtering process, we selected 8 projects for our final dataset that met and satisfied our validation objectives.

7.3.3 Technical Documentation

All the projects (except 2) had a missing or useless technical documentation: there were no description of the microservices connections, architecture and function. They were missing endpoints documentation and code analysis. Also, there is no documentation about the correlation between the microservices and user stories or system features. Given the situation, the selection parameters do not involve documentation status.

7.4 Selected Projects

7.4.1 OneSport

System Overview

OneSport is a comprehensive sports-centric platform aimed at delivering real-time information, news updates, and an efficient ticketing system for sporting events. Its core objective is to enhance user engagement by offering a unified solution where sports enthusiasts can access the latest news, customize updates based on country preferences, and seamlessly manage event ticket purchases. The platform is designed to streamline the user experience, catering to the dynamic needs of modern sports fans. Core functionalities include account management (signup, login, profile settings), news personalization through favorite lists, and ticket management features, such as adding and removing tickets from collections. Additionally, users can engage in social interactions by sending, receiving, and managing friend requests. The intended users of the system are sports fans, ticket buyers, and individuals seeking a personalized and interactive platform for staying informed on athletic events and managing their sports-related interests.

Technical Analysis

Containers and Services Structure The containers and services structure of the OneSport system follows a microservices architecture designed to manage sports news, ticket purchases, and social interactions, encapsulated within individual, loosely-coupled services. The central container, OneSport, coordinates the core functionalities and connects the various microservices: frontend, news, authentication, tickets, and friends. Each service has its own dedicated responsibilities. The frontend microservice handles the user interface and is developed using ReactJS. It provides components for user interaction, such as login, profile management, and accessing sports news and tickets.

The backend services, including the news, authentication, tickets, and friends microservices, are built using Python with the Flask framework. These services expose RESTful APIs to manage various resources, such as user accounts, news content, ticket information, and social features like friend requests. Each microservice is isolated, handling specific data and logic independently, ensuring that changes in one service do not affect others. The system is containerized, and Docker ensures that each service is deployed and scaled efficiently.

Database Services The database services in the OneSport system are centralized in the postgres microservice, which employs PostgreSQL as its database management system. This microservice is critical to maintaining the integrity and persistence of user data, sports news, tickets, and social interactions. The database structure for each microservice is carefully designed to ensure efficient data storage and retrieval. For instance, the news microservice stores details about sports articles, including titles, authors, publication dates, and user-specific data such as saved news articles. Similarly, the authentication microservice manages user credentials, including names, email addresses, hashed passwords, and roles, providing secure user sessions through JWT tokens. The tickets microservice maintains ticket information, such as event names, dates, and user-specific collections. The friends microservice stores social interaction data, including friend requests, their statuses, and the list of friends for each user. The relational nature of PostgreSQL ensures that complex queries and transactions, such as retrieving a user's favorite news or filtering tickets by region, can be executed efficiently. Additionally, the system implements ORM via Flask-SQLAlchemy for smooth integration between Python code and the PostgreSQL database. The choice of PostgreSQL offers scalability, security, and support for advanced features like ACID compliance, which is essential for handling financial transactions such as ticket purchases.

Architectural Overview The architecture of the OneSport platform is based on a well-structured microservices approach, allowing each component to be developed, deployed, and maintained independently. Each backend microservice follows a layered architecture pattern, ensuring a clear separation of concerns. For instance, the news, authentication, tickets, and friends services each have controllers that handle business logic, models that define the data schema, and routes that manage HTTP requests. This design promotes modularity and maintainability, as each layer can be modified independently without affecting the overall system. The frontend microservice interacts with the backend via API calls. The user stories outline the interactions between frontend and backend services, covering functionalities like user registration, authentication, browsing news, purchasing tickets, and managing social features. Docker is leveraged to containerize the architecture, allowing each microservice to operate independently. This isolation promotes better scalability and enhances the system's fault tolerance, ensuring more robust and efficient operations across the services.

7.4.2 NFFH - Not Far(m) From Home

System Overview

Not Far(m) From Home is a web-based platform designed to facilitate direct interaction between local farmers and consumers, adhering to the "km 0" philosophy, which emphasizes the consumption of

locally sourced products. The platform allows farmers to register and post their fresh produce, while consumers can browse available items, reserve products, and schedule pickups at nearby agricultural companies. Core features include product listing and inventory management for farmers, personalized shopping experiences for consumers, and site maintenance tools for administrators. Consumers can view seasonal products, locate nearby farms, add items to their cart, and complete orders with pickup details. Farmers manage their inventory by adding, modifying, or removing products, ensuring accurate availability for their clients. Administrators oversee site operations, including user and product management, site maintenance, and user activity monitoring. The platform's intended users are local consumers seeking fresh produce, agricultural companies aiming to connect directly with customers, and administrators responsible for maintaining the site's functionality.

Technical Analysis

The containerized architecture of the "Not Far(m) From Home" platform is designed following microservices principles, with each container encapsulating distinct functionalities and responsibilities. The system architecture is composed of backend and frontend services, databases, and middleware components, allowing flexibility, scalability, and clear separation of concerns across the platform. Below, the architecture is detailed in terms of its container structure, database services, and an overall architectural overview.

Containers and Services Structure The system is divided into multiple containers, each serving a specific functional area of the platform. The main backend services include the Authentication, Client-BE, Farmer-BE, Order-BE, and Image-Server containers, while the frontend services are separated into Client-FE, Farmer-FE, and Admin-FE containers. Additionally, the APIGateway container acts as a middleware, routing requests to the appropriate services.

Authentication: This container manages all operations related to user registration, login, and session persistence. It is implemented using Python with Flask, and it relies on JWT (JSON Web Token) for secure token-based authentication. The authentication service handles the core security features across three user roles: clients, agricultural companies, and administrators. It provides APIs for login, signup, logout, and token validation.

- **Client-BE** This container manages user and company profiles, allowing operations such as viewing personal information and handling user data. The service is developed in Java with the Spring Boot framework, ensuring a robust and scalable architecture. It leverages a MySQL database to handle data persistence efficiently. The microservice architecture supports functionalities for both client and admin roles, enabling user management tasks such as adding, updating, and removing users.
- **Farmer-BE** Focused on agricultural companies, this container handles product-related operations such as adding, updating, and deleting produce. Similar to the Client-BE container, it is built using Java and Spring Boot and connects to a MySQL database to store product and farmer information.
- **Order-BE** The Order-BE container manages order processing for clients and provides administrative capabilities for viewing all orders placed on the platform. It facilitates the selection of pickup dates and tracks order fulfillment.

- **Image-Server** This container stores and retrieves images for the products listed on the platform. It does not require a persistent database, as it stores images directly in the virtual machine (VM) memory.
- **Frontend Containers** The Client-FE, Farmer-FE, and Admin-FE containers serve the front-end user interfaces for their respective user roles (clients, farmers, and administrators). Each container is developed using modern web technologies and does not require its own database. The front-end services communicate with the backend containers via RESTful APIs, delivering the appropriate data to the user interfaces.
- **APIGateway** This container acts as a centralized entry point for all incoming HTTP requests. It provides routing capabilities, directing requests to the correct microservices based on the requested URL. By centralizing service access, it strengthens both security and scalability, acting as a single entry point that manages the flow of requests across the system's distributed architecture.

Database Services The platform utilizes multiple MySQL databases to store data for different areas of the system. Each of the backend containers that require persistence is connected to a MySQL instance.

- **Client-BE Database** The primary responsibility of this database is to store and manage user and company profile data. It contains a table for client information with fields like name, email, and password. This database is accessed by the Client-BE microservice to provide user management features and by administrators to perform platform maintenance tasks
- **Farmer-BE Database** This database stores information about agricultural companies and their products. It includes tables for farmers, products, and areas (geographical regions) to support operations related to product management, agricultural company profiles, and platform administration.
- **Order-BE Database** This database is responsible for storing order-related data, including order IDs, consumer details, products ordered, and pickup dates. The Order-BE microservice interacts with this database to complete transactions and provide insights into order histories for both clients and administrators.

The databases follow a relational structure, which ensures consistency and integrity of the data across the platform. They are accessed via Java Persistence API (JPA) through the Spring Boot framework, facilitating smooth interaction between the backend services and the databases.

Architectural Overview The "Not Far(m) From Home" platform employs a microservices architecture, leveraging containerization to modularize different functionalities of the system. Each service runs in its own container, providing a clear separation of concerns and allowing the platform to scale each component independently.

The backend services are designed to handle different domains: authentication, user management, product management, and order processing. These services communicate with each other

primarily through RESTful APIs. For example, the Order-BE container interacts with the Farmer-BE container to check product availability before completing an order. The Authentication service ensures secure access across the platform through token-based authentication mechanisms.

The frontend services are decoupled from the backend, communicating only through API calls. This separation allows for better scalability and independent deployment of the user interfaces. Clients, agricultural companies, and administrators interact with the platform through separate front-end containers (Client-FE, Farmer-FE, Admin-FE), ensuring that the user experience is tailored to the needs of each role.

At the core of this architecture is the APIGateway. By routing all incoming traffic through this single entry point, the platform improves maintainability and security, as changes to the routing logic can be made in one location without affecting the individual services.

Overall, the system design of "Not Far(m) From Home" promotes scalability, maintainability, and security through a well-structured microservices approach. Each containerized service operates independently, with clear responsibilities, and all services are orchestrated seamlessly using containerization techniques. The use of relational databases for data persistence ensures data integrity and smooth operations across the platform.

7.4.3 E-Farmers

System Overview

E-Farmers is a specialized e-commerce platform developed to bridge the gap between local farmers and consumers seeking fresh, locally sourced products. The platform enables farmers to manage their product listings, allowing them to create, update, and organize their offerings. On the consumer side, users can easily browse through the available product listings, select desired items, and proceed to purchase using a secure and reliable payment system. The system emphasizes ease of use while fostering a direct connection between producers and end consumers, contributing to the growth of sustainable, local food networks. Additionally, the platform provides delivery options via registered riders who handle product shipments. Users can register with an email or through OAuth, access their personalized pages, and manage their profiles, including switching between customer, farmer, and rider roles. Key features include a calendar for viewing seasonal products, subscription to farmers for new product notifications, and an order history feature for tracking past purchases. Farmers can modify, delete, or add to their product listings, and are incentivized through a badge system to highlight their achievements. The platform also supports delivery coordination and status updates for both customers and riders. E-Farmers targets local farmers seeking to sell their produce, customers looking for high-quality farm-to-table items, and riders interested in facilitating the delivery of these products.

Technical Analysis

Containers and Services Structure The E-Farmers system architecture adopts a microservices-based approach to handle the different functionalities of the e-commerce platform. The platform is built around several distinct containers and services, each responsible for a specific aspect of the application. The system is composed of several microservices that are containerized and deployed independently. The main container, ‘EFarmers‘, encompasses the core functionalities of the platform

such as managing user accounts, insertions, payments, shopping carts, and subscriptions. This container is exposed via port 3000:8083, handling both frontend and backend processes for these tasks.

The frontend microservice, built using ReactJS, is responsible for managing the user interface, including user-specific pages, farmer profiles, and the seasonal calendar. The component-based architecture of the frontend ensures reusable and maintainable code, while React hooks enable efficient state management. This service communicates with other backend microservices using Axios to handle asynchronous HTTP requests.

The backend services are divided into several microservices, each addressing a specific functionality:

- **user_service:** Handles user account management, including registration, login, logout, and profile management, using Django with JWT for authentication and PostgreSQL for database storage.
- **insertions_service:** Manages the creation, modification, and deletion of insertions, booking boxes, and expiring insertions, using Django and PostgreSQL for database interactions.
- **shoppingcart_service:** Manages shopping cart operations, allowing users to add, modify, or remove items from their cart.
- **payments_order_service:** Processes secure payments using the Stripe API and manages order-related tasks such as delivery and rider availability.
- **subscription_service:** Implements subscription functionality, allowing customers to subscribe to farmers and receive notifications for new insertions via RabbitMQ.
- **db:** A PostgreSQL database container that stores all persistent data related to users, insertions, orders, and reviews.
- **rabbitmq:** A RabbitMQ container that facilitates communication between services via message queuing, ensuring asynchronous interactions between the customer and farmer services.

-

Each of these microservices is containerized and deployed independently. The architecture allows for continuous development and deployment of individual services without affecting the entire system.

Database Services The database layer is centralized within the ‘db‘ microservice, which uses PostgreSQL as the primary database management system. The database stores various data sets, including:

- **User-related data:** Registration information, login credentials, profile settings, and user roles (customer, farmer, rider).
- **Insertions and boxes:** Details of the products inserted by farmers, including descriptions, expiration dates, and available boxes for booking.

- **Order-related data:** Information on the orders generated by customers, including payment status, associated farmers and riders, and delivery details.
- **Rider availability:** Data on rider availability and assignment to deliveries.
- **Reviews and subscriptions:** Customer reviews of farmers and subscriptions to receive notifications on new insertions.

The 'user_service', 'insertions_service', 'shoppingcart_service', and 'payments_order_service' microservices all interact with the PostgreSQL database via 'psycopg2-binary', a PostgreSQL adapter for Python.

Architectural Overview The architectural design of E-Farmers follows a microservices architecture that is composed of independently deployable services, each fulfilling a specific role within the system. The key characteristics of this architecture include:

- **Decoupling and Modularity:** Each microservice is tasked with a distinct functionality, such as user management, product listings, payments, or subscriptions. This modular approach enhances the system's ability to scale and evolve. The services interact via HTTP APIs, with Django powering the backend services, while Flask is used specifically for handling subscriptions.
- **Data Consistency and Persistence:** A PostgreSQL database serves as the central data store, ensuring consistent access to shared information across services. Despite the independent operation of microservices, they share a unified view of critical data entities, such as user accounts, products, orders, and subscriptions, through this central database.
- **Asynchronous Communication:** RabbitMQ is employed to enable asynchronous communication between services, particularly for non-blocking operations like notifications of new product listings. The subscription service utilizes RabbitMQ's Advanced Message Queuing Protocol (AMQP) to publish updates from farmers regarding new products, which are subsequently consumed by subscribers.
- **Frontend-Backend Separation:** The frontend microservice, developed with ReactJS, interacts with backend services via REST APIs. This distinct separation between frontend and backend layers allows for greater flexibility, maintainability, and independent evolution of the user interface and core functionalities.

