POLITECNICO DI TORINO

Master's Degree course in Computer Engineering

Master's Degree Thesis

Kube: a cloud ERP system based on microservices and serverless architecture

Supervisor
prof. Alessandro Fiori

Candidate
Thomas Cristofaro

Academic Year 2022-2023

A me e alla mia
determinazione

Summary

In the context of digital transformation, the necessity for agile, efficient, and scalable corporate information systems is paramount, with Enterprise Resource Planning (ERP) systems at the forefront of integrating and optimizing business processes. Confronted with growing complexity and the demand for flexibility, traditional ERP architectures exhibit considerable limitations. This thesis delves into the innovative realms of microservices architecture and serverless technologies, such as Function-as-a-Service (FaaS) and cloud services, proposing them as robust solutions to these challenges. An extensive analysis and a real-world application case study form the core of this research, focusing on the design and development of an ERP software platform tailored for small and medium-sized enterprises (SMEs). This platform not only meets the specific requirements of SMEs but also encapsulates modern concepts of distributed architecture and novel cloud-based programming paradigms. Aiming to optimize cost-effectiveness for end-users, even with a pricing model divergent from existing platforms, the study culminates in the creation of a prototype ERP system. This is achieved by integrating various cloud services and developing a cross-platform client front-end using Flutter, showcasing the viability and advantages of the proposed architectural shift. The thesis also tackles the intricacies of this technological evolution, discussing potential challenges and establishing best practices. Ultimately, it provides practitioners and business decision-makers with comprehensive insights and practical tools, supporting a successful transition to more resilient, agile, and future-ready ERP systems.

Acknowledgements

I would like to extend my heartfelt thanks to all those who supported me throughout my university journey. A special mention to my friends, with whom I shared countless beautiful moments, enriching my experience.

I am deeply grateful to all my family, in particular to: Veronique, Roberto, and my sister Erica, who have always believed in me. Their faith has been a constant source of strength. To my little nephews, Leonardo and Alessandro, whose presence brings immense joy to my life, thank you for brightening my days.

I am also immensely grateful to Angelica, whose unwavering support and continuous encouragement have been crucial, especially during these last few challenging months of thesis writing.

This accomplishment would not have been possible without their collective support. Thank you all.

I extend my gratitude to my thesis advisor, Prof. Alessandro Fiori, for granting me the opportunity to work on this intriguing project. Additionally, I am thankful to Horsa Way Srl for allowing me to undertake this project alongside their team.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's era of digital transformation, there's an urgent need for agile, efficient, and scalable information systems in the corporate world, with Enterprise Resource Planning (ERP) systems playing a critical role in integrating and optimizing business processes. However, the increasing complexity and the requirement for flexibility highlight the limitations of traditional ERP architectures. These platforms often rely on outdated technology, are built on monolithic architectures, and follow archaic pricing models. Many such systems struggle with compatibility issues on modern operating systems, suffer from obsolete graphical interfaces, and are not user-friendly.

The aim of this thesis, therefore, is to develop an ERP platform called Kube that encapsulates ERP functionalities within a modern technological framework. This includes seamless cloud integration and an updated pricing model. The envisioned platform is designed to be cross-platform compatible, ensuring accessibility across all current operating systems without the hassle of complex installations. This approach is intended to address the shortcomings of traditional ERP systems by offering a solution that is both technologically advanced and adaptable to the evolving digital landscape.

1.1 Overview of the Thesis

This document presents a thorough exploration of the key arguments and design choices essential to achieving the set objectives. The thesis starts with the Chapter 2, an in-depth look into Enterprise Resource Planning (ERP) systems. It starts with a discussion on the motivations behind implementing ERP, highlighting the benefits and strategic advantages they offer. The chapter then critically assesses the drawbacks of ERP systems, providing a well-rounded perspective on their limitations and challenges. It also details the various modules of ERP systems and illustrates how they collaborate to enhance business processes. A significant portion of this chapter is dedicated to the technology underpinning ERP systems, with a special focus on the Three-Tier Client-Server Architecture. Additionally, the market-related aspects of ERP systems are examined, including an analysis of the costs involved in their implementation and maintenance, and a comparison of the competitive landscape.

Chapter 3 marks the beginning of our deep dive into the foundational narrative of our thesis. This chapter introduces the key design concepts and architectural principles behind the Kube platform. It emphasizes cloud computing as the cornerstone of the platform's operating environment. The chapter also delves into the concept of microservices, an architectural approach that segments applications into smaller, independently functioning services. A critical element of Kube's architecture is its serverless framework, which promotes scalability and operational efficiency. The implementation of sagas is discussed, outlining their role in managing failures and maintaining data consistency across services. The chapter concludes with an exploration of event-driven architecture, highlighting its importance in enabling reactive programming and facilitating responsive interactions within the platform. These elements collectively define the architectural design of Kube, with a focus on flexibility, scalability, and seamless service integration.

Then, the Chapter 4 of the thesis transitions into the practical application, specifically focusing on the technologies implemented in the Kube platform. Central to this are the AWS services, which include Lambda for serverless computing, SQS and SNS for effective messaging, and RDS for efficient database management. These services form a robust and scalable infrastructure essential to Kube's functionality. The platform also capitalizes on serverless architecture to enhance resource efficiency and minimize operational costs. Additionally, the Go programming language is employed for its effectiveness in constructing high-performance applications. The user interface of Kube, designed using Flutter, offers both versatility and aesthetic appeal, significantly improving the platform's user experience. Collectively, these technologies are pivotal in ensuring Kube's high performance and user satisfaction.

Finally, in Chapter 5, the focus shifts to the actual implementation of the application. The chapter begins by delineating the fundamental requirements that have guided the Kube platform's development, emphasizing strategic goals, technical necessities, and business rationale. This sets the stage for a comprehensive exploration of the platform's final architecture, which is marked by a Microservices framework executed through a Function as a Service (FaaS) model, utilizing AWS services. The narrative then moves to practical use cases, showcasing how the platform functions in real-world scenarios. The chapter concludes by examining the client application, specifically the development and design of its user interface using Flutter. This not only augments the robustness of Kube's backend but also significantly enhances user interaction. This chapter aims to demystify the complexities of the Kube platform, underlining its significance as a revolutionary tool in enterprise resource planning and its vast potential in the field.

Chapter 2

ERP: Enterprise resource planning

Business systems known as enterprise resource planning (ERP) systems consolidate and simplify data from several organizational departments into a single, comprehensive solution that satisfies the demands of the whole corporation. ERP systems function by seamlessly integrating and coordinating activities and tasks that were previously fragmented and supported by older, stand-alone, and separate legacy systems. The basis of an ERP system is a well-structured database that supports the operational and decision-making requirements of end users across the company and generates information for external constituencies, such as regulatory agencies and investors.

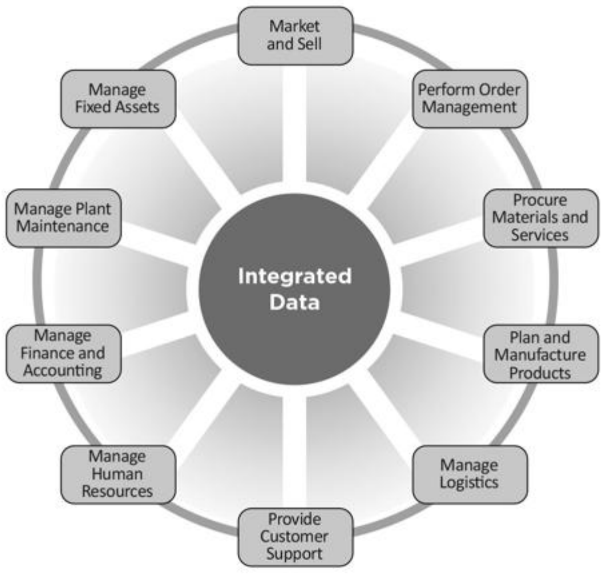


Figure 2.1. ERP-Supported Business Processes

ERP systems are regarded as **cross-functional** in nature, since they satisfy the information needs of all end users, and also **process-centered** because they offer a clear, full, logical, and precise view of the business processes of the firm, which are groups of interconnected tasks that bring value to the enterprise. Business operations frequently cross departmental boundaries and, in many circumstances, cross organizational boundaries, sharing data and information with external business partners like clients and suppliers, making ERPs essential within a company. Some key business processes incorporated in ERP systems are shown in Figure 2.1.

2.1 Reasons for Implementing ERP

Businesses that use ERP systems the most often have a lot of the same issues and frustrations. Figure 2.2 lists the primary justifications for ERP adoption by businesses. A few of these reasons are explored below.

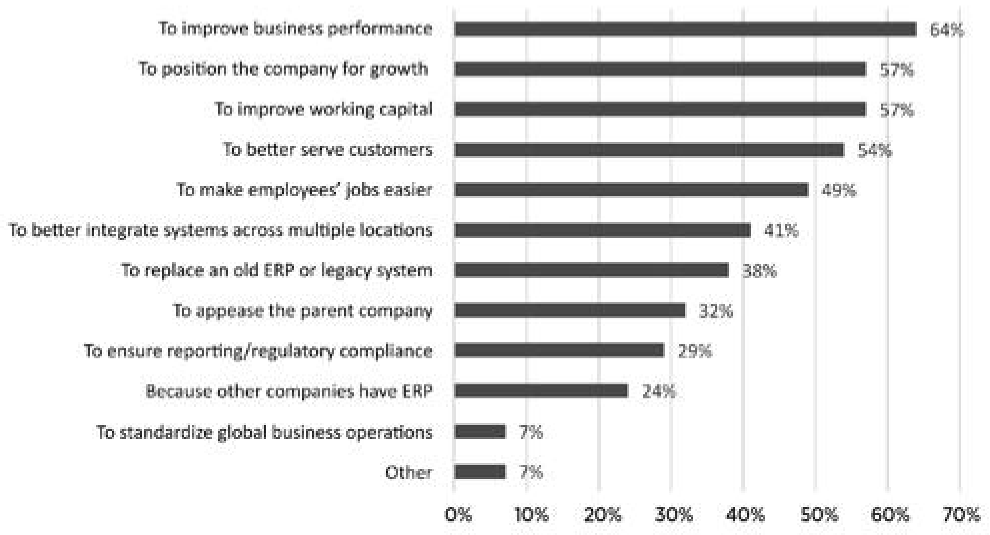


Figure 2.2. Reasons for Implementing ERP

Improve business performance

ERP improves business performance through the numerous best practices embedded within the several business processes. A **best practice** is a business procedure that is generally regarded as being more successful and/or efficient than others in a certain sector. Companies that implement ERP will end up redesigning their previously disjointed, erroneous, slow, and ineffective processes to align with best practices in the software and can decrease operational costs, such as lower inventory costs, production costs, or purchasing costs, and increase revenue-generating processes, such as time to market, marketing and

sales, and customer service.
Each view of best practices distinguishes one ERP vendor's software from another's, thus finding which ERP system's best practices match a buyer's demands is essential when choosing an ERP vendor's product since this fit influences the implementation's final success. In order to find best practices across different industries and implement them into their solutions, the suppliers fund significant research and development (R&D) initiatives. Additionally, it enables an ERP provider to provide niche versions of its software known as **vertical solutions**, which are essential due to the unique characteristics that each industrial sector has.

Desire for growth

Examples of growth strategies include market expansion and penetration, product diversification, and mergers and acquisitions (M&A). ERP systems help with market expansion through demand forecasting, which generates predictions to estimate the future requirements for items. Advanced rule-based pricing is another feature of ERP software. This capability enables businesses to comprehend the present patterns and trends of the sector, consumers, and rivals before making any pricing modifications. Businesses may expand their product lines and provide new items and features to their clients by diversifying their product offerings. Data on which items are selling and to whom may be found in ERP systems, as well as information on which products are just taking up space on the shelves. Finally, an ERP may assist in standardizing procedures across organizations during an M&A activity in order to integrate them into a common platform.

Facilitate employees' work

ERP systems are also recognized to make the duties of employees easier. ERP does this in part by providing employees with real-time access to information; this feature significantly enhances operations, corporate governance, and enterprise risk management, resulting in a horizontally "connected up," process-centered organization. ERP systems also offer an unified user interface and tool set that improves accuracy, encourages collaboration, and reduces misunderstanding. Finally, ERP systems empower users by providing them with access to data that was previously impossible to get due to fragmented procedures supported by many older systems.

Lack of compliance

Government and institutional compliance requirements continue to grow and evolve. Navigating through numerous legal, regulatory, and supply chain mandates has never been tougher. ERP systems can help companies comply with these requirements, such as GDPR, SOX or Food and Drug Administration.

Data integration

With ERP systems, data is better integrated since it is only gathered once and then shared throughout the company, reducing the risk of inaccuracies and duplications and eliminating time-consuming data checking, and reconciliation across systems. Because all users have access to up-to-date, accurate, and comprehensive data, this feature is advantageous to all of them. With ERP, since data is now kept in a single data repository, the process of fixing errors is made simpler because they only need to be fixed once. Processes are also better integrated because they are managed within one system, not spread across multiple systems that have been cobbled together. When a corporation's systems are patched together from several sources, the scenario can cause problems on the operations designed to keep the organization functioning efficiently.

Replacement an old ERP

Having multiple disparate systems or operating an out-of-date ERP system, that runs on obsolete technology or that cannot support a company's business processes, creates an IT maintenance nightmare. These systems may be complicated to customize, and installing fixes and upgrades can take up valuable time and resources. Additionally, because the vendor could no longer be in operation, it might not be viable to upgrade these systems.

2.2 Disadvantages of ERP Systems

An ERP system implementation is significantly more involved than merely installing commercially available software; it is a labor-intensive process that requires a variety of different tasks and, if managed incorrectly, might lead to the project's failure. Companies shouldn't take the choice to deploy an ERP system lightly due of its importance. All workers, from functional users to IT professionals to top management, must be aware of the goals of the ERP project and collaborate to make the deployment successful. Companies that are thinking about implementing an ERP system should perform due diligence in selecting the solution that best fits their needs and collaborating with experts who can help with different implementation-related tasks.

People issues

Top management can be a major problem if they do not establish a convincing "tone at the top" that the ERP system is a priority or if they don't allocate adequate resources to its deployment. Lack of support from the employees may also be a concern. The legacy systems that employees have used for years may make them feel quite at ease. They could oppose to the additional training, organizational adjustments, and modifications to business processes that are unavoidable, or they can claim that the system is too challenging, constrictive, or inflexible. Employees who are resistant to the ERP system may create unproductive workarounds or create their own "shadow IT," such as spreadsheets or old systems, as a result of which they fail to use the system as intended.

Software issues

Because ERP systems are sophisticated and intricate, installing them sometimes necessitates paying high-priced system integrators. Companies frequently struggle to take control of technology and to use it to transform business processes in a quantifiable and sustainable way. A level of complexity that has not before been encountered and is difficult to absorb may also be added by the various capabilities, options, and setup requirements for businesses with relatively straightforward business requirements.

Price tag

ERP system deployments can cost millions of dollars and take years to complete, especially for big, international businesses. Additionally, once established, the ERP system requires ongoing "care and feeding" to keep it current, stable, and compatible with a variety of constantly evolving software programs with which it may interact. Companies often update and do larger improvements to the ERP system. This component of ERP might wind up costing more overall than the initial software licensing and implementation fees combined since maintenance charges are required annually.

Standardization

The above noted benefit of business process standardization may also be a drawback if the rigidity is inconsistent with the firm's culture or expectations. Additionally, a problem that must be resolved for the ERP installation to be effective is that the current corporate culture may not promote information exchange among business units or divisions. Vendors and system integrators actively advise businesses to adopt the best practices for ERP systems rather than customizing the software to fit their unique workflows. An exception, though, would be if businesses customized the software in accordance with a special business strategy that set them apart from rivals. The general guideline is that customizing the ERP software to get this capability is necessary if a certain procedure makes a business competitive or is required for compliance with a legislation.

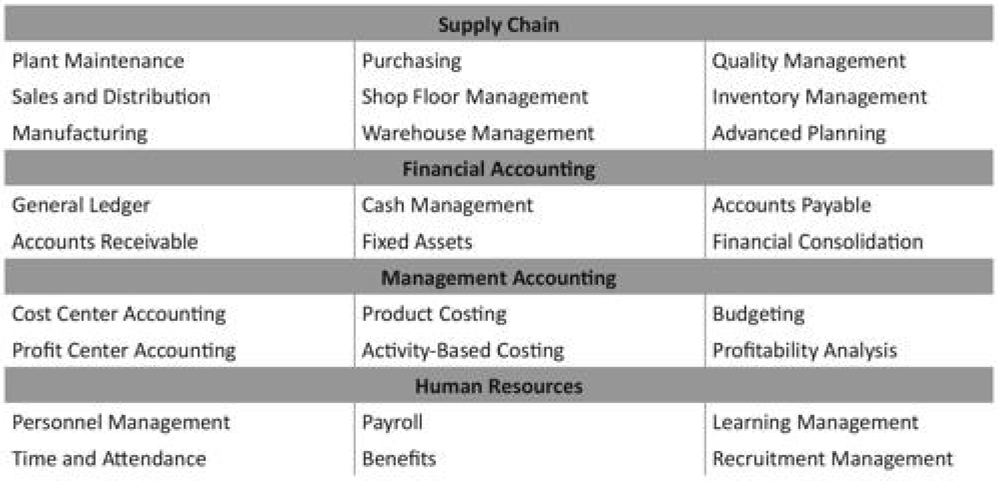
2.3 Modules

ERP systems are offered as modules, which are collections of connected software applications that handle key organizational tasks like accounting or production. Each module is designed to support a particular business process. The main modules, that provide basic functionalities for managing business processes, make up the **Core of the ERP**. This includes financial management, supply chain, human resources, customer relationship management, and other critical business processes. The "core" is the central and fundamental part of the ERP system and provides an integrated solution for managing company data. Here are some of the most common modules found in ERP systems:

- **Financial Management**: This module is responsible for managing financial transactions, such as accounts payable and receivable, general ledger, and financial reporting.
- **Accounting**: This module includes sub-modules for cost accounting, payroll, fixed asset management, and other accounting functions.

- **Human Resources**: This module manages employee information, benefits, payroll, and other HR functions.
- **Procurement**: This module covers all aspects of procurement, including vendor management, purchase order creation and management, and inventory management.
- **Supply Chain Management**: This module manages the flow of goods and services, including procurement, production, and logistics.
- **Production**: This module helps manage the production process, including planning, scheduling, and tracking.
- **Sales and Marketing**: This module supports sales and marketing activities, including lead management, opportunity tracking, and customer relationship management.
- **Customer Relationship Management (CRM)**: This module supports customer-facing activities, such as marketing, sales, and customer service.

The specific modules included in a particular ERP system can vary based on the size of the organization and its unique business requirements. Most ERP software is flexible enough to allow businesses to purchase only the components they require "a la carte", this allows companies to have a solution "tailor-made" to its needs. Modular architecture has the advantage of enabling ERP suppliers to create product solutions for specific industries. Sometimes modules that support a major business area are called a **suite** which comprises multiple sub-modules, or components. An example is shown in Figure 2.3, where ERP modules for a manufacturing company are depicted.



	Supply Chain	
Plant Maintenance	Purchasing	Quality Management
Sales and Distribution	Shop Floor Management	Inventory Management
Manufacturing	Warehouse Management	Advanced Planning
	Financial Accounting	
General Ledger	Cash Management	Accounts Payable
Accounts Receivable	Fixed Assets	Financial Consolidation
	Management Accounting	
Cost Center Accounting	Product Costing	Budgeting
Profit Center Accounting	Activity-Based Costing	Profitability Analysis
	Human Resources	
Personnel Management	Payroll	Learning Management
Time and Attendance	Benefits	Recruitment Management

Figure 2.3. Examples of ERP Modules for a manufacturing company

In summary, the various modules in an ERP system work together to provide a comprehensive solution for managing business processes and data. By integrating all business functions into a single system, organizations can streamline processes, reduce data duplication, improve business analytics, and make more informed decisions.

2.4 Technology

An ERP system has a far-reaching impact that affects users across an entire organization, as well as its customers, suppliers, and other business partners. With the need to support a large number of users who have different processing and reporting requirements, it is important to have advanced and adaptable software that utilizes cutting-edge technology. Given that the ERP system plays a crucial role in fulfilling an organization's operational and information needs, it is essential to have a thorough understanding of the technology that supports the integrated system, and provide a robust, scalable, and user-friendly solution for managing a wide range of business processes and data.

2.4.1 Three-Tier Client-Server Architecture

The client-server architecture[19] is widely used in modern computing, and is a fundamental aspect of many systems, including ERP systems. This architecture is a computing model in which a server (**Back-end application**) provides services to clients (**Front-end application**) over a network. The client requests a service or resource from the server, and the server responds by providing the requested information or performing the requested task. This architecture allows for efficient and scalable distribution of resources and tasks, as the server can handle requests from multiple clients simultaneously.

In this context we can separate an application into three logical components **(3-Tier architecture)**, which are the client tier, the application tier, and the database tier. This architecture provides a scalable, flexible, and secure solution for software applications. In Figure 2.4 is showed a detailed explanation of each tier.

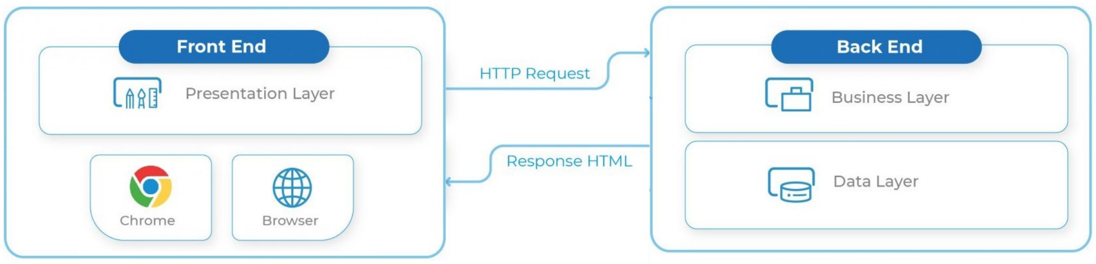


Figure 2.4. Web Application Architecture

- **Client Tier - Presentation layer**: This tier is responsible for presenting the user interface to the end-user. It provides the interface through which users interact with the application. The client tier can be implemented as a standalone application or as a web application accessed through a web browser.
- **Application Tier - Business layer**: This tier is responsible for processing the user requests and returning the results to the client tier. It is responsible for handling the business logic and data processing. The application server tier is typically

implemented as a web server.

- **Database Tier - Data access layer**: This tier is responsible for storing and managing the vast amount of data generated by the application. It is a key component of a web application that stores and manages information for a web app. You can search, filter and sort information based on user request. It is typically implemented using a robust database management system.

The three-tier architecture allows for a separation of concerns, with each tier having a specific role and responsibility. This separation makes it easier to develop, maintain, and upgrade the application, as changes can be made to one tier without affecting the others. Additionally, by dividing the system into separate tiers, the performance and security are improved. The client does not have direct access to the data, instead, all data passes through the application server which controls and regulates access to the information. This allows for more efficient and secure management of data. The ability to deploy application servers on multiple machines provides higher scalability, better performance and better re-use.
The Three-Tier Client-Server Architecture is a widely used architecture for ERP systems, as it provides a scalable, flexible, and secure platform for managing complex business processes and data.

2.4.2 Deployment

An ERP system can be deployed in two ways: On-premise or on Cloud, each with its own advantages and disadvantages. The best deployment method depends on the specific needs and requirements of the organization.

On-Premise

The conventional approach to ERP deployment is **on-premise ERP**. In this deployment method, the ERP software is installed and run on computers within an organization's own physical facilities. This allows for complete control over the software and data, but also requires the organization to provide the necessary hardware, storage, and technical support. Companies choosing the "on-prem" option are usually larger companies with bigger budgets, an existing IT infrastructure in place, and knowledgeable IT personnel to support the software and infrastructure. Due to the large upfront cost required, which often includes the cost of both hardware and software, on-premise ERP is typically seen as a capital investment.

Cloud

Cloud ERP deployment is becoming increasingly popular, where the ERP system is hosted by a vendor or third party on shared computing resources that can be accessed through the internet. These resources are maintained in data centers dedicated to hosting various applications on multiple platforms. This deployment method offers more scalability and flexibility, as well as reduced hardware and technical support expenses, but it requires an organization to have trust in a third party with access to its data. Customers have access to the ERP system as needed and pay for the software on a monthly or yearly basis. This method of paying for ERP software on a subscription basis is called **software as a service (SaaS)**, an attractive option for businesses looking to reduce upfront expenses and to budget for ERP long-term. Nowadays, nearly all ERP vendors offer some form of cloud deployment because it has many advantages compared to on-premise deployment.

Advantage

- One key advantage is that ERP cloud providers maintain, upgrades and handle maintenance for the infrastructure of the ERP system.
- Cloud ERP is more scalable than on-premise, which is ideal for startups and fast-growing businesses.
- Companies that choose cloud ERP over on-premise can now enjoy more peace of mind that the cloud provider has up-to-date controls in place such as data backup, dual factor authentication, encryption for confidential data, and a disaster recovery plan.

Disadvantages

- Many vendors offering cloud solutions are primarily focused on just one particular area. Very few cloud providers are offering a suite of products to meet the needs of medium-to-large organizations.
- Many cloud ERP solutions are limited in what the customer can do in terms of customization.
- Although cloud ERP is generally thought to be less expensive than on-premise, research has shown that over a 10-year window, the total costs for each converge. While expenses for cloud ERP are less upfront than on-premise, the costs catch up over time. Thus, the costeffectiveness of cloud ERP is not as great as initially thought.

2.4.3 Customization

It's uncommon for an ERP system to fully meet a company's needs, especially if the company is a large, global organization. There are often problems that go beyond what the ERP software can accommodate through configuration. Examples of these issues include:

- Creating additional functionality not provided by the ERP system
- Establishing connections between the ERP system and third-party systems
- Adding extra fields to the ERP database

These types of problems usually require development and programming to enhance the ERP system. This is known as customization, which involves adding custom code to increase the capabilities and features of the ERP system. Customization is typically performed when all efforts to find a solution through configuration have failed. As customization requires time and money, companies should aim to minimize it.

2.5 Market

The ERP market is estimated at \$43.72 billion in 2020, and is projected to reach \$117.09 billion by 2030, at a compounded annual growth rate of 10.0%[52]. The increase in the ERP market can be attributed to the growing interest from small and medium-sized businesses and the development of new ERP applications for both cloud and mobile platforms. However, not all ERP vendors offer the same quality of software, which can be divided into three categories based on specific criteria, as presented in a table 2.1.

- **Tier I** (Enterprise Class) is software designed for large, worldwide corporations with significant market capitalization and annual revenues that exceed \$750 million. These solutions are very costly due to their extensive capabilities, including the ability to manage complex organizational structures and address international concerns, like multiple currencies and varying accounting regulations. Only a few ERP vendors have the necessary size, resources, and comprehensive functionality to support the high volume of daily transactions that Tier I companies typically encounter.