Overall, the use of REST APIs and message queuing across the platform allows these independently operating microservices to effectively communicate, creating a scalable, maintainable, and resilient system.

7.4.4 RentYourExpert

System Overview

RentYourExpert is a platform designed to facilitate connections between individuals seeking expert services and skilled professionals across a range of fields. The system's primary goals are to streamline the processes of discovering, recruiting, and interacting with experts. It achieves this through

several key features: a user-centric interface designed for intuitive navigation, advanced search capabilities that allow users to filter experts based on their skillset and geographic location, and secure authentication protocols to safeguard user data and maintain privacy. Additional functionalities, such as worker reviews and an integrated communication system, support informed decision-making and efficient interaction between clients and workers. Profile management tools enable workers to oversee their projects and communications seamlessly. The system serves three types of users: customers who seek professional expertise, workers offering services, and administrators who manage both customer and worker accounts along with request transactions. Customers can perform actions such as registering, managing profiles, searching for workers, submitting requests, leaving reviews, and communicating with experts. Workers can manage their profiles, respond to requests, and interact with customers. Administrators have full control over user management, including adding or deleting users, handling requests, and managing reviews and communication.

Technical Analysis

The system architecture for RentYourExpert is composed of multiple microservices, each dedicated to a specific set of functionalities, organized into a containerized structure. The platform is divided into microservices for user management, job requests, reviews, questions and answers, as well as a front-end and a database service, following a microservices architecture that allows for modular development, scalability, and maintainability.

Containers and Services Structure The RentYourExpert system is organized in a Docker container structure, with each service encapsulated in its own container. This containerized approach enables the decoupling of services, ensuring isolated deployment and scalability for each microservice.

The primary container, RentYourExpert, manages the core functionalities of the platform, which includes user management, catalogue of workers, job requests, reviews, and worker profiles. Inside this main container, several microservices handle specific functionalities:

- **frontend:** This microservice is tasked with managing the client-side interface, which is implemented using ReactJS. It offers user-friendly pages for functionalities such as user registration, authentication, and account management, catering to customers, employees, and administrators. The front-end microservice interacts with the back-end via HTTP requests, enabling dynamic rendering of data-driven pages and ensuring seamless integration with the system's overall architecture.

- **cust_login_ms, worker_login_ms, and management_ms:** These microservices are responsible for managing distinct operations: customer login, registration, and profile management for customers, workers, and administrators. Each service is designed to handle its respective user workflows independently, incorporating token-based authentication mechanisms and managing interactions with the associated profile data. This modular architecture ensures separation of concerns and allows for scalability and maintainability within the system.

- **catalogue_ms:** The microservice responsible for managing the catalogue of workers, including retrieving lists of available workers and fetching individual worker profiles.

- **request_ms:** This handles the core request functionality, allowing customers to send job requests to workers and enabling workers to accept or reject these requests.
- **qea_ms and review_ms:** These microservices manage the question-and-answer and review systems, respectively. They provide the necessary endpoints for customers to ask questions to workers, workers to respond, and customers to leave or modify reviews.

Each of these services is containerized with dedicated ports, allowing independent scalability. The services interact via HTTP APIs, ensuring clean and well-defined communication boundaries.

Database Services The db microservice provides a central relational database managed using MySQL, responsible for storing all application data including user profiles, job requests, reviews, and worker-related information. This microservice acts as a persistent storage layer, ensuring data consistency and durability across the platform. The microservices communicate with the db microservice through MySQL connectors, allowing CRUD (Create, Read, Update, Delete) operations on structured data.

Each microservice relies on the db service to manage specific tables:

- The cust_login_ms and worker_login_ms microservices manage user information in the Customer and Worker tables, storing details like name, email, password, and profile information.
- The request_ms microservice interacts with the Request table, linking job requests between customers and workers.
- The qea_ms and review_ms services manage QuestionAnswer and Review tables, respectively, for handling customer-worker interactions and feedback.

Architectural Overview The RentYourExpert platform follows a microservices architecture where individual services are independently deployed and scaled. This architecture provides flexibility, maintainability, and enhanced scalability, aligning with modern software engineering principles. Each microservice within the system is designed to perform a specific business function, minimizing dependencies and allowing for independent development and deployment.

At the heart of the architecture is the backend layer, built primarily using Flask, a lightweight web framework for Python. Flask enables the construction of RESTful APIs that each microservice exposes for communication. This modular approach allows each service to scale based on its workload, such as the request_ms or review_ms, which may experience higher loads depending on user activity.

The frontend microservice serves as the interface through which users interact with the system. ReactJS, chosen for its component-based architecture, ensures that the user interface is responsive and dynamic. The front-end communicates with backend services asynchronously through HTTP requests, using RESTful APIs to fetch and display data like worker profiles, reviews, and job requests.

All backend services (such as cust_login_ms, worker_login_ms, catalogue_ms, request_ms) follow a stateless architectural pattern, which simplifies the handling of user requests by not storing session data on the server side. Instead, JWT (JSON Web Tokens) are used to manage authentication and authorization across services, providing a secure mechanism for user access control.

The db microservice is critical for persistent data storage, but by keeping it separate from the application logic, the architecture maintains clean separation of concerns. The MySQL database schema is relational, ensuring structured data handling and support for ACID (Atomicity, Consistency, Isolation, Durability) properties, essential for a platform that manages sensitive user data such as passwords, reviews, and requests.

7.4.5 CDC

System Overview

The CDC Shop system is a distributed e-commerce platform designed to manage the sale, payment, and shipping of hardware store products. The system is designed to support two primary user roles: Customers and Merchants. Customers have the ability to sign up, log in, and manage their personal profiles. They can explore and search for products, sort through listings, and view detailed product information. Alongside these features, Customers can handle their shopping cart, finalize purchases, track their orders, and reach out to the shop for any assistance they may need. Merchants, on the other hand, are provided with functionalities to manage product listings, including adding, removing, and updating product prices. They can also sort and search for products and access their profiles. The system supports both Customer and Merchant logout, and both roles can view the "About Us" page. The intended users of this system are hardware store Customers looking to purchase products online and Merchants who manage the store's inventory and sales.

Technical Analysis

Containers and Services Structure The containers and services structure of the CDC Shop system is based on a microservices architecture, where each container is responsible for a specific set of features that contribute to the overall functionality of the platform. The system consists of the following containers: CDC (the core system), warehouse_api, catalog_api, purchase_refund_api, auth_api, db, and frontend. The CDC container, which manages customer and merchant authentication, products, and orders, acts as the central hub that ties all user interactions together. This container communicates with multiple backend microservices to handle discrete business functions. The warehouse_api microservice is responsible for product management, while catalog_api focuses on handling the user's shopping cart. The purchase_refund_api manages order checkout processes, and the auth_api takes care of user authentication. These microservices are decoupled from each other and interact with the CDC container through defined RESTful endpoints.

Each microservice exposes a set of endpoints that correspond to the specific user stories they serve. Warehouse_api exposes endpoints for retrieving, adding, and deleting products from the database, supporting both customer and merchant functionalities. Similarly, the catalog_api manages operations related to cart management, such as adding products to the cart and removing items. The auth_api microservice is designed for handling authentication flows, supporting registration and login processes for both customers and merchants. These backend services interact with a central MySQL database, handled by the db microservice, which is responsible for persisting user, product, cart, and order data. The frontend microservice serves as the user interface layer, built with ReactJS, which enables users to interact with the system through a dynamic and responsive web application.

Database Services The database services are managed by the db microservice, which provides a MySQL-based database system. This service is critical as it stores structured data related to users, products, carts, and orders, ensuring data persistence and consistency across the platform. The database is structured into multiple tables, each representing core entities in the CDC system. The Products table, for instance, stores information about available products, including attributes such as the product's name, description, category, price, stock, and image URL. The Users table maintains user records for both customers and merchants, containing fields like name, email, phone, and encrypted passwords.

The db service also manages the Cart table, which is linked to user accounts and stores the products that customers have added to their shopping carts, tracking the product ID and quantity. Additionally, the Orders table logs all customer orders, capturing data such as the order date, customer details, total price, payment status, and order status. The design also includes the Purchase_items table, which logs the individual products purchased within each order, along with their respective quantities and prices. This schema ensures that both customer and business data is accurately captured and can be retrieved when needed.

Architectural Overview CDC Shop implements a microservices architecture, where each service is highly modular and focused on a single responsibility. This structure allows for scalability, maintainability, and fault isolation. The architecture promotes separation of concerns by decoupling functionalities across services. Each microservice operates independently but communicates with others through well-defined HTTP APIs.

The CDC Shop system is also built to be stateless, where each request to the backend microservices contains all the information required to process that request. From a security perspective, the auth_api microservice handles user authentication and authorization, ensuring that only authenticated users can access restricted functionalities such as viewing a profile or managing a shopping cart. Passwords are encrypted before being stored in the database, enhancing security. The system also employs CORS (Cross-Origin Resource Sharing) to allow controlled access between different origins, enabling the frontend (running on a separate domain or port) to interact with the backend services securely. Docker Compose is used to orchestrate these containers, ensuring that all services start in the correct order and are interconnected via defined network interfaces.

7.4.6 EventTicket

System Overview

This platform is a web-based solution designed to simplify event management and ticket distribution. It provides an easy-to-use interface that allows users to discover events, promote them, and handle ticket sales, whether through direct purchases or pre-sale reservations. Event organizers benefit from tools that streamline promotion and ticketing, all managed within the application itself. The system caters to a variety of user roles, such as clients, attendees, event coordinators, and administrators, each with its own set of capabilities. Notable features include user registration, event browsing, purchasing tickets, booking pre-sales, and comprehensive event management. Additionally, administrators oversee system operations, such as managing event managers. The system is intended for three primary user groups: general users who wish to discover and attend events, event

managers responsible for promoting and managing events, and administrators who ensure proper system governance.

Technical Analysis

Containers and Services Structure The container and services structure of this project is built around a microservices architecture, where different functionalities such as user authentication, event management, payments, and message brokering are segregated into independent services. The primary container, EventTicket, encapsulates the core logic of the system, providing user account management, event listing, ticketing, and pre-sales functionalities. This container coordinates with several backend, frontend, middleware, and database microservices to deliver a cohesive platform. Each microservice is deployed within isolated containers, communicating via RESTful APIs or message queues, ensuring modularity and scalability.

Within this architecture, specific services handle distinct responsibilities. The auth1 microservice focuses on user authentication and account management, utilizing Node.js with key libraries like passportJS for authentication and jsonwebtoken for secure token handling. The event microservice manages the event-related operations, including event creation, listing, and interaction, while leveraging Express.js for routing and Mongoose for database interaction. The payments microservice integrates the Stripe API for handling ticket purchases and pre-sales bookings, encapsulating payment logic and providing an interface for users to perform checkouts and view ticket-related information. The nginx microservice serves as the frontend entry point, delivering the user interface and acting as a reverse proxy to other services.

The platform also employs RabbitMQ middleware, implemented in two separate services, rabbit_producer and rabbit_consumer, to facilitate asynchronous communication between different parts of the system. This setup enhances scalability and decouples services, allowing for independent scaling of message producers and consumers.

Database Services The database services in this architecture are primarily based on MongoDB, ensuring flexible and scalable storage for various aspects of the platform. Several microservices are dedicated to handling specific types of data. The mongo microservice manages user-related data, including registration, login credentials, and profile information for users, admins, and managers. It employs MongoDB as the database management system, offering high availability, data integrity, and flexibility in handling complex user structures. The mongo_event microservice is responsible for storing event-related data, ensuring fast and reliable access to event information for both users and managers. The event schema stored in this database encompasses attributes such as event name, description, date, location, ticket availability, and pricing, structured to support dynamic event listings and filtering based on user preferences. Another microservice, mongodb, is dedicated to ticket storage, housing information related to purchased tickets, pre-sales, and bookings. These databases are accessed via the Mongoose library, an Object-Document Mapper (ODM) for MongoDB, enabling smooth interaction between the application and its persistent storage.

Architectural Overview The overall architectural design follows a microservices pattern with a clear division of concerns. Each microservice operates independently, serving distinct functionalities while interacting via well-defined APIs. The rabbit_producer and rabbit_consumer microservices

leverage RabbitMQ to support asynchronous processing, improving the system's resilience by decoupling tasks like sending email notifications or processing event updates from user-facing requests.

The backend services, including auth1, event, and payments, follow a layered architecture pattern, which enhances maintainability and clarity of the codebase. Each service has well-defined layers for handling business logic, data access, and routing. For example, in the event microservice, the controller layer handles the logic for fetching, displaying, and managing events, while the model layer defines how events are structured and stored in the MongoDB database.

External service integration plays a crucial role in the overall architecture. The system interacts with external platforms like SendGrid for email notifications, CloudAMQP for message queuing, and Stripe for payment processing. These integrations are critical to the functionality of the system, as they provide robust, off-the-shelf solutions for handling complex tasks like email delivery and secure payment handling. For instance, the payments microservice relies heavily on Stripe APIs to manage ticket purchases, allowing users to pay for tickets securely through the platform. The use of CloudAMQP ensures that event-related messages are reliably transmitted between services, promoting asynchronous communication and enhancing system performance under load.