- **Tier II** (the Mid-Market Class) category of ERP systems is intended for medium-sized companies and can be further divided into upper and lower sub-categories. Upper Tier II systems are for companies with annual revenues ranging from \$250 million to \$750 million. Lower Tier II systems typically are for companies with annual revenues between \$10 million and \$250 million. These vendors offer software that is designed for either single or multiple legal entities and locations, but with limited functionalities compared to Tier I vendor solutions. As a result, these ERP systems are less expensive than Tier I systems. Typically, they are easier to implement and support and are designed specifically for only a few industries.

- **Tier III** (Small Business Class) ERP systems are made for smaller businesses with annual revenue below \$10 million, operating within a single country. They are the most affordable among the different tiers of ERP systems. The market is saturated with many software providers in this category, some of which offer robust point solutions that can be utilized to enhance a Tier I or Tier II ERP system.

The ERP market is experiencing a trend where Tier II and Tier III vendors are aiming to serve larger companies by improving their software's capabilities and scalability, while Tier I vendors are reaching smaller companies by offering simplified versions of their software and acquiring cloud vendors. This is causing the boundaries between ERP tiers to become less distinct as vendors strive to increase their market share. Table 2.2 presents some example of ERP systems in tiers.

Tier I	Tier II	Tier III
High complexity	Medium complexity	Low complexity
Highest cost	Medium cost	Lowest cost
Many industry solutions	Fewer industry solutions	Fewest industry solutions
Large companies	Mid-market companies	Small companies
Support global functionality	Operate in more than one country	Does not support global functionality

Table 2.1. Characteristics of ERP Vendor Tiers

Tier I	Tier II	Tier III
SAP S4/HANA	Microsoft Dynamics 365	Sage
Oracle EBS	NetSuite	Aptean
Infor LN	SAP Business All-in-One	ASC
Infor M3	ODOO	ECI

Table 2.2. Example ERP Vendors in Tiers

2.5.1 Cost

Numerous ERP projects run over budget, often as a result of unanticipated organizational or technological problems, scope expansion, or an unrealistic project budget. Budgeting for ERP can be difficult since some costs are difficult to predict at first. This section will go into depth about every expense that goes into calculating the system's total cost of ownership **total cost of ownership (TCO)**. Some of these costs will be one-time costs, while others will be recurring[16].

Software License Costs

Typically, an ERP system's price tag depends on the:

- Number of employees that will be using the system
- Vendor tier being deployed, Tier 1 software is more expensive than Tier 2, while Tier 3 would be the least expensive
- Number of modules purchased

The vast majority of ERP software licenses are supplied using a **perpetual licensing model**, which requires paying an upfront licensing price before the vendor grants access to the program for an endless amount of time. Additionally, customers must pay annual maintenance costs in order to get support, updates, and future software upgrades. For on-premise implementations, perpetual licensing is standard. With perpetual licensing, a couple of license methods are used:

- **Named user licensing**. A company determines how many unique users will use the ERP system and pays a licensing charge for each of them. Numerous ERP vendors provide different named user categories, such as heavy user licensing, for users who utilize more system capability and are thus paid a larger license fee, or casual user licensing, for users who just read reports or lists.

- **Concurrent user licensing**. A perpetual license type enables an unlimited number of designated users and accounts, but restricts the number of individuals who can actively use the software at one time. Concurrent user licensing is often cheaper than named user licensing as it only requires payment for the estimated number of simultaneous users. However, it's essential to accurately predict the number of concurrent users, otherwise, employees may experience difficulties logging on and using the software.

Third-Party Software License Costs

In some cases, a company has specific requirements that cannot be fulfilled by the ERP system they have purchased, and modifying the ERP to meet those requirements is too costly. To address this, third-party software known as "bolt-ons" can be used. These provide additional functionality or logic to help solve specific business needs. To ensure seamless integration with the ERP system, it is advisable to get recommendations from the ERP vendor or system integrator on which bolt-on solution will work best.

Hardware and IT Infrastructure Costs

The implementation of an ERP system requires a strong and up-to-date IT setup. If the ERP is run on-site, a company will need to invest in IT hardware such as servers, routers, backup, storage devices, desktops, laptops, tablets, and printers. They will also need to consider measures for failover, network access, power supply, and security. This may involve hiring additional staff and increasing data center space, with costs ranging from one-time expenses like purchasing servers to ongoing expenses like utility bills and salaries.

Database License Costs

The cost of the database in an ERP project can range from a few thousand dollars to hundreds of thousands of dollars, depending on several factors such as the type of database, the size of the data being stored, and the number of users accessing the database. ERP vendors will provide the specifications for the type of database needed.

Implementation Costs

The expenses related to the implementation of ERP software are among the most pricey parts of the total cost of ownership (TCO). The cost of functional and technical consultants, who play a key role in the implementation, can be a major part of these expenses, depending on how much a company is going to rely on the system integrator. The complexity of the project and consultants from different geographical locations may also affect the hourly rate charged.

Maintenance and Support Costs

The cost of an ERP system doesn't end once the software is up and running. To ensure the system continues to run smoothly, companies need to have a plan for ongoing maintenance and support, also known as **application management services (AMS)**. This includes functional and technical support, updating and patching, monitoring the software, and backup and recovery. Some companies can handle these tasks in-house, while others may hire a third-party company to provide these services. The cost of maintenance typically ranges from 18% to 25% of the original software license cost and can be paid directly to the ERP vendor or to the third-party company managing the system.
For software support, various levels are available, with the higher levels offering more services at a higher cost. Premium support could include having a representative from the ERP vendor on-site during the project implementation and for a period of time post-implementation, as well as prioritized access to help tickets. Basic support might only provide access to a help portal where customers can log tickets and get answers to questions.

Cloud

The Total Cost of Ownership (TCO) in a cloud context is the same as what we previously discussed, but often, all of the costs are included in a single **subscription-based licensing**. Customers subscribing to this model are granted program access for a set duration, such as monthly or annually, which covers not only the use of the software but also maintenance, support, scheduled updates, and upgrades provided by the cloud vendor. This comprehensive subscription often includes a basic database offering, with the flexibility to expand storage for an additional fee.

Predominantly, ERP applications are marketed as **Software as a Service (SaaS)**, a cloud-based delivery model wherein the vendor shoulders the responsibility for the underlying infrastructure, including hosting, maintenance, and support. This implies that the costs for IT infrastructure, when the ERP system is hosted by the vendor or a third-party provider, are bundled into the monthly subscription fee. Companies utilizing public cloud platforms like Microsoft Azure would typically pay a recurring fee for the infrastructure lease in addition to the software licensing fees due to the ERP vendor.

2.5.2 Competitors

The ERP (Enterprise Resource Planning) market is experiencing a significant shift with the emergence of smaller, more agile players. These smaller vendors are challenging the traditional dominance of larger, established companies like SAP, Oracle, and Microsoft. Characterized by their innovative, cloud-native solutions, these emerging players focus on delivering specialized and industry-specific ERP systems. This approach contrasts with the one-size-fits-all, monolithic systems traditionally offered by the larger vendors[45].

This trend towards smaller, more nimble ERP vendors is driven by the increasing popularity of cloud-based solutions. These solutions offer benefits such as lower costs, greater scalability, and flexibility, making them particularly appealing to small and medium-sized enterprises. The pandemic has further accelerated this shift, as businesses seek solutions that support remote work and offer greater operational agility. As a result, the ERP market is becoming more fragmented and competitive, providing businesses with a wider array of choices to suit their specific needs[38].

This section provides an overview of the key features of some cloud ERP products based on research from the ERP Research comparison platform[53]. These features are critical for businesses to consider when choosing an ERP system, as they directly impact usability, scalability, and overall business efficiency.

SAP Business One

PROS	**CONS**
• A complete business management solution for SMEs. • Exceptional performance in handling business functions. • Simple user interface and internal controls. • Can be highly customized to adapt to business needs. • Mature product with major functionalities supported. • Deployment flexibility for On-Prem, SaaS or Private Cloud.	• Limited Human Capital and Manufacturing functionalities supported. • The limitation to customize the dashboards and cockpit feature. • The Firefox web browser is currently the only web browser supported. • Heavy reliance on partner addons for deeper and wider functionality. • Requires heavy customization which can lead to IT debt.

Table 2.3. Pros and Cons of SAP Business One.

Microsoft Dynamics 365

PROS	CONS
• Good integration with other software and technologies. • User friendly, easy to train users. • Secure and permission-based account setup. • Flexible and customizable for all company needs. • Extensive filtering capabilities.	• Difficult migration from old ERP. • Some functions could be more user friendly and intuitive. • User documentation needs improvement. • Can be expensive due to high level of customization.

Table 2.4. Pros and Cons of Microsoft Dynamics 365.

ODOO

PROS	CONS
• Low-cost when investing in a small number of modules. • Free "Community" version available. • Offers a comprehensive selection of Odoo apps and integrates with many thirdparty add-on software apps. • Uses Open Source software which can be easily customized.	• Requires IT knowledge to install and maintain, this is not a "plug and play" solution. • Steep learning curve on initial implementation. • Costs may rise with the use of numerous Odoo Modules and third-party apps. • Has a shorter history compared to other established ERP players.

Table 2.5. Pros and Cons of ODOO platform.

Oracle NetSuite

PROS	CONS
• Wide and deep functionality across several key business areas. • True SaaS Cloud ERP offering. • Largest Cloud ERP customer base and ecosystem. • Strong localization capabilities for international businesses. • Strong consultant market and availability.	• License pricing is complex and can produce hidden costs. • Some localisations and functionality is not provided out of the box put as part of partner extensions. • Many acquisitions have led to the solution being more loosely connected instead of a cohesive, integrated suite. • Strong consultant market and availability.

Table 2.6. Pros and Cons of Oracle NetSuite.

Chapter 3

Fundamentals

In this chapter, we embark on a journey to construct the narrative foundation that underpins our thesis. Here, we will introduce the essential design concepts and the architectural principles that have been employed in the creation of the platform known as Kube. Central to this discussion is cloud computing, which serves as the bedrock for the platform's operational environment. We then navigate through the intricacies of microservices, an architectural style that breaks down applications into smaller, independently deployable services. A pivotal aspect of Kube's architecture is its serverless framework, which facilitates greater scalability and efficiency. Complementing this is the implementation of sagas, a sequence of transactions that manage failures and ensure data consistency across services. Lastly, the event-driven architecture of Kube plays a crucial role, enabling reactive programming and responsive interactions within the platform. Together, these concepts embody the architectural design of Kube, prioritizing flexibility, scalability, and robust service integration.

3.1 Cloud Computing

The advent of cloud computing has revolutionized the way we approach computing and data management. This section of the thesis will explore the paradigm of cloud computing, which has emerged as a transformative force in the technological landscape. We will examine the fundamental aspects of cloud computing, including its service models, deployment strategies, and the pivotal role it plays in modern IT infrastructure.

3.1.1 Definition

The United States National Institute of Standards and Technology's definition of cloud computing is[49]:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand

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

Cloud computing offers a highly flexible service delivery model, enabling on-demand access to various resources, such as storage, processing power, and applications, via the internet. This eliminates the need for local servers, shifting data handling to online remote servers and offering a cost-effective, pay-for-what-you-use pricing model. Services like Amazon Web Services (AWS) provide the technology infrastructure, allowing users to scale operations with ease. Additionally, this model promotes economic efficiency, as organizations pay only for resources they consume, supporting a scalable and agile approach to resource management. The network, often the internet, serves as a conduit between users and cloud services, ensuring data is managed with strong security measures.

Essential Characteristics

Five essential characteristics of cloud computing are[49]:

- **On-Demand Self-Service**: Users can independently set up and manage their computing needs, such as server time and network storage, automatically, eliminating the need for direct interaction with service providers.

- **Broad Network Access**: Services are accessible over the internet using standard methods that support a diverse range of devices, including smartphones, tablets, laptops, and desktops.

- **Resource Pooling**: The provider's computing resources are aggregated to serve various customers under a multi-tenant model. Resources are dynamically allocated and reallocated based on user demand, with the user typically not knowing or controlling the exact physical location of the resources but may be able to designate a location at a broader level (such as country, state, or data center).

- **Rapid Elasticity**: Services can be quickly scaled up or down to match demand, with the scaling process sometimes occurring automatically. From the user's perspective, the supply of available resources often seems boundless and can be acquired in any volume at any time.

- **Measured Service**: Cloud platforms automatically monitor, control, and report resource usage with a metering function that operates at an appropriate level of detail, depending on the type of service, such as storage space, processing power, bandwidth, or active user numbers. This metering provides clear visibility into the usage of services for both the provider and the consumer.

3.1.2 Service Models

Cloud computing has revolutionized the way businesses approach technology, offering a spectrum of services that cater to varying requirements for control, flexibility, and management. As organizations transition to cloud-based solutions, understanding the different service models becomes crucial for leveraging the full potential of the cloud.

The figure 3.1 shows the three primary cloud service models, forming what is often referred to as the cloud computing "stack," include Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). These service models are designed to build upon one another, offering layers of abstraction and increasing levels of managed services[3].

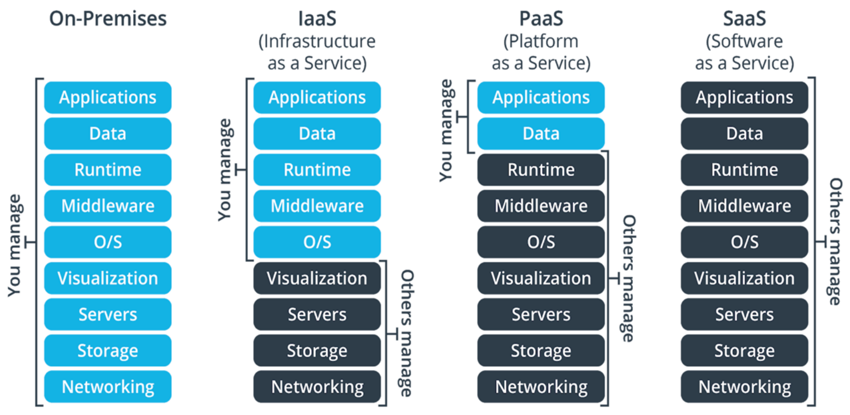


Figure 3.1. Cloud service models [35].

Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) is a transformative approach to managing IT resources, offering flexible and on-demand access to essential infrastructure services through the internet. This includes virtual machines, storage, and networking that can be customized and billed based on actual usage. IaaS grants organizations unprecedented control over their IT resources, closely resembling traditional on-premises infrastructure. It allows for easy scalability without the need for costly upfront investments in hardware. IaaS empowers consumers with the ability to provision processing, storage, and networking resources, deploying and running various software, including operating systems and applications. This level of customization enables organizations to tailor their IT environment to their specific needs, ensuring a seamless and efficient operation in the cloud.

Platform as a Service (PaaS)

Platform as a Service (PaaS) is a significant innovation in cloud computing, offering comprehensive hardware and software resources for cloud-based application development.

Leading PaaS providers simplify development by effectively managing the underlying infrastructure, allowing a laser focus on application creation. With PaaS, you're free from infrastructure oversight, enabling dedicated attention to application deployment and management, improving operational efficiency by eliminating resource provisioning, capacity planning, and maintenance. PaaS provides an environment for building, testing, and managing software applications without the need to manage the underlying cloud infrastructure. As a user, you control your applications and their hosting settings, simplifying the development process by allowing you to focus solely on creating and deploying your applications.

Software as a Service (SaaS)

Software as a Service (SaaS) is a cloud computing model that delivers software applications over the internet on a subscription basis. In this approach, cloud providers manage and host the applications, ensuring their availability, performance, and security. Well-known examples of SaaS offerings include Google Workspace, Microsoft Office 365, and Salesforce. SaaS provides a complete software solution over the internet, including the application and its underlying infrastructure, which is fully maintained by the cloud service provider. This approach spares users from managing the infrastructure, as the provider handles software updates and security measures. Users can access these applications through different devices and web browsers, enjoying an accessible and simplified experience. SaaS enables users to efficiently utilize applications hosted on the cloud, focusing solely on using the software rather than its maintenance.

3.1.3 Deployment Models

Cloud computing, a key driver in modern IT resource management, offers various deployment models tailored to different business needs, security requirements, and scalability demands. This exploration focuses on Public, Private, Hybrid, and Community Cloud models, discussing their unique features, benefits, and considerations.

Community Cloud

Community-dedicated cloud infrastructure is exclusively used by a specific community of organizations with shared interests, including mission objectives, security requirements, policies, and compliance regulations. This infrastructure can be managed by one or more organizations within the community, external providers, or a combination of both, and it can be located on-premises or off-premises to accommodate community preferences[49].

Private Cloud

A private cloud is a form of cloud computing providing exclusive resources and services via a private network, dedicated solely to one organization. It offers enhanced security and data isolation, with the ability to tailor infrastructure and software to specific needs and workflows. While offering robust security and control, private clouds can be costlier due to the organization's responsibility for infrastructure management and scaling. These can be deployed on-site or hosted by third-party providers, catering specifically to businesses requiring high security and compliance standards. The combination of cloud computing benefits with heightened data security and customization makes private clouds an attractive option for businesses handling sensitive data.

Public Cloud

In public cloud computing, a third-party service provider manages all the level of infrastructure, including servers, storage, and computing resources. Clients access these resources via the internet and are billed based on their actual usage, creating a cost-effective and flexible pay-as-you-go model. These cloud environments eliminate the need for substantial investments in expensive infrastructure, democratizing access to cloud computing. It's important to note that public clouds operate on shared infrastructure, which offers cost efficiencies but raises concerns about data isolation and privacy, as multiple customers' data and applications coexist on the same infrastructure.

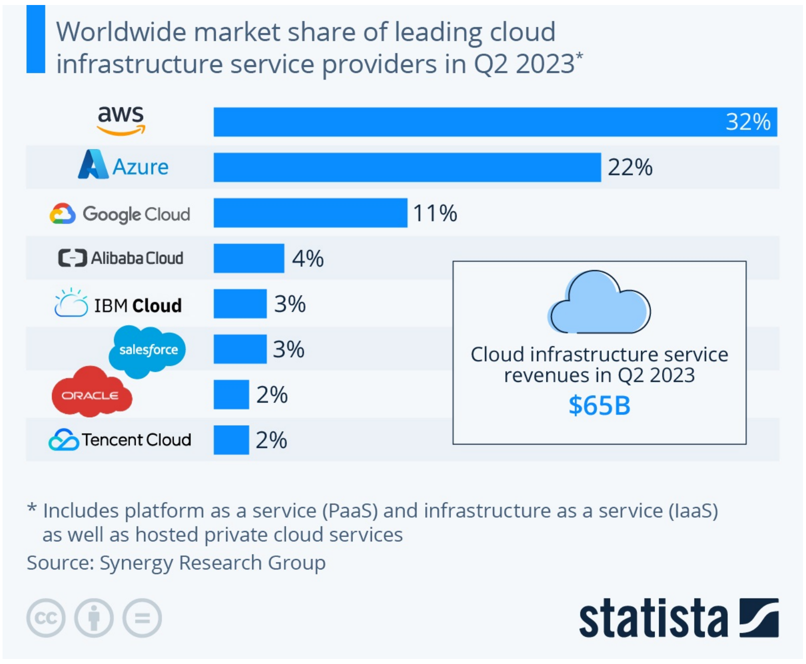


Figure 3.2. Worldwide market share of leading cloud infrastructure service providers in Q2 2023[57].

The figure 3.2 show how the public cloud marketplace consists of numerous cloud providers. Amazon, Microsoft and Google account for 65% of the total 2023 cloud market. The remaining public cloud market is divided among IBM, Alibaba, Oracle and several smaller players. The table 3.1 make a comparison between the major cloud providers.

Name	Cost/hour	Pros	Cons
Amazon AWS	$0.0255	Reliability, Quality, Professional Support	Expensive despite regular lowering of price
Google GCP	$0.0475	Reliability, Affordable	Limited features and services
Microsoft Azure	$0.043	Best infrastructure configuration	Unsatisfactory customer experience and technical support
IBM Cloud	$0.04	Flexibility, Speed, Interoperability	Complicated pricing model and platform can be slow

Table 3.1. Comparative overview of major cloud service providers[51].

Hybrid Cloud

The hybrid cloud is an advanced cloud computing model that blends public and private clouds, giving organizations the flexibility to distribute their applications and workloads as needed. This setup allows for greater control and scalability than using only public clouds, letting businesses keep sensitive data secure while still enjoying public cloud efficiency. It's ideal for companies that need both strong security and the ability to quickly adapt and scale. The hybrid cloud offers a versatile IT infrastructure that adjusts to the complex needs of modern businesses, improving security, compliance, and overall efficiency. This makes it a valuable asset for companies navigating the rapidly changing digital world.

3.1.4 Benefits

Cloud computing is a big shift from the traditional way businesses think about IT resources. Here are seven common reasons organizations are turning to cloud computing services[40]:

- **Cost**: Cloud computing reduces the capital expense of buying hardware and software, setting up and running on-site datacenters, which can quickly add up.

- **Speed**: Cloud services are typically on-demand, allowing vast amounts of computing resources to be provisioned in minutes, offering businesses flexibility and easing capacity planning.

- **Global Scale**: It includes the ability to elastically scale IT resources, providing the right amount of computing power, storage, and bandwidth when and where needed.

- **Productivity**: Cloud computing eliminates many time-consuming tasks associated with managing on-site datacenters, allowing IT teams to focus on more important business goals.

- **Performance**: Cloud services run on a worldwide network of secure datacenters, regularly upgraded to the latest generation of fast and efficient computing hardware, offering benefits like reduced network latency and greater economies of scale.
- **Reliability**: Data backup, disaster recovery, and business continuity are easier and less costly, as data can be mirrored at multiple redundant sites on the cloud provider's network.
- **Security**: Cloud providers typically offer a broad set of policies, technologies, and controls to strengthen security, protecting data, apps, and infrastructure from threats.

3.1.5 Market Overview

The global cloud computing market, valued at USD 483.98 billion in 2022, is expected to exhibit a robust compound annual growth rate (CAGR) of 14.1% from 2023 to 2030[55]. This remarkable growth is attributed to the cloud's capacity to significantly enhance business performance in large enterprises, the increasing demand for hybrid and Omni-cloud systems, and the adoption of pay-as-you-go models. Cloud services have gained popularity in developing countries, thanks in part to government initiatives aimed at safeguarding data integrity and security. The COVID-19 pandemic has expedited the adoption of cloud computing, driven by the shift towards hybrid work models. While data privacy and security concerns remain, large enterprises are increasingly turning to cloud-based technologies to optimize costs. Moreover, cloud adoption is on the rise among small and medium-sized organizations, and governments in developing nations are making substantial investments in cloud delivery models to enhance productivity. The Figure 3.3 illustrates the growth of the U.S. cloud computing market.

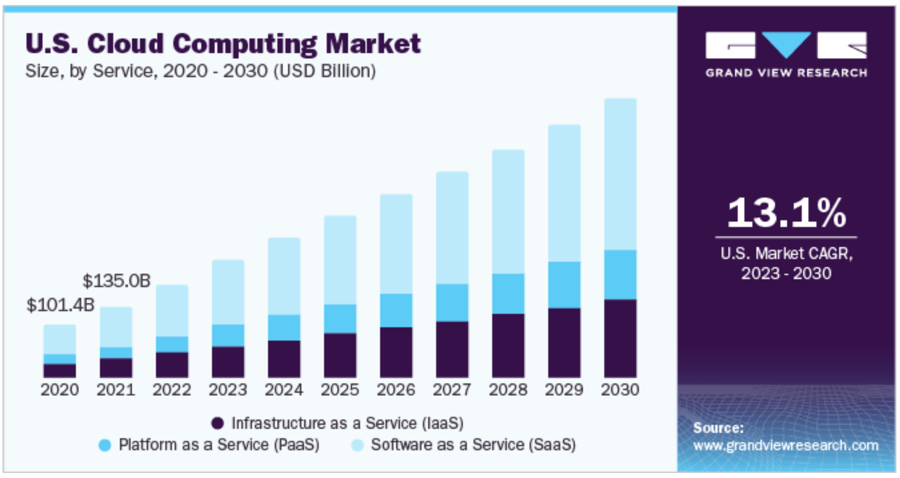


Figure 3.3. U.S. cloud computing market[55].

3.2 Microservices

This section of the thesis is dedicated to a comprehensive exploration of microservices as an architectural choice that has seen an increasingly popularity over the past half-decade. It aims to unpack the intricacies of microservices, given a broad overview of the core ideas behind this technology and some reasons why these architectures are used so widely.

3.2.1 Characteristics

Microservices architecture is a modular approach to software development, breaking complex applications into smaller, independent components—microservices—tailored to specific business domains like inventory management or order processing. These microservices, with well-defined interfaces, can be developed and deployed independently, ensuring flexibility and the evolution of each component without affecting others. This architecture supports a service-oriented approach, emphasizing independent deployability and technology neutrality, suitable for diverse technical challenges[44].

Internally, microservices encapsulate their functionality, operating via network endpoints and hiding implementation details like programming languages or data storage. This ensures effective complexity management, with each service maintaining its own data storage, avoiding shared database issues.

Externally, microservices act as 'black boxes,' offering functionality without exposing internal processes. This approach protects against impacts from internal changes, as long as interfaces remain compatible. It enables seamless updates and maintenance, supporting independent development and continuous integration.

Key characteristics of microservices include loose coupling for flexibility and high cohesion for maintainability. They allow targeted scalability, parallel team work, and robust security, with each service secured separately. This makes microservices ideal for creating adaptable, scalable, and sustainable software in rapidly evolving business and technology environments.

Microservices in the Context of Cloud Computing

The integration of microservices and cloud computing marks a significant progression in software architecture, fostering dynamic, scalable, and resilient systems. The decentralized nature of microservices aligns seamlessly with cloud environments, providing agility and scalable infrastructure to meet varying service demands. Cloud platforms enhance resource optimization, ensuring efficient and cost-effective operations. This synergy enables organizations to exploit cloud computing's robustness, supporting microservices' complex interactions for heightened scalability and resilience. It allows for continuous integration and deployment, promoting rapid innovation. Additionally, strategic distribution of microservices across various regions in the cloud enhances fault tolerance and ensures a consistent user experience globally.

Example of a Microservice Architecture

A prime example of microservices in action is the streaming giant Netflix, which has become synonymous with the successful implementation of this architectural style. The backend architecture of Netflix, a detailed account of which is provided in an article on DEV.to[46], is a testament to the company's innovative engineering approach.

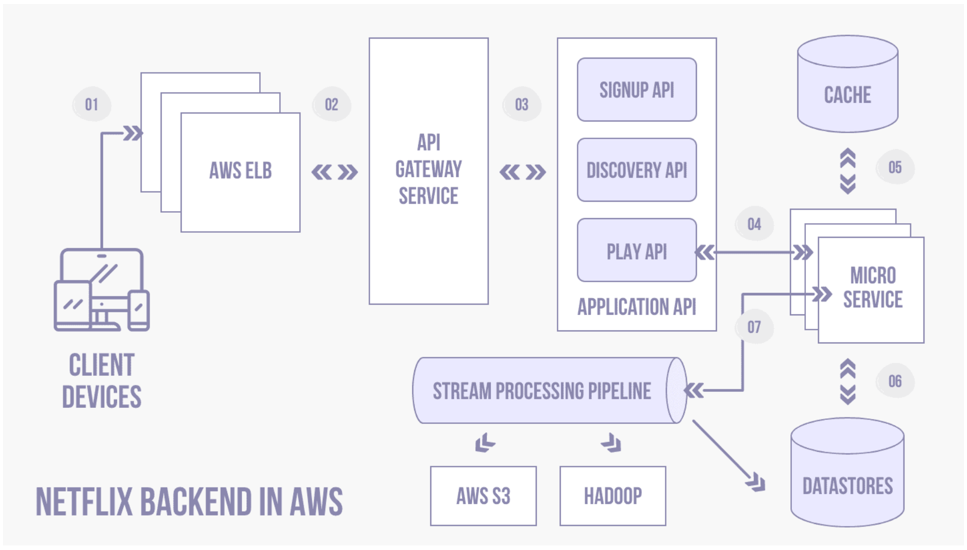


Figure 3.4. Netflix backend in AWS[61].

How we can see in Figure 3.4, Netflix's backend is a conglomeration of microservices that operate on Amazon Web Services (AWS), enabling them to serve a staggering amount of content globally with high availability and resilience. Each microservice is designed to perform a specific function, such as handling login requests, processing user recommendations, or managing customer support interactions. This division of responsibilities allows for independent scaling and development of services, which is crucial given the diversity of Netflix's content and the variability in demand. The microservices architecture is not only a core component of their backend system but also underpins their Open Connect content delivery network (CDN), ensuring optimal streaming performance by placing servers within Internet Service Provider (ISP) networks around the world. This architecture facilitates rapid and reliable delivery of complex applications at scale, illustrating the microservices model's capacity to support large-scale enterprise systems efficiently and effectively.

3.2.2 Benefits

This section delves into the multifaceted advantages of adopting microservices. From enhanced scalability to independent deployment cycles, microservices promise a range of benefits that cater to both technical and business needs. By dissecting these advantages, this section aims to elucidate why microservices are becoming the architectural choice for many modern enterprises, providing them with the flexibility and agility required to thrive in a competitive market[26].

Scalability

Microservices excel in scalability due to their ability to be scaled individually. This granular scalability allows for precise allocation of resources to different components based on fluctuating demands, leading to enhanced efficiency in resource utilization. Unlike monolithic architectures, where scaling often requires scaling the entire application, microservices operate independently. This independence facilitates the seamless addition, removal, updating, or scaling of each service without causing interruptions to the rest of the system. Organizations benefit from this by being able to dynamically allocate resources to microservices experiencing spikes in demand—such as during peak shopping seasons—and similarly, scale them down when demand wanes, thereby optimizing the use of resources and computing power across the service landscape.

Robustness

Microservices architecture enhances the robustness of software applications by leveraging its inherent decoupling. Individual services can fail without precipitating a system-wide shutdown, thereby preventing a single point of failure from causing cascading breakdowns. In comparison, monolithic architectures are susceptible to the domino effect, where a single component's failure can paralyze the whole application. Microservices inherently design for failure, allowing the system to degrade gracefully and maintain functionality even when certain services are down. However, network and machine failures are inevitable, and strategies must be in place to handle these incidents without significantly affecting the user experience.

Technology agnostic

Microservices architecture stands out for its technological flexibility, granting teams the liberty to select the most suitable technology stack for each distinct service. This technology-agnostic approach decouples services from any singular, early-stage technology decisions that often constrain entire projects. Within this paradigm, each microservice can be developed using different programming languages and data storage solutions, according to what best serves its purpose. This not only streamlines development by aligning with teams' existing proficiencies but also avoids the overhead of learning new languages unnecessarily. For instance, organizations like Netflix and Twitter predominantly utilize the Java Virtual Machine (JVM) as their operational platform [44]. Their choice is driven by a deep familiarity with this technology.

Distributed Development

Microservices architecture allows development teams to independently build, deploy, and manage their services, speeding up updates and feature additions with minimal disruption to the overall system. This facilitates swift adaptation to changing business needs. Unlike monolithic applications, which require large-scale deployments for minor updates, microservices support targeted, independent changes to specific services. This reduces deployment risks, enables quick error recovery, and hastens the delivery of new features to customers. Companies like Amazon and Netflix leverage microservices to bypass obstacles in software delivery, ensuring rapid and reliable service to their users[44].

Team optimization

Microservices architecture enhances team productivity by adhering to the "two-pizza rule," where smaller teams—ideally just large enough to be fed with two pizzas—tend to produce higher quality outcomes due to improved focus and manageability. This approach, pioneered by Amazon, ensures that each team works on a discrete codebase, fostering efficiency and faster achievement of goals. The flexibility inherent in microservices also allows for easy reassignment of service ownership, facilitating a seamless adaptation of the architecture to align with evolving organizational structures, thereby maintaining efficiency and effectiveness in the long term.

3.2.3 Challenges

While the microservices architecture offers numerous benefits such as enhanced scalability and improved team productivity, it also introduces a set of challenges that can complicate system design and maintenance: the complexity of orchestrating numerous services, maintaining data consistency across distributed systems and managing inter-service communication efficiently. Addressing these challenges is essential for a smooth microservices architecture.

Complexity

The decentralized approach of microservices inherently leads to systems with a high degree of complexity. As the number of services increases, the overall system can become more challenging to oversee and manage. Debugging exemplifies this complexity; with each microservice generating its logs, pinpointing the source of an issue can become a substantial challenge. This complexity requires robust logging and monitoring solutions that can aggregate and correlate logs from across services, providing a cohesive view of the system's health and facilitating faster problem resolution. Additionally, the complexity demands that developers and operators have a clear understanding of the system's architecture and communication patterns, ensuring they can effectively trace and troubleshoot issues as they arise.

Data Consistency

Ensuring data consistency across microservices poses significant challenges due to their distributed design. Unlike monolithic systems that rely on a single database, microservices often use separate databases, making traditional transaction-based consistency difficult to maintain. As a result, developers need to shift toward patterns like sagas and embrace eventual consistency, which can be a major paradigm shift, especially when adapting existing systems. It's crucial to decompose applications incrementally, allowing for careful evaluation of each change's impact on the system's data integrity.

Inter-service Communication

In microservices architecture, especially in cloud environments, inter-service communication adds complexity due to distributed network use. Each microservice's unique API requires careful management to ensure compatibility, a significant task when hundreds or thousands of APIs are involved. Disaggregating processes into multiple network-dependent services increases serialization, transmission, and deserialization, potentially adding to latency. This impact on performance, hard to predict in design or development, highlights the need for a gradual transition to microservices, allowing for an assessment of changes on system latency.

3.2.4 Comparison with Monolithic Architecture

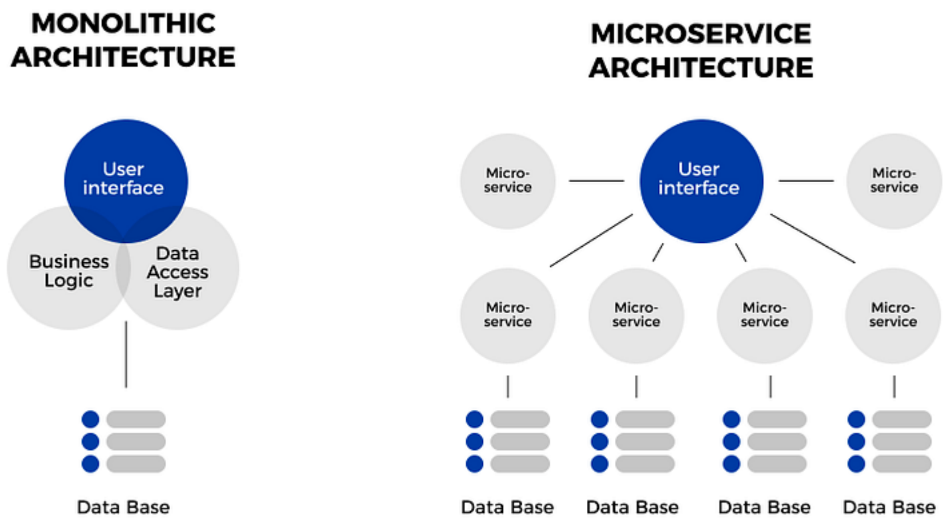


Figure 3.5. Monolithic and Microservices architectures[33].

In Figure 3.5, the distinction between a microservices approach and monolithic architecture is illustrated. The latter is characterized by tightly coupled and interdependent software components, any changes require building and deploying the entire stack, which can be slow and error-prone. Microservices are designed to overcome these limitations by decomposing functionality into separate services, each with a specific role, thus providing a more flexible and scalable architecture. The table 3.2 contains an overview of the differences between the two approaches.

	Monolithic	**Microservices**
Deployment	Simple and fast deployment of the entire system	Requires distinct resources, making orchestrating the deployment complicated
Scalability	It is hard to maintain and handle new changes; the whole system needs to be redeployed	Each element can be scaled independently without downtime
Agility	Not flexible and impossible to adopt new tech, languages, or frameworks	Integrate with new technologies to solve business purposes
Resiliency	One bug or issue can affect the whole system	A failure in one microservice does not affect other services
Testing	End-to-end testing	Independent components need to be tested individually
Security	Communication within a single unit makes data processing secure	Interprocess communication requires API gateways raising security issues
Development	Impossible to distribute the team's efforts due to the huge indivisible database	A team of developers can work independently on each component

Table 3.2. Comparison of Monolithic and Microservices architectures[13].

3.3 How to Model Microservices

This section is dedicated to exploring key principles such as information hiding, coupling, and cohesion, which are crucial in shaping our approach to defining the limits of our microservices. We will place a particular focus on domain-driven design, a highly effective strategy that plays a crucial role in establishing the boundaries of your microservices. This approach not only maximizes their advantages but also effectively reduces potential risks.

3.3.1 Boundaries

Our goal is to design microservices that can be independently modified, deployed, and have their features released to users without relying on others. The ability to update a single microservice independently from the others is crucial. At their core, microservices represent a type of modular decomposition, but with network interactions between modules. This means we can rely on a lot of prior art in the space of modular software to assist in defining our boundaries. Bearing this in mind, we will delve into three essential concepts crucial for identifying effective microservice boundaries: information hiding, cohesion, and coupling[44].

Information Hiding

Information hiding is a concept developed by David Parnas to look at the most effective way to define module boundaries[47]. Information hiding aims to conceal as much detail as possible within a microservice boundary. Parnas write[48]:

> The connections between modules are the assumptions which the modules make about each other.

Reducing assumptions between modules in microservices simplifies their connections, making it easier to modify one module without affecting others. This approach also allows developers to make safer changes, as they understand how their module is used by others, preventing the need for changes in upstream components. Additionally, in microservices, such modifications can be deployed independently, enhancing the benefits outlined by Parnas: faster development, better comprehensibility, and increased flexibility.

Coupling and Cohesion

The concepts of coupling and cohesion are integral to the structure and stability of microservice architectures. Understanding their interplay helps in designing systems that are both stable and efficient. Achieving the right balance between these two aspects is crucial for the effective functioning of microservices.

- **Cohesion**: Cohesion in microservices is about strategically grouping related business functionalities to reduce the need for changes across multiple areas. It emphasizes consolidating similar behaviors in a single location, which streamlines the process of modification and deployment. This approach leads to strong cohesion, where closely related functionalities are contained within a single microservice, thereby enabling faster and more secure updates and changes.

- **Coupling**: Coupling in the context of microservices involves designing services in a way that changes in one do not require modifications in others. This design principle promotes minimal inter-service knowledge, thereby reducing dependencies between different services. The ideal state of loose coupling is achieved when services have minimal interactions with each other, maintaining their independence. This approach significantly reduces the risks associated with tightly interconnected systems, ensuring more robust and flexible service architecture.

This balance is not only about the technical aspects but also about making pragmatic decisions that fit the specific context and challenges of the project.

Types of Coupling

The concept of coupling in system design is nuanced and not as straightforward as it might initially appear. While it's true that excessive coupling can lead to various challenges in system architecture, some level of coupling is inevitable and, in certain cases, necessary. The key objective in effective system design is not to eliminate coupling entirely but to manage and minimize its extent.

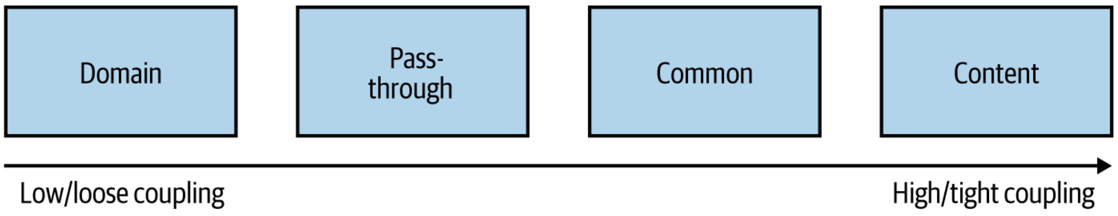


Figure 3.6. The different types of coupling, from loose (low) to tight (high)

.

The different types of coupling[44], as depicted in Figure 3.6, provide a comprehensive spectrum, ranging from low to high. Low coupling is generally desirable as it indicates a system where components operate independently, enhancing flexibility and ease of maintenance. High coupling, on the other hand, suggests a tightly interlinked system where changes in one component can have significant ripple effects, making it less desirable due to the increased complexity and risk involved. Understanding these variations and their implications is crucial for designing robust, scalable, and maintainable systems.

- **Domain Coupling**: Domain coupling refers to a scenario where one microservice depends on another for specific functionalities. While such interactions are largely inevitable in a microservice architecture, where collaboration among multiple services is essential for system operation, it's important to minimize these interactions. It is a form of loose coupling in microservices, but can lead to issues if a service relies too heavily on many downstream services, suggesting over-centralization of logic. Problems may also arise from exchanging complex data sets between services. It's advisable to share only essential information and minimize data exchange.
- **Pass-Through Coupling**: Pass-through coupling in microservices occurs when one microservice transmits data to another solely for the use of a subsequent downstream service. This form of coupling is particularly challenging within implementation strategies, as it suggests that the initiating service is aware not only of the direct recipient microservice but may also need to understand the functioning of the microservice further down the chain. This creates a complex interdependency where knowledge of multiple services and their interactions becomes necessary, complicating the architecture.
- **Common Coupling**: Common coupling in microservices refers to the scenario where multiple services utilize the same data set, such as a shared database, memory, or filesystem. This coupling becomes problematic when changes to the data's

structure affect several services simultaneously. For example, if the schema of a commonly used database changes incompatibly, all services relying on it need updates. Additionally, common coupling can lead to resource contention issues, as multiple services accessing the same database or filesystem may strain or even incapacitate that resource. While sometimes manageable, common coupling often indicates a lack of cohesion in the system and can pose operational challenges, making it one of the less desirable forms of coupling.

- **Content Coupling**: Content coupling occurs when an upstream service intrusively modifies the internal state of a downstream service, commonly by directly accessing and altering the latter's database. This is subtly different from common coupling, where multiple services interact with a shared dataset, but acknowledge it as an external, uncontrollable dependency. Content coupling blurs ownership lines, complicating system modifications for developers. A clear distinction in microservices between changeable and unchangeable elements is crucial. Developers must be aware of the service contract exposed to external parties to avoid disrupting upstream consumers. While common coupling shares some issues with content coupling, the latter introduces additional complexities, often termed pathological coupling. Direct external access to a database challenges the definition of what can be safely altered and what cannot, undermining the principle of information hiding. Therefore, content coupling is best avoided due to these inherent complications.

3.3.2 Domain-Driven Design

In defining microservice boundaries, we primarily focus on the domain itself, applying domain-driven design (DDD) to model our domain more effectively. DDD, as introduced by Eric Evans in "Domain-Driven Design"[22], provides key concepts that are crucial in this context. These include Ubiquitous Language, which ensures uniform language usage across the domain; Aggregates, which group related domain objects into a single unit; and Bounded Context, which sets the scope of applicability for a particular model. These principles play a vital role in guiding our microservice architecture strategy.

Ubiquitous Language

Ubiquitous language emphasizes the importance of aligning the terminology in our code with the terms used by the users. This commonality of language between the development team and the end users simplifies modeling the real-world domain and enhances communication. Integrating real-world language into the code streamlines the development process. It allows developers, when handling tasks or stories, to quickly grasp the requirements and objectives, as these are expressed in terms familiar to both the product owner and the development team.

Aggregates

Aggregates in microservice architecture are envisioned as self-contained units, each with its own state, identity, and life cycle that mirrors real-world entities. These aggregates are apt for implementation as state machines, given their inherent life cycles. The design focuses on consolidating the code that manages state transitions with the aggregate's state itself. Typically, a single microservice is responsible for one aggregate, but it may handle several. For instance, an Invoice aggregate would include various line payments, each significant only within the context of the overall Invoice aggregate.

A microservice's role extends to managing the life cycle and data storage of one or several types of aggregates. Should a different service need to modify an aggregate, it must either directly request this change or prompt the aggregate to initiate its own state transitions, possibly in response to events from other microservices. Aggregates are designed with the capability to reject inappropriate state transition requests, underscoring the importance of preventing illegal state changes in their implementation.

Bounded Context

A bounded context usually reflects a larger section of an organization, with clear responsibilities within its boundaries. This concept focuses on concealing the finer details of implementation, safeguarding internal aspects that aren't necessary for external understanding or involvement.

In terms of structure, a bounded context comprises one or more aggregates. While some of these aggregates might be visible externally, others remain internal to maintain the integrity of the context. Bounded contexts can also form relationships with other contexts, translating into dependencies between services in a microservice architecture.

For instance, a warehouse service can be seen as a bounded context, bustling with activities like processing outgoing orders, receiving new inventory, and other logistical tasks. In a different bounded context, such as the finance department, the focus shifts to less dynamic but equally vital functions like managing payroll and handling financial transactions. Each context operates within its own realm of responsibilities but may interact with or depend on other contexts, reflecting the interconnected nature of services in a microservices setup.

Domain-Driven Design in Microservices

Domain-Driven Design (DDD) is effective in microservices architecture due to its focus on bounded contexts which conceal internal complexities and present clear boundaries to the system. These contexts aid in maintaining stable microservice boundaries by ensuring that internal changes do not affect other system parts. When systems are segmented along bounded contexts, modifications for business needs are confined to specific microservices, streamlining deployment and reducing the complexity of changes.

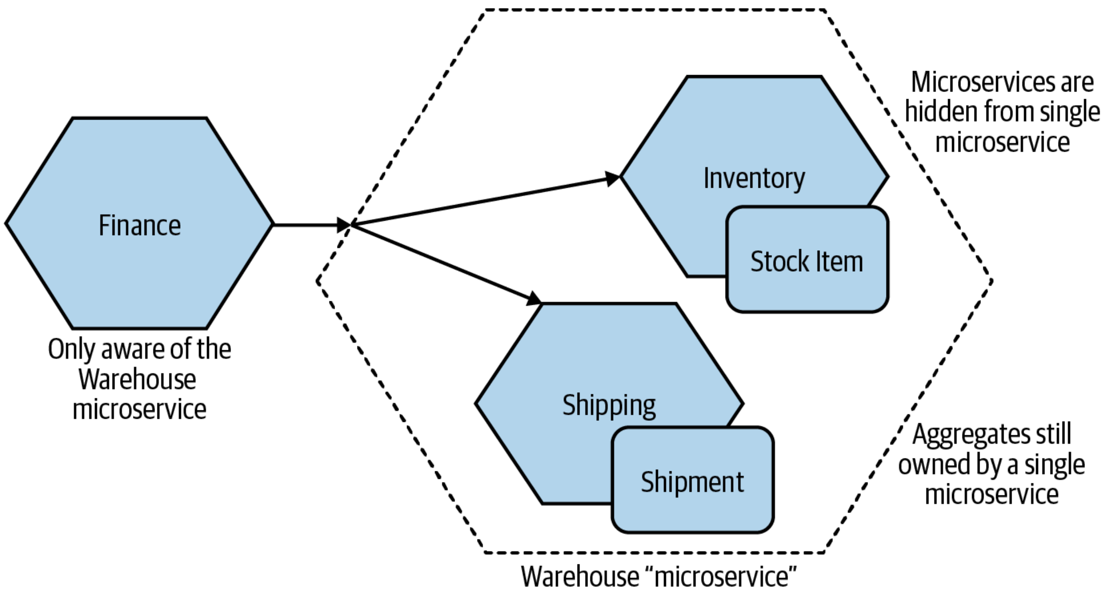

Finance
Only aware of the Warehouse microservice
Inventory
Stock Item
Shipping
Shipment
Microservices are hidden from single microservice
Aggregates still owned by a single microservice
Warehouse "microservice"

Figure 3.7. The Warehouse service internally has been split into Inventory and Shipping microservices

.

Aggregates and bounded contexts both provide cohesive units with clear interfaces to the larger system. Aggregates are focused state machines for single domain concepts, while bounded contexts group these aggregates and represent them to the outside world. These constructs are ideal for defining microservice boundaries. Initially, it's beneficial to work with services that cover complete bounded contexts. If needed, services can later be divided into smaller ones without splitting individual aggregates, keeping such internal decisions invisible to external stakeholders. For instance, a Warehouse service may internally be divided into Inventory and Shipping, but externally it remains a singular Warehouse microservice to users, as depicted in the figure 3.7.

3.3.3 Event-Driven Architecture

Event-Driven Architecture (EDA) is a software design paradigm that facilitates communication between services or components through the asynchronous exchange of events. It is essential in microservices for enabling decoupling, scalability, and reactive programming. This architecture allows real-time information flow between applications, microservices, and connected devices as events occur, promoting what is known as loose coupling. In EDA, applications and devices communicate without needing to know the specific sources or destinations of the information, maintaining isolated services with single responsibilities within a system[60].

Example Architecture

The image 3.8 depicts an event-driven architecture for an e-commerce site, showcasing how different components interact via events to enable a responsive and resilient system. Event Producers like a retail website, mobile app, and point-of-sale system generate events such as new orders or stock queries. These events are routed by the Event Router, which ingests, filters, and directs them to the appropriate Event Consumers. The consumers, such as the Warehouse Management Database, Finance System, and Customer Relations, act upon these events to update inventory, financial records, and customer service actions, respectively. This architecture ensures the site remains operational and efficient, even during high traffic, by reacting to real-time data without overloading the system.

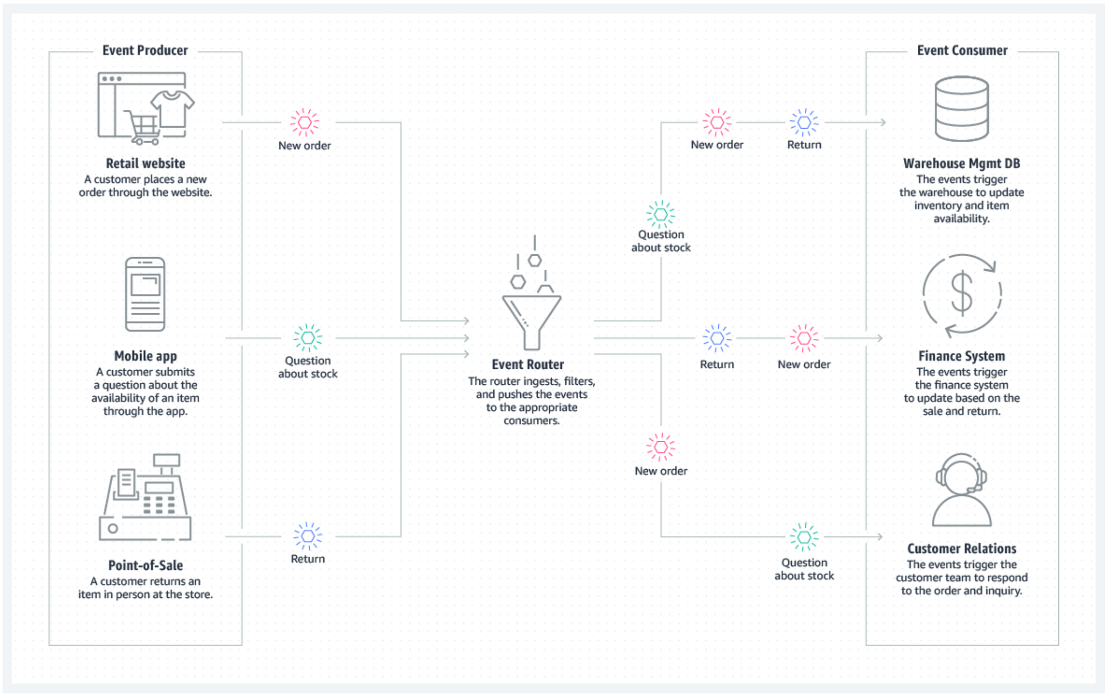

Event Producer
Retail website
A customer places a new order through the website.
Mobile app
A customer submits a question about the availability of an item through the app.
Point-of-Sale
A customer returns an item in person at the store.
New order
Question about stock
Return
Event Router
The router ingests, filters, and pushes the events to the appropriate consumers.
New order
Return
Question about stock
Return
New order
New order
Question about stock
Event Consumer
Warehouse Mgmt DB
The events trigger the warehouse to update inventory and item availability.
Finance System
The events trigger the finance system to update based on the sale and return.
Customer Relations
The events trigger the customer team to respond to the order and inquiry.

Figure 3.8. Example of an event-driven architecture[4].

How it works

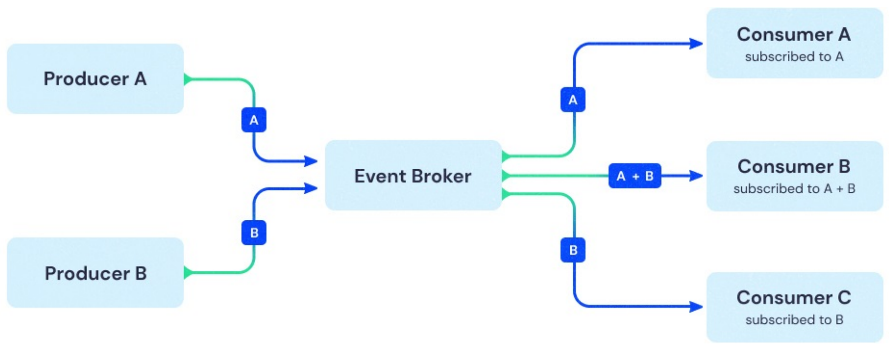


Figure 3.9. How event-driven architecture works[2].