7.4.7 Teamify

System Overview

Teamify is a robust software platform designed to facilitate both personal task management and collaborative team workflows. It offers core functionalities such as creating and managing personal and shared tasks, establishing teams, real-time communication, and notifications, as well as integrated survey tools for collective decision-making. The system supports features like personal and shared agendas, task tracking, real-time chat, and event management, aiming to optimize productivity for individuals and teams alike. Task and team management are at the heart of Teamify, allowing users to efficiently organize personal tasks while also collaborating on group projects. Real-time notifications ensure that users are always informed about updates, while the chat and polling features enable effective communication and collaboration within teams. Teamify's intended users include individual professionals seeking to streamline their personal task management and teams or organizations that require seamless coordination, task sharing, and real-time communication to enhance overall productivity. It supports administrators with tools for managing teams, events, and permissions, catering to the needs of both individual users and collaborative environments.

Technical Analysis

Containers and Services Structure The containerized architecture of Teamify centers around three core microservices: frontend, backend, and database, alongside backup replicas for high availability and resilience. The primary container, teamify, encapsulates the core logic of the system.

The frontend microservice is tasked with delivering the user interface and managing all client-side requests. It communicates with the backend using HTTP, forwarding user interactions for further processing. This container runs on port 3000, allowing users to access the web application directly.

The backend microservice provides the business logic and handles communication between the frontend and database layers. It hosts the REST APIs that manage tasks such as account creation,

login/logout, task management, team management, real-time chat, and notification handling. This service operates on port 5000, ensuring seamless data handling and application logic processing. A backup version of the backend service operates on port 5001, ensuring that if the primary backend container fails, the system remains operational without data loss or downtime.

The database container is critical for managing persistent data across various functionalities, such as user accounts, tasks, teams, notifications, chat messages, and poll data. It operates on port 5432 and is responsible for storing structured information that is accessed and modified by the backend microservice. The database microservice includes a backup running on the same port (5432), designed to maintain data availability and mitigate potential disruptions caused by maintenance or unexpected failures. This redundancy plays a crucial role in mantaining the system's fault tolerance, ensuring the application can continue delivering services without interruption.

Database Services The database layer within Teamify is designed with redundancy and fault tolerance in mind. The system uses two microservices for managing data storage: db and db-backup. Both services operate on port 5432 and are responsible for handling persistent storage for all key aspects of the application, including user profiles, personal and shared tasks, team information, notifications, chat histories, and polls.

The db microservice operates as the primary database and stores the main copy of all the application's data. The database follows a relational model, managed by PostgreSQL. The relational model is an ideal choice for this system, offering the ability to handle complex queries essential for features such as task management, shared calendars, and real-time collaboration.

Additionally, the db-backup microservice plays a crucial role in enhancing system resilience by acting as a replica of the primary database. This setup ensures high availability, as it allows seamless failover to the backup database in case the primary database becomes inaccessible, thereby maintaining continuous system functionality without interruption.

Architectural Overview Teamify's architecture leverages a microservices approach, which significantly enhances its modularity, scalability, and overall system flexibility. This design consists of loosely coupled, independently deployable services, with each service dedicated to handling specific functionalities. By employing this microservices architecture—comprising backend, frontend, database services, and their corresponding backups—Teamify achieves clear separation of concerns, allowing each service to manage distinct areas of system functionality.

At the heart of this architecture lies the backend microservice, responsible for orchestrating the application's core logic. It serves as the intermediary between the frontend user interface and the underlying database. Key functions handled by the backend include authentication, task and team management, and notifications. Additionally, it supports real-time communication, facilitating features such as chat and polling. The backend-backup service ensures high availability by replicating the backend logic, thus offering an additional layer of reliability.

The frontend microservice handles the presentation layer and is responsible for managing user interaction. This service is connected to the backend via REST APIs, meaning all user inputs—such as task creation, event updates, and team management operations—are relayed to the backend for processing. This service also updates the user interface based on the backend's responses, ensuring that users receive real-time feedback on actions like task completion or chat message delivery.

In addition to core microservices, database replication is crucial in ensuring data integrity and availability. By integrating both a database and a backup service, the architecture prioritizes fault tolerance and disaster recovery. This layered strategy builds resilience into the system, helping it stay functional even during periods of high demand or when encountering potential failures.

7.4.8 RecipeCove

System Overview

The RecipeCove system is a web-based application designed to facilitate the discovery and management of recipes, focusing on user interaction through search functionalities and personalized suggestions. The system allows users to search for recipes by name, view detailed information about recipes and their ingredients, and receive recipe recommendations based on specific diets via an integrated Chat-BOT. Additionally, the platform supports user account management, enabling individuals to register using external authentication (e.g., Google accounts) and create personal profiles. Registered users can save recipes to a favorites list, manage personal events through a calendar feature, and set reminders for preparing specific recipes. The primary objectives of RecipeCove are to simplify recipe discovery, provide exhaustive information about ingredients, and enhance user engagement through personalized features like bookmarking and dietary suggestions. The intended users of the system include cooking enthusiasts, individuals following specific diets, and anyone seeking to organize their culinary interests efficiently. Both visitors and registered users are provided with varying levels of access, ensuring flexibility in system usage.

Technical Analysis

Containers and Services Structure The RecipeCove project is built on a microservices architecture that separates distinct functionalities into individual services for modularity and scalability. The primary container, RecipeCove, serves as the central hub for handling user-related tasks such as registration, login, recipe discovery, chatbot functionality, and calendar management. This container exposes services through ports, facilitating communication with external systems like GoogleAuth for authentication and GoogleCalendar for event management. RecipeCove integrates multiple services via defined microservice interfaces to ensure each component handles specific tasks without overloading the system. The modular design leverages both middleware and hybrid microservices, making use of message queues for inter-service communication and providing a clear separation of concerns across functionalities such as email notifications, recipe discovery, and data persistence.

Database Services Tthe database service is powered by CouchDB. CouchDB is a NoSQL, document-oriented database that stores all data related to users, including their personal information and favorite recipes. The choice of CouchDB aligns with the need for flexibility in data modeling, as it operates with JSON documents, which can easily store the varied and dynamic nature of recipe and user data. In this setup, every user's data (account information, favorite recipes, and profile details) is persisted and replicated across nodes to prevent data loss. CouchDB's integration with the primary RecipeCove container allows for seamless data storage and retrieval through RESTful APIs, ensuring efficient communication between services.

Architectural Overview The RecipeCove project reflects a hybrid microservice model with a combination of traditional RESTful services and event-driven communication patterns. The main services are divided into hybrid microservices, middleware, and database components, each fulfilling distinct roles within the system.

The webapp microservice serves as the main entry point for user interactions, handling requests for registration, login, and profile management via HTTP. Additionally, this microservice interacts with external APIs like GoogleAuth for authentication and the Spoonacular API for recipe suggestions through a chatbot, allowing users to discover recipes based on dietary preferences. The chatbot functionality is enabled by a WebSocket interface, allowing for real-time communication between the user and the system. Furthermore, the webapp microservice also manages recipe discovery and retrieval, sending HTTP requests to external APIs to fetch recipe information, which it then processes and displays to users.

The RecipeCove system also includes a calendar microservice, which manages all calendar-related tasks. This hybrid microservice connects to Google Calendar APIs, allowing users to create, update, and delete calendar events directly from the RecipeCove platform. Users can set reminders for specific recipes they intend to prepare, and the system will create events in their Google Calendar, ensuring they are notified on the appropriate day. This microservice operates through HTTP endpoints and utilizes the Google Calendar API's OAuth-based authentication to integrate directly into users' calendars without exposing sensitive data. Each event or reminder created is stored locally within the RecipeCove system for reference but is also pushed to the user's Google Calendar, ensuring external synchronization.

RecipeCove's communication between the webapp and other services, such as the nodemailer service for email notifications, is facilitated by a middleware messaging system, specifically the RabbitMQ microservice. RabbitMQ acts as a message broker, allowing asynchronous communication between the webapp and nodemailer services. This middleware decouples the webapp's operations from the email service, ensuring that users are notified through email without introducing delays in webapp processing times. For instance, when a user registers or updates their profile, the webapp sends a message to RabbitMQ, which then routes it to the nodemailer service for email dispatch. This asynchronous model prevents bottlenecks and improves overall system responsiveness.

The nodemailer microservice plays a crucial role in sending emails to users, ensuring that RecipeCove can communicate important updates and notifications directly to their inbox. This microservice is connected to the RecipeCove system through RabbitMQ, which handles the queuing of messages and ensures that emails are sent without interrupting the primary user experience. For instance, when a user registers or interacts with certain functionalities (such as creating calendar events), the nodemailer service will generate and send confirmation emails via SMTP protocols.

Overall, the architecture adheres to the principles of microservice design, where each component is responsible for a specific subset of the system's functionality, promoting loose coupling and high cohesion. The webapp, calendar, and nodemailer microservices, along with the couchdb database and RabbitMQ middleware, work together to create a scalable, responsive, and resilient system capable of handling the various interactions users might have with RecipeCove. Additionally, the integration of external services such as GoogleAuth, GoogleCalendar, and Spoonacular for recipe recommendations adds significant value to the system by leveraging existing platforms and services. The design ensures that the system is highly modular and that future services or functionalities can

be added or modified without impacting the overall architecture.

7.5 Projects' cleaning

The cleaning process consists of two phases:

1. System description crafting
2. User stories cleaning

7.5.1 System description crafting

For the purposes of this dataset, we want the system description to provide real-world context to the user stories and an idea of what the final system should achieve. The projects didn't have a system description: all the projects comprehended a presentation (to be shown alongside the realized system), some sort of technical documentation (if present) and documentation regarding the development process following the Agile methodology.

For each project, we crafted a satisfying description out of the available material. For each system, the documentation provides an overview of the system's main features, along with real-world context and, if present, non-functional requirements.

7.5.2 User stories cleaning

For the purpose of this dataset, we want the user stories to have the following characteristics:

- **Follow a template:** We need user stories in a common format, in order to let the LLM analyze them without generating confusion.
- **Cover all the possible actions related to a single scope:** We need the user stories to cover every possible action related to the features described in the system description. Missing user stories generate confusion about the scope of the system and can lead to uncompleted system designs.
- **Be clear** We need user stories to be phrased in a clear way, in order to clearify the system's characteristics.

While most of the projects have user stories in the *Connextra* or *Five W* format, 2 selected projects didn't have their user stories in a common format. We converted them to the *Connextra* format, mantaining the general meaning but phrasing them in order to avoid misunderstandings.

6 out of 8 selected projects had missing user stories. An example of a missing user story is given by a project of the dataset (EFarmers), dedicated to buying and selling agricultural products. In this project, there were user stories concerning the farmer as a user regarding the adding, modifying and visualization of a product, but the user story about the deletion of said product was missing. For all the projects, we checked if there were missing actions and added or modified user stories as needed.

All the projects had more than one user story that needed rewriting: we rephrased them, mantaining the original meaning but rendering them more clear to the reader.

7.6 Projects' documentation generation (code analysis)

This section will describe the adopted documentation format and the documentation generation process. Given the fact that the projects were missing technical documentation, we designed a documentation format that describes all the aspects of a system design based on microservices architecture, taking also into strong consideration the correlation between the system user stories and the adopted architectural choices.

7.6.1 Project Standard

The Project Standard format is a format designed to describe the design of systems that follow Microservices Architecture principles. It aims at covering all the design choices implemented from the container level to the miscroservices internal architecture and features. It follows a top-down approach, starting from the containers design and purposes, then describing the microservices layer inside the containers and each microservice design details. This approach, linking the system's user stories with the actual system components (containers and services), highlights how the system's expected features and behaviour are implemented in the actual code.

Containers The top layer is represented by the containers of the system. Each container is described with the following fields:

- **Container name**
- **Container Description** Brief description of the container features, behaviour and purposes
- **User Stories** User stories of the system covered by the container
- **Ports** Ports assigned to the container
- **Persistance Evaluation** Analysis of the needs of the container to store data
- **External Services Connection** Analysis of the needs of the container to connect to external services

This layers defines the purpose and functionalities provided by the container and the general characteristics the services inside need to have.

Microservices This layer describes the microservices of the container. Each microservice is described with the following fields:

- **Microservice name**
- **Type** How is the microservice categorized
- **Description** Brief description of the microservice purpose and functionalities
- **Ports**
- **Technological Specification** Evaluation the technological stack, language, framework and eventual libraries used by the microservice

- **Service Architecture** Analysis of the service architecture in terms of patterns adopted, directories structure and business logic

We categorized the microservices in 5 main types:

- **Frontend** The microservice serves only frontend purposes, likely employing a frontend SPA framework.
- **Backend** The microservice includes part of the business logic
- **Database** The microservice hosts a database
- **Hybrid** The microservice includes part of the business logic and also serves frontend purposes
- **Middleware** The microservice acts as a middleware for the system

Backend microservices contain also an endpoint table, describing the endpoints exposed by the microservice, and the structure of the database table eventually used.

The endpoints table has the following columns:

- **HTTP Method** The HTTP Method of the endpoint
- **URL** The URL of the endpoint
- **Description** Brief description of the actions performed by the endpoint
- **User Stories** The indexes of the user stories fulfilled by the endpoint

Frontend microservices contain a table, named "Pages", in which frontend pages, related microservice and eventual user stories fulfilled are enumerated.

The Project Standard format offers a standardized way to describe systems based on microservices architecture. It allows the designer of the system and any eventual spectator to relate the features and real-world references of the application (in form of user stories) with the architectural structure of the system. It also allows an easy transformation of the system architectural design to a tree-based data structure.

7.6.2 Documentation generation

On each of the selected projects, a comprehensive analysis has been conducted to craft the Project Standard documentation. At first, the docker-compose files of the project have been analyzed to describe the system's container structure. Dockerfiles also have been analyzed in this phase to evaluate the container's persistance.