In this architecture pattern, events are generated by producers and captured by an event bus, which routes them to the appropriate consumer services that process the event and may generate new events. This model enables decoupled and asynchronous interactions, allowing microservices to operate independently while responding to changes and inputs. EDA is scalable and allows for immediate response to events, with no need for point-to-point integrations, making it easy to add new consumers to the system[41].

Component	**Description**
Event	The core of EDA, an event is a significant change in state, which other parts of the application can listen to and react upon.
Event Producer	A component that generates events.
Event Consumer	A component that processes events.
Event Bus	A channel through which events are routed from producers to the appropriate consumers.

Table 3.3. Event-driven architecture components[41].

Models

The table 3.4 summarizing the different models of Event-Driven Architecture along with their descriptions and examples[15]:

Model	Description	Example
Process Manager	The orchestrator that manages a workflow or process, performing business logic and triggering events to other consumers.	AWS Step Functions
Event Sourcing	Stores events to calculate state, with downstream projections using this to calculate their view of the world. It's great for auditing.	Amazon DynamoDB
Event Streaming	Used for real-time information processing, such as user interactions. Messages are placed onto a stream for consumers to process.	Amazon Kinesis
Point-to-Point Messaging	Sends messages to a channel for downstream consumers to process, allowing for concurrent processing and scalability.	Amazon SQS
Change Data Capture	Reacts to changes made against data, with consumers attached to these changes for processing.	Amazon DynamoDB
Pub/Sub	Fires notifications out to downstream consumers, fanning out events and allowing consumers to get their own copy of the event.	Amazon EventBridge

Table 3.4. Models of Event-Driven Architecture.

Advantages

Event-Driven Architecture offers several compelling advantages when applied to microservices. One of its primary benefits is loose coupling, which allows services to operate independently, thereby reducing dependencies and simplifying maintenance. This loose coupling also contributes to the scalability of the system; services can be scaled up or down independently, and new consumers can be integrated without significant disruption to the existing ecosystem. Moreover, EDA provides flexibility and dynamism, enabling the system to quickly adapt to new business requirements by simply adding new event consumers. Finally, from a cost perspective, EDA systems are inherently efficient since they are push-based rather than pull-based, eliminating the need for continuous polling to check for events, which can lead to significant cost savings on compute and network resources[58][4].

Disadvantages

While Event-Driven Architecture presents numerous benefits for microservices, it also introduces a set of challenges. The inherent complexity of designing and managing an event-driven system is non-trivial, often requiring a sophisticated understanding of distributed systems. Testing, debugging and monitoring in a highly distributed systems adds another layer of complexity, necessitating robust and comprehensive strategies for tracing and logging. Performance can be impacted as well, given that the event broker or bus acting as a middle man between producers and consumers might introduce latency, potentially leading to longer execution times for event processing. Finally, the principle of eventual consistency in EDA means that services might process and react to the same event at different times, which can complicate transactional integrity and require careful handling to maintain system accuracy and reliability[58][41].

3.3.4 Workflow Management

In the environment of microservices, the complexity of interactions extends beyond the simple communication between two services. A critical aspect is the orchestration of multiple microservices working together to execute comprehensive business processes. This orchestration requires a nuanced approach to maintain the system's integrity and efficiency.

In this section, we'll explore how microservices can collaborate on workflows and processes. We'll delve into strategies like distributed transactions, which attempt to address these coordination challenges, and examine the saga pattern, an advanced concept that provides a structured approach to manage long-running, distributed business transactions within microservice architectures.

Database Transactions

In computing, transactions are a series of actions completed as a single unit, ensuring all changes are made or none if an error occurs. This concept is crucial in databases, where transactions (like insertions, deletions, or updates) must be successful, often spanning multiple tables. The term database transactions usually refers to ACID transactions[25], which is explained in table 3.5.

Letter	Stands for	Description
A	Atomicity	Ensures that all parts of a transaction are completed successfully, or none at all.
C	Consistency	Guarantees that a transaction only brings the system from one valid state to another.
I	Isolation	Ensures that transactions are performed independently and transparently.
D	Durability	Assures that once a transaction is committed, it will remain so, even in the event of a failure.

Table 3.5. ACID properties of database transactions

In microservices, ACID transactions apply to local operations within a single microservice, complicating atomic operations across multiple services. Unlike a monolithic database that ensures atomicity through ACID properties, a distributed microservices system handles changes across separate databases, as depicted in Figure 3.10. This leads to independent transactions that may succeed or fail separately, lacking atomicity for the entire operation.

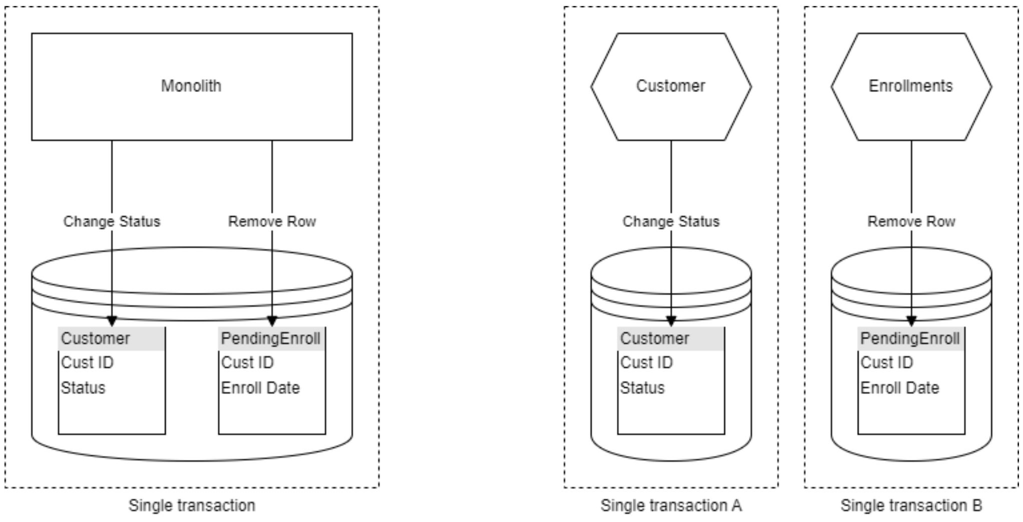


Figure 3.10. Example that show the difference between monolithic and microservices transactions.

Distributed transaction - Two-Phase Commit

The Two-Phase Commit (2PC) algorithm facilitates transactional updates across distributed systems, ensuring atomic transaction commits across multiple nodes. Essentially, it mandates that all involved nodes must either commit or abort together, maintaining the principle of atomic transactions[36]. Unfortunately, 2PC is often viewed as impractical for microservice architectures[44], thus this section explores the reasons behind this perspective, analyzing the limitations and challenges of applying 2PC in such contexts.

In the figure 3.11 we can see an example of the 2PC protocol process. The protocol is divided into two phases: the prepare phase and the commit phase. Initially, in the prepare phase, microservices are prompted to ready themselves for a potential atomic data change. Following this, the commit phase involves directing these microservices to execute the actual changes. Central to this process is a global coordinator, responsible for overseeing the transaction's lifecycle and engaging with microservices during both the prepare and commit phases. This coordinator is pivotal in determining whether the nodes can commit the proposed transaction and in issuing the final command to commit or abort[36] [63].

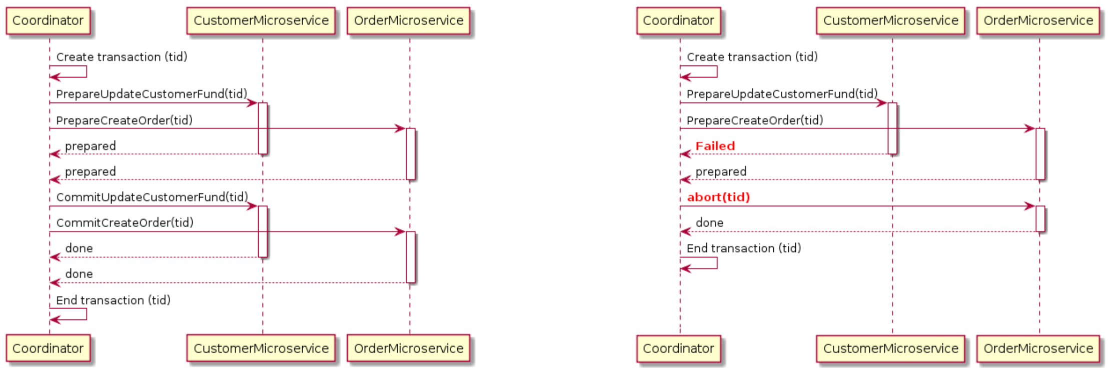


Figure 3.11. Example of the 2PC protocol process[63].

The major benefit of 2PC protocol is that it is a robust mechanism for ensuring consistency across distributed systems. Its dual phases — prepare and commit — assure that transactions are atomic, making all microservices commit successfully or none at all, preventing partial updates. Additionally, 2PC enforces read-write isolation, ensuring that any modifications remain invisible until the coordinating node finalizes the commit, maintaining transaction integrity throughout the process[63].

Although 2PC ensures atomicity, its limitations make it less suitable for numerous microservice-based systems[63]. The main issue of 2pc protocol are[24]:

- **Blocking**: The protocol locks objects until a transaction is complete, causing potential delays and deadlock.
- **Latency**: Waiting for all participant responses before proceeding adds to the transaction time.
- **Coordinator Risk**: The Transaction Coordinator is a critical point that can fail, blocking all transactions.
- **Participant Performance**: The entire transaction's speed is tied to the slowest participant, with failures necessitating full rollbacks.

Saga distributed transactions pattern

As described so far, to avoid coupling between microservices, the database-per-microservice pattern is utilized, allowing each service to manage its own data. This method offers several advantages: select the most suitable data store type, scale it independently, and maintain isolation from failures in other services[42]. This pattern has a drawback: it does not support ACID transactions across multiple services. To overcome this limitation, the Saga pattern can be employed.

Unlike a two-phase commit, the saga pattern is designed to effectively coordinate multiple state changes, ensuring data consistency and avoiding resource locks across microservices. It accomplishes this by decomposing the process into separate, independently executable activities. The adoption of the saga pattern necessitates the explicit modeling of business processes, which can yield significant benefits[44].

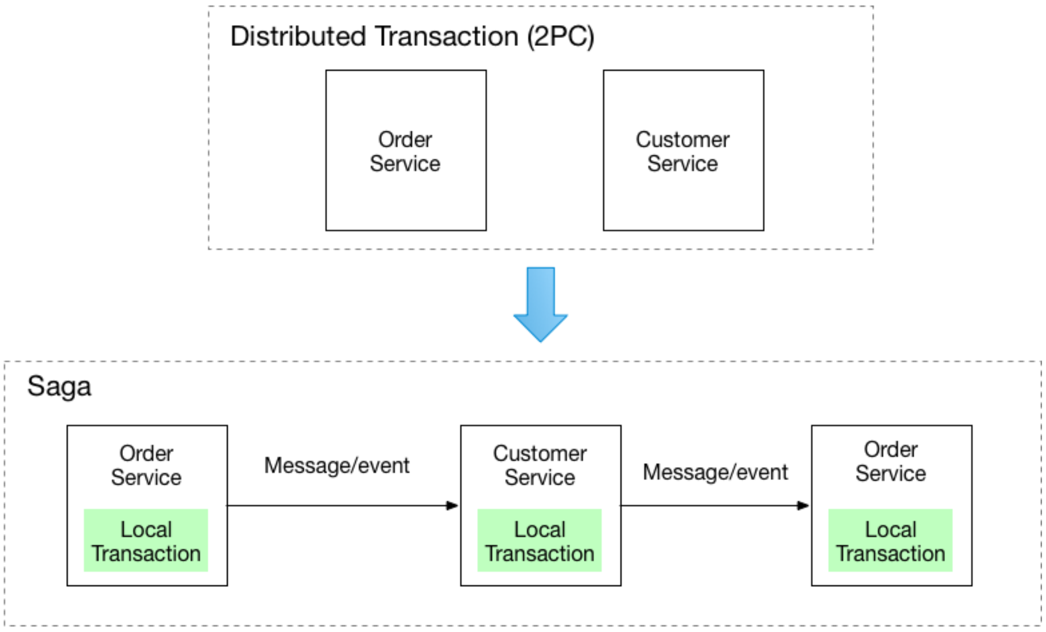


Figure 3.12. From 2pc to Saga pattern[56].

How showed in figure 3.12, the Saga pattern manages transactions across multiple services through a sequence of local transactions, each serving as an atomic work effort by a saga participant. In this pattern, every local transaction updates the database and publishes a message or event to initiate the subsequent local transaction within the saga. If any local transaction fails, typically due to a violation of business rules, the saga responds by executing compensating transactions to reverse the changes made by earlier local transactions. This approach ensures consistent and reliable transaction management in complex, distributed systems[42][56].

The key benefit of Saga is maintaining data consistency across multiple services without needing distributed transactions. However, it introduces complexities: developers must create compensating transactions to reverse earlier changes, and debugging becomes challenging, especially as the number of services involved increases. When a client initiates a saga through a synchronous request (like an HTTP POST), determining the saga's outcome is crucial. This can be managed in several ways[56]:

- **Immediate Response Post-Completion**: The service responds after the saga completes, ensuring a definitive outcome but possibly causing delays.

- **Initiation Acknowledgement with Periodic Polling**: The service acknowledges the saga's start, and the client periodically checks for the outcome.
- **Initiation Acknowledgement with Event Notification**: The service sends an initial response and notifies the client via an event (e.g., websocket) upon saga completion.

There are two prevalent methods for implementing the Saga pattern: choreography and orchestration. Each method presents unique challenges and requires specific technologies to effectively coordinate the workflow.

Choreography-based saga

Choreography in sagas refers to a decentralized method of coordination where participants communicate through the exchange of events, without relying on a central control point. In this approach, each local transaction emits domain events that activate local transactions in other services[42].

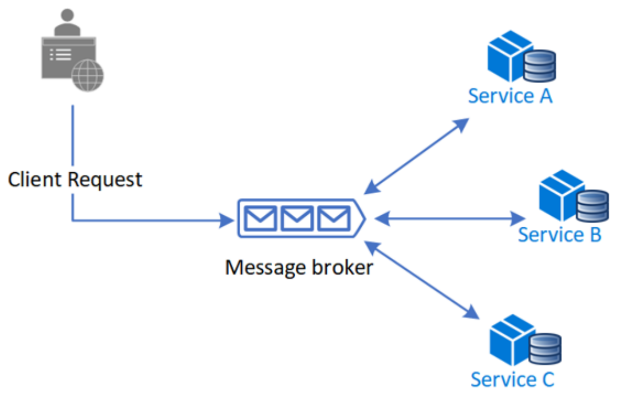


Figure 3.13. Choreography-based saga[42].

The workflow is well-suited for simpler processes with fewer participants, as it does not necessitate complex coordination logic or the implementation and maintenance of an additional service. This decentralization also prevents the emergence of a single point of failure. However, the approach has its limitations; as the workflow grows, it becomes increasingly challenging to track the interactions and commands between saga participants, potentially leading to cyclic dependencies. Integration testing can be complicated, requiring all services to be active to effectively simulate a transaction, posing a considerable challenge in practical applications[42].

Orchestration-based saga

Orchestration in sagas involves a central controller directing saga participants on which local transactions to carry out. The orchestrator manages all transactions, instructing participants on specific operations in response to events. It is responsible for executing saga requests, maintaining and interpreting the state of each task, and managing failure recovery through compensating transactions[42].

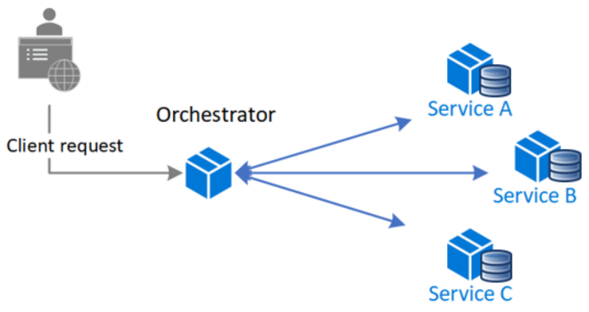


Figure 3.14. Orchestration-based saga[42].

This pattern is advantageous for complex workflows with numerous participants or when incorporating new participants over time, as it allows complete control over each participant and their activities. This approach eliminates cyclical dependencies by having the orchestrator solely depend on the saga participants and simplifies business logic by clearly separating concerns. Anyway, it introduces additional design complexities by requiring the implementation of specific coordination logic, and the orchestrator itself becomes a potential point of failure, managing the entire workflow and thus presenting a risk to the system's stability[42].

3.4 Serverless

3.4.1 Introduction

The advent of serverless computing marks a significant shift in the way applications are developed and managed, especially within the domain of microservices. As the architecture of an application expands, operational responsibilities grow correspondingly. Public cloud providers have risen to this challenge by offering an extensive suite of managed services, ranging from managed database instances and Kubernetes clusters to message brokers and distributed file systems. Leveraging these managed services translates to offloading substantial operational workload to third-party experts, who are often more equipped to manage these complex tasks[44].

Within the serverless offerings, services that facilitate Event-Driven Architecture hold a place of prominence. Serverless platforms inherently support EDA by abstracting away the infrastructure, allowing developers to concentrate on code and event flows rather than server management. Products such as message brokers, storage solutions, and databases are designed to seamlessly integrate with event-driven models, providing a responsive and efficient means to trigger and scale microservices based on real-time events[14].

Function as a Service (FaaS), a cornerstone of serverless offerings, perfectly complements EDA by providing a mechanism to deploy code that automatically responds to events without the need for explicit server provisioning. Developers simply deploy their functions, which are then executed in response to events, scaling automatically with the volume of requests. This serverless, event-driven model significantly reduces the complexity of scaling and managing infrastructure, allowing developers to focus on building responsive, efficient, and modern cloud-native applications.[44].

Example of Serverless

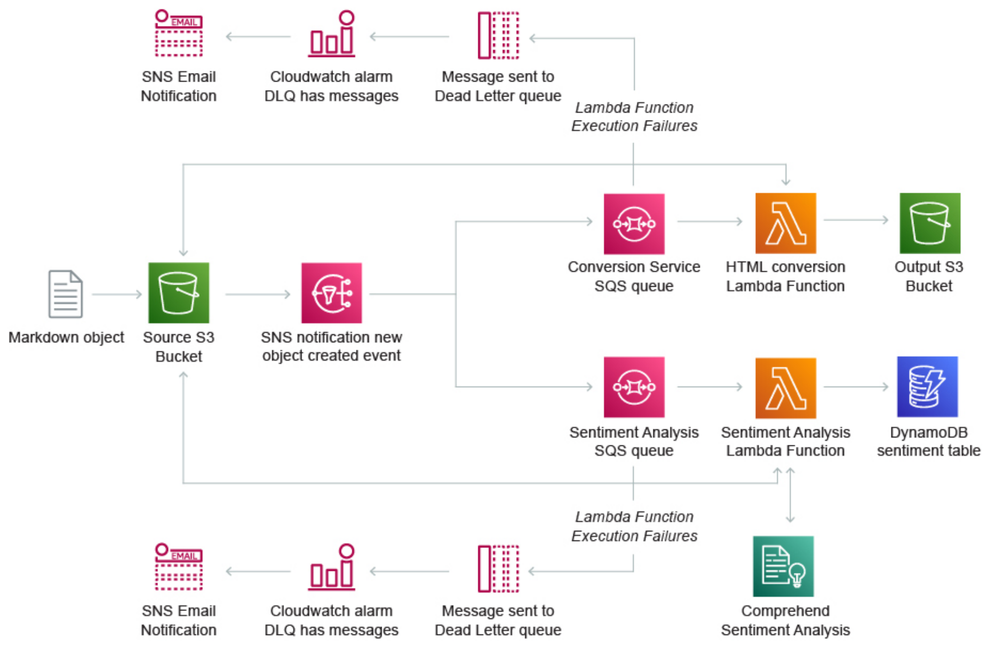

SNS Email Notification
Cloudwatch alarm DLQ has messages
Message sent to Dead Letter queue
Lambda Function Execution Failures
Conversion Service SQS queue
HTML conversion Lambda Function
Output S3 Bucket
Markdown object
Source S3 Bucket
SNS notification new object created event
Sentiment Analysis SQS queue
Sentiment Analysis Lambda Function
DynamoDB sentiment table
Lambda Function Execution Failures
SNS Email Notification
Cloudwatch alarm DLQ has messages
Message sent to Dead Letter queue
Comprehend Sentiment Analysis

Figure 3.15. Example of Serverless Architecture.

The diagram 3.15 presents a serverless data processing architecture that enables automatic handling of interview notes. When documents in Markdown format are uploaded to a cloud-based storage service, the system is designed to trigger multiple automated workflows. The first workflow involves a function-as-a-service platform, which is activated to convert the Markdown documents into HTML files; these are then saved back to the storage service in a different location for future access. Simultaneously, a second workflow commences, where another function is triggered through a message queuing service to perform sentiment analysis on the text. The results of this analysis are stored in a NoSQL database, allowing for efficient retrieval and analysis of the processed data.

The entire process is monitored by a cloud-based monitoring system that watches over the message queues and function executions. In case of any errors or malfunctions, an alert system is activated to send notifications, ensuring that any issues are addressed promptly. This serverless architecture exemplifies the modern approach to cloud computing, emphasizing automated, event-driven processes that facilitate quick, scalable, and efficient data processing without the need for managing server infrastructure[5].

3.4.2 Benefits

Cost-effective

Serverless computing provides a cost-efficient model by charging only for the backend services when code is executed, following an event-based model. This stands in contrast to traditional hosting models, where costs are incurred for dedicated servers regardless of whether they are in active use. By eliminating the need to provision, manage, and maintain physical servers, serverless computing allows for significant cost savings. It is particularly economical for applications that do not run continuously, as you pay solely for the resources utilized during the execution time, without the overhead of server maintenance and upgrades[62][37]. However, it is worth noting that for applications with constant runtime, serverless computing may not always be the most cost-effective option[43].

Scalability

Seamless scalability is a hallmark of serverless computing, allowing applications to adapt quickly to fluctuating demands without the need for server management. It operates much like a bus system that adjusts its capacity according to the number of passengers; serverless infrastructure automatically scales up or down as user demand changes. This model ensures that server capacity is precisely aligned with the necessary demand, providing ample resources when the number of user requests increases without being constrained by the limitations of server storage and performance capabilities[62]. This adaptability is considered one of the primary benefits of serverless computing, enabling businesses, especially small and medium enterprises (SMEs), to handle traffic surges efficiently and cost-effectively, ensuring service continuity and performance without the disruptions commonly associated with traditional hosting models[37].

Reliability

Serverless computing inherently enhances application reliability due to multiple layers of built-in redundancy. Since serverless applications are not tethered to a single origin server, they have the flexibility to execute code from various locations, optimizing the function execution closer to the end-user[62][37]. This distributed nature not only helps in reducing latency, thereby improving performance, but also contributes to high availability. The risk of service outages or failures is significantly reduced, ensuring that end-users have consistent and reliable access to the application functions[43].

Increased productivity

Serverless computing streamlines the development process, making it much faster and more efficient to build and deploy applications. Developers are freed from the time-consuming tasks of server setup and infrastructure maintenance, allowing them to concentrate on innovation and creativity without being hindered by server constraints. As a result, developers can launch products more swiftly, as there is no need for traditional server-side installations or workflow monitoring. The serverless model enables direct code uploads and immediate function execution on the cloud provider's infrastructure. This agility extends to application updates and patches, which can be implemented incrementally, targeting individual functions rather than disrupting the entire service. This approach not only enhances development velocity but also reduces the resources allocated to DevOps, leading to cost savings and allowing developers to focus purely on coding and product improvement[62][37].

Resource Utilisation

The efficiency of serverless computing in terms of resource utilization marks a substantial advancement towards greener technology practices. By engaging resources solely during code execution, serverless platforms significantly minimize waste and avoid the energy costs associated with powering idle servers. This responsiveness to real demand underscores serverless computing as an eco-friendly choice, particularly appealing to organizations aiming to reduce their environmental impact and achieve sustainability goals. Such a model is not only in line with ecological considerations but also optimizes operational expenditures, positioning serverless computing as a judicious approach for backend operations that are both environmentally and economically conscious[20].

3.4.3 Drawbacks

Performance Issues

Serverless architectures can encounter performance hiccups, particularly when reactivating idle applications. Known as "cold start" latency, this issue arises when the serverless platform takes additional time to provision resources for an application that hasn't been used recently. While steps such as reducing code length can mitigate the impact, they may lead to a trade-off where developers must manage a larger number of smaller, more manageable functions. Despite these efforts, the initial delay in resource setup inherent to cold starts can still lead to slower response times when an application is invoked after being dormant[37].

Limited Control

In serverless computing, the cloud provider's management of the infrastructure results in developers having limited control over the environment and certain application parameters. This lack of direct control can hinder customization efforts and may pose issues if specific needs arise that require adjustments at the infrastructure level. Additionally, cloud providers may offer limited support for certain programming languages and runtime environments, which can restrict the technology stack options for development[62]. Another consideration is the potential for vendor lock-in; reliance on a single provider's serverless architecture can create dependencies that complicate migrating to a different provider's services[43]. These factors highlight the trade-offs between the convenience of serverless models and the need for flexibility in application deployment and management.

Increased Complexity

While developers benefit from the reduced need to manage servers and scalability concerns, they are faced with the intricacies of serverless architectures, such as state management, integration with other services, and the unique security considerations these environments entail. Moreover, the serverless paradigm often requires a different approach to application design, with a focus on individual functions and services, which can be a departure from traditional monolithic architectures. These elements can sometimes increase the cognitive load on developers and complicate the application lifecycle management, especially for those accustomed to more control over the server environment[62][37].

Testing & Debugging

The abstraction inherent in serverless computing can obscure backend processes, complicating tasks such as testing and debugging. Without traditional backend visibility, developers might find it difficult to conduct in-depth inspections, detect faults, and identify errors within the cloud environment, adopting more comprehensive strategies to anticipate and address potential issues. This added complexity in monitoring and diagnosing problems requires a proactive approach to fault prediction and management to minimize service disruptions for users[37]. Debugging, in particular, presents a challenge as the infrastructure's concealed nature takes away some of the direct troubleshooting tools that developers rely on in more conventional setups[43].

Security concerns

In serverless computing, the responsibility for security configurations largely rests with the cloud provider, which can lead to concerns over limited control and potential security risks if the provider's measures are not up to par. Careful evaluation of a provider's security protocols is crucial, and applications must be designed with security best practices in mind to mitigate risks. Despite the robust security measures typically employed by service providers, some users may still be apprehensive about relinquishing direct control over security. Serverless architecture often involves sharing resources between different clients on the provider's infrastructure, which introduces the need for strong isolation policies to ensure that one client's activities do not compromise another's security. Complying with these security considerations requires trust in the provider's ability to safeguard the environment while also necessitating that developers stay vigilant and informed about the security posture of their serverless applications[37][43].