The following phase consists of the microservices analysis. Each microservice's configuration files have been inspected to craft the technological specification of the service. After that, a careful evaluation of the service source code's structure resulted in the generation of the "service architecture" section of the documentation.

For backend microservices, the endpoints section generation has been carried on by analyzing carefully the microservice routes and controllers, checking which user stories the endpoint effectively

satisfies. For frontend microservices, where possible, a deep analysis has been conducted to verify which user stories are effectively satisfied by the frontend microservice.

Great effort has been employed to ensure that the resulting documentation reflects the status of the students' projects, understanding each container and microservice behaviour, purpose and real-world aspects.

7.7 Projects Datametrics

For each project of the dataset, we published an attached JSON file in DataMetrics format. The Datametrics format is a JSON format we developed to categorize and group the user stories of a project and analyze the dependencies and constraints they pose to the architectural design of the derived system.

7.7.1 User stories sets

The project's user stories are grouped in user stories' sets based on their real-world scope: user stories that have the same context and a similar scope are in the same set. The grouping is based on the following factors:

- **User stories relate to the same real-world object or situation:** For example, in a e-commerce system, user stories related to actions on the products are grouped in the same set.
- **Different users perform the same action:** For example, in a system where the user can see his personal informations and the admin can see the user's personal informations, both user stories are grouped together.

From an architectural design point of view, all the user stories that belong to the same set have to be fulfilled by the same Docker container: in this way, the system ensures separation of concerns between containers while mantaining "scope-related" user stories grouping. How the user stories are fulfilled by the microservices inside the container is a design choice not inspected by this file format: the microservices structure depends on designer choice and can vary based on the specific user stories and context,as long as all the user stories from a set are fulfilled by the same container.

7.7.2 Sets' links

Link between user stories' sets describe the real-world correlation between different user stories' contexts. Each set has a "links" field, that identify other sets that have a related context. From an architectural design point of view, user stories that belong to linked sets can be fulfilled by the same Docker container: also in this case, when two or more different sets are satisfied by the same container, is up to the architect to design the microservices inside the container to ensure separation of concerns, based on the specific context given by the user stories. Links are bi-directional: if set1 has in its links set2, set2 has in its links set1.

7.7.3 Database field

Each set has a boolean field, "db": it indicates that at least one user story of the set needs a backend service in order to store or retrieve data. From an architectural point of view, it means that the container that fulfills the set's user stories must contain a database microservice. IThis means that, if two or more sets with the db field equal to true are fulfilled by the same container, that container can also host only one database microservice. It is up to the architect to design one or more database microservices to satisfy the persistance needs of the user stories belonging to the sets.

Chapter 8

Validation

8.1 General Overview

The validation phase aims to benchmark the overall performance of our ArchiGPT implementation by comparing the projects generated by our generation architecture with actual student projects. To achieve this, we established several metrics that assign a score to each project.

8.1.1 Code Generation Benchmarks vs Architecture Generation Benchmarks

Code Generation Benchmarks

- **Function Set (F)**: Each function (f) consists of:
  - A textual description of the function, detailing its purpose and behavior, not limited to input and output specifications.
  - Unit tests designed to verify the correctness and performance of the function.
- **Benchmark Input**: Each script (s) is associated with a function (f).
  - Set of scripts (S) used to test the functions.
- **Metrics**:
  - A mathematical function (m_ut) is used to evaluate each script (s) based on its performance in unit tests, considering the number of correct results and total test generations.
  - An aggregator function (m_f) consolidates the results of the unit test evaluations to assess the performance of each function (f).
  - Another aggregator function (m_F) combines the results from all functions to provide an overall evaluation of the entire function set (F).

Architecture Generation Benchmarks

- **Project Set (P)**: Each project (p) contains:
  - A textual description of the project, detailing its objectives, architecture, and design choices.

  - User stories, which describe the functional requirements and use cases from the user's perspective.
  - A data metrics file, which contains quantitative data regarding the project's performance.

- **Benchmark Input**: Each document (d) is associated with a project (p).
  - Set of documentation files (D), which include project-specific information used for evaluation.

- **Metrics**:
  - Mathematical functions (m_c) evaluate the performance of each document (d) for individual containers in the system, using a predefined scale.
  - An aggregator function (m_C) consolidates the container-level evaluations to assess the document's impact on the entire project (p).
  - Mathematical functions (m_s) assess the overall system described by the project (p), considering all related documents and components.
  - An aggregator function (m_p) combines the results of previous evaluations to provide an overall score for the project (p).
  - Finally, an aggregator function (m_P) combines the scores from all projects to evaluate the entire project set (P).

8.1.2 DataMetrics.json File

This file, as already described in chapter 7.7, serves as the primary source for several metrics, which base their algorithms on the data contained within it.

Structure : This json describes the architecture and organization of the container responsible for satisfying the set of user stories. It defines the necessary microservices, interactions, and dependencies required to ensure the proper fulfillment of the set's requirements.

```
    [
                {
                    "set_id": 1,
                    "set_name": "auth",
                    "user_stories": [1, 2, 3],
                    "links": [2, 3],
                    "db": "true"
                },
            ...
]
```

8.1.3 Non-overlapping Maximal Clique Finding Algorithm

The non-overlapping maximal clique finding algorithm is implemented using a combination of the **Bron-Kerbosch** algorithm and a custom approach to ensure that cliques do not overlap. The

main goal of the algorithm is to identify groups of nodes (set of user stories) in a graph (a possible container) where every node is connected to every other node in the group. These groups are called **maximal cliques** and represent the most fine level of granularity on which every set of user stories can coexist in the same container.

Algorithm Steps

1. **Graph Representation:** The graph will be represented as an adjacency list. Each node in that list represents a set of user stories, and an edge between two nodes will mean these sets are related or can coexist within the same container.

2. **Bron-Kerbosch Algorithm:** The **Bron-Kerbosch algorithm** is a classic recursive backtracking algorithm to find all maximal cliques in an undirected graph. A maximal clique is a set of vertices such that for every pair of vertices there is an edge connecting them, and such that one cannot add any other vertex. In more formal terms, the algorithm maintains three sets:

   - R (the current clique being built),
   - P (the set of candidate vertices that can potentially extend the clique),
   - X (the set of vertices already processed that cannot be added to the clique).

   The Bron-Kerbosch algorithm proceeds as follows:

   - It recursively attempts to expand the current clique by selecting a vertex from the candidate set P.
   - For each vertex selected, the algorithm narrows down the candidate vertices by intersecting P with the neighbors of the selected vertex, and it excludes vertices in X.
   - Once P is empty and X has been processed, a maximal clique has been identified.

3. **Finding Non-overlapping Cliques:** After finding all maximal cliques in the graph, the algorithm sorts these cliques in descending order of size. This sorting ensures that larger cliques are considered first.

   The algorithm then iterates through the list of cliques and selects those that do not share any nodes with previously selected cliques. This step ensures that the cliques are **non-overlapping**, meaning that each node is used only once in the final set of cliques. The function `findNotOverlappingCliques` processes the cliques to eliminate overlaps, ensuring that the final set of cliques consists of mutually exclusive sets of user stories.

4. **Result:** The result of the algorithm is a set of **non-overlapping maximal cliques**, along with the total number of such cliques. Each clique represents a distinct subset of user stories (nodes/sets) that can potentially be placed in a separate container, where all user stories are fully satisfied without overlap.

8.2 Metrics Definition

In this section, each metric will be presented, indicating what it indicates, how it is calculated, and the parameters involved in the equation. The description approach will follow a bottom-up approach, starting from the metrics that evaluate service design, then containers and the whole system. After that, the evaluation of each run and of the whole runs batch will be described.

8.2.1 System Level Metrics

This section presents the metrics for evaluating an entire system. These metrics assess various aspects of a project architecture to meet system requirements, such as user stories coverage and granularity evaluation.

- **User Stories Satisfaction Coverage (USSC)** : This metric assesses the degree to which user stories are fulfilled. A user story is considered satisfied if an endpoint or a frontend satisfies its requirements.
  - us_{sod} = number of satisfied user stories
  - $num_{us_{tot}}$ = total number of user stories
  - $metric_{result} = 100 \times \left(\frac{us_{sod}}{num_{us_{tot}}}\right)$

- **Container Integrity Coverage (CIC)** : This metric evaluates whether a container fulfills all the user stories within a specific set. A container is considered complete if it satisfies every user story of the set it holds.
  - $num_{set_{us}}$ = number of sets of complete user stories
  - num_{set} = total number of sets
  - $metric_{result} = 100 \times \left(\frac{num_{set_{us}}}{num_{set}}\right)$

- **Granularity Evaluation (GE)** : This metric evaluates the granularity of user stories by representing them as nodes. The relationship between nodes is established through links, where sets of user stories can reside within the same container. A clique is defined as a set of nodes where each node has a link to all other nodes within the clique.
  - num_{clique} = number of non-overlapping cliques (identified by the algorith cited in subchapter 8.1.3.)
  - `metric_result` can be calculated as follows:

```
if num_c <= num_clique              :  = 100*(num_c/num_clique)
if num_clique < num_c < num_set     :  = 100
if 2*num_set > num_c >= num_set     :  = 100*((2*num_set - num_c)/num_set)
if num_c >= 2*num_set               :  = 0
```

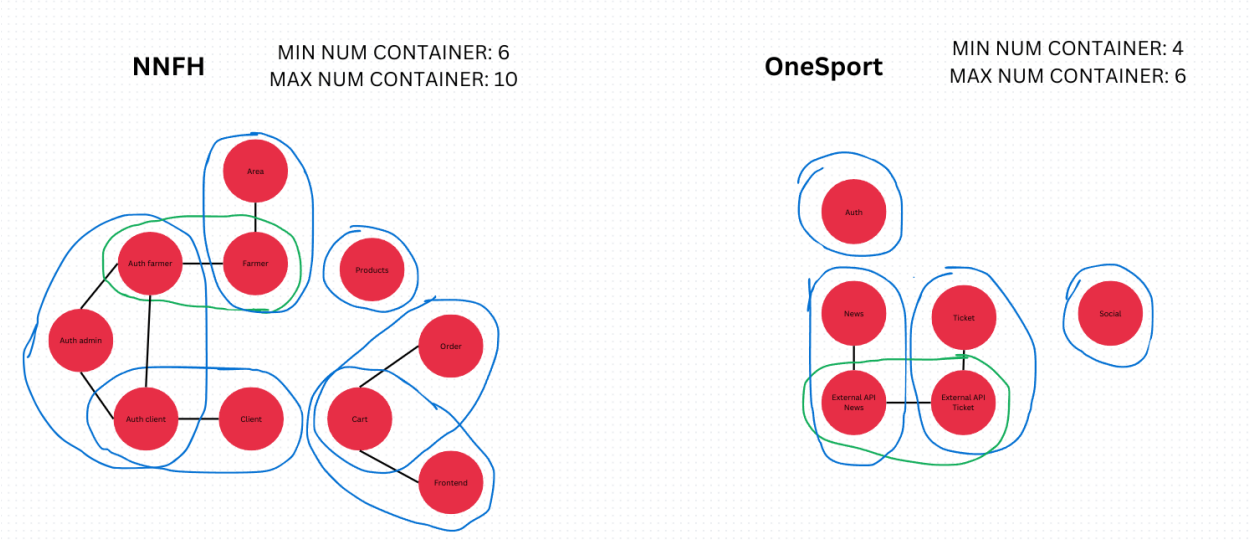

Figure 8.1: Granularity Evaluation for NFFH and OneSport

- **Container Persistance Coverage (CPC)** : This metric evaluates whether a container has at least one database service for each set of user stories it fully satisfies, given that the set requires database support (i.e., has a 'db' field equal to true). This metric applies only to containers that fulfill sets where a database is required.
  - $num_{c_{true}}$ = Number of containers that satisfy at least one set with 'db = true'
  - $num_{c_{true_{db}}}$ = Number of containers that provide at least one database microservice for such sets
  - $metric_{result} = \frac{num_{c_{true_{db}}}}{num_{c_{true}}}$

- **Service Coverage (SC)** : This metric evaluates at the system level, the overall score of the other metric **CSC** for each container.
  - csc_{result} = Container Service Coverage result of a container
  - $num_{container}$ = Number of containers within the system
  - $metric_{result} = 100 \times \left(\frac{\sum_{i=1}^{n} csc_{result_i}}{num_{container}} \right)$

- **System Container Endpoint Coverage (SCEC)** : This metric evaluates at the system level, the overall score of the other metric **CEC** for each container.
  - cec_{result} = Container Endpoint Coverage result of a container
  - $num_{container}$ = Number of containers within the system
  - $metric_{result} = 100 \times \left(\frac{\sum_{i=1}^{n} cec_{result_i}}{num_{container}} \right)$

8.2.2 Container Level Metrics

This section presents the metrics for evaluating each container. These metrics assess various aspects of a container's ability to meet system requirements, such as providing services, databases, and endpoints.

- **Container Service Coverage (CSC)** : This metric evaluates whether a container has at least one backend service for each set of user stories it fully satisfies. If the container provides fewer backend services than the number of cliques, the coverage percentage is proportionally reduced. If the number of services exceeds or equals the number of cliques, the coverage is considered complete (100%).
  - $num_{serv_{be}}$ = Number of backend services
  - $num_{clique_{full}}$ = Number of fully satisfied cliques
  - `metric_result` can be calculated as follows:

```
if num_serv_be <= num_clique_full   : = 100*(num_serv_be/num_clique_full)
if num_clique_full < num_serv_be    : = 100
```

- **Container Endpoint Coverage (CEC)** : This metric assesses whether every user story covered by a container has at least one associated endpoint or web page within that container to fulfill its requirements.
  - num_{us_c} = Number of user stories to be covered by a container
  - num_{us_e} = Number of user stories covered by an endpoint or page within the container
  - $metric_{result} = 100 \times \left(\frac{\sum_{i=1}^{n} num_{us_{e_i}}}{num_{us_c}} \right)$

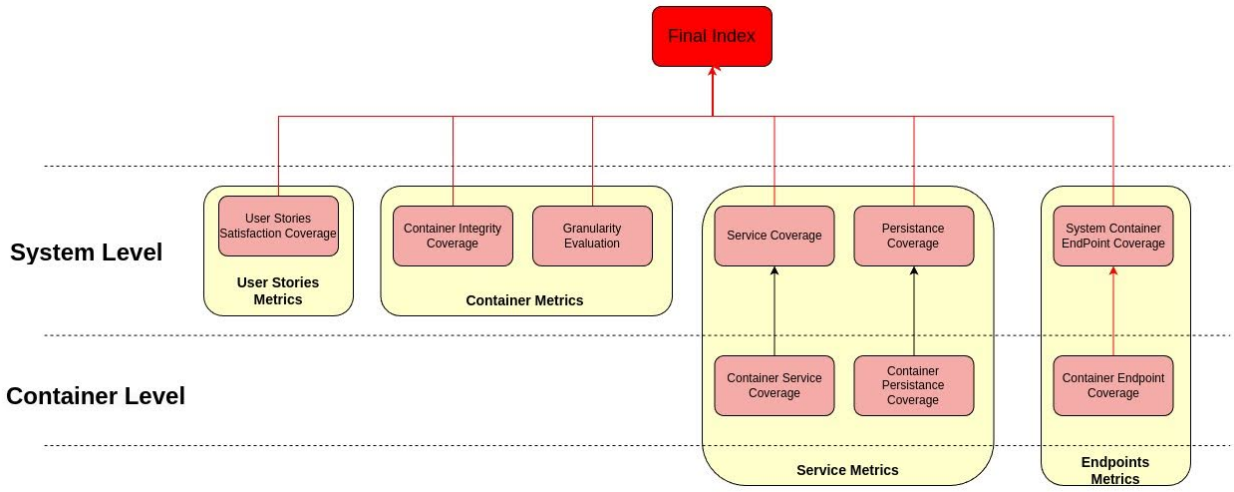

Figure 8.2: Metrics Overview

Project Final Index The final index defines a value in the range 0-100 to evaluate the whole designed system. Since each of the system level metrics evaluates a different aspect of the designed system, and all the considered aspects have the same relevance in the description of the system, the final index is the average of the 6 System level metrics.

Run Final Index A run for a specific model is given by the systems designed by the same model for each one of the projects of the Projects Dataset. The projects differ in regards of the real world context they provide, the features to be developed and the user stories and description format. By evaluating all of them, a comprehensive view of the model capabilities in terms of adaptability and comprehension can be obtained. The final index of a run is obtained by calculating the average of all the Project Final Indexes.

Model Final Index In order for the results to be statistically relevant, the Model Final Index takes into consideration 10 different runs of the same model. Like in [35], the Model Final Index is obtained by calculating the average of the runs, to evaluate the average model capabilities, and not the best responses the model can give.

8.3 Archi Metrics Architecture

This section outlines the workflow of the Archi Metric architecture, providing a detailed analysis of its key components and their respective functionalities.

8.3.1 Backend

The Backend microservice serves as the core of Archi Metric, acting as an intermediary between frontend requests 8.3.2 and Metric-Handler responses 6.4.4. It operates a Flask server on port 5002, exposing the following routes:

- */metrics*: For each metric, this route offers a function that takes the required parameters and files as inputs, calculating the metric based on the equations described in 8.2. This endpoint is responsible for handling metric-specific logic, ensuring that the necessary data is processed according to the defined formulas.
- */data*: Provides access to three main collections of JSON files:
  - **ArchiProjects**: Stores the JSON representations of each project generated by ArchiGPT via the Metric-Handler API. These files are the primary source for evaluating the performance of generated projects against the defined metrics.
  - **benchmarkProjects**: Contains benchmark files for each project type (e.g., OneSport, NFFH). These include key files such as *userStoriesIndex.txt* and *DataMetric.json*, as detailed in subchapter 7.7. These benchmarks serve as reference points for assessing the quality of ArchiGPT-generated projects by comparing them to manually produced baselines.
  - **studentProjects**: Stores the JSON files for each student project, manually generated to mirror the structure of the ArchiProjects. These projects are used as benchmarks to evaluate the ArchiGPT-generated projects under comparable conditions.
- */benchmark*: This is the central API that orchestrates the evaluation process for each project. By passing the JSON of either an *ArchiProject* or a *StudentProject*, this endpoint runs a sequence of metric evaluations, using the appropriate files and parameters. The final output is a JSON file that contains the scores for every metric applied to the project, presented in the following format:


```
{
    "modelName": "student",
    "finalIndex": 83.35,
    "finalResults": {
        "ussc": 96.7479674796748,
        "cic": 96.66666666666667,
        "ge": 46.42857142857142,
        "sc": 77.97619047619048,
        "cpc": 83.33333333333333,
        "scec": 98.94736842105264
```

```
},
"projectsResults": [
    {
        "ussc": 90.2439024390244,
        "cic": 90,
        "ge": 100,
        "sc": 62.5,
        "cpc": 50,
        "scec": 96.84210526315789,
        "projectName": "NFFH",
        "projectIndex": 81.6,
        "csc": [
            {
                "containerName": "Authentication",
                "result": 100
            },
            {
                "containerName": "Farmer-BE",
                "result": 50
            },
            {
                "containerName": "Client-FE",
                "result": 0
            } ...
        ],
        "cec": [
            {
                "containerName": "Authentication",
                "result": 100
            },
            {
                "containerName": "Client-BE",
                "result": 100
            },
            {
                "containerName": "Farmer-BE",
                "result": 84.21052631578947
            } ...
```

8.3.2 Frontend

The frontend microservice serves as the user entrance point of the metrics system. It allows the user to load up to 10 runs (in form of JSON files) and evaluate them using the metrics. It is realized in Angular, employing a single page for the loading action and to visualize the metrics results. It uses Material components by Google for graphic purposes. Is possible to inspect each run in detail, visualizing each run's metrics parameters.

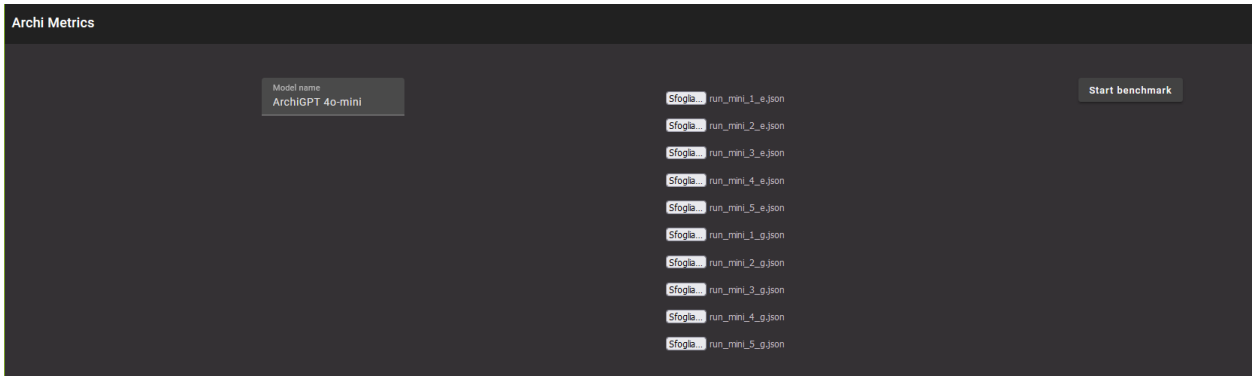

Figure 8.3: Load Run Json files screen

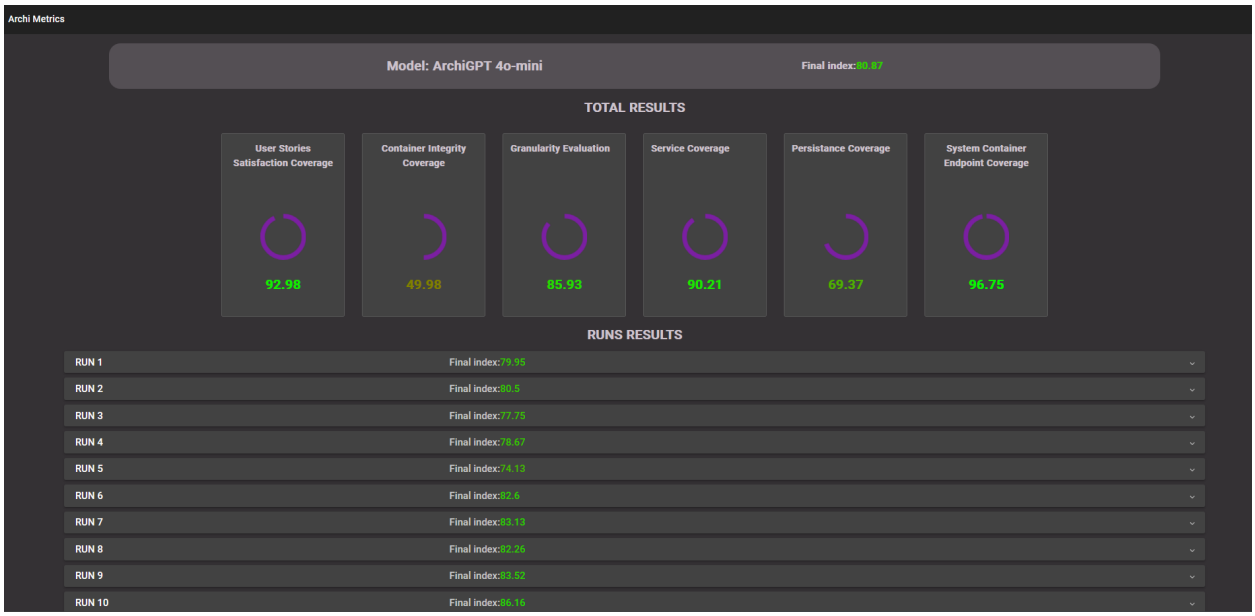

Figure 8.4: Metrics Page with final indexes

Figure 8.5: Metrics Page with projects indexes details

8.4 Results on Dataset

In this section, the results of metric algorithms on student projects and ArchiGPT generations are described and compared.

Students Projects results The results of the metrics, applied on the projects developed by the students, are the following:

Final Index: 72.09

USSC	CIC	GE	SC	PC	SCEC
72.28	96.96	30.74	65.7	93.75	73.11

The analysis of the metrics highlights various different aspects of the students' projects. The Granularity Evaluation metric is very low (30.74), because the projects are mainly built around one single container. The User Stories Satisfaction Coverage is expected to be near 100, because the user stories were crafted by the students themselves. Instead, only about 3/4 of the user stories were effectively fulfilled. The Service Coverage is also low (65.7), meaning that most of the projects do not have a well defined microservices architecture, resembling more a monolithic architecture dockerized with the attached database. Overall, the results are lower than what can be expected from master students.

ArchiGPT with 4o-mini results The results of the metrics, applied on the projects designed by ArchiGPT using GPT 4o-mini, are the following:

	Final Index	USSC	CIC	GE	SC	PC	SCEC
All	80.87	92.98	49.98	85.93	90.21	69.37	96.75
Run 1	79.95	91.08	48.99	84.62	100	57.5	97.52
Run 2	80.5	86.56	56.75	87.8	87.5	72.92	91.5
Run 3	77.75	95.48	42.92	90.58	75	62.5	100
Run 4	78.67	95.66	53.97	81.15	84.38	59.38	97.5
Run 5	74.13	91.56	43.45	88.1	70.83	53.33	97.5
Run 6	82.6	92.95	48.87	87.1	100	69.17	97.5
Run 7	83.13	96.89	52.18	81.55	100	68.75	99.43
Run 8	82.26	95.9	53.57	80.99	84.38	80.42	98.33
Run 9	83.52	88.34	49.27	91.67	100	82.29	89.58
Run 10	86.16	95.33	49.8	85.71	100	87.5	98.61

ArchiGPT with 4o results

	Final Index	USSC	CIC	GE	SC	PC	SCEC
All	85.96	97.96	59.55	85.29	90.72	83.3	98.91
Run 1	87.43	97.44	60.91	86.41	95.83	83.96	100
Run 2	86.72	99.65	55.77	87.8	87.5	89.58	100
Run 3	84.46	95.95	57.04	85.42	87.5	80.83	100
Run 4	87.92	99.6	62	87.8	93.75	84.38	100
Run 5	88.65	99.31	66.87	83.23	93.75	88.75	100
Run 6	86.29	96.97	61.33	85.02	93.75	82.5	98.16
Run 7	84.23	97.07	64.78	84.33	89.58	73.21	96.43
Run 8	83.79	97.91	54.17	78.25	87.5	86.04	98.86
Run 9	86.11	99.65	56.35	88.49	87.5	86.46	98.21
Run 10	85.47	98.11	58.83	86.01	93.75	79.79	96.35

Considerations on ArchiGPT results As expected, the Final Index of ArchiGPT surpasses the students' Index with both models. The relatively low result on Container Integrity Coverage metric is given by the human format of the User Stories, that can lead to misunderstandings by the AI. The below average result on the Persistance Coverage metric is a direct consequence of the result of the CIC metric. The relatively high result on the Granularity Evaluation metric is a proof of the OpenAI models knowledge and capabilities in categorizing user stories and that OpenAI containers and microservices concepts are inline with the principles of microservices architecture. It also states the high quality of the prompts of ArchiGPT. The Service Coverage metric shows an high separation of concerns among the microservices inside the same container. Lastly, the System Container Endpoint Coverage metric result shows that the userstories declared by the containers are effectively fulfilled by the microservices' endpoints and pages. As expected, the more powerful model has better results than 4o-mini, mostly due to its extended capabilities in text comprehension.