3.4.4 Function-as-a-Service

Functions-as-a-Service (FaaS) represents a modern "serverless" approach that has gained widespread popularity for its ability to simplify the development and operation of applications. In FaaS, functions - pieces of code - are executed in response to specific triggers, running only until their task is completed. This model allows for efficient scaling, as the number of function executions automatically adjusts based on the load, making it highly effective for event-driven systems where workload can be unpredictable.

FaaS operates on a consumer/producer basis, where functions are transient, ceasing to exist after a set duration along with their connections and state. This design necessitates careful planning in function development but offers significant benefits. The cost efficiency is notable, as you incur charges only for the time your code is running. Additionally, the FaaS platform manages the operational complexities, such as automatically handling the spinning up and down of functions and ensuring high availability and robustness with minimal effort from the developer.

The integration of FaaS in application development reduces operational overhead substantially, making it an ideal choice for implementing simple to moderately complex solutions within event-driven architectures. This approach allows developers to focus more on building functionality without the burden of managing infrastructure, leading to more agile and responsive application development[14].

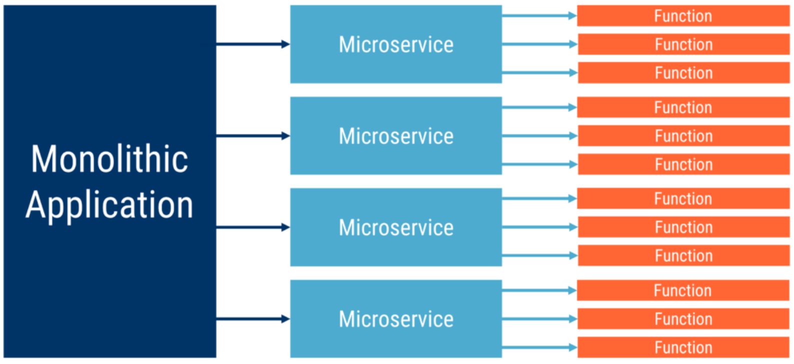


Figure 3.16. Evolution of architectures[18].

Cold Start and Warm Starts

A "cold start" refers to the initial phase of a function in a FaaS (Function-as-a-Service) environment when it is first activated or after a period of dormancy. During this phase, the system initiates a container, loads the code, establishes connections with the event broker, and sets up other necessary external resource connections. Once these preparatory steps are complete, the function transitions to a "warm state," primed to process events.

In this state, the function actively processes events until it either completes its task or reaches a timeout, at which point it enters a suspended or hibernation mode.

FaaS frameworks typically strive to optimize resource usage by reactivating these suspended functions for subsequent tasks. In scenarios where there is a consistent flow of events, a function may cycle between active processing and brief termination due to timeout. However, if the connections to the event broker and state stores remain active during these short periods of inactivity, the system can quickly resume processing with the reactivated function instance. This reuse of function instances enhances efficiency, reducing the overhead and latency associated with initializing a function from a cold start[14].

Maintaining State

FaaS (Function-as-a-Service) platforms are designed for stateless operations, primarily due to the ephemeral nature of functions. Since these functions have a limited lifespan, any FaaS-based solution that requires statefulness typically relies on external stateful services. This design choice aligns with the objective of FaaS providers to offer rapid, highly scalable processing resources that are not constrained by the location of data. Local state dependence, where a function needs to access state from previous executions, restricts execution to specific nodes where this state is available. This limitation hinders the inherent flexibility of FaaS.

To maintain scalability and flexibility, FaaS providers often implement a policy that prohibits local state within functions, directing that all stateful data be stored externally. Functions then interact with these external state stores in the same manner as any other client: by establishing a connection and utilizing the appropriate API. Consequently, it's essential for functions to explicitly persist and retrieve any state as part of their operational process. This approach ensures that while the functions themselves remain stateless, stateful interactions are still possible through external mechanisms[14].

Integrated with Event-Driven Architectures

This technology is inherently stateless and is activated by specific triggers or events, making it an ideal match for event-driven architectures. In an event-driven architecture, components react to events such as user actions, sensor outputs, or messages from other systems. FaaS fits seamlessly into this paradigm because it is designed to respond to events automatically. When an event occurs, it triggers a specific function, which executes and then terminates, ensuring a minimal use of resources. This model is highly efficient and scalable, as it allows for on-demand, ephemeral computing resources that are only used when needed. The stateless nature of FaaS ensures that each function execution is isolated and independent, enhancing both scalability and reliability. This integration of FaaS with event-driven architecture represents a powerful and flexible approach to building and operating modern, responsive, and efficient applications.

Mapping to microservices

There are two primary modes of mapping microservices using Function-as-a-Service technology: "Function per Microservice" and "Function per Aggregate".

In **Function per microservice** method, a single instance of a microservice is deployed as a singular function, as illustrated in figure 3.17. This strategy maintains the integrity of the microservice instance as the fundamental unit of deployment in the FaaS platform.

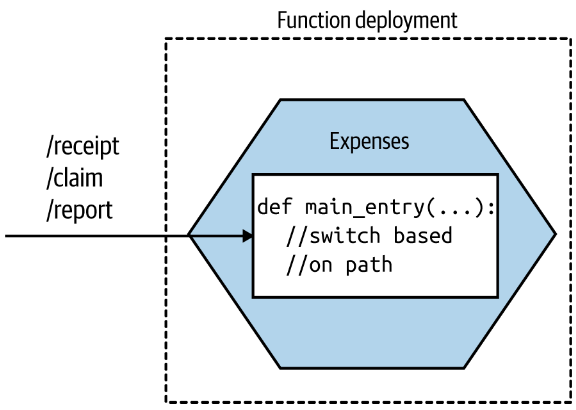


Figure 3.17. Example of Function per microservice.

In this setup, when the function is invoked, the FaaS platform activates a singular entry point within the deployed function. This necessitates a mechanism within the function for routing the invocation to various functionalities that the microservice comprises. For instance, if you have a microservice like the Expenses service, designed as a REST-based system, it could expose different endpoints such as /receipt, /claim, or /report. In a single function deployment model, requests to any of these endpoints are funneled through the same entry point. Consequently, there must be an internal dispatch system within the function that discerns the path of the incoming request and directs it to the correct segment of the microservice for processing. This design ensures streamlined handling of various functionalities within a microservice through a unified FaaS-based interface[44].

The **Function per aggregate** approach adopts a detailed perspective, dividing a single microservice into several functions, where each function is responsible for a distinct part or "aggregate" of the microservice's overall role. In this context, an aggregate refers to a group of domain objects treated as a cohesive unit for the purposes of data manipulation. This approach aligns well with domain-driven design, where aggregates are typically defined as collections of objects managed together and often represent real-world entities.

If a microservice manages multiple aggregates, a practical solution is to dedicate a function to each aggregate, as depicted in figure 3.18. This strategy ensures that all the logic pertaining to an individual aggregate is encapsulated within a single function. Such an arrangement simplifies the management and ensures consistent implementation of the aggregate's life-cycle[44].

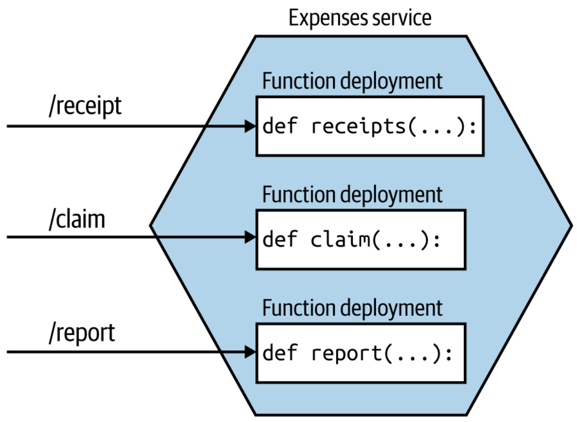


Figure 3.18. Example of Function per aggregate.

Adopting this model alters the traditional concept of a microservice instance from a single deployment unit to a more conceptual framework composed of various functions. These functions, representing different parts of the microservice, can theoretically be deployed independently. This not only provides flexibility in deployment but also allows for more granular scaling and maintenance of the service.

3.4.5 Market Overview

The serverless computing market is poised for rapid expansion, with projections estimating a compound annual growth rate (CAGR) of 22.2% from 2024 to 2032, fueled by an escalating demand for cost-effective computing. Organizations are keen to diminish capital expenditures on IT infrastructure, and serverless computing meets this need by delivering backend services that only incur costs when actively used. This pricing model significantly enhances organizational cost-efficiency and augments scalability, reliability, and the pace of product development to market[54].

Contributing to this growth is a concerted move towards sustainable IT practices, with companies embracing serverless computing to cut down on IT hardware dependency and improve energy efficiency, thereby supporting their environmental, social, and governance (ESG) objectives. The adaptability of serverless computing, compatible with numerous programming environments, is finding traction across vital sectors like manufacturing, IT, and banking. This is further amplified by the boom in web and mobile application development, thanks to broader mobile and internet accessibility, where serverless computing's efficient resource utilization and speedy deployment significantly lower operational and maintenance costs. Collectively, these factors are making serverless computing a compelling strategy for organizations seeking to bolster user experiences and maintain a competitive edge, indicating a strong and dynamic future for the serverless computing market[54].

Trends and Success Rates

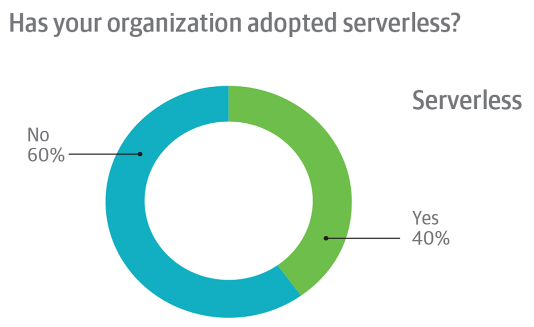


Figure 3.19. Serverless adoption among survey respondents' organizations.

The O'Reilly serverless survey[50], with over 1,500 participants, showcased a high level of engagement with serverless architecture, reflecting its growing significance in the tech community. The figure 3.19 shows around 40% of respondents' organizations had embraced serverless, drawn by the promise of reduced operational costs and the benefit of automatic scaling. These adopters spanned across various industries and company sizes, indicating that serverless solutions are not limited to any specific sector or business scale. The survey also revealed that while a substantial 60% of organizations had not yet adopted serverless, the reported benefits and the growing success rates among experienced users could serve as a catalyst for wider adoption. The benefits driving this interest include not only cost efficiency and scalability but also the agility to adapt to new business requirements rapidly.

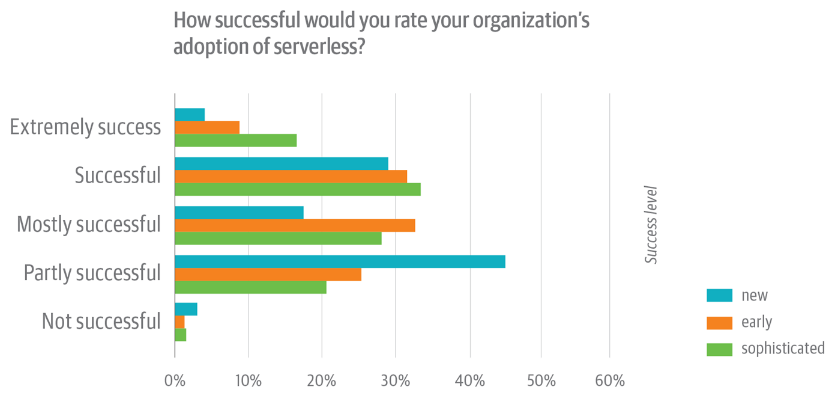


Figure 3.20. Success rating of serverless adoption among survey respondents, broken down by serverless experience level.

In figure 3.20 is illustrate how adoption rates varied by experience, with half of those using serverless for over three years considering their implementation a success, compared to a lower success rate among newer adopters. This suggests a correlation between experience with serverless technologies and successful deployment, highlighting the importance of familiarity and understanding of serverless paradigms in achieving positive outcomes.

The data indicates that the serverless model is gaining traction beyond the realm of early adopters and innovators, with a diverse range of professionals from various fields showing interest. This is indicative of a broader trend towards infrastructure management models that offer greater efficiency and flexibility. The survey paints a picture of serverless as a maturing field with significant potential for growth, as more organizations begin to realize and leverage its advantages for building modern, scalable, and cost-effective applications.

Costs

The cost of using a FaaS model can be lower than using a Software as a Service model, as you only pay for the specific code you run, rather than paying a monthly or yearly fee for access to the entire software. Additionally, with a FaaS model, you only pay for the resources that your code actually uses, like computing time and data storage, rather than paying for the entire server or infrastructure even when your code is not running. This can result in significant cost savings; it tends to be more cost-effective than SaaS for short-lived, event-driven workloads as the customer only pays for the specific computation and resources used by each function[1].

In this section, we delve into a practical case study drawn from 'Horsa' the company hosting my thesis research, to illustrate the impact of Enterprise Resource Planning systems in small business environments. The case involves a small Italian company utilizing Microsoft Business Central, with a user base of seven employees. By extrapolating this data and applying it to a Function as a Service environment, we aim to demonstrate the cost-efficiency and potential benefits of embracing FaaS technology in small-scale business operations. This analysis not only highlights the adaptability and scalability of FaaS solutions but also presents a tangible example of how transitioning to cloud-based platforms can offer significant cost savings and operational advantages for small businesses.

Object	Usage per day	Conversion	Request/Day
Pages	3K	1 Page = 2 function request	6K
		1 to read - 1 to write	
Report	150	1 Report = 1 function req.	150
WebService	76K	1 WS = 1 function req.	76K

Table 3.6. Example of ERP usage

Inside the teble 3.6, we can see the usage of the ERP system. Now applying these data to the FaaS cost model. Considering a month with 26 working days we have:

$$(150 + 6000 + 76000) * 26 = 2135900 \cong 2.2M \text{ Request per month} \tag{3.1}$$

We will use the FaaS service called Lambda provided by Amazon AWS, prices are calculated based on the number of calls, duration, and memory allocation for the respective function. By definition, Lambda functions must be fast and lightweight, so we will use half a second as the duration of each request and an allocated memory amount of 256 Mb, we will calculate the price not considering the free plan offered by Amazon:

$$\text{Total processing time: } 2200000 * 0.5sec = 1100000.00sec \tag{3.2}$$

$$256 \text{ MB} * 0.0009765625 = 0.25 \text{ GB} \tag{3.3}$$

$$\text{Total processing: } 0.25 \text{ GB} * 1100000.00sec = 275000 \text{ GB/s} \tag{3.4}$$

$$\text{Total processing cost: } 275000 \text{ GB/s} * 0.0000195172 \text{ USD} = 5.3672 \text{ USD} \tag{3.5}$$

$$\text{Request cost: } 2200000 * 0.00000023 \text{ USD} = 0.51 \text{ USD} \tag{3.6}$$

$$\text{Total monthly lambda cost: } 5.3672 \text{ USD} + 0.51 \text{ USD} = 5.88 \text{ USD} \tag{3.7}$$

To this we must also add the cost of the API Gateway for routing requests:

$$2200000 * 0.00000117 \text{ USD} = 2.57 \text{ USD} \tag{3.8}$$

Finally, the total cost of the enviroment is 8,45 USD per month, that show the flexibility and cost-effectiveness of the system. This example gives us the reason why the serverless has rapidly become a popular choice for many organizations, even if the concept is still relatively new.

Chapter 4

Technologies

This chapter focuses on the specific technologies utilized in implementing Kube. Foremost among these is the suite of AWS services, including Lambda for serverless computing, SQS (Simple Queue Service) and SNS (Simple Notification Service) for messaging, and RDS (Relational Database Service) for database management. These services collectively provide a resilient and scalable infrastructure for Kube. Additionally, the platform leverages the serverless architecture to optimize resource utilization and reduce operational overhead and the Go programming language for its efficiency and suitability for building high-performance applications. The user interface of Kube is crafted using Flutter, a versatile UI toolkit, which enhances the platform's accessibility and aesthetic appeal. Together, these technologies form the cornerstone of Kube, enabling it to deliver good performance and user experience.

4.1 Amazon AWS Services

AWS is recognized as the world's most comprehensive and broadly adopted cloud platform. It offers over 175 fully featured services from data centers globally. AWS is utilized by a diverse range of customers, including rapidly growing startups, large enterprises, and leading government agencies, to reduce costs, increase agility, and accelerate innovation. The platform provides a wide array of cloud-based products including compute, storage, databases, analytics, networking, mobile, developer tools, management tools, IoT, security, and enterprise applications, all available on-demand with pay-as-you-go pricing[6]. In this section we will analyze the most important services offered by Amazon AWS that we will use in the development of the project.

Example of AWS services integration

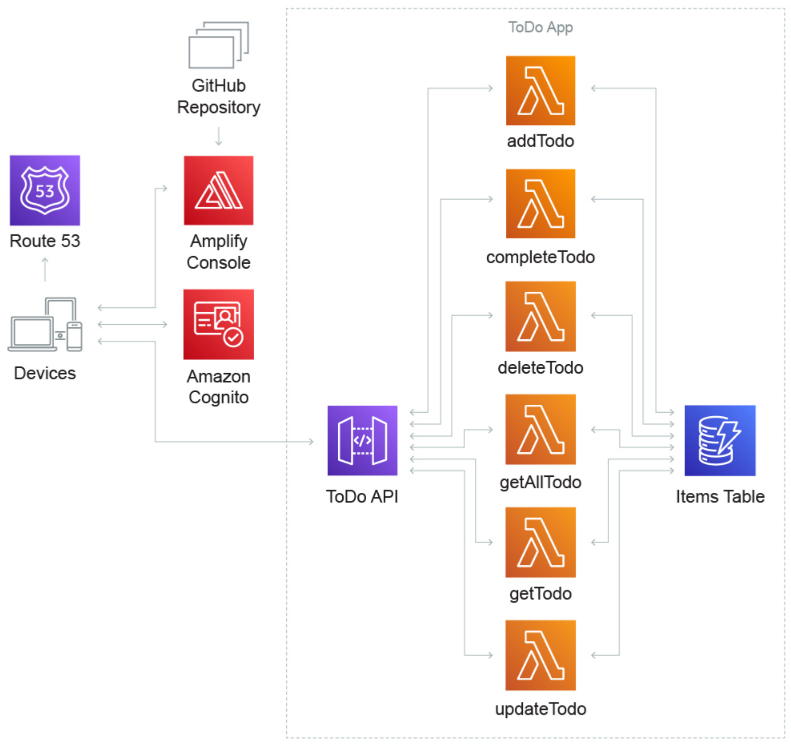


Figure 4.1. Example of AWS services integration.

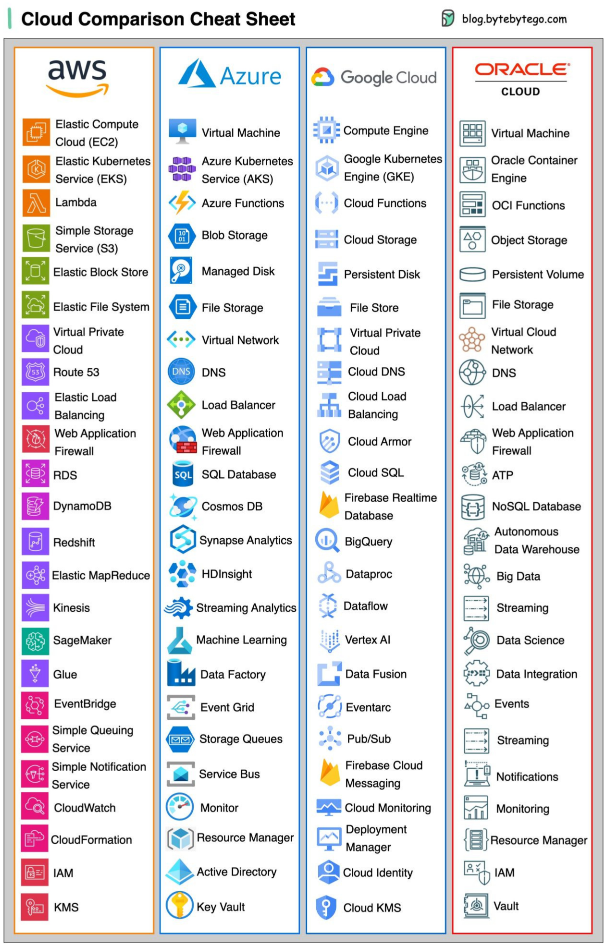
The image 4.1 illustrates the architecture of a simple "to-do list" web application built using serverless technology on the Amazon public cloud. This event-driven application is designed to allow registered users to manage their tasks efficiently. Users interact with the application to create, update, view, and delete to-do items. The architecture integrates several Amazon Web Services to achieve this functionality. Each component of the architecture is orchestrated to respond to specific events triggered by user actions, such as adding a new to-do item or marking one as complete, making the entire application responsive and scalable without the need to manage server infrastructure[5].

4.1.1 Pros and cons

Amazon Web Services (AWS) is renowned for its extensive array of over 200 fully-featured services, ranging from fundamental infrastructure technologies like compute, storage, and databases to cutting-edge fields such as machine learning, AI, data lakes, analytics, and IoT. This diversity offers tailored solutions for different applications, optimizing both cost and performance. However, the sheer breadth of services can be overwhelming for less experienced users and creates a potential dependency on AWS, making it challenging to switch providers. Additionally, while AWS allows for cost and performance flexibility, improper resource management or unsuitable service choices can lead to high expenses.

4.1.2 Alternatives

Microsoft Azure and Google Cloud Platform (GCP) stand as significant alternatives to Amazon Web Services. Azure, backed by Microsoft's legacy in enterprise software, excels in integrating with existing Windows-based environments, making it a preferred choice for organizations deeply embedded in Microsoft's ecosystem. It offers a strong focus on hybrid cloud, AI, and machine learning capabilities. On the other hand, GCP is highly regarded for its deep expertise in data analytics, machine learning, and open source technologies, leveraging Google's pioneering work in these areas[40][27]. The table 4.2 shows a comparison between the major four cloud services providers.


Cloud Comparison Cheat Sheet

blog.bytebytego.com

aws	Azure	Google Cloud	ORACLE CLOUD
Elastic Compute Cloud (EC2)	Virtual Machine	Compute Engine	Virtual Machine
Elastic Kubernetes Service (EKS)	Azure Kubernetes Service (AKS)	Google Kubernetes Engine (GKE)	Oracle Container Engine
Lambda	Azure Functions	Cloud Functions	OCI Functions
Simple Storage Service (S3)	Blob Storage	Cloud Storage	Object Storage
Elastic Block Store	Managed Disk	Persistent Disk	Persistent Volume
Elastic File System	File Storage	File Store	File Storage
Virtual Private Cloud	Virtual Network	Virtual Private Cloud	Virtual Cloud Network
Route 53	DNS	Cloud DNS	DNS
Elastic Load Balancing	Load Balancer	Cloud Load Balancing	Load Balancer
Web Application Firewall	Web Application Firewall	Cloud Armor	Web Application Firewall
RDS	SQL Database	Cloud SQL	ATP
DynamoDB	Cosmos DB	Firebase Realtime Database	NoSQL Database
Redshift	Synapse Analytics	BigQuery	Autonomous Data Warehouse
Elastic MapReduce	HDInsight	Dataproc	Big Data
Kinesis	Streaming Analytics	Dataflow	Streaming
SageMaker	Machine Learning	Vertex AI	Data Science
Glue	Data Factory	Data Fusion	Data Integration
EventBridge	Event Grid	Eventarc	Events
Simple Queuing Service	Storage Queues	Pub/Sub	Streaming
Simple Notification Service	Service Bus	Firebase Cloud Messaging	Notifications
CloudWatch	Monitor	Cloud Monitoring	Monitoring
CloudFormation	Resource Manager	Deployment Manager	Resource Manager
IAM	Active Directory	Cloud Identity	IAM
KMS	Key Vault	Cloud KMS	Vault

Figure 4.2. Cloud services comparison[17].

4.1.3 AWS API Gateway

Amazon API Gateway is a fully managed service that simplifies the creation and maintenance of APIs, serving as the gateway for applications to access backend data and services. It supports various workloads, including real-time communication through RESTful or WebSocket APIs, and handles tasks like traffic management, security, and monitoring. Plus, there are no upfront fees, and you pay based on your API usage, with pricing that scales according to your needs[7].

4.1.4 AWS Lambda

AWS Lambda is a serverless computing service from Amazon Web Services, designed to run code without server provisioning or management. It executes code in a high-availability environment, handling all aspects of computational resource administration. This approach allows for code organization into Lambda functions, which are executed and automatically scaled as needed, with billing based on compute time used. Ideal for scenarios requiring rapid scaling, Lambda supports diverse applications like real-time data processing with Amazon S3, streaming data handling with Amazon Kinesis, and creating scalable web and serverless back-ends for IoT and mobile devices. Key features include easy function configuration, environment variables, version management, container image support, packaging libraries, monitoring tools, HTTP(S) endpoints, streaming responses, and code signing for security. Lambda's flexibility, scalability, and cost-effectiveness make it an attractive solution for various cloud computing needs, emphasizing efficiency and developer focus[8].

AWS Lambda functions can be invoked in various ways, depending on the needs of the application. One common method is through event triggers, which can include changes in data within AWS services like S3 bucket updates or DynamoDB table updates. These events can be configured to invoke a Lambda function synchronously or asynchronously how showed in figure 4.3.

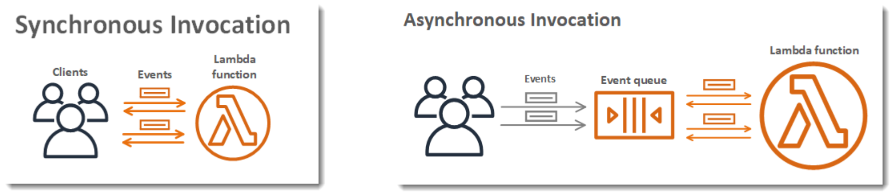


Figure 4.3. Lambda invocation types.

Synchronous invocation, where the caller waits for the function to process the event and return a response, is commonly used in scenarios like API executions through Amazon API Gateway. Asynchronous invocation is employed when the order of execution is not critical. In this case, events are placed in a queue before being sent to the function, and AWS Lambda manages the function's invocation rate. For asynchronous execution, AWS also provides services like Amazon Simple Queue Service (SQS) for queueing messages or Amazon Simple Notification Service (SNS) for delivering messages to subscribing endpoints or functions. These services can be directly integrated with Lambda to handle event-driven, scalable computing architectures, allowing developers to focus on code rather than infrastructure management.