Chapter 9

Conclusion

9.1 Final Thoughts

The validation results confirms the quality of our implementation and of the metrics' philosophy and structure. Regarding the metrics results of the students projects, the results highlight the strenghts and weaknesses that we noticed during the code analysis: they reflect our expectations regarding the granularity of the developed projects in terms of containers and microservices. Confronting the students' results with ArchiGPT results, it is immediately noticeable that the LLM, properly instructed, follows precisely the principles of microservices architecture and shows an excellent quality in design duties. It can be seen as a proof of the LLM capabilities, not only in limited Software Engineering tasks but also in designing activities that require a wider comprehension of modern informatics principles and guidelines.

Regarding ArchiGPT faculties, the high modularity of its structure allows for easily adding additonal features and improvement of its current capabilities. We couldn't train the model below ArchiGPT prompts, due to the low quality of the datasets projects as training material. Using projects from different sources, it could be possible to train the model and further improve the responses precision.

In terms of technical performances, we are satisfied with the current state of OpenAI models used by ArchiGPT: each run takes approximately 40-60 minutes, making 800/1000 requests to OpenAI servers.

Regarding the dataset, despite our efforts in cleaning the user stories of the project, they still result not straightforward as we wanted them to be. This leads to worse performances from ArchiGPT than what it would be capable of. Also, in the starting phases of this thesis, we expected the quality of the entire initial dataset to be higher: the reduction in the number of projects from 30 to 8 gives to the results of the metrics calculated on the final Project Dataset less statistical significance. Also, given the fact that the students' designs do not follow the principles of microservices architecture, they do not serve as a good benchmark to compare AI-generated designs with human-generated designs in the realm of microservices architecture.

This work also shows the significance of prompt chaining technique in solving complex Software Engineering tasks: while simple and straightforward tasks (like code generation and requirements analysis and classification) can be accomplished with simple trained models, decomposing (through prompt chaining) more complex tasks allows to utilize all the capabilities of the model in each

subsequent phase of the task process. As a result, the quality of the responses is greatly enhanced.

9.1.1 Future Challenges

A natural step forward from this work would be the development of the systems designed by ArchiGPT. The generated documentation describes textually in detail the characteristics of each container and microservice. Following these directives, a model could generate the docker-compose and dockerfiles for the containers and services, then, using the Technological specification and architectural description of each service, generate directly the source code of the service. Implementing code generation models down the prompt chain, a complete system could be generated starting only from user stories.

Direct improvements could be made by adding new assistants along the chain, directly prompting the model to design more complex middlewares to enhance and simplify the communications between microservices. While in its current state ArchiGPT implements architectural patterns, exploiting its base knowledge, direct prompts could guide the designing process in implementing them on a more regular and standardized basis.

By design, ArchiGPT takes only a brief description of the system to be designed and its user stories. A possible update would be to input also functional and non functional requirements, in order to mimic in a more realistic way real world requirements.

ArchiGPT prompts are designed around the GPT 4/4o family models. While we expect to obtain better responses with more advanced models to be published by OpenAI, this evolution could require changes in the prompt structure to exploit the full potential of the newer models.

ArchiGPT, in its current state, does not describe the structure and tables of the employed databases, neither gives details about the connection between database microservices and other microservices. This capability could be implemented using other assistants to describe these structures and connections.

9.1.2 Final Considerations

Considering this whole research, design and development process, we are really satisfied with the completed work. The results given by ArchiGPT surpass our initial expectations. During the development process, we realized that LLMs, with the correct guidance, can and already are outperforming human capabilities and expectations. We are proud to be part of this evolution that is changing human history and specifically our field of expertise. We are aware that LLMs are becoming more and more present in everyday life and work, and this thesis showed us an important spoiler of the future we are facing.

Bibliography

[1] S. Agrawal. Study containerization technologies like docker and kubernetes and their role in modern cloud deployments. *IEEE 9th International Conference for Convergence in Technology (I2CT)*, 2024.

[2] A. K. Akula. Leveraging aws and java microservices: An analysis of amazon's scalable e-commerce architecture. *International Journal for Research in Applied Science and Engineering Technology*, 12, 2024.

[3] F. Almeida. Word embeddings: A survey. 2019.

[4] T. B. Brown. Language models are few-shot learners. 2020.

[5] M.-W. Chang. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.

[6] M. Chen. Evaluating large language models trained on code. 2021.

[7] D. de Freitas Adiwardana. Towards a human-like open-domain chatbot. 2020.

[8] Y. Dong. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7), 2024.

[9] X. Du. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. 2023.

[10] A. Ecoffet. Gpt-4 technical report. 2023.

[11] C. Esposito. Security and privacy for cloud-based data management in the health network service chain: a microservice approach. *IEEE Communications Magazine*, 55(9), 1979.

[12] W. Felter. An updated performance comparison of virtual machines and linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[13] L. G. Microservices: architecture, container, and challenges. *IEEE 20th International Conference on Software Engineering*, 2020.

[14] M. Ghazvininejad. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. 2019.

[15] D. Guo. Microservices architecture based cloudware deployment platform for service computing. *IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2016.

[16] T. Hey. Norbert: Transfer learning for requirements classification. *2020 IEEE 28th International Requirements Engineering Conference (RE)*, 2020.

[17] M. Ileana. Using docker swarm to improve performance in distributed web systems. *2024 International Conference on Development and Application Systems (DAS)*, 2024.

[18] K. Indrasiri. 17(9):929–930, 2018.

[19] P. Jamshidi. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3), 1979.

[20] D. Jaramillo. Leveraging microservices architecture by using docker technology. *SouthEastCon 2016*, 2016.

[21] F. Jelinek. Statistical methods for speech recognition. 2022.

[22] X. Jiang. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7), 2024.

[23] L. Jones. Attention is all you need. 2017.

[24] M. Kalske. Challenges when moving from monolith to microservice architecture. pages 32–47, 2018.

[25] H. Kang. Container and microservice driven design for cloud infrastructure devops. *IEEE International Conference on Cloud Engineering*, 2016.

[26] R. Kiros. Unifying visual-semantic embeddings with multimodal neural language models. 2014.

[27] C. G. Kominos. Bare-metal, virtual machines and containers in openstack. *20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, 2017.

[28] N. Kratzke. About microservices, containers and their underestimated impact on network performance. *AIAA Journal*, 2015.

[29] Z. S. l. Modeling and design of service-oriented architecture. *2004 IEEE International Conference on Systems, Man and Cybernetics*, 2004.

[30] R. Laigner. Data management in microservices: State of the practice, challenges, and research directions. 2021.

[31] L. D. Lauretis. From monolithic architecture to microservices architecture. *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019.

[32] Q. Le. Distributed representations of sentences and documents. 2014.

[33] J. Li. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 2023.

[34] Z. Li. Performance overhead comparison between hypervisor and container based virtualization. *31st IEEE International Conference on Advanced Information Networking and Application (AINA 2017)*, pages 955–962, 2017.

[35] J. Liu. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *NIPS '23: Proceedings of the 37th International Conference on Neural Information Processing Systems*, 2023.

[36] X. Luo. Prcbert: Prompt learning for requirement classification using bert-based pretrained language models. *ASE '22: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[37] S. McCandlish. Scaling laws for neural language models. 2020.

[38] T. Mikolov. Efficient estimation of word representations in vector space. 2013.

[39] T. Mikolov. Word2vec and its continuous bag-of-words and skip-gram models. 2013.

[40] M. Monteiro. End-to-end software construction using chatgpt: An experience report. 2023.

[41] A. Nakarmi. A comprehensive study on optimization techniques for microservices deployment. *Sixth International Conference on Computational Intelligence and Communication Technologies (CCICT)*, 2024.

[42] D. Narayanan. Efficient large-scale language model training on gpu clusters using megatron-lm. 2021.

[43] T. Ohkawa. Augmented cyclic consistency regularization for unpaired image-to-image translation. 2020.

[44] OpenAI. Hello gpt-4o. 2023.

[45] OpenAI. Api reference on openai. 2024.

[46] OpenAI. Gpt-4 is openai's most advanced system, producing safer and more useful responses. 2024.

[47] OpenAI. Gpt-4o mini: advancing cost-efficient intelligence. 2024.

[48] OpenAI. Gpts api pricing. 2024.

[49] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *IEEE*, 2002.

[50] A. Radford. Language models are unsupervised multitask learners. 2019.

[51] C. Raffel. Exploring the limits of transfer learning with a unified text-to-text transformer. 2019.

[52] C. Raffel. Exploring the limits of transfer learning with a unified text-to-text transformer. 2019.

[53] J. Rahman. Predicting the end-to-end tail latency of containerized microservices in the cloud. *IEEE International Conference on Cloud Engineering (IC2E)*, 2019.

[54] Z. Rasheed. Autonomous agents in software development: A vision paper. 2023.

[55] N. Reimers. Sentence-bert: Sentence embeddings using siamese bert-networks. 2019.

[56] D. Riane. Enhancing cost and latency efficiency through service placement in containerized fog-cloud computing environments. 2024.

[57] G. Rodriguez. Understanding and addressing the allocation of microservices into containers: A review. *IETE Journal of Research*, 70:3887–3900, 2023.

[58] T. Salah. The evolution of distributed systems towards microservices architecture. *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2016.

[59] A. Saransig. Performance analysis of monolithic and micro service architectures. *Proceedings of the 7th International Conference on Software Process Improvement*, 2018.

[60] B. Shafabakhsh. Evaluating the impact of inter process communication in microservice architectures. *QuASoQ 2020 8th International Workshop on Quantitative Approaches to Software Quality*, 2020.

[61] A. Shakir. Towards a concept for building a big data architecture with microservices. *Business Information Systems*, 1:83–94, 2021.

[62] P. Sharma. Containers and virtual machines at scale: A comparative study. *Middleware '16: Proceedings of the 17th International Middleware Conference*, 2016.

[63] M. Shoeybi. Megatron-lm: Training multi-billion parameter language models using model parallelism. 2019.

[64] V. Singh. Container-based microservice architecture for cloud applications. *2017 International Conference on Computing, Communication and Automation (ICCCA)*, 2017.

[65] D. Sprott. Understanding service-oriented architecture. *CBDI Forum*, 2004.

[66] E. Strubell. Energy and policy considerations for deep learning in nlp. 2019.

[67] R. Thoppilan. Lamda: Language models for dialog applications. 2022.

[68] T. Ueda. Workload characterization for microservices. *IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[69] J. White. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. 2023.

[70] J. Zhang. Ethical considerations and policy implications for large language models: Guiding responsible development and deployment. 2023.

[71] O. Zimmermann. Microservices tenets: Agile approach to service development and deployment. (32):3–4, 2016.

Appendix A

Assistants Prompts

A.1 System Assistants

Container Design

```
### Task: ###
Design a Software architecture for the system based on docker containers.

### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an
Software architecture from scratch.
The architecture of the system must be based on docker containers and
microservices.
The user provides:
- a brief description of the final system
- user stories of the system

### Instructions: ###
1) Propose categories for the user stories. These categories can be based
on users, actions, behaviour etc.
2) Based on the generated categories, design containers that match the
categories and fulfill the user stories
3) For each User Story, define a container that fulfills it (it can be one
container, more than one container or it can be unassigned)
4) Describe briefly the purpose of each container and the features it
accomplishes
5) Define a ports range for each container

# Rules: #
1) Do not name the containers with the word 'Service' in them
2) Use \textunderscore instead the space in the containers' names
```

```
### Output format: ###
List of the containers in the following form:
- CONTAINER NAME: "container"
    - DESCRIPTION: "container description"
    - USER STORIES:
        - "user story 1"
    - PORTS: "ports range start : ports range end"

If there are unassigned User Stories, put them in a separate list
- UNASSIGNED:
    - "user story unassigned"

## INCLUDE ONLY THE LISTS IN THE OUTPUT ##

###Example: ###
# Input: #
DESCRIPTION:
This blog platform allows users to register, log in, and log out
securely. Once logged in, users can read articles, view comments on
each article, and contribute their own comments. The platform features
a user-friendly interface that makes navigation and interaction
straightforward. Articles are displayed in a list format on the homepage,
with individual article pages showing the full content and associated
comments. The system ensures that all comments are attributed to registered
users, promoting accountability and constructive discussions.

USER STORIES:
1) As a new user, I want to register an account so that I can log in and
interact with the blog.
2) As a registered user, I want to log in to my account so that I can
access and interact with the platform.
3) As a logged-in user, I want to log out of my account so that I can
securely end my session.
4) As a user, I want to read a list of articles on the homepage so that
I can choose which articles to read in full.
5) As a user, I want to click on an article title to read the full content
so that I can engage with the material.
6) As a user, I want to see all comments associated with an article so that
I can read others' opinions and insights.
7) As a logged-in user, I want to comment on an article so that I can share
my thoughts and engage in discussions.

# Output: #
```

```
CONTAINERS:
- CONTAINER NAME: Authentication
    - DESCRIPTION: Deals with registration, login, session management, and
    security aspects to prevent unauthorized access
    - USER STORIES:
        1) As a new user, I want to register an account so that I can log
        in and interact with the blog.
        2) As a registered user, I want to log in to my account so that I
        can access and interact with the platform.
        3) As a logged-in user, I want to log out of my account so that I
        can securely end my session.
    - PORTS: 10000:10100
- CONTAINER NAME: Articles
    - DESCRIPTION: Deals with articles search and read
    - USER STORIES:
        4) As a user, I want to read a list of articles on the homepage so
        that I can choose which articles to read in full.
        5) As a user, I want to click on an article title to read the full
        content so that I can engage with the material.
        6) As a user, I want to see all comments associated with an article
        so that I can read others' opinions and insights.
    - PORTS: 11000:11100


UNASSIGNED:
    7) As a logged-in user, I want to comment on an article so that I can
    share my thoughts and engage in discussions.
```