4.1.5 AWS RDS

Amazon Relational Database Service (Amazon RDS) is a web service from AWS that simplifies setting up, using, and scaling relational databases in the cloud. It offers scalable, cost-effective solutions for standard industry relational databases, managing common administrative tasks, thus allowing users to focus more on their applications and user engagement. As a fully managed service, Amazon RDS handles the majority of management tasks, relieving users from manual and time-consuming database maintenance. It supports various popular database engines, such as Amazon Aurora, MySQL, MariaDB, PostgreSQL, Oracle, and SQL Server. Additionally, Amazon RDS provides deployment flexibility, including on-premises options with Amazon RDS on AWS Outposts. This combination of versatile database engine support, automated management, and deployment options make Amazon RDS a comprehensive and user-friendly solution for managing relational databases in the AWS Cloud[9].

RDS vs DynamoDB

AWS does not offer only SQL options with RDS; there is also DynamoDB, a NoSQL alternative for different database needs. While RDS excels in structured data management and complex querying capabilities via SQL, DynamoDB provides a flexible schema with key-value and document data models, delivering quick and predictable performance. RDS is preferable for traditional applications that need transactional support, complex joins, and other SQL operations. In contrast, DynamoDB is tailored for modern applications that demand scalability, low-latency data access, and where database management overhead should be minimized. The choice between RDS and DynamoDB hinges on the application's data requirements and scalability demands.

4.1.6 AWS SQS

Amazon Simple Queue Service (SQS) is a fully managed message queuing service for microservices, distributed systems, and serverless applications, offering secure and reliable data transfer without losing messages or depending on other services' availability. Amazon SQS provides key benefits such as security, with user-controlled access and server-side encryption options using AWS Key Management Service, and durability, ensuring message storage across multiple servers. Its high availability is maintained through a redundant infrastructure for consistent message access. The service is highly scalable, processing each request independently to manage load spikes, and guarantees reliability by locking messages during processing to support multiple producers and consumers. Amazon SQS also allows for customization, like setting default delays on queues or storing large message contents on Amazon S3 or DynamoDB. These features make Amazon SQS an efficient and versatile tool for handling large volumes of messages in various application architectures, offering a combination of reliability, scalability, security, and customization[10].

4.1.7 AWS SNS

Amazon Simple Notification Service (Amazon SNS) is a fully managed service offering effective Pub/Sub messaging for both application-to-application (A2A) and application-to-person (A2P) communication. It facilitates high-throughput, push-based messaging among distributed systems, microservices, and serverless applications, integrating seamlessly with Amazon SQS, AWS Lambda, and other services. A2P messaging extends capabilities to customer communications through SMS, push notifications, and emails. Amazon SNS stands out for simplifying messaging architectures while reducing costs through features like message filtering, batching, ordering, and deduplication. It also enhances message durability with storage, delivery retries, and dead-letter queues. Additionally, it supports strict FIFO message delivery, ensuring accuracy and consistency across applications. This combination of features makes Amazon SNS a versatile tool for a wide range of messaging scenarios, from system integration to direct customer engagement[11].

4.1.8 AWS Cognito

Amazon Cognito is a comprehensive service designed to implement secure, frictionless customer identity and access management (CIAM) in a scalable manner. With Amazon Cognito, you can offer a smooth management of customer identity and access, thanks to its affordable and customizable platform. It includes features such as adaptive authentication, support for compliance, and data residency requirements.Amazon Cognito is capable of scaling to millions of users with a fully managed, high-performance, and reliable identity store. It also provides access to federation using OpenID Connect (OIDC) or SAML 2.0 and integrates with a broad range of AWS services and products. This platform supports up to 50,000 active users per month for free under the AWS free tier plan, making it a cost-effective solution for businesses of varying sizes[12].

4.2 Firebase Cloud Messaging

Firebase Cloud Messaging (FCM)[31] is a powerful cloud solution for messages on iOS, Android, and web applications for free. It provides a reliable and efficient connection between servers and devices that allows for the delivery of notifications or messages. FCM offers versatile messaging options including topic messaging, which allows you to send a message to multiple devices that have opted in to a particular topic, device group messaging, allowing for messages to devices that belong to a group, and direct messaging to individual devices. This scalability makes it an essential tool for developers looking to engage their user base effectively, with the added benefits of analytics and performance tracking.

4.2.1 How it works

Firebase Cloud Messaging employs a client-server architecture where the FCM backend is responsible for handling and routing messages. The client app on a user's device communicates with the FCM via an SDK, which manages the registration process and token generation. This token uniquely identifies the app instance and enables secure message delivery to the device. Messages sent from the developer's server to the FCM backend can be payload-specific, directing the FCM to deliver them as notification or data messages. FCM then optimizes message delivery by queuing them, managing priority, and even aggregating messages for network efficiency. This architecture supports a high level of scalability and reliability in delivering messages across platforms and devices globally.

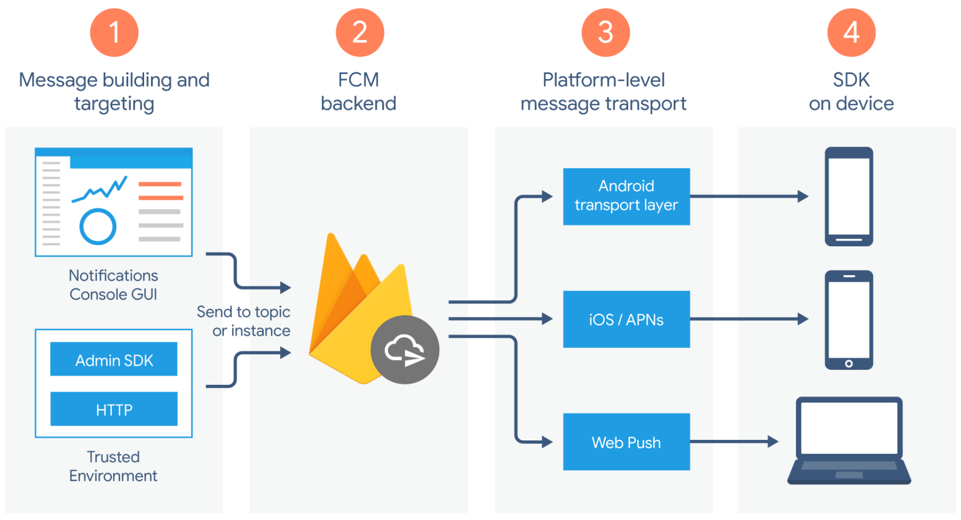

1
Message building and targeting
2
FCM backend
3
Platform-level message transport
4
SDK on device
Notifications Console GUI
Admin SDK
HTTP
Trusted Environment
Send to topic or instance
Android transport layer
iOS / APNs
Web Push

Figure 4.4. FCM architecture.

The image 4.4 show the FCM's operational architecture:

1. Tools for crafting message requests, including a GUI via the Notifications composer for notifications, and server environments like Cloud Functions for Firebase or App Engine, supported by the Firebase Admin SDK or FCM server protocol, for comprehensive message type handling.

2. The central FCM server that handles incoming message requests, manages topic-based message distribution, and assigns metadata like message IDs.

3. A device-level transport system that ensures messages reach their destination, varying by platform: Android devices, Apple devices and Web push protocols for browsers

4. The FCM SDK, which resides on the end-user's device, and is responsible for displaying notifications or processing messages, depending on the app's state and custom

logic.

4.3 GO Language

For my project, I chose the Go language[28] due to its inherent cloud-native characteristics and enhanced performance in cloud environments. It presents numerous advantages and capabilities:

- Concurrency in Cloud Computing: Go is tailored for building highly reliable concurrent applications, a necessity in cloud computing where coordinating access to shared resources is crucial. This makes Go an excellent choice for scalable cloud systems.
- Development Cycle and Server Performance: Go addresses the trade-off between development cycle time and server performance. A significant portion of Cloud Native Computing Foundation projects use Go. Its fast build times, lower memory and CPU utilization, and instant server start-up times make it a cost-effective option for cloud applications.
- Addressing Modern Cloud Challenges: Go provides standard idiomatic APIs and built-in concurrency to leverage multicore processors. Its low-latency and "no knob" tuning offer a balance between performance and productivity, enabling teams to adapt quickly to changing needs.
- Strong Ecosystem for Service Development: Go's standard library includes tools for HTTP servers and clients, JSON/XML parsing, SQL databases, and security/encryption. The runtime includes tools for race detection, benchmarking, code generation, and static code analysis. Major cloud providers and open-source libraries offer Go APIs, supporting a wide range of services and functionalities.

It have the drawback too, it has faced criticism for its until-recent lack of generics, leading to less flexible code, and its verbose error handling approach. While Go's standard library is comprehensive, it sometimes falls short in specialized third-party libraries compared to other languages. In essence, Go excels in backend and cloud services development but might not be ideal for all project types.

4.3.1 GO CDK

The Go Cloud Development Kit[29] is an open-source project aimed at enhancing the experience of developing cloud applications with Go. It provides vendor-neutral, commonly used generic APIs that work across different cloud providers, supporting hybrid cloud deployments and the integration of on-premises (local) and cloud tools. Go CDK's main focus is on portable APIs for cloud programming, targeting major cloud providers like AWS, GCP, and Azure, along with local (on-prem) implementations. The project enables developers to write application code once using these APIs, test locally, and then deploy to a cloud provider with minimal changes. The Go CDK is open-source and released under the Apache 2.0 License.

The Go CDK provides APIs like blob.Bucket or runtimevar.Variable as specific, concrete types rather than interfaces. This design distinguishes the generic logic from the specific interface. The generic logic resides in what we call the 'portable type,' whereas the 'driver' represents the interface. This is illustrated in figure 4.5.

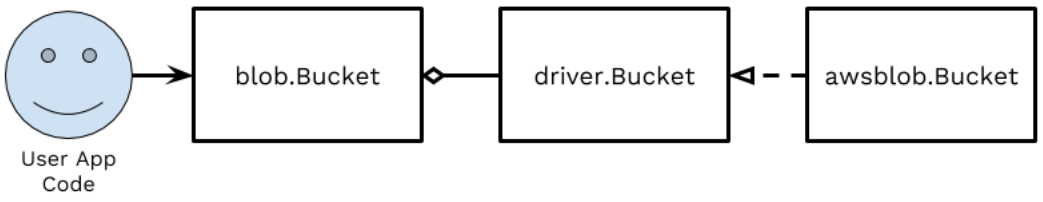


Figure 4.5. Go CDK API architecture.

This approach offers several advantages:

- The portable type can handle complex logic internally, simplifying the driver's interface. For example, in the blob service, the NewWriter method of the portable type can determine the content type before interacting with the driver.
- It allows for the addition of new methods to the portable type without affecting backward compatibility, unlike modifying an interface which would be a breaking change.
- The portable type can seamlessly integrate new operations introduced in the driver through optional interfaces, eliminating the need for the user to perform type assertions.

4.3.2 GORM

GORM[32] is a prominent ORM (Object-Relational Mapping) library for Go (Golang), designed to be developer-friendly. It offers a full-featured ORM system with capabilities such as associations (various relationship types), hooks (for create, save, update, delete, find operations), and eager loading using Preload and Joins. GORM supports transactions, nested transactions, context, prepared statement mode, and dry-run mode. It also provides functionality for batch insert, SQL building, upserts, locking, and auto migrations. The library includes a logger, extendable plugins, and is test-backed for each feature. Additionally, GORM supports composite primary keys, indexes, and constraints, emphasizing its flexibility and developer-friendly nature.

4.4 Serverless framework

The Serverless Framework[59] is a leading tool for deploying serverless architectures. Developed after the release of AWS Lambda in 2014, it enables building applications on cloud infrastructure that auto-scales and incurs no charges when idle. Key highlights include:

- Empowering developers to focus more on building and less on managing.
- Supporting a wide range of serverless use-cases.
- Automated deployment of both code and infrastructure.
- Simple syntax for deploying AWS Lambda functions without needing in-depth cloud expertise.
- Multi-language support.
- Full lifecycle management of serverless architecture.
- Built-in support for multiple stages and environments.
- Extensibility through plugins.

The Framework streamlines serverless application development, offering tools for building, deploying, updating, monitoring, and troubleshooting serverless architectures.

4.4.1 Pros and cons

The Serverless Framework offers a host of benefits for serverless architecture, such as facilitating more efficient building with less management, supporting a wide range of use-cases, automating code and infrastructure deployment, and providing simple syntax for safe AWS Lambda function deployment. It supports multiple languages and manages the full lifecycle of serverless architecture, accommodating large projects and teams with multi-domain and multi-environment support. The framework is also highly extensible through its plugin ecosystem. However, it presents challenges like a steep learning curve for newcomers, potential dependency issues, limited control over fine infrastructure details, complexities in handling large-scale projects, and possible performance bottlenecks. Also the variability in plugin reliability, the risk of vendor lock-in, and challenges in cost management are significant considerations.

4.4.2 Alternatives

Based on the image 4.6 taken from a Datadog research[21], it is evident that there are several alternatives to the Serverless Framework for deploying functions, and their adoption varies according to the size of the organization, measured by host count. In smaller companies with zero hosts, the Serverless Framework is the clear frontrunner, indicating its preference among startups or for small-scale projects. For mid-sized organizations, with

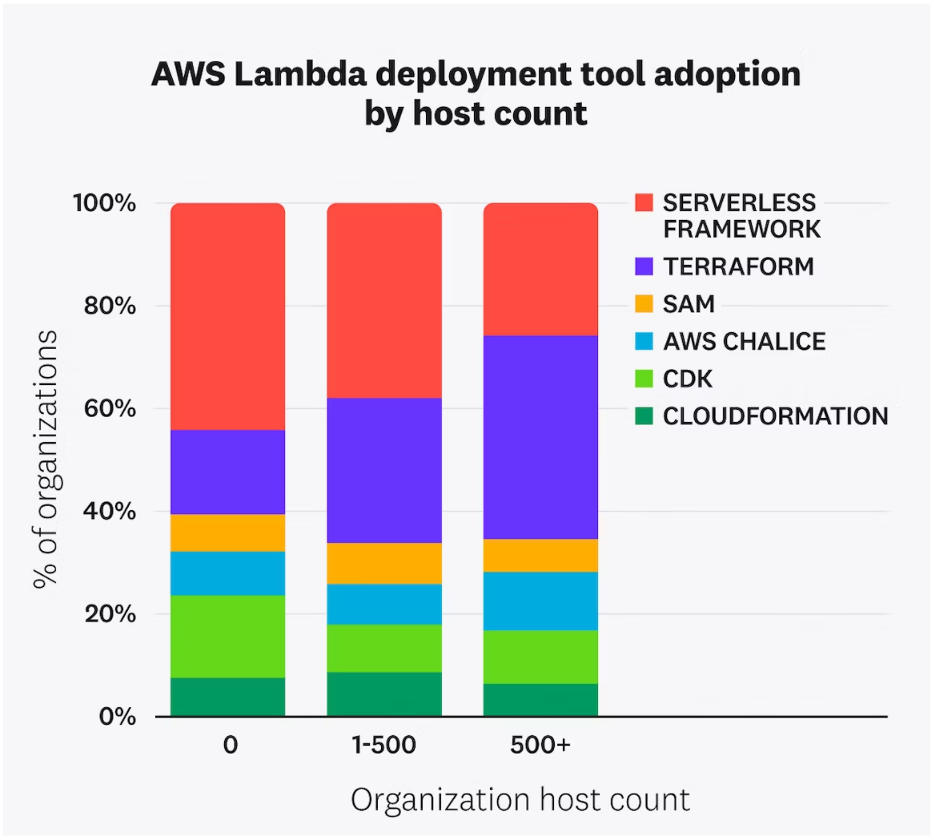


Figure 4.6. Serverless Framework alternatives.

host counts between 1 and 500, there is a more even split between the Serverless Framework and Terraform, showing that both tools are well-suited for medium-scale operations. However, in larger companies with over 500 hosts, Terraform emerges as the dominant tool. This trend suggests that Terraform's versatility, its support for multiple cloud providers, and its widespread adoption by DevOps teams make it the preferred choice for larger organizations that likely have more complex and diverse cloud infrastructures.

4.5 Flutter

Flutter[23] is an open-source UI software development kit created by Google. It's used for building natively compiled applications for mobile, web, and desktop from a single codebase. Flutter provides a fast development cycle with a "hot reload" feature that allows instant updates without losing the state of the app. It offers expressive and flexible UI with a rich set of widgets and a layered architecture that enables full customization. Flutter's native performance is achieved through the use of Dart, which compiles to ARM or JavaScript code. This toolkit is popular for its ability to create visually attractive and natively compiled applications across platforms efficiently.

4.5.1 Pros and cons

Flutter, as a framework for app development, offers several compelling advantages alongside a few drawbacks. Its ability to maintain consistency across multiple platforms with a single codebase streamlines development, while the stateful hot reload feature significantly boosts developer productivity. The framework's growing popularity is evident from its strong community support and open-source nature, which fosters continual improvement and innovation. Flutter's approach to app development with customizable widgets and excellent documentation facilitates faster and more flexible app creation. However, being a relatively new entrant in the cross-platform arena, it faces challenges such as limited learning resources, fewer plugins and packages compared to more established frameworks, and larger app sizes due to its use of built-in widgets. Additionally, Dart, the programming language for Flutter, has a smaller community, which might limit resources for learning and development. These factors make Flutter a powerful but nuanced choice for app developers, balancing its efficiency and ease of use against the considerations of newness and community size.

4.5.2 Architecture

As depicted in figure 4.7, Flutter is constructed as a modular, layered framework. It comprises a set of autonomous libraries where each is built upon the layer beneath it. There's no special access granted to any layer over the one it rests on, ensuring a democratic structure. Additionally, the framework is engineered to be adaptable, with each segment crafted to be optional and replaceable.

To the base operating system, Flutter apps are wrapped and delivered just like any native app. The platform-specific embedder acts as the initial entry point, interfacing with the OS for critical services such as rendering surfaces, accessibility, inputs, and managing the messaging event loop. This embedder layer is adaptable, written in the native languages of the platform such as Java and C++ for Android, Objective-C/Objective-C++ for iOS and macOS, C++ for Windows and Linux, and JavaScript for the web. Flutter's flexibility allows its code to be embedded within existing apps as a module, or to form the entirety of a new application, supported by a variety of embedders tailored for common target platforms, as well as third-party options.

Central to Flutter's functionality is the engine layer, primarily crafted in C++. This engine underpins every Flutter app by providing the essential components they require. It's tasked with rasterizing composited scenes for painting new frames and underlies the core API of Flutter, encompassing graphics (employing Impeller on iOS, soon on Android, and Skia elsewhere), text, file and network I/O, accessibility, plugin infrastructure, and the Dart runtime with compilation tools.

The bridge between the Flutter engine and the framework is dart:ui, which translates the engine's C++ capabilities into Dart classes. This critical library introduces base

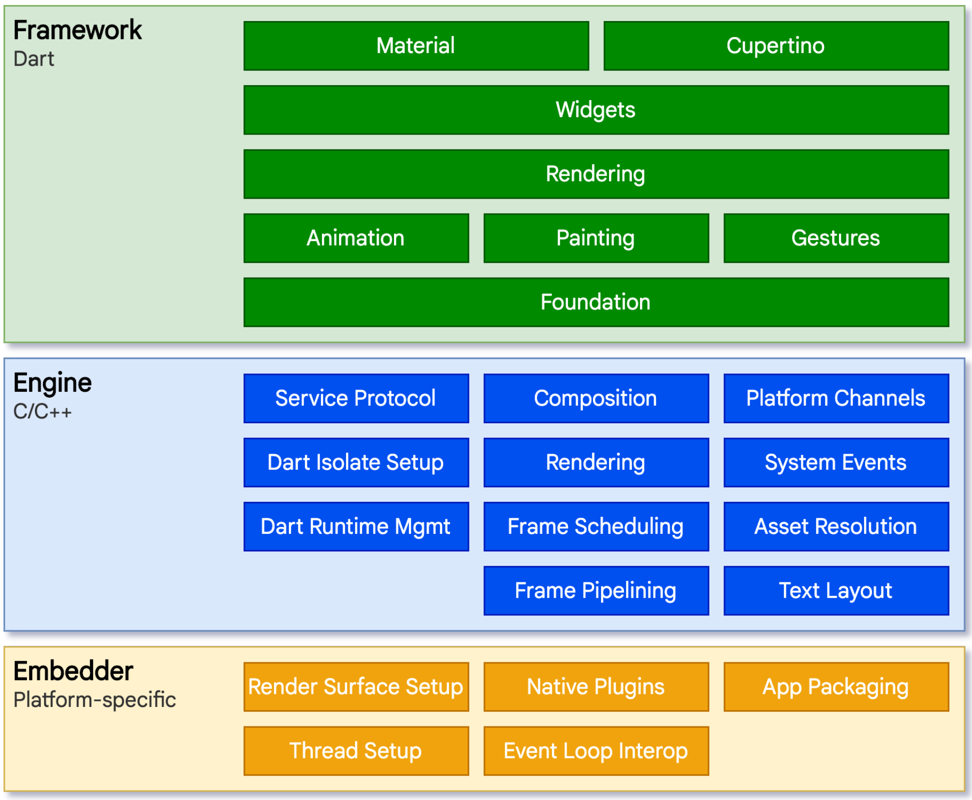


Figure 4.7. Flutter architecture.

primitives, enabling the manipulation of input, graphics, and text rendering. Although the core Flutter framework is compact, it is extendable, with numerous high-level functions available through additional packages, not unlike the independent libraries stacked in the image, which collectively form the robust and extensible framework that Flutter is known for.

4.5.3 Alternatives

Two major alternatives to Flutter for app development are React Native and Angular.

React Native[39], developed by Facebook, is an open-source framework tailored for building mobile applications. It allows developers to create natively rendered apps for both iOS and Android using React, a popular JavaScript library. This framework embraces a 'learn once, write anywhere' philosophy, enabling the use of a single codebase for multiple platforms while retaining the capability to include platform-specific features. Known for its time and resource efficiency, React Native can seamlessly integrate with existing native code, offering versatility in development. It features live reloading for immediate reflection of code changes, boosting developer productivity. With extensive community support, a wealth of libraries, and third-party plugin compatibility, React Native is ideal for developers aiming for efficient cross-platform development with a strong native performance and feel.

Angular[30], developed by Google, is a comprehensive solution for web application development. This platform and framework are geared towards building single-page client applications using HTML and TypeScript. Angular streamlines the development and testing process with tools for declarative templates, dependency injection, and integrated best practices. It features a two-way data binding, reducing the need for additional code and enhancing efficiency for interactive applications. Angular's architecture supports the rapid development of readable, maintainable, and testable code, making it suitable for enterprise-level and complex web projects that demand scalability and productivity. With its extensive libraries and strong community support, Angular is an excellent choice for creating dynamic, high-performance web applications.

Chapter 5

Kube Platform

In this chapter, we commence on an in-depth exploration of the Kube platform, an implementation of innovative ERP platform prototype. We start by outlining the fundamental requirements that have shaped the development of the Kube platform, highlighting the strategic objectives, technical needs, and business logic that inform its design. This leads us into a detailed examination of the platform's final architecture, which showcases a Microservices framework implemented through a Function as a Service (FaaS) model, using AWS services. We then transition to practical applications, where specific use cases demonstrate the platform's operational flow and its real-world applicability. The chapter ends with a focus on the client application, delving into the design and development of a user interface using Flutter, which not only complements the platform's robust backend but also enhances the overall user experience. Throughout this chapter, we aim to unravel the complexities of the Kube platform, illustrating its role as a transformative force in the field of enterprise resource planning and highlighting its potential.

5.1 Requirements

Before starting development, requirements are established to define the properties of the product. It's important that these requirements are thorough and coherent, covering all necessary features without conflicts or inconsistencies. However, creating these documents can be challenging and errors may occur, such as incomplete or ambiguous feature descriptions, redundancy, or important details being omitted. To address these challenges, software engineering techniques have been developed to formalize the requirements and minimize the occurrence of errors.

ISO/IEC 25010[34], an international standard, serves as a comprehensive framework for assessing software product quality. This standard enumerates several key quality characteristics, including functionality, reliability, usability, efficiency, maintainability, security, compatibility, and portability. Its widespread application in software engineering and quality assurance offers a standardized approach to evaluating and articulating software quality. This facilitates more informed decisions in software acquisition, development, and maintenance. ISO/IEC 25010 aids in identifying the actors involved and delineating both functional and non-functional requirements.

5.1.1 Stakeholders

A stakeholder refers to any role, person, group, or organization that has an interest in a software project or system being developed. This could include end-users, customers, investors, project managers, developers and other individuals or groups involved in the development, deployment, and maintenance of the software. Identifying all relevant stakeholders is important for considering diverse perspectives and generating relevant requirements for the system. As shown in Table 5.1, numerous stakeholders play a role in the process.

Stakeholder	**Description**
End-users	These are the people who will use the ERP system in their day-to-day work. They may include employees from various departments within the organization.
Developers	These are the individuals responsible for creating the software code that makes up the ERP system.
Admin/IT staff	These are the individuals responsible for installing, configuring, and maintaining the ERP system.
Customers	These are the organizations or businesses that are purchasing the ERP system. They have a vested interest in ensuring the system meets their needs and requirements.
Vendors	These are the organizations that provide the ERP software and related services, such as installation, configuration, and support.
Cloud Vendors	These hosts the system and provide the necessary infrastructure for its operation. They are responsible for system availability, scalability, and security.

Table 5.1. Stakeholders of a Cloud ERP System.

5.1.2 Functional and Non-functional

Functional requirements and non-functional requirements are two types of requirements that are used to specify what a system or software application should do and how it should perform. They are important for the successful development and implementation of a system or software application. The functional requirements ensure that the software application meets the needs of its users, while the non-functional requirements ensure that the system is reliable, efficient, and secure.

Functional requirements

Functional requirements describe what the system should do in terms of its functionalities, features, and capabilities. They define the specific tasks that the software application should be able to perform to meet the needs of its users. For distinguish one requirement from another it is important to assign for each functionality an ID, in order to easy identify it and trace throughout the life cycle of the project (Table 5.2).

ID	Description
FR1	Sign-up users by email and password
FR2	Login users by email and password
FR3	Logout users
FR4	Activate notifications
FR5	Deactivate notifications
FR6	View chart for andamento mensile degli ordini
FR7	View chart for distribuzione degli ordini per cliente
FR8	Customize the menu bar
FR9	View the log of the platform events
FR9.1	View the detailed log of an event
FR9.2	Delete a log event
FR10	View the customers list
FR10.1	View the detailed customer
FR10.2	Insert a customer
FR10.3	Update a customer
FR10.4	Delete a customer
FR11	View the sales order list
FR11.1	View the detailed sales order with sales lines
FR11.2	Insert a sales order
FR11.3	Update a sales order
FR11.4	Delete a sales order
FR11.5	Insert a sales order line
FR11.6	Update a sales order line
FR11.7	Delete a sales order line
FR12	Launch the posting order event
FR13	show notifications of change status on order
FR14	View the shipment list
FR14.1	View the detailed shipment with sales lines
FR14.2	Delete a shipment
FR15	View the invoice list
FR15.1	View the detailed invoice with sales lines
FR15.2	Delete an invoice

Table 5.2. Functional requirements for the platform.