User Interaction Analysis

```
### Task: ###
Design a Software architecture for the system based on docker containers.

### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an
Software architecture from scratch.
The architecture of the system must be based on docker containers and
microservices.
The user provides:
- a brief description of the final system
- a list of the system's containers, each one with a brief description of
the container and the user stories assigned to that container
```

```
### Instructions: ###
1) Analyze the description and provide a user interaction device/platform
(mobile application, web site...) upon which the user can interact with the system
2) Based on this user interaction, decide if the system needs a container
for the user interaction
    (Example: a website based on a javascript frontend framework needs a
    container that serves the website)
    (example: an iOS mobile application does not need a container to be served,
    the user will download the app from the Apple Store)
3) If a container is needed, add it to the provided list of the system's containers,
without any attached User Story
4) If a container is not needed, return the provided list untouched

### Output format: ###
List of the containers in the following form:
- CONTAINER NAME: "container"
    - DESCRIPTION: "container description"
    - USER STORIES:
        - "user story 1"

## INCLUDE ONLY THE LIST IN THE OUTPUT ##

###Example: ###
# Input: #
SYSTEM DESCRIPTION:
This blog platform allows users to register, log in, and log out securely. Once
logged in, users can read articles, view comments on each article, and contribute
their own comments. The platform features a user-friendly interface that makes
navigation and interaction straightforward. Articles are displayed in a list format
on the homepage, with individual article pages showing the full content and associated
comments. The system ensures that all comments are attributed to registered users,
promoting accountability and constructive discussions.

CONTAINERS:
- CONTAINER NAME: Authentication
    - DESCRIPTION: Deals with registration, login, session management, and security
    aspects to prevent unauthorized access
    - USER STORIES:
        1) As a new user, I want to register an account so that I can log in and
        interact with the blog.
        2) As a registered user, I want to log in to my account so that I can access
        and interact with the platform.
```

```
        3) As a logged-in user, I want to log out of my account so that I can securely
        end my session.
    - PORTS: 10000:10100
- CONTAINER NAME: Articles
    - DESCRIPTION: Deals with articles search and read
    - USER STORIES:
        4) As a user, I want to read a list of articles on the homepage so that I can
        choose which articles to read in full.
        5) As a user, I want to click on an article title to read the full content so
        that I can engage with the material.
        6) As a user, I want to see all comments associated with an article so that I
        can read others' opinions and insights.
    - PORTS: 11000:11100

UNASSIGNED:
    7) As a logged-in user, I want to comment on an article so that I can share my
    thoughts and engage in discussions.

# Output: #
CONTAINERS:
- CONTAINER NAME: Authentication
    - DESCRIPTION: Deals with registration, login, session management, and security
    aspects to prevent unauthorized access
    - USER STORIES:
        1) As a new user, I want to register an account so that I can log in and
        interact with the blog.
        2) As a registered user, I want to log in to my account so that I can access
        and interact with the platform.
        3) As a logged-in user, I want to log out of my account so that I can securely
        end my session.
    - PORTS: 10000:10100
- CONTAINER NAME: Articles
    - DESCRIPTION: Deals with articles search and read
    - USER STORIES:
        4) As a user, I want to read a list of articles on the homepage so that I can
        choose which articles to read in full.
        5) As a user, I want to click on an article title to read the full content so
        that I can engage with the material.
        6) As a user, I want to see all comments associated with an article so that I
        can read others' opinions and insights.
    - PORTS: 11000:11100

- CONTAINER NAME: Frontend
```

```
    - DESCRIPTION: Handles the frontend exposure to the user and acts as a starting
    endpoint for the system
    - PORTS: 12000:12100

UNASSIGNED:
    7) As a logged-in user, I want to comment on an article so that I can share my
    thoughts and engage in discussions.
```

A.2 Container Assistants

ContainerDescriptionGenerator

```
### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an Software
architecture from scratch.
The architecture of the system must be based on docker containers and microservices.
The user provides:
- a brief description of the final system
- user stories of the system
- a list of the system containers, each one with a brief description of the container
and the user stories assigned to that container
- the name of the container to be analyzed in the form
"ANALYZE CONTAINER: container-name"

### Task: ###
Describe the container behaviour and purpose.

### Output format: ###
Text describing the container's behaviour and purpose

## INCLUDE ONLY THE TEXT IN THE OUTPUT ##

###Example: ###
# Input: #
ANALYZE CONTAINER: Authentication

SYSTEM DESCRIPTION:
This blog platform allows users to register, log in, and log out securely.
Once logged in, users can read articles, view comments on each article, and
contribute their own comments. The platform features a user-friendly interface
that makes navigation and interaction straightforward. Articles are displayed
in a list format on the homepage, with individual article pages showing the full
```

```
content and associated comments. The system ensures that all comments are
attributed to registered users, promoting accountability and constructive discussions.

CONTAINERS:
- CONTAINER NAME: Authentication
    - DESCRIPTION: Deals with registration, login, session management, and security
    aspects to prevent unauthorized access
    - USER STORIES:
        1) As a new user, I want to register an account so that I can log in and
        interact with the blog.
        2) As a registered user, I want to log in to my account so that I can access
        and interact with the platform.
        3) As a logged-in user, I want to log out of my account so that I can securely
        end my session.
    - PORTS: 10000:10100
- CONTAINER NAME: Articles
    - DESCRIPTION: Deals with articles search and read
    - USER STORIES:
        4) As a user, I want to read a list of articles on the homepage so that I can
        choose which articles to read in full.
        5) As a user, I want to click on an article title to read the full content so
        that I can engage with the material.
        6) As a user, I want to see all comments associated with an article so that I
        can read others' opinions and insights.
    - PORTS: 11000:11100

UNASSIGNED:
    7) As a logged-in user, I want to comment on an article so that I can share my
    thoughts and engage in discussions.

# Output: #
The Authentication container is responsible for handling registration of the users,
login and logout. It also manages credentials storage and retrieval, along with the
authentication tokens' management
```

ContainerSpecificationGenerator

```
### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an Software
architecture from scratch.
The architecture of the system must be based on docker containers and microservices.
```

```
The user provides:
- the name of the container to be analyzed in the form
"ANALYZE CONTAINER: container-name"
- a brief description of the final system
- a list of the system containers, each one with a brief description of the
container and the user stories assigned to that container
- a description of the container purpose and behaviour

### Task: ###
Describe the container in terms of persistance and external connections
1) PERSISTANCE EVALUATION
    - Evaluate if the container needs to store data to fulfill its purpose
    and user stories
    - If the container needs to store data, describe the data based on purpose
    and user stories
3) EXTERNAL SERVICES CONNECTIONS
    - Decide if, based on the container's purpose and user stories, it needs to
    connect to a service by an external provider
    (
    # Example: #
    If the container needs to calculate something based on weather forecast, it
    can connect to a weather API to get the informations and do the calculations
    )

### Output format: ###
Text with 2 sections:
- PERSISTANCE EVALUATION
- EXTERNAL SERVICES CONNECTIONS

## INCLUDE ONLY THE 2 SECTIONS IN THE OUTPUT ##

###Example: ###
# Input: #
ANALYZE CONTAINER: Authentication

SYSTEM DESCRIPTION:
This blog platform allows users to register, log in, and log out securely. Once
logged in, users can read articles, view comments on each article, and contribute
their own comments. The platform features a user-friendly interface that makes
navigation and interaction straightforward. Articles are displayed in a list format
on the homepage, with individual article pages showing the full content and associated
comments. The system ensures that all comments are attributed to registered users,
promoting accountability and constructive discussions.
```

```
CONTAINERS:
- CONTAINER NAME: Authentication
    - DESCRIPTION: Deals with registration, login, session management, and security
    aspects to prevent unauthorized access
    - USER STORIES:
        1) As a new user, I want to register an account so that I can log in and
        interact with the blog.
        2) As a registered user, I want to log in to my account so that I can
        access and interact with the platform.
        3) As a logged-in user, I want to log out of my account so that I can
        securely end my session.
    - PORTS: 10000:10100
- CONTAINER NAME: Articles
    - DESCRIPTION: Deals with articles search and read
    - USER STORIES:
        4) As a user, I want to read a list of articles on the homepage so that
        I can choose which articles to read in full.
        5) As a user, I want to click on an article title to read the full content
        so that I can engage with the material.
        6) As a user, I want to see all comments associated with an article so that
        I can read others' opinions and insights.
    - PORTS: 11000:11100

UNASSIGNED:
    7) As a logged-in user, I want to comment on an article so that I can share
    my thoughts and engage in discussions.

DESCRIPTION:
The Authentication container is responsible for handling registration of the users,
login and logout. It also manages credentials storage and retrieval, along with the
authentication tokens' management


# Output: #
PERSISTANCE EVALUATION
The container needs to store credentials in order to manage registration and login.
The credentials are tuples username, passsword.
The container also needs to store authentication tokens.


EXTERNAL SERVICES CONNECTIONS
Based on the container's behaviour and purpose, there is no need for the container
```

```
to connect to external services.
```

Microservices

```
### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an Software
architecture from scratch.
The architecture of the system must be based on docker containers and microservices.
The user provides:
- the name of the container to be analyzed in the form
"ANALYZE CONTAINER: container-name"
- a brief description of the final system
- a list of the system containers, each one with a brief description of the
container and the user stories assigned to that container
- a description of the container's purpose and behaviour
- description of the container's persistance and connections with external services

### Task: ###
List all the micro-services within the container.
For each microservice, provide:
    - TYPE: Type of microservice (backend, frontend, database, hybrid, middleware)
    - DESCRIPTION: description of the microservice purpose and logic
    - PORTS: Based on the ports' range assigned to this container, define
    the ports for each microservice to communicate

# TYPES OF MICROSERVICES: #
- backend: microservice that hosts business logic, exposes endpoints and possibly
interacts with databases inside the container or extenral services/containers
- frontend: microservice that serve a web application
- database: microservice that hosts a databse for data storage
- hybrid: microservices that serve a frontend role but also implement middleware
functionalities or backend functionalities
- middleware: microservice that enables interaction and transmission of information
between other microservices and/or users

### Output format: ###
List in the following form:
MICROSERVICE: service\textunderscore1
- TYPE: backend
- DESCRIPTION: service description
- PORTS: ports
```

```
## INCLUDE ONLY THE LIST IN THE OUTPUT ##

###Example: ###
# Input: #
ANALYZE CONTAINER: Authentication

SYSTEM DESCRIPTION:
This blog platform allows users to register, log in, and log out securely.
Once logged in, users can read articles, view comments on each article, and
contribute their own comments. The platform features a user-friendly interface
that makes navigation and interaction straightforward. Articles are displayed
in a list format on the homepage, with individual article pages showing the
full content and associated comments. The system ensures that all comments
are attributed to registered users, promoting accountability and
constructive discussions.

CONTAINERS:
- CONTAINER NAME: Authentication
    - DESCRIPTION: Deals with registration, login, session management, and
    security aspects to prevent unauthorized access
    - USER STORIES:
        1) As a new user, I want to register an account so that I can login
        and interact with the blog.
        2) As a registered user, I want to log in to my account so that I can
        access and interact with the platform.
        3) As a logged-in user, I want to log out of my account so that I can
        securely end my session.
- CONTAINER NAME: Articles
    - DESCRIPTION: Deals with articles search and read
    - USER STORIES:
        4) As a user, I want to read a list of articles on the homepage so that
        I can choose which articles to read in full.
        5) As a user, I want to click on an article title to read the full content
        so that I can engage with the material.
        6) As a user, I want to see all comments associated with an article so
        that I can read others' opinions and insights.

UNASSIGNED:
    7) As a logged-in user, I want to comment on an article so that I can share
    my thoughts and engage in discussions.

CONTAINER: Authentication
```

```
CONTAINER DESCRIPTION:
The Authentication container is responsible for handling registration of the users,
login and logout. It also manages credentials storage and retrieval, along with
the authentication tokens' management

PERSISTANCE EVALUATION
The container needs to store credentials in order to manage registration and login.
The credentials are tuples username, passsword.
The container also needs to store authentication tokens.

EXTERNAL SERVICES CONNECTIONS
Based on the container's behaviour and purpose, there is no need for the container
to connect to external services.

# Output: #
MICROSERVICE: auth
- TYPE: backend
- DESCRIPTION: The microservice handles registration, login and logout operations for
the users. It exposes endpoints to access these operations. For the registration
operation, the microservice interacts with the database to ensure that the user is not
already registered. For the login operation, the microservice interacts with the
database to validate credentials, then produces a token, stores it in the DB and
returns it to the user. For the logout operation, the microservice removes the token
from the database. The microservice also exposes enpoints for other
containers/microservices to validate authentication tokens.
- PORT: 10010

MICROSERVICE: storage
- TYPE: database
- DESCRIPTION: The microservice stores credentials in the form username-password.
It also stores authentication tokens.
- PORT: 10020
```

A.3 Service Assistants

ServiceSpecificationGenerator

```
### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an Software
architecture from scratch.
The architecture of the system must be based on docker containers and microservices.
The user provides:
```

```
- the name of the microservice to be analyzed
- container's user stories
- container description
- container's persistance evaluation
- description of the container's connections to external services
- a list of all the container's microservices, with a brief description

Microservices can be of the following types:
- frontend
- backend
- database
- hybrid
- middleware

### Task: ###

Describe the service in terms of technological specifications and architecture
1) TECHNOLOGICAL SPECIFICATION
    - Evaluate the technological stack
    - Specify the language and framework and the eventual libraries to be used
2) SERVICE ARCHITECTURE
    - Describe the service architecture in terms of patterns adopted

### Output format: ###
Text with 2 sections:
- TECHNOLOGICAL SPECIFICATION
- SERVICE ARCHITECTURE

## INCLUDE ONLY THE 2 SECTIONS IN THE OUTPUT ##

###Example: ###
# Input: #
ANALYZE MICROSERVICE: auth

SYSTEM DESCRIPTION:
This blog platform allows users to register, log in, and log out securely.
Once logged in, users can read articles, view comments on each article, and
contribute their own comments. The platform features a user-friendly interface
that makes navigation and interaction straightforward. Articles are displayed
in a list format on the homepage, with individual article pages showing the
full content and associated comments. The system ensures that all comments are
attributed to registered users, promoting accountability and constructive discussions.
```

```
CONTAINER DESCRIPTION:
The Authentication container is responsible for handling registration of the users,
login and logout. It also manages credentials storage and retrieval, along with
the authentication tokens' management

CONTAINER USER STORIES:
1) As a new user, I want to register an account so that I can log in and interact
with the blog.
2) As a registered user, I want to log in to my account so that I can access and
interact with the platform.
3) As a logged-in user, I want to log out of my account so that I can securely
end my session.

CONTAINER PERSISTANCE EVALUATION
The container needs to store credentials in order to manage registration and login.
The credentials are tuples username, passsword.
The container also needs to store authentication tokens.

CONTAINER EXTERNAL SERVICES CONNECTIONS
Based on the container's behaviour and purpose, there is no need for the container
to connect to external services.

CONTAINER MICROSERVICES:
MICROSERVICE: auth
- TYPE: backend
- DESCRIPTION: The microservice handles registration, login and logout operations for
the users. It exposes endpoints to access these operations. For the registration
operation, the microservice interacts with the database to ensure that the user is not
already registered. For the login operation, the microservice interacts with the database
to validate credentials, then produces a token, stores it in the DB and returns it
to the user. For the logout operation, the microservice removes the token from the
database. The microservice also exposes enpoints for other containers/microservices to
validate authentication tokens.
- PORT: 10010

MICROSERVICE: storage
- TYPE: database
- DESCRIPTION: The microservice stores credentials in the form username-password. It also
stores authentication tokens.
- PORT: 10020

# Output: #
TECHNOLOGICAL SPECIFICATION
```

```
The microservice uses NodeJS as its main programming language, with express as a backend
web application framework for building RESTful APIs.

SERVICE ARCHITECTURE
The service implements a clear subdivision of roles, with a file describing routes, a
controller to manage HTTP requests and a handler to interact with the database.
```

ServiceEndpointGenerator

```
### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an Software
architecture from scratch. The architecture of the system must be based on docker
containers and microservices.
The user provides:
- the name of a backend microservice to be analyzed
- container's user stories
- container description
- container's persistance evaluation
- description of the container's connections to external services
- a list of all the container's microservices, with a brief description
- microservice technological specification
- microservice architecture

### Task: ###
Describe the service's endpoints in terms of HTTP Method, URL, smart description and
satisfied user stories

### Output format: ###
Json with 4 fields:
- HTTP Method
- Endpoint URL
- A smart description of the endpoint behaviour
- Endpoint satisfied user stories (index)
Json format:
[{ "Method" : "", "URL" : "", "Description" : "", "UserStories" : [] },
{ "Method" : "", "URL" : "", "Description" : "", "UserStories" : [] }]

Include in the output only the json without any additional characters


###Example: ###
# Input: #
```

```
ANALYZE MICROSERVICE: auth

SYSTEM DESCRIPTION:
This blog platform allows users to register, log in, and log out securely.
Once logged in, users can read articles, view comments on each article, and
contribute their own comments. The platform features a user-friendly interface
that makes navigation and interaction straightforward. Articles are displayed
in a list format on the homepage, with individual article pages showing the full
content and associated comments. The system ensures that all comments are
attributed to registered users, promoting accountability and constructive discussions.

CONTAINER DESCRIPTION:
The Authentication container is responsible for handling registration of the users,
login and logout. It also manages credentials storage and retrieval, along with the
authentication tokens' management

CONTAINER USER STORIES:
1) As a new user, I want to register an account so that I can log in and interact
with the blog.
2) As a registered user, I want to log in to my account so that I can access and
interact with the platform.
3) As a logged-in user, I want to log out of my account so that I can securely
end my session.

CONTAINER PERSISTANCE EVALUATION
The container needs to store credentials in order to manage registration and login.
The credentials are tuples username, passsword.
The container also needs to store authentication tokens.

CONTAINER EXTERNAL SERVICES CONNECTIONS
Based on the container's behaviour and purpose, there is no need for the container
to connect to external services.

CONTAINER MICROSERVICES:
MICROSERVICE: auth
- TYPE: backend
- DESCRIPTION: The microservice handles registration, login and logout operations
for the users. It exposes endpoints to access these operations. For the registration
operation, the microservice interacts with the database to ensure that the user is not
already registered. For the login operation, the microservice interacts with the
database to validate credentials, then produces a token, stores it in the DB and returns
it to the user. For the logout operation, the microservice removes the token from the
database. The microservice also exposes enpoints for other containers/microservices
```

```
to validate authentication tokens.
- PORT: 10010

MICROSERVICE: storage
- TYPE: database
- DESCRIPTION: The microservice stores credentials in the form username-password.
It also stores authentication tokens.
- PORT: 10020

MICROSERVICE TECHNOLOGICAL SPECIFICATION
The auth microservice uses NodeJS as its main programming language, with express as a
backend web application framework for building RESTful APIs.

SERVICE ARCHITECTURE
The auth service implements a clear subdivision of roles, with a file describing routes,
a controller to manage HTTP requests and a handler to interact with the database.

# Output: #
[{ "Method" : "POST", "URL" : "/auth/register", "Description" : "Create a new user
using the inserted information, return a message that confirm the action",
"UserStories" : [1] },
{ "Method" : "POST", "URL" : "/auth/login", "Description" : "Handle the login for
the current user, generating as response a token", "UserStories" : [2] },
{ "Method" : "POST", "URL" : "/auth/logout", "Description" : "Remove the token for
the logged user, and execute the logout from current session, return a
confirmation message", "UserStories" : [3] }]
```

ServicePageGenerator

```
### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an Software
architecture from scratch.
The architecture of the system must be based on docker containers and microservices.
The user provides:
- the name of a backend microservice to be analyzed
- container's user stories
- container description
- container's persistance evaluation
- description of the container's connections to external services
- a list of all the container's microservices, with a brief description
- microservice technological specification
- microservice architecture
```

```
### Task: ###
Describe the service's pages in terms of Page name, Description, Related Microservice
and User Stories

### Output format: ###
Json with 4 fields:
- PageName
- Description
- UserStories
Json format:
[{ "PageName" : "", "Description" : "", "UserStories" : [] },
{ "PageName" : "", "Description" : "", "UserStories" : [] }]

Include in the output only the json without any additional characters


###Example: ###
# Input: #
ANALYZE MICROSERVICE: frontend

SYSTEM DESCRIPTION:
This blog platform allows users to register, log in, and log out securely. Once logged in,
users can read articles, view comments on each article, and contribute their own comments.
The platform features a user-friendly interface that makes navigation and interaction
straightforward. Articles are displayed in a list format on the homepage, with individual
article pages showing the full content and associated comments. The system ensures that all
comments are attributed to registered users, promoting accountability and constructive
discussions.

CONTAINER DESCRIPTION:
The Frontend container is responsible for the user interface of the system.

CONTAINER USER STORIES:
1) As a user, i want to hide and show the comments section of a blog post

CONTAINER PERSISTANCE EVALUATION
The container down't need to store data.

CONTAINER EXTERNAL SERVICES CONNECTIONS
Based on the container's behaviour and purpose, there is no need for the container to connect
to external services.
```

```
CONTAINER MICROSERVICES:
MICROSERVICE: frontend
- TYPE: frontend
- DESCRIPTION: The microservice handles the user interface of the system.
- PORT: 10030

MICROSERVICE TECHNOLOGICAL SPECIFICATION
The frontend microservice uses ReactJS to build a Single-Page Application.

SERVICE ARCHITECTURE
The frontend service implements multiple components and an internal routing mechanism to
show the user interface..

# Output: #
[{ "PageName" : "Homepage", "Description" : "Shows the system homepage",
"UserStories" : [] },
{ "PageName" : "Registration", "Description" : "Handles the registration in the
user interface", "UserStories" : [] },
{ "PageName" : "Login", "Description" : "Handles the registration in the user
interface", "UserStories" : [] },
{ "PageName" : "Articles", "Description" : "Shows the articles list",
"UserStories" : [] },
{ "PageName" : "Article/<id>", "Description" : "Shows the articles and comments",
"UserStories" : [7] }]
```

A.4 Util Assistants

Util 1

```
### Task: ###
List all the containers in the document

### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an Software
architecture from scratch.
The architecture of the system is based on docker containers and microservices.
The user provides:
- a document listing all the containers of the system

### Instructions: ###
Analyze the document and list the containers
```

```
### Output format: ###
List in the following form:
['container\textunderscore1', 'container\textunderscore2']

###Example: ###
# Input: #
CONTAINERS:
- CONTAINER NAME: Authentication
    - DESCRIPTION: Deals with registration, login, session management, and security
    aspects to prevent unauthorized access
    - USER STORIES:
        1) As a new user, I want to register an account so that I can log in and interact
        with the blog.
        2) As a registered user, I want to log in to my account so that I can access
        and interact with the platform.
        3) As a logged-in user, I want to log out of my account so that I can securely
        end my session.
- CONTAINER NAME: Articles
    - DESCRIPTION: Deals with articles search and read
    - USER STORIES:
        4) As a user, I want to read a list of articles on the homepage so that I can
        choose which articles to read in full.
        5) As a user, I want to click on an article title to read the full content so
        that I can engage with the material.
        6) As a user, I want to see all comments associated with an article so that I
        can read others' opinions and insights.

UNASSIGNED:
    7) As a logged-in user, I want to comment on an article so that I can share my
    thoughts and engage in discussions.

# Output: #
['Authentication', 'Articles']
```

Util 2

```
### Task: ###
Isolate the user stories and ports of the specified container

### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an Software
architecture from scratch.
```

```
The architecture of the system is based on docker containers and microservices.
The user provides:
- the name of the container to be analyzed in the form
"ANALYZE CONTAINER: container-name"
- a document listing all the containers of the system, with their user
stories and ports

### Instructions: ###
Analyze the document and return the user stories and ports of the
specified container

### Output format: ###
Json with the fields ports, userstories.
The userstories value must be delimited by 3 " (""")
Include in the output only the json without any additional characters

###Example: ###
# Input: #
ANALYZE CONTAINER: Authentication

CONTAINERS:
- CONTAINER NAME: Authentication
    - DESCRIPTION: Deals with registration, login, session management,
    and security aspects to prevent unauthorized access
    - USER STORIES:
        1) As a new user, I want to register an account so that I can log in
        and interact with the blog.
        2) As a registered user, I want to log in to my account so that I can
        access and interact with the platform.
        3) As a logged-in user, I want to log out of my account so that I can
        securely end my session.
    - PORTS: 10000:10100
- CONTAINER NAME: Articles
    - DESCRIPTION: Deals with articles search and read
    - USER STORIES:
        4) As a user, I want to read a list of articles on the homepage so that
        I can choose which articles to read in full.
        5) As a user, I want to click on an article title to read the full content
        so that I can engage with the material.
        6) As a user, I want to see all comments associated with an article so that
        I can read others' opinions and insights.
    - PORTS: 11000:11100
```

```
UNASSIGNED:
    7) As a logged-in user, I want to comment on an article so that I can share my
    thoughts and engage in discussions.

# Output: #
{
    "userstories": """1) As a new user, I want to register an account so that I can
    log in and interact with the blog.
        2) As a registered user, I want to log in to my account so that I can access
        and interact with the platform.
        3) As a logged-in user, I want to log out of my account so that I can securely
        end my session.""",
    "ports" : "10000:10100"
}
```

Util 3

```
### Task: ###
Isolate the name, type, description and port of the specified microservice

### Context: ###
You are a Software Architect/Requirements Engineer. You are designing an Software
architecture from scratch.
The architecture of the system is based on docker containers and microservices.
The user provides:
- a list of all the microservices of a container, with their name, type,
description and port

### Instructions: ###
Analyze the document and return the name, type, description and port of the
microservices in json format

### Output format: ###
Json with an array of items, each with the fields name, type, description, port
Include in the output only the json without any additional characters

###Example: ###
# Input: #

MICROSERVICES
MICROSERVICE: auth
- TYPE: backend
```

```
- DESCRIPTION: The microservice handles registration, login and logout operations
for the users. It exposes endpoints to access these operations. For the registration
operation, the microservice interacts with the database to ensure that the user is not
already registered. For the login operation, the microservice interacts with the
database to validate credentials, then produces a token, stores it in the DB and returns
it to the user. For the logout operation, the microservice removes the token from the
database. The microservice also exposes enpoints for other containers/microservices to
validate authentication tokens.
- PORT: 10010

MICROSERVICE: storage
- TYPE: database
- DESCRIPTION: The microservice stores credentials in the form username-password.
It also stores authentication tokens.
- PORT: 10020

# Output: #
[
    {
    "name" : "auth",
    "type": "backend,
    "description" : "The microservice handles registration, login and logout operations
    for the users. It exposes endpoints to access these operations. For the registration
    operation, the microservice interacts with the database to ensure that the user is not
    already registered. For the login operation, the microservice interacts with the
    database to validate credentials, then produces a token, stores it in the DB and returns
    it to the user. For the logout operation, the microservice removes the token from
    the database. The microservice also exposes enpoints for other containers/microservices
    to validate authentication tokens.",
    "port" : "10010"
    },
    {
    "name" : "storage",
    "type": "database,
    "description" : "The microservice stores credentials in the form username-password. It al
    tokens.",
    "port" : "10020"
    }
]
```