Non-functional requirements

Non-functional requirements describe how the system should perform in terms of its functionality, reliability, usability, efficiency, maintainability, security, compatibility, and portability, all aspects that are not directly related to the specific functionalities to be implemented. They refer to operating methods and constraints, such as response times, supported platforms, choice of languages, required resources, tools and various implementation techniques. They must be measurable and may be more critical than functional requirements. They are identified with a unique code and it is also necessary to specify their type associated to the ISO properties and which functional requirements they refer to (Table 5.3).

ID	Type	Description
NFR1	Usabilty	Application should be used with no training by any user
NFR2	Efficiency	All functions should complete in < 0.5 sec
NFR2	Efficiency	All functions must optimize the resource utilization
NFR3	Portability	System must work on Chrome, Firefox, Safari, Edge, Android and iOS
NFR4	Portability	No installation is needed
NFR5	Compatibility	Platform must be compatible with Azure and AWS cloud
NFR6	Security	All data must be stored in a secure database
NFR7	Maintainability	All functionalities must be tested indipendently
NFR8	Reliabilty	Downtime allowed is of one hour per year

Table 5.3. Non-functional requirements for the platform.

5.2 Server application

In this section, we will discuss the final architecture designed for the Kube platform, encompassing its data structures and services. We will explore the database setup, highlighting the tables crafted for each microservice. Following this, we'll shed light on the REST APIs tailored for every microservice, along with the integration of queues and events designed for asynchronous event management. Concluding our discussion, we'll outline a SAGA implementation that has been incorporated into the platform.

5.2.1 Final Architecture

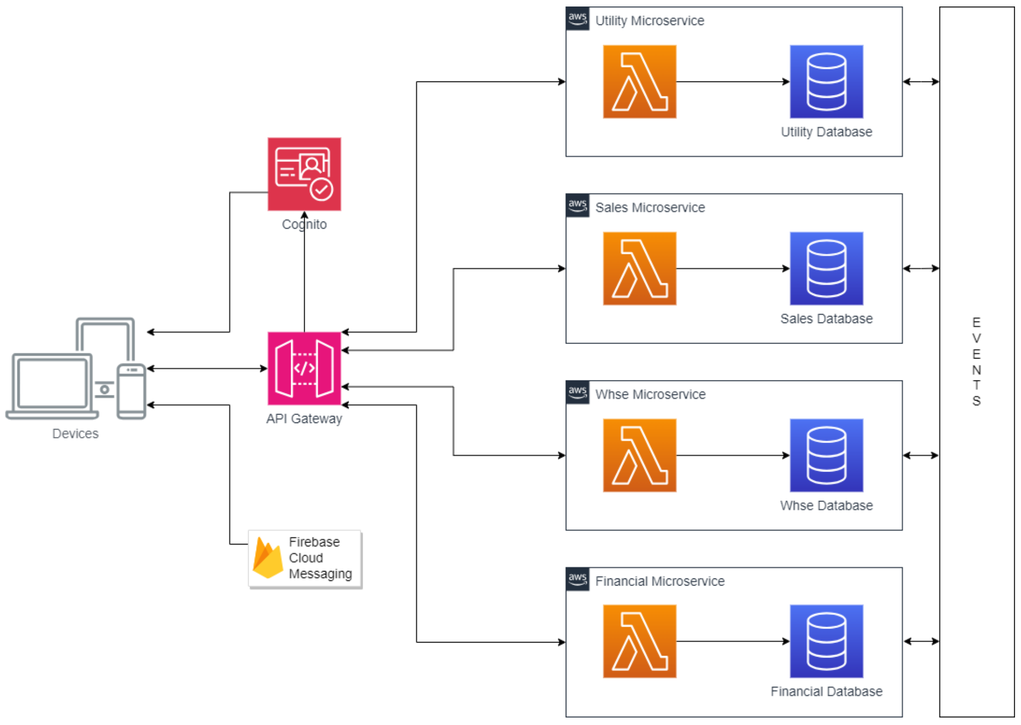


Figure 5.1. Final architecture of the platform.

The architecture diagram in Figure 5.1 showcases a modern, serverless microservices-based architecture for the Kube system. Here's a description based on the elements and their interconnections:

- **User Authentication**: the service which is responsible for user authentication is 'Cognito'. It is the entry point for security, ensuring that only authenticated users can interact with the platform and with all the API.
- **Firebase Cloud Messaging**: At the bottom of the diagram, we can see 'Firebase Cloud Messaging', it is an integration with a cloud solution for sending notifications to devices, allowing for real-time user engagement.
- **API Gateway**: The 'API Gateway' is the central hub through which all client requests pass. It acts as a front door, directing incoming requests from various devices (such as computers and mobile phones) to the appropriate microservices.
- **Microservices Architecture**: Actually each microservice is deployed to AWS (Amazon Web Services), and utilize various AWS serverless features for optimizing performance.

  - **Utility Microservice**: This service handles log functions and events, and is backed by a 'Utility Database' for storing log records. It also manage home page graph function and data and navigations/menu API and data.
  - **Sales Microservice**: Dedicated to handling sales-related operations, like handle customer and sales order, this service interacts with a 'Sales Database'.
  - **Whse Microservice**: this service manages shipment and warehouse-related data, backed by its own 'Whse Database'.
  - **Financial Microservice**: This handles financial transactions and invoice, with a separate 'Financial Database' for storing related records.

- **Database Pattern**: The architecture uses a database-per-microservice pattern with SQL databases, ensuring that each service has its own datastore, thus maintaining database isolation and decoupling services.

- **Event-Driven Architecture**: all microservices are connected through an event-driven architecture, which allows them to communicate asynchronously and facilitates loose coupling. This architecture is implemented using Amazon Simple Queue Service (SQS) and Amazon Simple Notification Service (SNS), which are managed message queues and notification services.

All services are managed and deployed using the serverless framework, which allows to easily integrate a CI/CD pipeline manage the entire infrastructure as code. With this setup, the platform can be easily deployed to other cloud providers and add other cloud services and features. How we can see, the Kube platform's architecture is designed to be scalable, flexible, and maintainable, with a focus on modern cloud-native principles and best practices for microservices development.

Microservices components

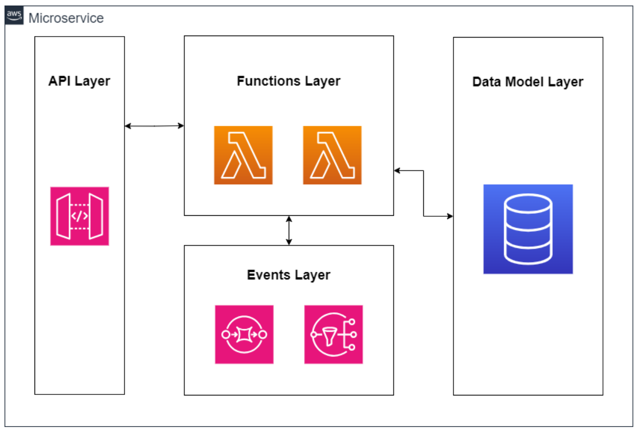


Figure 5.2. Microservices components.

Each microservice in the Kube platform is a composite of four essential components that work in concert to deliver its specific functionality:

Component	Service	Description
Data Models	RDS	At the core of each microservice is a set of data models. These models define the structure of the data that the microservice handles, ensuring data integrity and consistency. By having its own models, each microservice encapsulates the necessary information to perform its tasks, leading to a clear delineation of responsibilities within the system.
REST APIs	API Gateway	The REST APIs serve as the interfaces through which external services or client applications interact with the microservices. These APIs are carefully designed to provide a clear and consistent contract for accessing and manipulating data. They follow REST principles, allowing for stateless communication and enabling clients to perform standard HTTP operations such as GET, POST, PATCH, and DELETE.
Functions	Lambda	The functions are the operational units within each microservice, containing the business logic that processes requests, manipulates data models, and performs the necessary computations. These functions are likely implemented as serverless functions, which are executed in response to events, scaling automatically with the number of requests and reducing the need for managing server infrastructure.
Events	SNS, SQS	Each microservice also incorporates an event-driven mechanism, signaling and reacting to various conditions and triggers. These events facilitate asynchronous communication between microservices, thereby enhancing the platform's responsiveness and efficiency. By decoupling microservices through events, the system can better handle load variations and failure modes, contributing to overall resilience.

Table 5.4. Microservices components.

Together, these components create a modular and cohesive microservice that is self-contained, scalable, and robust. The data models ensure that each service can independently manage its segment of the data. The REST APIs provide the necessary endpoints for interaction, while the functions encapsulate the business logic. Finally, the event system allows the services to react to and communicate changes across the platform without direct coupling, promoting a reactive architecture that can quickly adapt to changing conditions.

5.2.2 Code Structure

In our microservices architecture, all lambda functions are written in the Go language. This decision was influenced by several key factors. Firstly, Go is a compiled language, which leads to significantly faster execution times and results in smaller executable files. Another important reason for choosing Go is the availability of the Go CDK library. This library is unique to Go and aids in making functions as portable as possible across different cloud providers. While it's true that shifting functions between providers requires some manual adjustments, the process could be streamlined by developing CLI tools for automation.

However, adopting Go was not without its challenges. Unlike object-oriented languages, Go doesn't fully integrate all object-oriented principles. It has its own way of handling certain programming concepts, which required a learning curve. Additionally, Go is not as versatile in some aspects; for instance, the main function is bound to have a dedicated 'main' package. This can pose difficulties in managing multiple functions, each with its own 'main,' making the overall management a complex task.

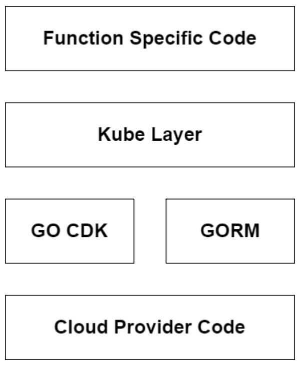


Figure 5.3. Stack abstractions layer.

In every function architecture, I've implemented an abstraction layer specifically designed for managing REST APIs of various resources. This unified approach, applied across all microservices, is facilitated by a layer we've named 'kube'. Built on top of the GORM and Go CDK libraries, this layer simplifies and accelerates API management. Connecting a GORM model to a 'kube' layer structure called 'page' requires minimal coding. The 'page' concept is central to our approach: it automatically generates a suite of REST APIs for a resource, significantly reducing development time and standardizing structures across resources. Moreover, these 'pages' can define a UI schema, which clients can use to dynamically build pages for resources in their applications. The APIs that are automatically constructed from a GORM model are structured as indicated inside the following tables.

http://api.example.com/:pageid?field=value

Method	Description
GET	Returns the list of records linked to the page. The records can be filtered by inserting the fields to be filtered in the URL's query string.
POST	Creates a new record in the table linked to the page.
PATCH	Edits the record in the table linked to the page.
DELETE	Deletes records linked to the page. The records can be filtered by inserting the fields to be filtered in the URL's query string.

Table 5.5. REST API for page.

http://api.example.com/:pageid/schema

Method	Description
GET	Returns the UI schema for building the page in the client.
POST	Not allowed.
PATCH	Not allowed.
DELETE	Not allowed.

Table 5.6. Methods for page schema.

http://api.example.com/:pageid/button?button_id=value

Method	Description
GET	Initiates the functionality linked to the button and returns the result.
POST	Not allowed.
PATCH	Not allowed.
DELETE	Not allowed.

Table 5.7. Methods for page buttons.

5.2.3 Utility Microservice

This microservice is responsible for managing the home page graph data, the navigation menu, and the logging system. It is the first microservice to be deployed, as it have the entry points of the client application. In this microservice there are three crucial componentes of the platform: the home page, the logging system and the navigation functionalities.

Home Page

The home page is the first page that the user see when he log in the client application. It is composed by two graphs, one for the monthly trend of the orders and one for the distribution of the orders by customer. The data of these graphs are stored in the database of the microservice and are updated by the final event of the posting order chain, it call the graph update function. The home page is implemented by a page model, so through a Lambda function, which is triggered by HTTP API by the client. The Lambda function is responsible for updating the home page table.

Logging System

The logging component allows for monitoring and debugging the system. It is implemented through a FIFO Queue, using Amazon SQS, and a Lambda function, which is triggered by the queue. The Lambda function is responsible for writing the log to the database of the microservice, inside the Log tabel. Thanks to the thesis layer in the framework stack, a message to the logging queue is automatic sended when an error occurs in the platform (even in other microservices).

Navigation

The navigation component is responsible for managing the menu bar of the client application. It have a list of all the page ids that most be present in the client menu bar. This list is stored in the table navigation and is updated by the client application when the user customize the menu bar. It not have only ids information but also the order of the pages in the menu bar, the icon to show, the caption name to show and the entry point of the page. The entry point is the url of the page that the client application must call when the user click on the page in the menu bar, and it depends on the microservice that manage the page. The navigation component is implemented by a page model, so through a Lambda function, which is triggered by an HTTP API by the client. The Lambda function is responsible for updating the navigation table.

Functions

In the table 5.8 are summarized all the lambda functions of the utility microservice, with the relative description and the trigger event (type and name).

Name	Type	Event Name	Description
Home	API	/home	handle home page GET request
GraphUpdate	SNS	OnFinishPostOrder	Update graph data for home page
NavigationList	API	/navigationlist	represents the list page of navigation model
NavigationCard	API	/navigationcard	represents the detailed page of navigation model
LogList	API	/loglist	represents the list page of log model
LogCard	API	/logcard	represents the detailed page of log model
LogMessage	SQS	LogMessageQueue	for logging the message in RDS database

Table 5.8. Functions of utility microservice.

5.2.4 Sales Microservice

This microservice is specifically designed to handle and manage customer and sales order data. It includes dedicated page functionalities that allow for efficient management and access to both customer and sales order information. The data related to these entity are securely stored in a specialized database, named 'sales', which is an integral part of this microservice. Additionally, this service is equipped with a lambda function that plays a crucial role in orchestrating the saga of posting orders. This function ensures that the process of managing and posting sales orders is conducted smoothly and effectively.

Functions

In the table 5.9 are summarized all the lambda functions of the sales microservice, with the relative description and the trigger event (type and name).

5.2.5 Whse and Financial Microservice

These two microservices are tailor-made to manage shipments and invoices, each equipped with its respective page model and database. They incorporate lambda functions integral to the posting order saga. These functions are tasked with creating shipments and invoices, as well as updating the status of sales orders.

Functions

In the table 5.10 are summarized all the lambda functions of the Warehouse and Financial microservice, with the relative description and the trigger event (type and name).

Name	Type	Event Name	Description
CustomerList	API	/customerlist	represents the list page of customer model
CustomerCard	API	/customercard	represents the detailed page of customer model
SalesOrderList	API	/salesorderlist	represents the list page of sales order model
SalesOrderCard	API	/salesordercard	represents the detailed page of sales order model, it has the button for start the posting process
SalesOrderLineList	API	/salesorderlinelist	represents the list page of sales order lines
SalesOrderLineCard	API	/salesorderlinecard	represents the detailed page of sales order lines
ChangeOrderStatus	SQS	ChangeOrderStatusQueue	the function that orchestrates the saga

Table 5.9. Functions of sales microservice.

Name	Type	Event Name	Description
ShipmentList	API	/shipmentlist	represents the list page of shipment model
ShipmentCard	API	/shipmentcard	represents the detailed page of shipment model
PostShipment	SNS	OnPostShipment	the function that create shipment and recall the change order status
InvoiceList	API	/invoicelist	represents the list page of invoice model
InvoiceCard	API	/invoicecard	represents the detailed page of invoice model
PostInvoice	SNS	OnPostInvoice	the function that create invoice and recall the change order status

Table 5.10. Functions of Whse. and Financial microservices.

5.2.6 Posting Order Saga

The posting order saga, designed for users to post sales orders, operates through a series of orchestrated lambda functions. At its core is the ’ChangeOrderStatus’ function from the sales microservice, which manages the sequence of steps in the saga. This function is triggered by an SQS queue, which initially receives data when a user clicks the ’Post Order’ button via the ’SalesOrderCard’ function. Once activated, the ’ChangeOrderStatus’ function begins the saga, leading to a sequence of lambda functions, each set off by an

SNS event. Key functions in this saga include 'CreateShipment' from the warehouse microservice and 'CreateInvoice' from the financial microservice. This well-organized method ensures a seamless and efficient process for posting orders.

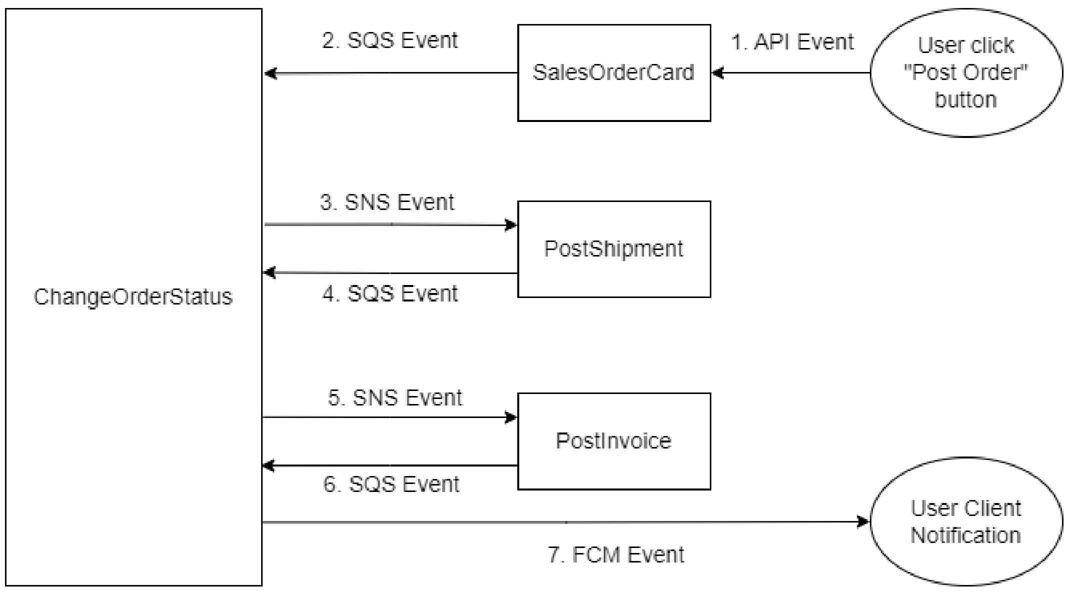


Figure 5.4. Sequence diagram of the posting order saga.

The figure 5.4 shows the sequence diagram of the posting order saga. Here's a detailed description of the steps involved:

1. User initiates the process by clicking the "Post Order" button, which triggers an API event which is processed by the SalesOrderCard function.
2. The SalesOrderCard places a message in an SQS queue, the ChangeOrderStatusQueue. It is a FIFO queue, ensuring that messages are delivered to the recipient in the same order they were received by the queue.
3. ChangeOrderStatus make the check of posting order, change the status of order in "Pending Shipment" and triggers an SNS event on topic OnPostShipment.
4. The SNS event invokes the PostShipment function. It create the shipment and send an SQS message on ChangeOrderStatusQueue.
5. ChangeOrderStatus change the status of order in "Shipped" and triggers an SNS event on topic OnPostInvoice.
6. The SNS event invokes the PostInvoice function. It create the Invoice and send an SQS message on ChangeOrderStatusQueue.
7. Upon successful invoice creation, the status of order is changed in "Invoiced" and a final Firebase (FCM) event is triggered, sending a notification to the user client.

5.3 Client Application

The Kube application client, a multifaceted and versatile platform, is constructed using Flutter, a well-known cross-platform framework. This comprehensive solution seamlessly supplies a wide array of platforms including Android, iOS, and Web, while also extending support to desktop environments like Windows, MacOS, and Linux, albeit with a current limitation in notification capabilities. At its core, the application smoothly integrates various robust systems and components, each playing a pivotal role in its functionality and are described in the following sections. These integral elements include a secure authentication system featuring a Cognito login page, an efficient state management setup utilizing the Provider package, a dynamic routing mechanism powered by the GoRouter package, a custom-made page builder tailored to specific needs, and a reliable notification system implemented through Firebase Cloud Messaging, Figure 5.5.

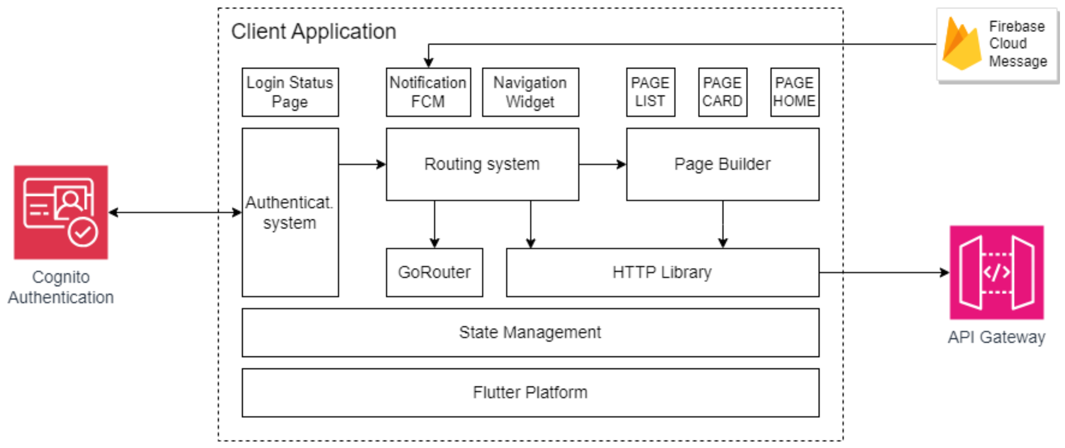


Figure 5.5. Client application architecture.

From an UI point of view, the application is designed to be simple and intuitive, with a clean and modern look. Visually it is mainly composed of a menu bar and a page area, where the content of the selected page is displayed. The Figure 5.6 shows what the application looks like.

To enhance its effectiveness, the client application is hosted on GitHub Pages and is seamlessly woven into a CI/CD pipeline orchestrated via GitHub Actions. This strategic implementation ensures a streamlined deployment process: each commit to the repository triggers an automated build and deployment sequence. This diligent approach guarantees that the latest iteration of the web application is consistently accessible, embodying the epitome of efficiency and continuous integration.

5.3.1 State Management

State management in the Flutter framework refers to the process of handling the data and UI states of an app effectively. In Flutter, "state" can be anything that affects the

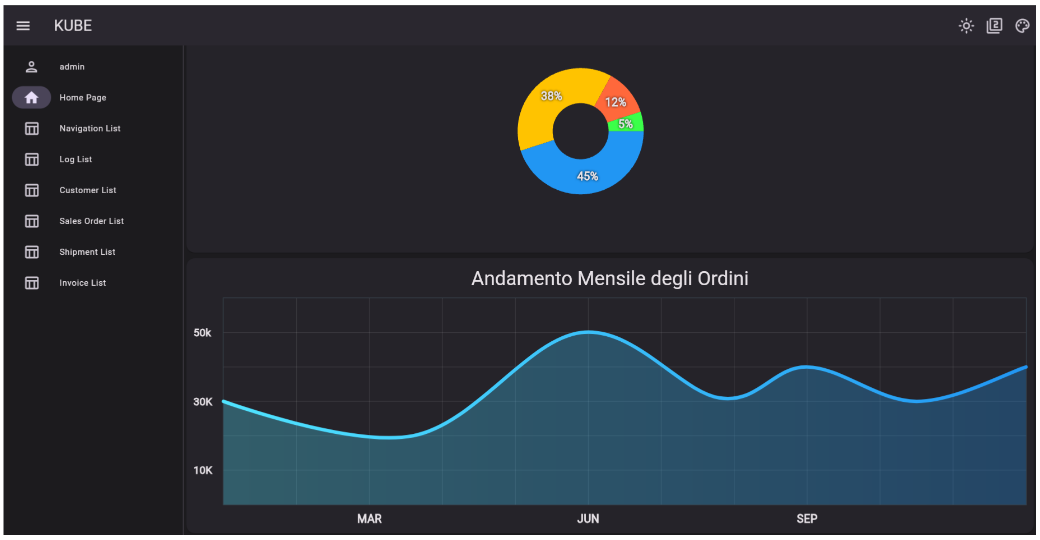


Figure 5.6. Client UI.

UI and needs to be tracked, such as user inputs, server responses, or even a timer's progress. Proper state management ensures that the UI reflects the current state of the app accurately and efficiently, without unnecessary rebuilds or updates. For the Kube application, the state is managed by the Provider package. it is a popular library and it give an efficient way to manage state. It works by using a mix of dependency injection and state management, allowing widgets to subscribe to changes in the state of the app.

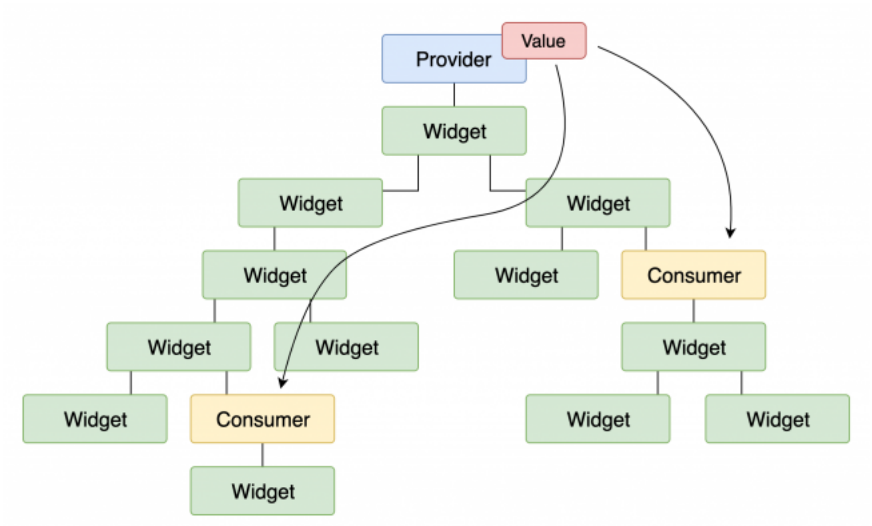
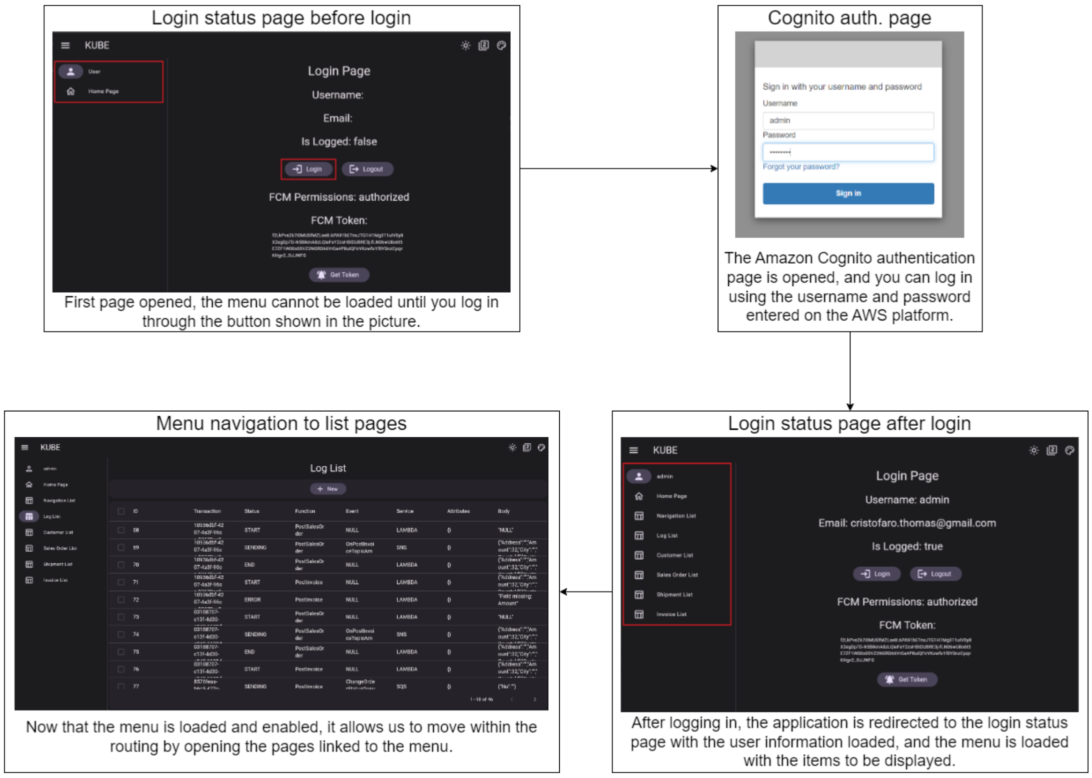


Figure 5.7. Provider package.

Provider simplifies the process of passing data and events down the widget tree. This approach not only makes the code more maintainable and scalable but also optimizes the app's performance by rebuilding only those widgets that need to be updated. The figure 5.7 shows how the Provider package works. In this project, the Provider system is utilized for managing various states within the application. It handles the user's authentication state, controls the navigation state of the menu bar, and manages the notification state of the application, and finally it is used for handling the controller page between several widgets on a single page.

5.3.2 Authentication System

The authentication system is a crucial component of the Kube application, ensuring that only authenticated users can access the platform. This system is implemented using Amazon Cognito, a comprehensive authentication service that provides secure user sign-up and sign-in functionality. Cognito is a fully managed service, which means that it handles all the authentication processes, including user registration, authentication, and account recovery. In this platform, the sign-up process is exclusively handled by the administrator, meaning that users are unable to register themselves. Instead, users can only sign in using credentials provided by the administrator.



First page opened, the menu cannot be loaded until you log in through the button shown in the picture.

The Amazon Cognito authentication page is opened, and you can log in using the username and password entered on the AWS platform.

After logging in, the application is redirected to the login status page with the user information loaded, and the menu is loaded with the items to be displayed.

Now that the menu is loaded and enabled, it allows us to move within the routing by opening the pages linked to the menu.

Figure 5.8. Authentication system.

The authentication is facilitated through a Cognito login page, which is accessible via the app's login page status - serving as the application's entry point. This process is illustrated in Figure 5.8. Once authenticated, users gain access to the menu bar and the various pages of the application, thereby unlocking the routing system which was previously restricted.

5.3.3 Routing System

Kube application's routing system, an important aspect for page navigation, is powered by the GoRouter package. This dynamic routing tool is both user-friendly and highly adaptable, supporting custom routes and parameterized paths. Integrated state management ensures navigation state updates in response to route changes, allowing the app to respond dynamically. The app features three primary routes: login, home, and page, as shown in Table 5.11.

Route	**Page**	**Description**
/login	LoginStatus	go to the page where are shown login information and login and logout buttons
/home	Home	call the home API to retrieve page home information e show it
/page/:pageId	Page Selected	open the page indicated in pageId parameter. Call the page API related and build the page based on schema

Table 5.11. Routes and pages.

These routes are established in the main.dart file, the application's starting point. Positioned under the navigation menu bar, which remains constantly visible, all routes are constructed accordingly. Post-login, the first constructed widget is the menu bar, set up by querying the navigation list API. This API, a GET request to the navigationlist function in the utility microservice, retrieves a list of all pages to be displayed in the menu bar, with the relative icon, caption and url. When a user interacts with the menu bar by clicking on its buttons, it initiates a change in the application's navigation state. This action prompts the menu bar to be reconstructed. In the menu, each button corresponds to a different page within the application. Therefore, selecting a button results in a change to the respective page route, which in turn triggers the refreshing of that page.

5.3.4 Page Builder

The Page Widget in our application is a custom-built, dynamic component that creates pages according to a JSON schema fetched from the page API. Its design is intentionally flexible, allowing for the construction of a variety of page types. The widget uses a switch statement to decide the type of page to construct, based on the 'page type' defined in the schema. Currently, it supports three types of pages: list, card, and home. The Figure 5.9 is how the page builder workflow operates:

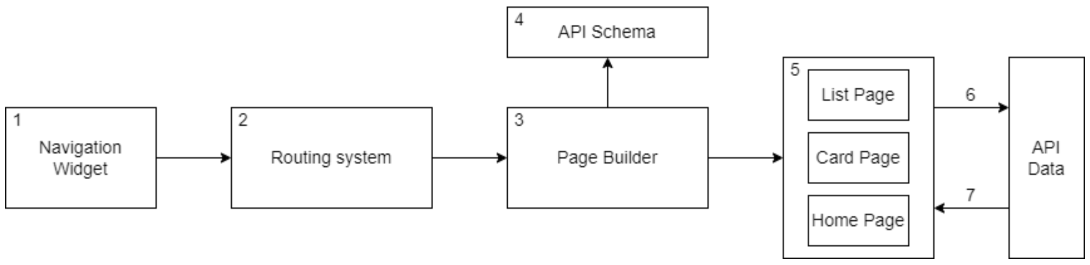


Figure 5.9. Page builder workflow.

1. The User clicks on a menu bar button, triggering a change in the navigation state.
2. The menu bar call the routing system to change the page with the pageId of the selected button.
3. The Page Widget is initiated with a specific 'pageId' parameter.
4. It then calls the page API to obtain layout information using the endpoint /:pageId/schema. This schema, a JSON format, outlines the structure of the page.
5. Based on this layout information, the widget builds the page. There are three distinct page types it can construct: list, card, and home. For each page type, a specialized widget is employed to build the page: PageList for lists, PageCard for cards, and PageHome for the home page.
6. These widgets then call their respective APIs, using '/:pageId' endpoint, to fetch the data needed for display. For the PageCard, a key filter is applied to retrieve specific records.
7. After receiving the data from the API, each widget finish to constructs the page, populating it with the relevant data.

5.3.5 Notification System

In Kube platform, the notification system is implemented through Firebase Cloud Messaging (FCM), a robust cross-platform messaging solution that ensures the reliable delivery of messages and notifications to client applications. FCM, a free service, supports messaging to devices using iOS, Android, and Web applications, although it does not extend to desktop applications. I have seamlessly integrated FCM into the global state management system of our application. This integration simplifies accessing and managing notification states, enabling the client application to receive notifications effortlessly. Each notification is displayed as a snackbar at the bottom of the screen, effectively informing users about important events on the platform, such as status updates on orders or the creation of new shipments. This feature enhances user engagement by keeping them promptly informed.

5.4 Use Case

A use case comprises various scenarios connected by a shared user objective, serving to illustrate how a system behaves under different circumstances when processing a request. Each use case should specify the system as an opaque entity, the user type (often referred to as the actor) interacting with the system, and the actor's functional objective achieved through the system. A single scenario represents a series of steps outlining the interaction between a user and the system. Additionally, each scenario requires a pre-condition that must be met before initiation and a post-condition fulfilled upon completion. This section will present several use cases, demonstrating the range of operations a user can execute on the platform.

5.4.1 User interaction

This use case is applicable to all entities of the platform that have a page list, so it is a generic use case. In this example we will see how the user can interact with the customer list page. We use the insert example, but the same steps are valid for the update and delete operations. The table 5.12 shows the steps of the scenario.

Actors involved	User
Precondition	The user U has already authenticated to the system and opened the customer list
Post-condition	The user U has created a new customer for the ERP system
Normal scenario	1. The user U clicks on the new button 2. Insert valid data in the opened detailed page 3. Click the button "Insert" for creating the new customer
Variant	The user inserts a non-valid data and the system inform him with an error message

Table 5.12. User interaction scenario

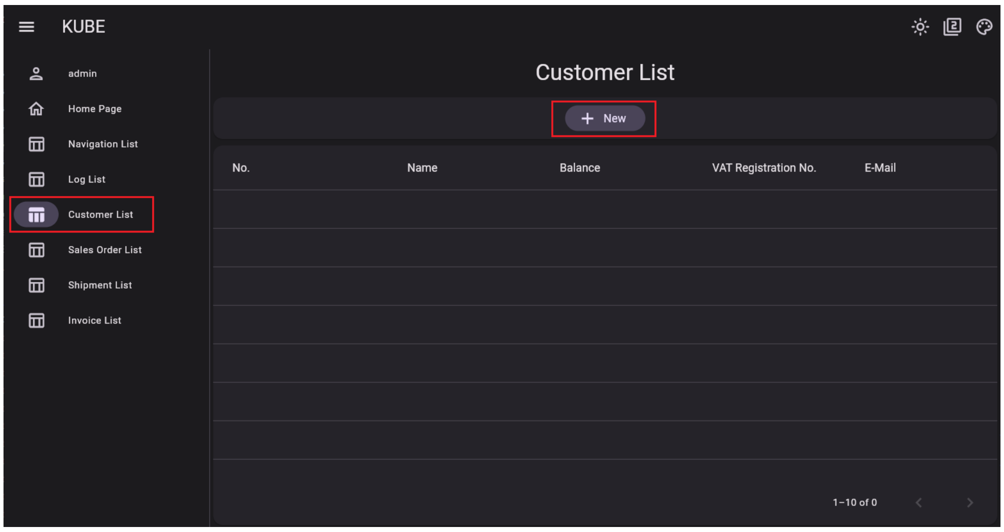
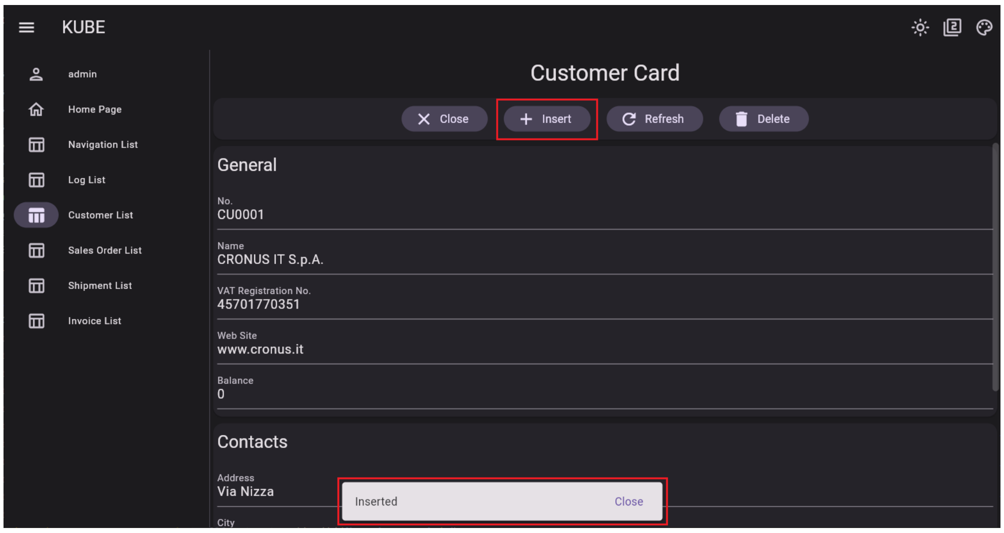


Figure 5.10. New button on customer list page.



No.
CU0001

Name
CRONUS IT S.p.A.

VAT Registration No.
45701770351

Web Site
www.cronus.it

Balance
0

Address
Via Nizza

City

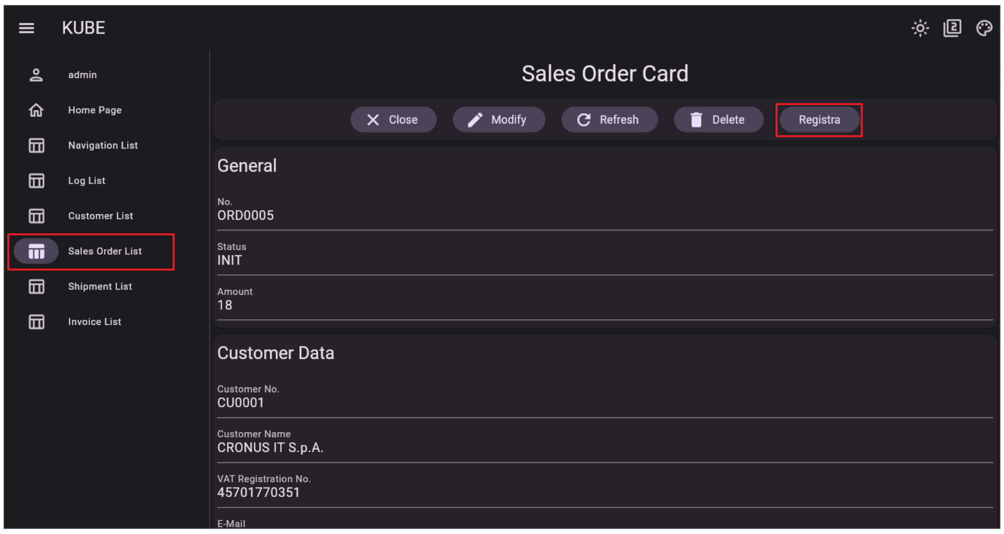
Figure 5.11. Insert Customer from detailed page.

5.4.2 Posting order

This use case show how the user can post an order. The table 5.13 shows the steps of the scenario.

Actors involved	User
Precondition	The user U has already authenticated to the system and opened the sales order card
Post-condition	The user U has posted a sales order, and created a shipment and invoice document
Normal scenario	1. The user U clicks on the Posting button 2. The system check if the order is valid 3. The system create the shipment document 4. The system create the invoice document 5. The system change the status of the order in "INV" 6. The system send a notification to the user U with the result of the posting order
Variant	The user inserts a non-valid data and the system inform him with an error notification

Table 5.13. User interaction scenario



Sales Order Card

General

No.
ORD0005

Status
INIT

Amount
18

Customer Data

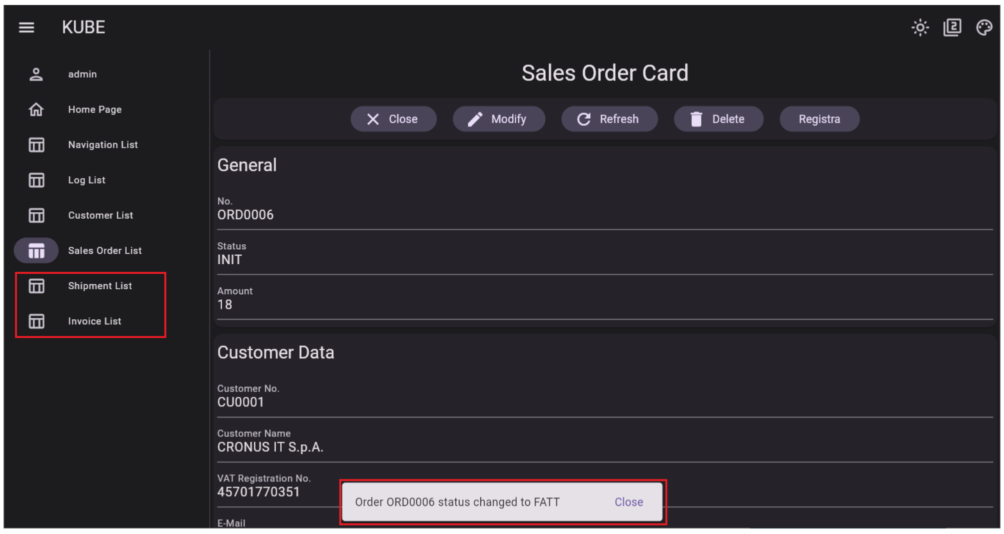
Customer No.
CU0001

Customer Name
CRONUS IT S.p.A.

VAT Registration No.
45701770351

E-Mail

Figure 5.12. Posting order button on sales order card.



No.
ORD0006

Status
INIT

Amount
18

Customer No.
CU0001

Customer Name
CRONUS IT S.p.A.

VAT Registration No.
45701770351

Order ORD0006 status changed to FATT

Figure 5.13. Posting order result notification.

Chapter 6

Conclusions and future works

As outlined in the introduction, this thesis sets out to develop a platform that handles key operations of an Enterprise Resource Planning system while incorporating microservices and serverless computing design patterns, ensuring accessibility across various platforms. With the completion of the platform's initial version, we have established a technical foundation that supports the ongoing development of diverse modules and functionalities. The platform is designed for easy extensibility and scalability, making it adaptable to a range of needs and requirements. Currently, the platform can be deployed on an AWS tenant using the Serverless Framework. Once user settings are configured, it becomes operational. The platform supports all CRUD (Create, Read, Update, Delete) operations for its main entities, such as customers, orders, and shipments. It features a dashboard displaying key statistics and allows for the posting of operations on orders. This triggers asynchronous processes like the creation of new shipments and invoicing for customers. Technically, the chosen technologies lay a valid foundation for future enhancements and the distribution of the platform across various cloud infrastructures. Despite encountering several challenges during development, the thesis reaffirms the belief in the future of platforms like Function as a Service, which streamline development by abstracting many underlying details. While FaaS may not be suitable for all applications due to certain constraints, it offers significant benefits for compatible systems, marking a progressive step in software development.

Looking ahead, there are multiple areas for future development of the platform. Building on its current capabilities, a wide array of ERP functionalities could be integrated, including warehouse management, production, accounting, and human resources management. Additionally, the introduction of a Command Line Interface (CLI) could offer a streamlined way to interact with the platform's code, enabling the automatic conversion of YAML files into templates for various cloud providers. Another promising avenue for expansion is the incorporation of a NoSQL database like AWS DynamoDB or Azure CosmosDB. This would allow for data storage in a truly serverless mode, enhancing the platform's serverless capabilities. Furthermore, the microservices architecture of the platform lends itself well to the integration of a Machine Learning (ML) module. Such a module could analyze platform data to provide valuable insights, such as identifying the best-selling products, top customers, and leading suppliers.

An additional area of future research could focus on the environmental implications of serverless computing. In the context of growing concerns about energy consumption in data centers, serverless computing, with its features like pay-per-use, auto-scaling, and multi-tenancy, could offer a more sustainable solution. This aspect of serverless computing is particularly relevant in today's world and warrants further exploration to understand its potential in reducing the environmental footprint of data centers.

Bibliography

[1] Joe Weinman Adam Eivy. Be wary of the economics of "serverless" cloud computing. *IEEE Cloud Computing*, 2017.

[2] adservio. Event-driven architecture, 2022. URL https://www.adservio.fr/post/event-driven-architecture.

[3] Amazon. What is cloud computing?, 2023. URL https://aws.amazon.com/what-is-cloud-computing/?nc1=h_ls.

[4] Amazon. What is an event-driven architecture?, 2023. URL https://aws.amazon.com/event-driven-architecture/?nc1=h_ls.

[5] Amazon. Serverless on aws, 2023. URL https://aws.amazon.com/serverless/?nc1=h_ls.

[6] Amazon. What is aws?, 2023. URL https://aws.amazon.com/what-is-aws/.

[7] Amazon. Amazon api gateway, 2023. URL https://aws.amazon.com/api-gateway/.

[8] Amazon. Amazon lambda, 2023. URL https://aws.amazon.com/it/lambda/.

[9] Amazon. Amazon rds, 2023. URL https://aws.amazon.com/it/rds/.

[10] Amazon. Amazon sqs, 2023. URL https://aws.amazon.com/it/sqs/.

[11] Amazon. Amazon sns, 2023. URL https://aws.amazon.com/it/sns/.

[12] Amazon. Amazon cognito, 2023. URL https://aws.amazon.com/it/cognito/.

[13] Avenga. Microservices vs. monoliths: which architecture will be best for your product?, 2022. URL https://www.avenga.com/magazine/microservices-vs-monoliths/.

[14] Adam Bellemare. *Building Event-Driven Microservices*. Oreilly & Associates Inc., 2020.

[15] David Boyne. Inside event-driven architectures, 2023. URL https://serverlessland.com/event-driven-architecture/visuals/inside-event-driven-architectures.

[16] Marianne Bradford. *Modern ERP: Select, Implement, and Use Today's Advanced Business Systems*. Lightning Source Inc., 2020.

[17] ByteByteGo. Cloud services cheat sheet, 2023. URL https://blog.bytebytego.com/p/ep70-cloud-services-cheat-sheet.

[18] Inc. CB Information Services. Why serverless computing is the fastest-growing cloud services segment, 2018. URL https://www.cbinsights.com/research/serverless-cloud-computing/.

[19] ClickIt. Web application architecture: The latest guide, 2022. URL https://www.clickittech.com/devops/web-application-architecture/.

[20] Codemotion. Green cloud computing strategies and best practices, 2022. URL https://www.codemotion.com/magazine/devops/cloud/green-cloud-computing-strategies-and-best-practices/.

[21] Datadog. The state of serverless, 2023. URL https://www.datadoghq.com/state-of-serverless/.

[22] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[23] Flutter. Flutter, 2023. URL https://flutter.dev/.

[24] Animesh Gaitonde. Distributed transactions & two-phase commit, 2021. URL https://medium.com/geekculture/distributed-transactions-two-phase-commit-c82752d69324.

[25] GeeksforGeeks. Acid properties in dbms, 2023. URL https://www.geeksforgeeks.org/acid-properties-in-dbms/.

[26] Gitlab. What are the benefits of a microservices architecture?, 2022. URL https://about.gitlab.com/blog/2022/09/29/what-are-the-benefits-of-a-microservices-architecture/.

[27] Google. What is cloud computing?, 2023. URL https://cloud.google.com/learn/what-is-cloud-computing?hl=en.

[28] Google. Go for cloud and network services, 2023. URL https://go.dev/solutions/cloud.

[29] Google. Go cloud cdk, 2023. URL https://gocloud.dev/.

[30] Google. Angular, 2023. URL https://angular.io/.

[31] Google. Firebase, 2023. URL https://firebase.google.com.

[32] Gorm. Gorm documentation, 2023. URL https://gorm.io/index.html.

[33] Taufiq Hidayat. Microservice versus monolithic architecture. what are they?, 2020. URL https://medium.com/javanlabs/micro-services-versus-monolithic-architecture-what-are-they-e17ddc8d3910.

[34] ISO. Software standards, 2023. URL https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[35] Joy Joel. An introduction to cloud computing, cloud service models, and cloud deployment models, 2021. URL https://aws.plainenglish.io/cloud-computing-intro-cloud-service-models-cloud-deployment-models-e013872064a4.

[36] James Kwon. What is two phase commit in distributed transaction?, 2022. URL https://hongilkwon.medium.com/when-to-use-two-phase-commit-in-distributed-transaction-f1296b8c23fd.

[37] Grace Lau. Serverless computing: The advantages and disadvantages, 2023. URL https://www.codemotion.com/magazine/devops/cloud/serverless-computing-the-advantages-and-disadvantages/.

[38] David Luther. 8 erp trends for 2023 and the future of erp, 2023. URL https://www.netsuite.com/portal/resource/articles/erp/erp-trends.shtml.

[39] Inc. Meta Platforms. React native, 2023. URL https://reactnative.dev/.

[40] Microsoft. What is cloud computing?, 2023. URL https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing/.

[41] Microsoft. Event-driven architecture style, 2023. URL https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven.

[42] Microsoft. Saga distributed transactions pattern, 2023. URL https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga.

[43] Sanad Mohamed. Pros and cons of serverless, 2023. URL https://www.linkedin.com/pulse/pros-cons-serverless-sanad-mohamed-mba-aws/.

[44] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. Oreilly & Associates Inc., 2021.

[45] Jim O'Donnell. Experts predict flexibility as a top erp trend in 2022, 2022. URL https://www.techtarget.com/searcherp/feature/Experts-predict-flexibility-as-a-top-ERP-trend.

[46] Elegberun Olugbenga. Netflix system design- backend architecture, 2021. URL https://dev.to/gbengelebs/netflix-system-design-backend-architecture-10i3.

[47] David Parnas. On the criteria to be used in decomposing systems into modules. *Journal contribution*, 1971.

[48] David Parnas. Information distribution aspects of design methodology. *Journal contribution*, 1972.

[49] Timothy Grance Peter M. Mell. The nist definition of cloud computing (technical report). *Special Publication 800-145*, 2011.

[50] C. Guzikowski R. Magoulas. O'reilly serverless survey 2019: Concerns, what works, and what to expect, 2019. URL https://www.oreilly.com/radar/oreilly-serverless-survey-2019-concerns-what-works-and-what-to-expect/.

[51] Recro. Top 5 cloud service providers: A comparison, 2019. URL https://recro.io/blog/top-5-cloud-service-providers/.

[52] Allied Market Research. Enterprise resource planning (erp) market, 2022. URL https://www.alliedmarketresearch.com/ERP-market.

[53] ERP Research. Independently compare erp systems and erp consulting, 2023. URL https://www.erpresearch.com/.

[54] Expert Market Research. Global serverless computing market outlook, 2023. URL https://www.expertmarketresearch.com/reports/serverless-computing-market.

[55] Grand View Research. Cloud computing market size, share and trends analysis report by service, by deployment, by enterprise size, by end-use, by region, and segment forecasts, 2023. URL https://www.grandviewresearch.com/industry-analysis/cloud-computing-industry.

[56] Chris Richardson. Pattern: Saga, 2023. URL https://microservices.io/patterns/data/saga.html.

[57] Felix Richter. Amazon maintains lead in the cloud market, 2023. URL https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/.

[58] Stephen Roddewig. What is event-driven architecture? everything you need to know, 2023. URL https://blog.hubspot.com/website/event-driven-architecture.

[59] Serverless. Serverless documentation, 2023. URL https://www.serverless.com/framework/docs.

[60] Solace. The complete guide to event-driven architecture, 2023. URL https://solace.com/what-is-event-driven-architecture/.

[61] Ketan Varshneya. Understanding design of microservices architecture at netflix, 2021. URL https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/.

[62] Jerry Wallis. 6 serverless computing benefits and drawbacks to help decide if it's right for you, 2023. URL https://intuji.com/serverless-computing-benefits-and-drawbacks/.

[63] Keyang Xiang. Patterns for distributed transactions within a microservices architecture, 2018. URL https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture.