NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RAFAEL SANTANA CARVALHO MIRA

BSc in Computer Science

AUTONOMIC MANAGEMENT OF MICROSERVICE-BASED WEB APPLICATIONS

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
July, 2025

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

AUTONOMIC MANAGEMENT OF MICROSERVICE-BASED WEB APPLICATIONS

RAFAEL SANTANA CARVALHO MIRA

BSc in Computer Science

Adviser: João Carlos Antunes Leitão
Associate Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
July, 2025

Abstract

Microservice architectures have become wildly popular for developing large-scale web applications, due to their inherent modularity, scalability and resilience, which together make them a perfect architectural choice for building complex and constantly evolving systems. Unfortunately, many operational decisions involved in managing microservice-based systems still rely heavily on human intervention. Tasks such as tuning configuration parameters or adapting to changing workloads often remain manual and require context-aware decision-making to balance performance, reliability, and user experience. This creates the opportunity for autonomic computing approaches that can enrich these architectures with self-management capabilities.

To explore these possibilities, in this work, we propose the design and implementation of a case-study CRM application based on a microservice architecture that is designed from the start to be enriched with autonomic capabilities. Our solution will focus on reducing the need for manual intervention by enabling context-aware reconfiguration of the application behavior at runtime. To assess its effectiveness, we evaluate the CRM-system prototype through large-scale emulation, demonstrating the feasibility and impact of autonomic control in realistic microservice environments.

Keywords: Microservices, Autonomic Computing, CRM

Resumo

Arquitecturas de microsserviços tornaram-se muito populares para o desenvolvimento de aplicações Web de grande escala, devido à sua modularidade, escalabilidade e resiliência inerentes, que, em conjunto as tornam uma escolha arquitetural perfeita para construir sistemas complexos e em constante evolução. Infelizmente, muitas decisões operacionais envolvidas na gestão de sistemas baseados em microserviços ainda dependem fortemente da intervenção humana. Tarefas como ajustar parâmetros de configuração ou adaptação a cargas de trabalho variáveis continuam a ser ajustadas manualmente e exigem uma tomada de decisão baseada no contexto para equilibrar o desempenho, com a fiabilidade e a experiência do utilizador. Isto cria uma oportunidade para abordagens de computação autonómica que podem enriquecer estas arquiteturas com capacidades de autogestão.

Para explorar estas possibilidades, neste trabalho, propomos a conceção e implementação de uma aplicação CRM, como um caso de estudo baseado numa arquitetura de microserviços concebida desde o início para ser enriquecida com capacidades autonómicas. A nossa solução centrar-se-á na redução da necessidade de intervenção manual, permitindo a reconfiguração do comportamento da aplicação durante o tempo de execução, com base no contexto envolvente. Para avaliar a sua eficácia, o protótipo do sistema CRM é testado por simulação em larga escala, demonstrando a viabilidade e o impacto do controlo autonómico em ambientes realistas de microserviços.

Palavras-chave: Microserviços, Computação Autonómica, CRM

Contents

List of Figures v

Acronyms vi

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Context . . . 3
- 1.3 Expected Contributions . . . 3
- 1.4 Document Structure . . . 4

2 Related Work 5
- 2.1 Customer Relationship Management (CRM) . . . 5
- 2.2 Monolithic Architectures . . . 6
- 2.3 Microservices Architectures . . . 8
- 2.4 Microservices Challenges . . . 10
  - 2.4.1 Communication . . . 10
  - 2.4.2 Granularity . . . 12
  - 2.4.3 Data Management . . . 13
  - 2.4.4 Distributed Transactions . . . 15
- 2.5 Microservice Patterns . . . 20
  - 2.5.1 Service Discovery . . . 20
  - 2.5.2 Externalized Configuration . . . 21
  - 2.5.3 API Gateway . . . 21
  - 2.5.4 Resilience . . . 22
  - 2.5.5 Observability . . . 23
- 2.6 Autonomic Computing . . . 24
  - 2.6.1 Self-* Properties . . . 24
  - 2.6.2 MAPE-K . . . 25
  - 2.6.3 Planning and Analysis Approaches . . . 26
- 2.7 Summary . . . 27

3 Planning **28**
3.1 Proposed Solution 28
3.1.1 CRM Implementation 28
3.1.2 Autonomic Tuning and Operations Manager (ATOM) 29
3.2 Evaluation 31

4 Work Plan **33**

Bibliography **34**

Appendices

Annexes

List of Figures

2.1 Example of a distributed transaction using the Two-Phase Commit protocol 16
2.2 Example of a distributed transaction using a orchestrated saga 17
2.3 Example of a distributed transaction using a choreographed saga 19
2.4 MAPE-K Control Loop (Image taken from [33]) 25

3.1 CRM Architecture Overview 29
3.2 Conceptual Overview of CRM with Autonomic Management 30

4.1 Gant chart of the expected work plan 33

Acronyms

2PC Two-Phase Commit *(pp. 15–17, 27)*

ATOM Autonomic Tuning and Operations Manager *(pp. iv, 29–31, 33)*

CRM Customer Relationship Management *(pp. iii, 3–6)*

DDD Domain-Driven Design *(p. 8)*

ECA Event-Condition-Action *(pp. 26, 27)*

Pub-Sub Publish-Subscribe *(p. 11)*

RPC Remote Procedure Calls *(pp. 11, 18)*

SOA Service-Oriented Architecture *(pp. 1, 8)*

1

Introduction

Software applications were often developed following a monolithic architecture, where all functionalities were tightly coupled into a single, indivisible deployable unit [61]. While simple to develop in their initial stages, monolithic applications often encountered significant challenges as they grew in complexity and scale. These challenges included difficulty in maintaining and updating the codebase, as even small changes required redeploying the entire application, and limited scalability, as the entire application had to be scaled even if only a single component faced high demand. Resilence was another issue, as a failure in one part of the application could bring down the entire system.

In the late 1990s, Service-Oriented Architecture (SOA) emerged as an attempt to address the challenges of large monolithic applications [62]. SOA promoted the decomposition of applications into distinct, reusable services that communicated through standardized protocols. While the principle of service reusability was appealing, it often introduced significant trade-offs. Increasing reusability frequently resulted in tight inter-service dependencies [62], making systems harder to manage and evolve. As a result, deploying a single service could require redeploying several others. Additionally, SOA's reliance on centralized infrastructure for orchestration [50] harmed the agility and efficiency that it initially promised, creating a continued need for a more lightweight and flexible architectural style.

In response to the limitations of prior architectural approaches, a new architecture known as microservices began to take shape. Microservice-based architectures provide a range of benefits that address the critical demands of modern systems, especially when compared to traditional centralized architectures. They address those limitations by offering greater modularity, scalability, and resilience. Their modular design enables teams to develop, deploy, and maintain components independently. Services can be scaled individually based on demand, optimizing resource usage. Additionally, fault isolation ensures that failures in one service do not cascade across the entire system, making microservices ideal for building robust, highly available applications.

1.1 Motivation

Over the past decade, the software development industry has been profoundly reshaped by the widespread adoption of microservice architectures [39, 53, 48]. This was significantly accelerated by concurrent advancements in cloud computing services that offered flexible and on-demand infrastructure, a crucial element for deploying and managing numerous small and independent components with ease [35]. Simultaneously, the rapid development of containerization technologies provided lightweight, portable, and isolated environments that perfectly complemented the microservice philosophy, simplifying deployment and operational complexity across diverse environments. Another factor that drove the shift towards this architectural style were the challenges faced by large-scale companies in managing monolithic systems under rapidly increasing user bases and data volumes [68]. Pioneers in this space, such as Netflix, publicly documented their transition to microservices architecture to achieve better scalability, agility, and continuous delivery capabilities required for their global streaming service [37]. This significantly popularized microservices architecture, demonstrating their practical advantages in handling massive scale and fostering rapid innovation [36].

But even though microservice-based systems provide numerous benefits, their management still presents significant challenges. The very characteristics that make them appealing also introduce considerable complexity at runtime. Many critical operational decisions, ranging from optimizing resource allocation to adjusting service behavior in real-time, still heavily depend on manual human intervention [56]. This often translates into ad-hoc adjustments which can be time-consuming, error-prone, and insufficient to maintain optimal performance and user experience in rapidly changing environments. The sheer volume of services, instances, and inter-service communications in a large-scale microservice deployment can quickly overwhelm human operators, leading to performance degradation, increased operational costs, and reduced system agility. While tools like Kubernetes [44] have significantly relieved the difficulties related to deployment, orchestration, and basic scaling of microservices, many complex operational decisions, particularly those requiring context-aware adjustments to application behavior or parameters, continue to demand human oversight.

This creates an opportunity for exploring, designing, and implementing more sophisticated and automated solutions. In this context, the paradigm of autonomic computing offers a promising avenue for research and development. Autonomic computing aims to imbue software systems with self-management capabilities [34] so that systems can continuously monitor their own state and operating environment, analyze collected data and then apply appropriate reconfigurations to the system itself in a timely fashion. By integrating these capabilities, it becomes possible to reduce the reliance on manual intervention, enabling microservice applications to dynamically adapt their behavior in a context-aware manner during execution. The shift from manual management to automated self-adaptation holds the potential to significantly enhance the efficiency and overall

robustness of large-scale microservice applications, allowing them to autonomously respond to fluctuating demands, evolving threats, and performance anomalies.

1.2 Context

As autonomic computing continues to advance, a clear opportunity emerges to explore the benefits of applying these capabilities to the dynamic and evolving domain of CRM systems. CRM systems are complex business-oriented tools that work as centralized platforms to manage and analyze all interactions and data related to customers throughout their lifecycle [55]. Their fundamental purpose is to enhance and strengthen business relationships, foster customer retention, and facilitate more effective interactions across all touchpoints with a customer. In today's highly customer-focused and data-driven world [10], using a CRM platform is crucial for companies striving to remain competitive as it enables them to deeply understand customer behaviors, anticipate their needs, and personalize experiences, leading to improved customer satisfaction and loyalty.

While the specific feature sets of CRM systems can vary widely depending on industry, business size, and specific needs, they typically converge on a common set of core functionalities designed to support various departments. CRM platforms commonly integrate robust capabilities for customer service, enabling efficient handling of inquiries, issue resolution, and tracking of customer interactions. For marketing departments, it provides tools for designing and executing targeted campaigns, managing lead generation and analyzing campaign effectiveness. Sales functionalities are also a key component, assisting teams in managing pipelines, tracking opportunities, forecasting revenue and automating aspects of the sales process, among others. Furthermore, data analytics are deeply integrated, offering reporting tools that allow for the identification of trends and providing helpful insights into various business domains. The rich, centralized data collected and processed through these diverse functionalities empowers companies to make informed and strategic decisions, driving business growth and efficiency.

Given the central role CRM systems play in direct customer engagement and strategic decision-making, their uninterrupted and efficient operation is crucial for any organization. Furthermore, any slowdown, instability, or difficulty in managing these systems can directly translate into missed sales opportunities, diminished customer satisfaction, and inefficient workflows for employees relying on the platform. The complexity and evolving demands of CRM applications make them particularly well-suited for implementation using microservices architecture as their numerous distinct business capabilities can benefit from independent development, deployment, and scaling.

1.3 Expected Contributions

The core of this work involves the design and implementation of a microservice-based CRM application with built-in autonomic capabilities. This will enable the CRM to

dynamically adapt its operational behavior, including automated tuning of configuration parameters and intelligent management of microservice instance lifecycles, to optimize its performance and resource utilization at runtime. Finally, we will perform experimental validation and evaluation of the developed autonomic CRM prototype through large-scale emulation. This evaluation will demonstrate the practical benefits and impact of integrating autonomic control into complex microservice applications, contributing insights into the autonomous management of such systems.

1.4 Document Structure

Besides this introductory chapter, this document is composed of:

- **Chapter** 2 provides a complete review of existing studies and main concepts relevant to this dissertation. It starts by presenting a detailed overview of CRM systems, followed by an extensive explanation of microservices architecture, covering its benefits, challenges, and known techniques and patterns for overcoming those challenges. The chapter will also explain the fundamental ideas of autonomic computing, including Self-* systems and the MAPE-K cycle. It finishes by discussing relevant technologies and tools that will help create the proposed solution.
- **Chapter** 3 describes our proposed solution and how we plan to evaluate and validate it. This chapter outlines the architectural design and the main components of the autonomic CRM system.
- **Chapter** 4 presents an estimated work plan to complete the tasks related to this dissertation.

2

Related Work

In this chapter, we present and discuss the concepts and previous work that form the foundation of our research and development plan. We begin by providing an overview of CRM systems, followed by a review of different software architectures commonly used to build web applications, starting with monolithic and then moving to microservices architectures. Here we will discuss in detail the challenges of working with microservices, as well as the practices and patterns that can be used to address them. Finally, we will introduce the concept of autonomic computing, which plays a significant role in our proposed approach.

2.1 CRM

CRM systems are complex business-oriented tools that work as centralized platforms to manage and analyze customer interactions [55]. Modern CRM systems are highly evolved, distributed applications whose primary purpose is to manage dynamic customer interactions and extensive, diverse datasets at scale. These systems are typically designed to operate in a multi-tenant environment [58], enabling multiple customers to share a single application instance while maintaining strict data isolation and security. This approach allows CRM providers to efficiently serve large user bases, ensuring both high availability and scalability.

CRM platforms are frequently built using a *cloud-native architecture* [54], which combines microservices and containerization technologies. While a CRM system offers numerous distinct functionalities, organizations often require only a subset of these features. Modularity is a key reason for adopting microservices architectures, as it enables the selective addition or removal of features in response to evolving business needs, thereby enhancing system agility and optimizing resource usage. Notable examples of platforms utilizing cloud-native architectures include Salesforce [69], HubSpot [32], and Pipedrive [59], all designed to support modularity and scalability across various cloud environments.

CRMs are inherently data-driven applications, designed to efficiently manage and

process large volumes of information. They aggregate and integrate data from diverse sources, utilizing both traditional data warehouses and more flexible data lakes. This data-centric foundation empowers CRMs to deliver comprehensive insights into customer behavior, sales trends, and marketing effectiveness, frequently leveraging advanced analytics and machine learning techniques to extract valuable insights from the data. Artificial intelligence is now a standard component of modern CRM systems [70], driving essential operations such as predictive sales forecasting, and the intelligent automation of routine tasks like data entry and follow-ups, fundamentally shaping their capabilities and value.

2.2 Monolithic Architectures

In this section, we examine the key characteristics of monolithic architectures and their implications in greater depth. A monolithic architecture is a traditional software design approach where all components of an application are integrated into a single deployable unit [61]. This means that the entire application's functionalities, encompassing everything from its core operational logic and data management to security concerns like authentication and authorization, are contained within one codebase. It is also common for all components of the application to use a single, shared database that stores all the application's data.

Monolithic Advantages: In its initial stages of development, a monolithic architecture can be particularly appealing due to its simplicity and ease of management. Since all components reside within the same codebase, developers can rapidly implement new features or extend existing ones without the need to coordinate across multiple independent services or complex deployment pipelines. This often leads to a faster initial development pace. Deployment is also straightforward, as the entire application is packaged and deployed as a single artifact, which significantly simplifies the release process.

Beyond development, monolithic architectures offer advantages in operational aspects. The application's unified structure simplifies debugging and performance tuning, as everything runs within a single process, making it easier to trace the root cause of issues and identify performance bottlenecks. Furthermore, data management is simpler in this architecture, as all data typically resides in a common database. This simplifies transaction management and data consistency, avoiding the complexities associated with distributed transactions found in other architectures [16].

Monolithic Challenges: Despite the short term benefits provided by this architecture, it is important to recognize that this approach can lead to significant challenges as the application grows in complexity and scale, and often becomes the reason why many applications struggle to adapt to changing requirements and increased user demands. One of the primary challenges in the monolithic architecture comes from the fact that

the entire application is treated as a single deployment unit. This characteristic implies that any modification, from a minor bug fix to a major feature implementation, implies rebuilding and redeploying the entire application. While this might not be a significant concern for small applications, for large-scale applications, comprising millions of lines of code, it can lead to substantial delays in development cycles, with each build and testing phases consuming considerable time. Furthermore, as the application expands, the codebase becomes increasingly challenging to navigate, and more importantly, to understand. This difficulty is linked to a lack of clear code structure and modularity, where various components are tightly coupled. Such tight coupling makes it exceedingly difficult to preview how changes in one section of the application might impact others. When this happens, it's often referred to as the *Big Ball of Mud* anti-pattern [15]

Beyond development challenges, monolithic architectures present significant limitations in terms of scalability. One option to scale is to increase the computing resources of the single server running the application, called vertical scaling [1]. While this approach is simple, it quickly hits physical limits and can become costly for applications with variable and unpredictable workloads due to the need for over-provisioning. The alternative is horizontal scaling, which involves distributing the application load across multiple machines to improve performance and availability [1]. However, this is often inefficient for monoliths because their components cannot be scaled independently, so the entire application must be replicated even if only one feature requires more resources. The inability to scale components independently also has implications for the resilience of the application. Even when multiple instances are deployed through horizontal scaling, the indivisible nature of the monolith means that each individual instance remains vulnerable due to the lack of isolation between its components. If a failure occurs in one part of the application, it can potentially affect the entire instance and bring down the whole application.

Modular Monoliths: A way to address some of the challenges of monolithic architectures is to adopt a modular monolith architecture, which can be presented as an intermediate step between monolithic and microservices architectures [3, 13]. In this architecture, the application is structured by dividing its business logic into well-defined modules with clear boundaries. All modules are still part of the same monolithic application, so they are deployed together and usually share the same database. The key difference is that when one module needs to communicate with another, it does not have direct access to that module's code or state. Instead, it does so through a well-defined API, as if they were separate, independently deployed services [13]. However, since they are deployed together, these inter-service communications are in-memory calls rather than network calls, making them much faster.

It can be seen as a monolithic application designed with modularity in mind, preparing the ground for a potential future migration to microservices. Its internal discipline improves code maintainability, simplifies testing and streamlines deployment processes

compared to undifferentiated monoliths. By establishing clear module boundaries and communication patterns upfront, it lays the groundwork for later extracting individual services into independent deployment units when scalability or other needs arise [13]. The primary benefit of using the modular monolith architecture is that it avoids the initial overhead of adopting a more complex architecture, such as microservices, when it is uncertain whether the application will need to scale or remain small and manageable.

2.3 Microservices Architectures

Moving beyond the confines of monolithic systems, modern software development increasingly embraces distributed architectures to address growing demands for more scalable, resilient, and flexible systems. This paradigm involves structuring applications as distributed systems, fundamentally altering how their components interact and operate. A distributed system can be defined as a set of autonomous computational entities, such as computers or processes, that are interconnected through some network where processes cooperate to execute tasks [71]. A famous and more informal definition of distributed systems was provided by Leslie Lamport, which states that "A distributed system is a system in which the failure of a computer you didn't even know existed can render your own computer unusable" [46]. Within this paradigm, various architectural styles have emerged, each offering distinct approaches to structuring and managing distributed components. In this section, we will focus on microservices architectures.

Microservices emerged as a response to the limitations of earlier architectures, such as monolithic and SOA [47]. Rather than being an entirely new concept, the microservices approach represent a refinement of existing distributed architectures that had already explored the possibility of building applications as collections of services, but often fell short due to misunderstandings about achieving true service independence and modularity. A conceptual definition of microservices was provided by James Lewis and Martin Fowler, who described them as **a way to develop applications as a collection of small, loosely coupled and independently deployable services, each built around business capabilities**, and use some lightweight mechanism to communicate with each other [47]. This definition served as a baseline for what was an undefined and misconceived territory at the time and remains strongly accurate with today's practices.

One of the defining characteristics of this architecture is how microservices are modeled. They are usually modeled through Domain-Driven Design (DDD) principles [50], which emphasize the importance of understanding the business domain and defining clear boundaries between different parts of the system, known as bounded contexts [11]. This means that each service is designed to encapsulate a specific business feature, promoting high cohesion within the service and low coupling with other services. Such characteristic provides the modularity benefit of microservices, as each service can be understood and changed independently. A key principle for achieving success with this approach is that each service should own its state [50]. Owning the state means that each service

is responsible for managing its own data and not relying on other services to provide it. This often leads to a decentralized data management approach, where each service manages its own database and does not share data with other services directly, commonly known as the *database per service* pattern [17]. In this approach, when services need to exchange data, they communicate with each other over the network using mechanisms designed for inter-service communication. This aspect of data decentralization is a key factor in effective microservices design, as it allows each service to have governance over its own data, thereby enabling them to evolve independently, as they can change their data models without affecting other services database schemas. Nevertheless, this decentralized approach involves challenges regarding data as we no longer have a single shared database, aspects like data consistency and transactions management become more complex. These challenges will be discussed in more detail in Section 2.4.

The modular nature of microservices allows for independent deployability [50]. Independent deployability means that we can change a microservice and deploy it, without needing to redeploy any other application service. This independence brings many benefits to the architecture. A major advantage is that each service can be scaled independently of other services. This has the potential to significantly reduce application costs, as resources can be allocated based on the actual needs of each service, rather than requiring the entire application to scale as a whole. Furthermore, without a monolithic codebase, there is no longer technology lock-in, and each service can be implemented using the most suitable technology for its specific requirements. As mentioned earlier, containerization technologies, such as Docker [9], were a key enabler for this particular characteristic of microservices. Containers provide isolated environments for each service, allowing different programming languages, frameworks and even distinct versions of the same runtime environment to coexist within the same application. This greatly simplifies the packaging and deployment of heterogeneous microservices, solidifying the benefits of independent technology choices.

Beyond independent scaling and technological flexibility, microservices significantly enhance application resilience through fault isolation. Since services are independent and loosely coupled, when the system is correctly designed and implemented, a failure occurring in one microservice does not bring down the entire application. Instead, the fault is isolated to that specific service, allowing other services to continue operating normally. This contrasts directly with monolithic architectures where a single component failure can lead to the entire application becoming unavailable. The combined characteristics of having each microservice as an independently deployable unit, modeled around a specific business domain, allow for services to be divided into smaller and more manageable pieces that can be developed and maintained by dedicated teams. This often leads to faster development, deployment, and testing cycles, enabling a faster time to market, as each team can work on their own service with minimal coordination with other teams.

2.4 Microservices Challenges

Despite the compelling benefits offered by microservice architectures, their distributed nature inherently introduces a set of challenges that require careful consideration during design and implementation. While centralized systems present their own difficulties, microservices shift these problems from within a single process to the network, demanding different approaches to solve them. Effectively managing these challenges is crucial for realizing the full potential of this architectural style. In this section, we will explore some of the primary difficulties encountered when building and operating microservice-based systems, including complexities related to inter-service communication, service granularity, data management and distributed transactions.

2.4.1 Communication

Unlike monolithic applications, where components interact through direct function in a shared memory space, microservices must communicate across a network, introducing concerns such as latency and network failures. In this section, we will review some important aspects of inter-service communication as well as common communication models used in this architecture.

2.4.1.1 Timing Assumptions

Communication in a microservice architecture involves fundamental assumptions about time. These assumptions shape the communication style and may have significant impact on the system's performance and resiliency. Inter-service interactions can be categorized based on their timing behavior as:

Synchronous: Involves direct requests and responses between services, where one service waits for the other to complete its operation before proceeding. This creates a blocking interaction, where the client pauses its operation until the server replies or a timeout occurs. This model introduces tight coupling between services and can lead to cascading failures if one service becomes unavailable or slow to respond.

Asynchronous: Allows a client service to send a message to a server service without immediately waiting for a response. The client continues its processing, and the response, if any, is handled separately, often via a callback mechanism or by polling for results. This non-blocking nature promotes loose coupling between services and enhances fault tolerance by allowing clients to operate even if a server is temporarily unavailable.

2.4.1.2 Communication Models

In distributed systems, choosing the right communication model is crucial to a successful architecture, and in microservices-based architectures this is no different. There are two

widely used approaches for inter-service communication [72]:

Remote Procedure Calls (RPC): Represent a communication model that aims to make remote service calls appear as if they were local function calls, typically within a client-server interaction using a request-reply pattern [72]. When a client invokes a RPC, the underlying RPC framework handles the network communication, data serialization and deserialization, presenting the remote operation as a familiar local procedure. This underlying framework often leverages standard network protocols, with HTTP being a common choice for implementing RPC-style APIs. While RPCs are most commonly associated with synchronous communication, they can also be used in asynchronous contexts. Some common use cases of RPC include REST [14], SOAP [4], gRPC [27] and GraphQL [12].

Message Brokers: Offer a robust approach to inter-service communication by introducing an intermediary layer between the sending and receiving parties. The model works by having services send messages to a message broker, which then reliably delivers them to the intended recipients. Messages are buffered by the middleware, often using persistent storage, until they can be processed by the receiving service. We can imagine this as a post office that receives letters from senders and delivers them to recipients. This model inherently promotes strong decoupling between communicating parties, as the sender and receiver do not need to be aware of each other's existence, making them particularly suitable for scenarios where it cannot be assumed the availability of the receiving service at the time a message is issued [72]. This form of communication is asynchronous, allowing client services to send messages and continue processing without waiting for an immediate response. Within this approach, distinct patterns are commonly employed.

One common pattern is the traditional message queue, which implements a point-to-point communication model. In this setup, a message sent by a producer is delivered to a queue and is consumed by exactly one consumer process, regardless of how many consumers are bound to that queue. Once a message is successfully processed and acknowledged by a consumer, it is typically removed from the queue, ensuring that each message is handled only once. One characteristic of this model is that the broker is actively managing the message delivery state and removing consumed messages, sometimes referred to as a "smart broker/dumb consumer" approach. A popular implementation of a message queue system is RabbitMQ, which implements the Advanced Message Queuing Protocol (AMQP) [29].

The other common use case is the Publish-Subscribe (Pub-Sub) [73] pattern, which operates differently. While also using an intermediary broker and facilitating asynchronous communication, Pub-Sub extends beyond point-to-point delivery to support one-to-many message distribution. Publishers send messages about a particular topic, and multiple independent subscribers can register their interest in that topic, each receiving their own copy of the message. A key differentiator in many Pub-Sub implementations is message

retention policy. Messages are typically not deleted immediately after consumption. Instead, they are stored durably for a configurable period, allowing consumers to replay historical data or new consumers to process messages from any point in time. In this model, consumers are often responsible for tracking their own progress, leading to a " dumb broker/smart consumer" approach. Systems like Apache Kafka [43] exemplify this model and are particularly well-suited for scenarios where multiple services need to react to the same event, enabling a highly decoupled event-driven architecture.

2.4.2 Granularity

Although microservices are often characterized as "small", this term lacks a strict, universally agreed-upon definition and can even be misleading [63]. The "micro" prefix does not imply that services must be microscopic. Rather, it emphasizes their focused scope and independent nature. Determining the optimal size and responsibility of a microservice is, in fact, a critical design decision, commonly referred to as *service granularity*. Inappropriate granularity can undermine many of the benefits microservices aim to provide. In extreme cases, instead of creating a more manageable and scalable architecture, it can lead to a system that is either too complex or too fragmented, exhibiting characteristics often associated with distributed monoliths [18].

A fundamental principle for defining service granularity lies in managing cohesion and coupling. When designing microservices, it is essential to ensure that each service exhibits high cohesion, meaning it encapsulates a single, well-defined business capability or domain concept, with all related functionalities residing within that service. At the same time, services should be loosely coupled, implying that changes within one service do not require changes in others, allowing them to evolve independently. Applying Domain-Driven Design principles allows development teams to identify natural boundaries and define bounded contexts for business features, ensuring that services are modeled with internal consistency and focus, which makes them easier to understand, develop, and maintain independently. Unfortunately, defining service boundaries is not a trivial task. Modeling microservices with extremely fine granularity risks creating a system with an excessive number of services, leading to substantial overhead in managing and coordinating them. In such scenarios, the complexity of communication, transactionality and orchestration of flows between many services can, and often does, outweigh the benefits of having fine-grained services [19]. But if modularity is not sufficiently considered and services are modeled with too coarse granularity, this risks creating large monolithic services, that are difficult to maintain and evolve, reintroducing the very same problems microservices aim to solve.

Therefore, achieving the correct granularity requires finding a balance between modularity, performance, and evolutionary capabilities. A correctly defined service granularity should be one that minimizes inter-service dependencies while maintaining strong cohesion. It should enable independent development and deployment, providing teams the

autonomy to work on their services without needing to coordinate with others. Perhaps the most important aspect of well-designed microservices is the minimization of distributed transactions [64]. Distributed transactions are a complex subject that can lead to significant performance issues in a microservice architecture. We will discuss distributed transactions in depth in Section 2.4.4.

2.4.3 Data Management

In microservice architectures, data management presents a distinct set of challenges compared to systems where a single, centralized database is used. The shift to a decentralized model, introduces complexities related to data consistency, availability, and transaction management across multiple independent services. In this section, we will explore some fundamental concepts regarding data management in microservices, including the CAP theorem, consistency models, and the trade-offs between ACID and BASE properties.

2.4.3.1 CAP Theorem

The CAP theorem [5] is a foundational concept in distributed data management, stating that across multiple replicas a distributed data store can simultaneously guarantee only two of the following three properties:

Consistency (C): All clients see the same data at the same time, regardless of which replica they connect to.
Availability (A): Every request receives a response, even if some replicas are unavailable, implying that the system is always operational.
Partition Tolerance (P): The system continues to operate even in the presence of network partitions.

Network partitions are unavoidable in highly distributed systems [26], making partition tolerance an essential requirement. This implies that the CAP theorem forces a fundamental choice between maintaining consistency or availability, a trade-off that directly impacts how data is managed and accessed across different services. Consequently, the optimal approach depends on the specific requirements and constraints of each system, as no universal solution exists for all scenarios.

2.4.3.2 Consistency Models

A consistency model defines the guarantees a distributed system provides concerning data visibility and the ordering of operations across its replicas. In practice, microservice architectures frequently adopt a combination of consistency models, allowing different services to prioritize either consistency or high availability based on their specific needs. Three primary models are commonly distinguished:

Strong Consistency: In a strongly consistent system, after a write operation is completed, all subsequent read operations across any replica are guaranteed to return the most recent version of data [42]. This model ensures that all replicas remain synchronized, providing a unified, up-to-date view of the data at any given moment. Achieving strong consistency typically involves synchronous replication of write operations across all relevant replicas [74]. This coordination introduces latency and can significantly impact the system's overall availability and performance, especially under network partitions.

Causal Consistency: Causal consistency is model that ensures all processes observe causally related operations in the same order, while independent operations may appear in different orders [74]. In a social media example, if a user posts a message *m1* and then replies to it with *m2*, in this model, anyone who sees the reply *m2* will also see the original message *m1* first. But it does not guarantee that if two users post unrelated messages at the same time, they will see those messages in the same order. It preserves cause-and-effect relationships in a system without the overhead of enforcing a global order, offering stronger guarantees than eventual consistency but less strict than strong consistency [42].

Eventual Consistency: In contrast with strong consistency, an eventually consistent system guarantees that if no new updates occur for a specific data item, all replicas of that item will eventually converge to the same value [76]. This model allows for temporary inconsistencies between replicas, as updates may not be immediately visible across all of them. It is commonly implemented through asynchronous replication, where the state of replicas is reconciled over time. While it introduces a delay in data propagation, it allows the system to remain operational and responsive even during network partitions or under heavy load.

2.4.3.3 ACID vs BASE

The different approaches to consistency influence how transactions are managed in distributed systems. Traditional monolithic applications often rely on strong transactional guarantees, typically provided by relational database management systems implementing *ACID* [30] properties, which stand for:

Atomicity (A): A transaction is treated as a single, indivisible unit of work, that either fully succeeds or completely fails.

Consistency (C): A transaction brings the database from one valid state to another, preserving all defined rules, constraints and invariants within the database.

Isolation (I): Concurrent transactions operate separately from each other, preventing their intermediate operations from interfering.

Durability (D): Once a transaction is committed, its changes are persisted and survive system failures.

While ACID transactions offer undeniable benefits for robust data integrity, implementing them across multiple independent services and their respective databases in a distributed environment is exceptionally complex and often impractical. Enforcing strict ACID guarantees across service boundaries introduces significant performance bottlenecks due to the necessity for heavy coordination mechanisms to handle distributed transactions. While possible, fully adhering to ACID properties across microservices is generally avoided, as it undermines the very advantages microservices aim to provide. In response to these challenges, distributed systems that prioritize availability and scalability over strong consistency, adopt transactional models aligned with *BASE* [24] properties, which stand for:

Basically Available (BA): All the services or systems involved are expected to be available.
Soft state (S): The system's state may change over time, even without new input, as data gradually propagates and converges, meaning consistency is not always immediate but eventually achieved.
Eventually consistent (E): Once there are no new updates, the system will eventually converge to a consistent state.

This shift from ACID to BASE is a fundamental characteristic of data management in microservice architectures. The strong guarantees of ACID are consciously traded for enhanced availability and consequently better performance. This allows microservice systems to prioritize continuous operation, acknowledging that data will eventually propagate and reconcile throughout the system. This strategic choice is crucial for designing the resilient and scalable applications that are characteristic of modern microservices.

2.4.4 Distributed Transactions

Transactions are the most challenging aspect of building a microservice-based system. As already discussed, in microservices, we abandon the idea of a single, shared database in favor of a decentralized data management approach, where each service owns its own data store. The consequence of this shift is that we can no longer rely on a single database to manage transactions across the entire application, therefore we no longer have the luxury of relying on a relational database management system (RDBMS) that handles transactions compliantly with ACID properties. As a result, maintaining data consistency in transactions that span multiple services, each with its own database, becomes a significant challenge.

2.4.4.1 Two-Phase Commit (2PC)

The most traditional approach to managing distributed transactions is the 2PC protocol [28]. 2PC is a protocol designed to ensure that all participating services in a distributed

transaction either collectively commit their changes or collectively abort them, thereby guaranteeing atomicity across multiple data stores.

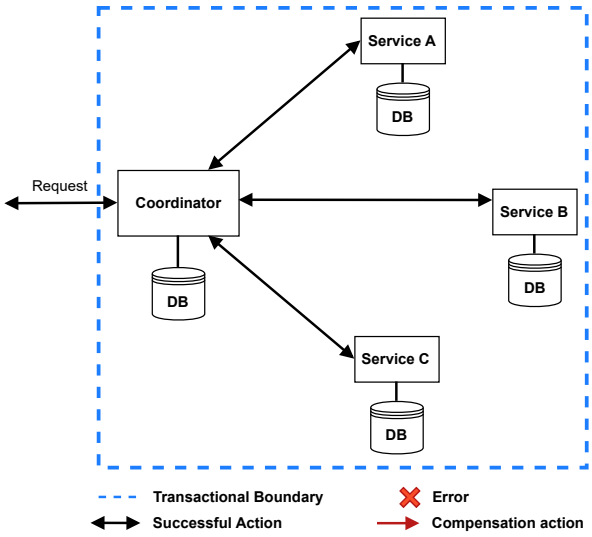


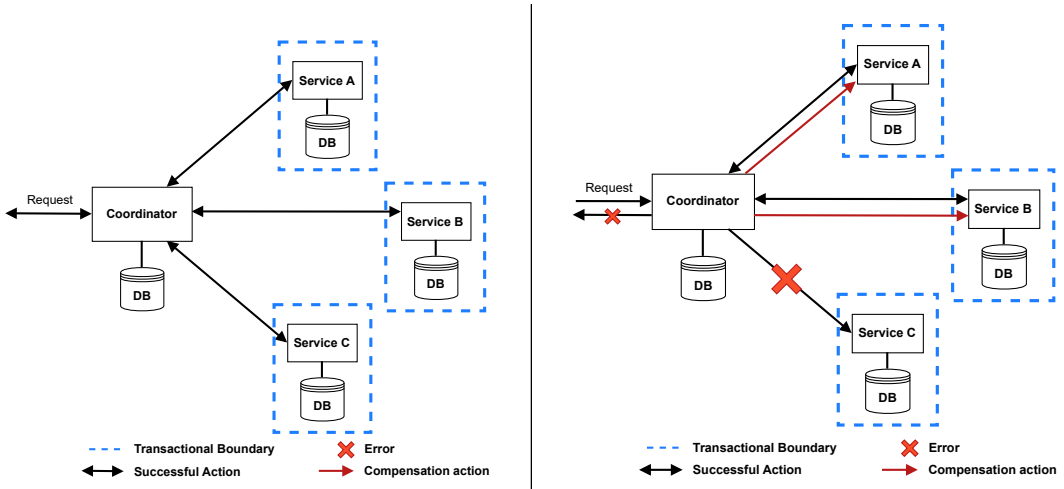
Figure 2.1: Example of a distributed transaction using the Two-Phase Commit protocol

The protocol involves a centralized coordinator service that manages the transaction across various participant services. In the first phase, often called the *prepare phase*, the coordinator contacts all participants to check if they are prepared to successfully complete their part of the transaction. Participants, after preparing their local actions, then inform the coordinator of their readiness or inability to proceed. Following this, the second phase, the *commit phase*, begins. If all participants signal their readiness, the coordinator makes a global decision to commit and instructs all participants to finalize their changes. However, if any participant indicates an issue or fails to respond, the coordinator decides to abort the transaction globally and instructs all participants to roll back their changes.

While 2PC provides strong consistency guarantees, it has significant performance drawbacks. The primary reason for 2PC's slow performance is its global transaction scope, which closely resembles a local ACID transaction by requiring all participating services to coordinate and lock resources until a global decision is reached. This, combined with the many communication steps between the coordinator and participants and the impact of network latency, often leads to significant bottlenecks within the system. Additionally, reliance on a centralized coordinator introduces a single point of failure and makes the protocol unscalable. An increasingly relevant limitation is that many modern technologies, such as NoSQL databases and message brokers, do not support 2PC [64]. These limitations make 2PC generally unsuitable for modern microservice architectures.

2.4.4.2 Sagas

Given the limitations of 2PC, we now present the concept of a saga, which is a solution for maintaining data consistency across multiple data stores without requiring a global, atomic transaction. A saga [25] can be defined as a sequence of local transactions, where each transaction executes within the context of a single service and updates its own data store. To better understand how sagas address distributed data consistency, it is helpful to examine them through three key dimensions: consistency, communication and coordination. We will start by discussing the **consistency** aspect of a saga.



(a) Orchestrated Saga

(b) Orchestrated Saga with failures

Figure 2.2: Example of a distributed transaction using a orchestrated saga

Figure 2.2 illustrates the same distributed transaction presented in the 2PC example on Figure 2.1, but this time modeled as a saga. The clear difference between the two approaches is the scope of the transaction. In the 2PC illustrated on Figure 2.1, the transaction has global scope, implying that all participating services lock their resources until the transaction is either committed or aborted, enforcing **atomic consistency** across all of them.

In the saga example on Figure 2.2(a), the scope of the transaction changes from a global scope to a local scope, where each service executes its own local transaction, updating its own data store without needing to coordinate with other services. With this change of scope, each local transaction can still be compliant with ACID properties. However, the overall business workflow is no longer fully compliant with them. We still manage to achieve atomicity by ensuring that if any local transaction fails, the saga as a whole is considered failed. In this case, we can execute countermeasures known as *compensating transactions* [25], to undo the effects of previously successful local transactions related to the saga. This is shown in Figure 2.2(b) where there was a timeout when trying to contact Service C and the saga coordinator decided to execute compensating transactions

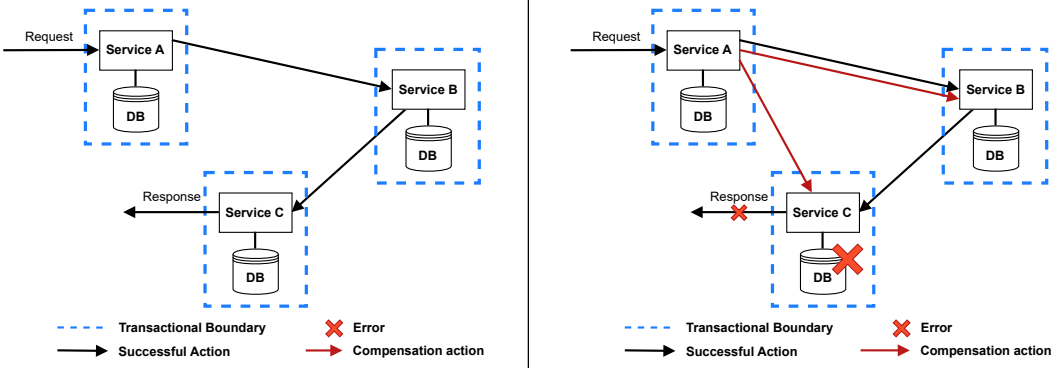
in Services A and B, which had already successfully executed their part of the saga. One important characteristic of compensating transactions is that they need to be *idempotent*. Idempotency means that executing the same operation multiple times has the same effect as if it were executed only once. This is crucial because compensating transactions may need to be retried in case of failures and the system must remains in a consistent state regardless of how many times a compensation transaction is executed.

Apart from atomicity, this also allows to achieve consistency, as each local transaction is designed to bring the service's data store from one valid state to another, ensuring that the business rules and constraints are respected. Durability is also preserved, as once a local transaction is committed, its changes are permanent and survive system failures. The issue lies in the fact that we cannot guarantee isolation at a global level. Since its possible to have many local transactions executing concurrently across different services, one saga may view the partial results of another saga [25] that is still in progress in some other service. A common approach to implement sagas is to prioritize availability and scalability by adopting BASE properties at a global level, favoring **eventual consistency**, but maintaining ACID properties at the local level to enhance data integrity within each service.

Communication is the next aspect to consider. Sagas can be implemented using either synchronous or asynchronous communication models, depending on the specific requirements of the application. Sometimes, a mixture of both models is the ideal solution, where some services communicate synchronously via RPCs while others use asynchronous messaging. This choice of communication model can be influenced by the nature of the business workflow, but in most cases, it represents a trade-off between performance and complexity.

The final aspect to consider is **coordination**. Coordination is responsible for determining the order of the saga steps and for handling compensating transactions in case of failures. There are two main approaches to coordinating sagas: orchestration and choreography. Starting with orchestration, this approach involves a centralized coordinator that manages the saga's execution as if it were a state machine. Here this central service is responsible for initiating the saga, coordinating the execution of local transactions, and handling compensating transactions in case of failures. This is the approach shown in Figure 2.2. Given this centralized nature, designing, implementing and monitoring the saga becomes easier, as the coordinator service has a global view of the saga's state and can easily track the progress of each local transaction. The drawback of this approach is the risk of centralizing too much logic in the orchestrator service. A way to address this is to keep the orchestrator as simple as possible, purely coordinating the saga's execution without implementing business logic [64]. Orchestration is often the preferred approach for implementing sagas, especially in scenarios where the business workflow is complex and scaling is not a primary concern.

Choreography, on the other hand, is a decentralized approach where each service involved in the saga knows what to do. This model is commonly implemented using an event-driven architecture, where services publish events to a message broker and subscribe to events from other services. An example of this is shown in Figure 2.3.



(a) Choreographed Saga

(b) Choreographed Saga with failures

Figure 2.3: Example of a distributed transaction using a choreographed saga

In this model, each service is responsible for reacting to events by executing its own local transaction and publishing events to notify other services of its completion. This approach allows for a more loosely coupled and scalable saga implementation at the cost of much higher complexity. The complexity arises from the lack of a central place where the saga's state is managed, making it harder to track the progress of the saga and to handle compensating transactions. One way to address this complexity is to use the first service in the saga to maintain the saga's state. This can be achieved by having the first service subscribe to acknowledgment events related to the saga request, and filter them based on the request's correlation ID [65]. This allows the first service to cancel the saga if it detects that a compensating transaction is needed, for instance, if a timeout threshold is reached.

Summary: Managing distributed transactions in microservice architectures represents a hard challenge, culminating all the challenges discussed up to this point on the chapter. The implementation of sagas, from their chosen consistency model and inter-service communication style to their coordination strategy, is essentially a series of critical trade-offs. There is no silver bullet solution that fits all scenarios, as the most suitable approach comes from a deep understanding of the system requirements and the specific business domain.

2.5 Microservice Patterns

Microservice patterns are established solutions to solve common problems that arise when building and running distributed systems. These patterns represent recurring designs that help manage complexity, enhance qualities such as resilience and scalability, and help general development practices within a microservice architecture. In this section we will explore some of the most common and important microservice patterns.

2.5.1 Service Discovery

Service discovery is a mechanism that allows a service to call another service without needing to know its network location in advance [8]. The necessity for this mechanism arises from the dynamic nature of microservice architectures. Unlike monolithic applications, where service locations might be static, microservice instances frequently change their network locations. This constant change is mainly due to the deployment of microservices in virtualized or containerized environments [8], where instances can be dynamically scaled up or down, removed due to failures, or replaced with newer versions of the service. In any service discovery mechanism, there is a core component named *service registry* [65], which acts as a database containing the network locations of all available service instances. There are two primary ways for implementing service discovery, differing on how microservices interact with the service registry:

Direct Interaction with the Service Registry: This approach combines *client-side discovery* with *self-registration* [65]. In the client-side discovery part, the client service is directly responsible for querying the service registry to fetch the network locations of available instances for the service it wants to call. Upon receiving them, the client then directly invokes that service and normally caches the results for subsequent requests to avoid repeated lookups. This allows for client-side load balancing, where the client can choose from multiple available instances. This approach saves an extra hop that would be needed to go through an intermediary load balancer, but it requires that the client implements the logic for load balancing. For service registration, the self-registration pattern is used, where each service instance registers itself in the service registry upon startup and deregisters when it shuts down. This places the responsibility on the service itself to manage its presence in the registry, often by sending periodic *"heartbeats"* to the registry to confirm its availability.

Infrastructure-Managed Service Discovery: This approach offloads service discovery responsibilities to the deployment infrastructure. This pattern combines *server-side discovery* with *third-party registration* [65]. Server-side discovery involves the client making a request to an intermediary component, such as a load balancer or API gateway. This intermediary then queries the service registry and routes the request to an available service instance

on behalf of the client. This simplifies the client's architecture, as the discovery logic is centralized within the intermediary, but it introduces an additional hop in the request path and a dependency on this intermediary component. For service registration, its used the third-party registration pattern. Here, a separate component, often called a registrar or controller, actively monitors service instances and automatically registers or deregisters them with the service registry. This approach offloads registration concerns from individual services, making it a common choice in container orchestration platforms like Kubernetes.

2.5.2 Externalized Configuration

The externalized configuration pattern decouples the service configuration from the service codebase [66]. It involves storing a service's configuration outside of its deployable unit and making it accessible at runtime. This approach is crucial in microservice architectures due to several factors. Firstly, applications typically need to be deployed across various environments, such as development, testing, and production, with each environment requiring different configurations. Secondly, this issue becomes even more significant for larger services, as embedding the configuration within the codebase requires rebuilding and redeploying the service whenever a configuration change is needed. This impacts development velocity and operational efficiency, as it can lead to unnecessary service restarts and consequent downtime.

While achieving externalized configuration can be readily accomplished using environment variables or configuration files, it is often beneficial to centralize configuration management in a dedicated service or system, often referred to as a *configuration Server*. Using a configuration server allows for a unified and consistent way to manage configurations across multiple services [66]. When combined with a secure storage solution, it can also enhance security by keeping sensitive information like credentials separate from the service codebase. Another major benefit of using a configuration server is that it provides a mechanism for dynamic reconfiguration of services at runtime without requiring a service restart.

2.5.3 API Gateway

The API Gateway serves as a single, unified entry point for all external client requests into a microservice architecture [67]. Its primary role is to abstract the complex internal structure of the microservice system from external clients, providing a simplified interface for clients to interact with the system. This abstraction is particularly important in microservice architectures, where multiple services may be involved in fulfilling a single client request. This form of request aggregation is often referred to *API composition* [67], where the gateway makes multiple calls to various internal services and combines their responses into a single response for the client. Beyond acting as a reverse proxy for request routing, an API

Gateway typically centralizes functionalities that would otherwise complicate individual microservices. It commonly handles authentication and authorization, verifying client identity, and doing preemptive checks on user permissions before requests are forwarded. It can also implement rate limiting, controlling the number of requests a client can make within a specific time frame to prevent abuse and ensure fair resource allocation. Additionally, it can also centralize request logs and metrics at the entry point to provide a clear view of external traffic patterns.

2.5.4 Resilience

Due to microservice architectures distributed nature, they are inherently susceptible to issues such as network latency, partial failures, and service unavailability. To maintain continuous operation and high service quality in these environments, resilience patterns are essential. The following patterns play a crucial role in enhancing the stability and robustness of distributed systems.

Bulkhead: The Bulkhead pattern isolates resources used by different components or services to prevent a failure in one from affecting others. Inspired by the compartmentalized design of ships [52], this pattern ensures that if one section of the system experiences a problem, it does not exhaust shared resources or consume all available capacity for other, unrelated operations. This is often achieved by limiting resources such as thread pools, connection pools, or, more effectively, by deploying different services to separate machines, providing fault isolation and improving overall system stability.

Retry: The Retry pattern allows a client to re-attempt a failed operation after a short delay. This is particularly useful for handling transient errors, such as temporary network errors or brief service unavailability, which might resolve themselves quickly. Effective implementation of the Retry pattern requires careful consideration of a maximum number of retries and a backoff strategy, which gradually increases the delay between successive attempts. This prevents overwhelming a struggling service with repeated requests and allows it time to recover.

Timeout: This is a client-side mechanism that ensures an operation does not take longer than a predefined duration to complete. If a response is not received within the specified time limit, the operation is automatically aborted. Timeouts are crucial in distributed systems to prevent clients from waiting indefinitely for a response, which could otherwise lead to resource exhaustion, such as tying up threads or connections, and propagate delays throughout dependent services.

Fallback: The Fallback pattern provides an alternative execution path when a primary operation fails or a service is unavailable. Instead of simply returning an error to the

client, a fallback mechanism allows for graceful degradation, providing a degraded but still functional response. This could involve returning cached data, default values, or redirecting to an alternative service instance. Fallbacks help improve the user experience by providing some level of functionality even during partial system failures.

Circuit Breaker: The Circuit Breaker pattern [52] is a foundational resilience mechanism that integrates elements of monitoring, timeouts and fallback to prevent cascading failures. It acts as a proxy for operations that might fail, preventing a client from repeatedly invoking a service that is likely to be unavailable or overloaded. By monitoring calls to a service, if the failure rate exceeds a predefined threshold, the circuit "opens", causing all subsequent calls to fail immediately without attempting to contact the service. After a configurable timeout, the circuit enters a "half-open" state, allowing a limited number of requests through to determine if the service has recovered. If these test requests succeed, the circuit "closes", resuming normal operation, otherwise, it returns to the "open" state. This pattern not only protects the consumer from waiting on a failing service but also gives the struggling service time to recover by reducing its load.

2.5.5 Observability

In complex distributed environments, observability is vital because it's not easy to know what is happening inside the numerous interconnected services. To achieve comprehensive observability in microservice systems, various mechanisms have become standard practices to gather and analyze data from across the application. Some of the most common mechanisms include:

Health Check API: This is a mechanism that involves implementing an API endpoint on each microservice to returns its health status [66]. Monitoring systems and service registries can use this endpoint to determine if an instance is alive and ready to handle requests. The status returned can simply be "up" or "down" or more detailed, including the health of underlying resources, like database connections or external service dependencies.

Application Metrics: Application metrics consist of quantitative data that reflect various aspects of a service behavior over time [66]. The collected metrics can be performance metrics such as average response time and throughput, or resource usage metrics such as CPU usage and memory consumption. These metrics offer valuable insights into monitoring the health of the system and identifying potential issues.

Log Aggregation: Application log aggregation is crucial for troubleshooting microservices because a single request can generate log entries across multiple services [66]. By centralizing all application logs in a searchable repository, it becomes possible to correlate related log events for the same request, often using a unique correlation ID that is included

in all log entries related to that request.

Distributed Tracing: Distributed tracing represents the end-to-end journey of a single request as it flows through multiple services in a distributed system [51]. The pattern works by assigning each external request a unique identifier, often called a correlation ID, which is propagated along side all subsequent inter-service calls. Each operation within a service generates a span, and these spans are linked together using the correlation ID to reconstruct the full path and timing of the request. This allows operators to visualize the request flow and analyze service latency across service boundaries, which is invaluable for debugging performance issues and understanding complex inter-service dependencies.

2.6 Autonomic Computing

With the growing complexity, scale, and heterogeneity of modern computing systems, their management has become an increasingly difficult challenge, often becoming unfeasible human oversight [57]. In response to this escalating challenge, IBM introduced the vision of autonomic computing [31]. Inspired by the human autonomic nervous system, which self-regulates vital bodily functions unconsciously [57], this vision aimed to imbue systems with capabilities that would allow them to manage themselves with minimal human intervention. In this section, we will explore the core principles of autonomic computing, starting with its self-managing properties, followed by a common framework for implementing these systems, known as the MAPE-K control loop, and will also focus on how these principles can be applied to microservice architectures.

2.6.1 Self-* Properties

The vision of autonomic computing is built upon four core properties, known as self-* properties [41, 31], each contributing to a system's ability to operate autonomously and adaptively:

Self-Configuration: This property refers to a system's ability to automatically configure itself in response to human-defined high-level goals, which focus on the state or behavior that the system should achieve. It involves dynamic adjustments and adaptations to its internal structure or parameters.

Self-Healing: A system can autonomously detect, diagnose and recover from disruptions or failures within its components. It aims to minimize downtime and maintain continuous operation by identifying problems and initiating corrective actions.

Self-Optimization: This refers to a system's ability to continuously tune its resources to achieve optimal performance and efficiency. A self-optimizing system dynamically

adjusts its parameters and resource allocation in response to changing workloads or, environmental conditions, striving to meet performance targets and improve overall resource utilization.

Self-Protection: This property refers to a system's ability to anticipate, detect, and defend itself against internal or external attacks and threats. It involves measures to safeguard its integrity, privacy, and security by automatically responding to detected anomalies or malicious activities.

2.6.2 MAPE-K

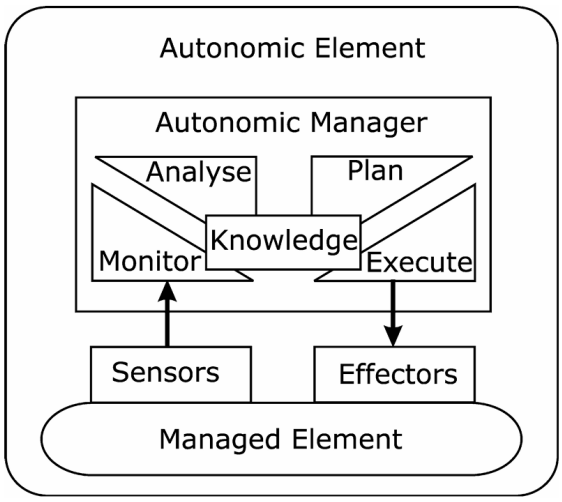


Figure 2.4: MAPE-K Control Loop (Image taken from [33])

The *MAPE-K* [34] loop was proposed by IBM and was adopted as a reference model for structuring autonomic systems [7]. MAPE-K is a control loop which consists of its four computational steps: *Monitor*, *Analyze*, *Plan*, and *Execute*, all operating over a shared *Knowledge* base [2]. This loop is an integrant part of an *autonomic manager* [34, 33] and represents a continuous cycle that enables the system to observe its own state and environment and adaptively respond to changes. These self-adaptive systems rely on *autonomic elements*. This can be any computing resource, like a database or a hardware component such as a disk [40]. Each autonomic element is made up of a *managed element*, which is the actual software or hardware resource that gains its autonomic capabilities when integrated with an *autonomic manager*.

The initial phase of the loop is executed by the *Monitor* component, which is responsible for gathering detailed information about the system's operational state and its surrounding environment [41, 33], and for storing it in the *Knowledge* component. The data is collected by *sensors*, which monitor the system's components *autonomic elements*.

After data collection, the *Analyze* component examines the monitored information to detect problems, identify optimization opportunities, or recognize deviations from

the desired state that may require adaptation. If the analysis indicates that changes are necessary, the *Plan* component devises a strategy to resolve the identified issues, aiming to restore system stability or enhance performance [41].

Subsequently, the *Execute* component is responsible for implementing the planned actions and applying the required changes within the system. This phase involves managing the concrete execution of reconfigurations, resource adjustments, or operational modifications through *actuators* [7]. It ensures that the adaptations devised by the *Plan* component are accurately and reliably applied to the running system, thereby completing the autonomic control loop.

Throughout the continuous operation of the MAPE control loop, the *Knowledge* Base acts as a central repository of context and information [34]. This shared knowledge encompasses policies, rules, historical data, and the current configuration and topology of autonomic elements. It is an essential resource that informs each phase of the MAPE loop, enabling intelligent, context-aware decision-making.

2.6.3 Planning and Analysis Approaches

The Analyze and Plan phases are central to decision-making and can employ several distinct approaches:

Policy-Based Planning: This approach, often realized through *Event-Condition-Action (ECA) rules*, relies on predefined "if-then" logic to govern adaptation actions [49]. In ECA rules, an *event* triggers the evaluation of a *condition*, and if the condition is satisfied, a predefined *action* is executed. For example, in a microservices environment, if a service observes that messages are accumulating because its polling interval for a Kafka topic is too long, the rule engine can automatically reduce the polling interval in the service's configuration to increase throughput. The main advantage of policy-based planning is its simplicity and predictability, making it transparent and easy to implement for well-understood adaptation scenarios. However, as the number of rules grows, management becomes more complex, and this approach is less effective at handling unforeseen situations or learning new optimal behaviors, often requiring manual updates [33]. In microservice architectures, this approach is particularly suitable for initial autonomic implementations that address clear, threshold-driven adaptations.

Architectural Model-Based Planning: This approach utilizes a formal model of the managed system and its environment to interpret system state, predict the effects of potential changes, and evaluate various adaptation strategies against desired goals or constraints [33, 40]. The model acts as a simulation environment where different adaptation plans can be tested to understand their potential outcomes before they are applied. The main benefit is the ability to explore complex scenarios and predict side effects, leading to more robust and optimized adaptation strategies. The significant drawbacks are the complexity of

creating and maintaining an accurate model of a dynamic distributed system, as well as the computational overhead required to run simulations [33]. Additionally, this approach is generally slower than rule-based planning because adaptation plans must be tested against the model before being applied.

Learning-Based Planning: This approach leverages artificial intelligence and machine learning techniques to enable the system to learn optimal adaptation strategies from historical data and runtime observations [33]. Machine learning algorithms are used to detect complex patterns, predict future states, or identify root causes. Based on learned behaviors or predicted outcomes, the system can then generate adaptation plans. The key advantages are the ability to handle highly complex and unforeseen scenarios, discover non-obvious optimal strategies, and continuously improve over time. The main benefits include effectively addressing a wide range of complex and unforeseen situations, identifying innovative solutions, and enabling ongoing adaptation and enhancement.

However, this approach requires significant amounts of high-quality data for training, and can sometimes make decision-making less transparent or interpretable [33].

2.7 Summary

This chapter critically explored the architectural evolution of system architectures from earlier monolithic approaches to modern distributed systems. While the monolithic paradigm served its purpose, the increasing demands of today's applications, particularly complex systems like CRM platforms, make them unsuitable for modern requirements. The examination of these established and emerging paradigms highlighted the critical necessity of balancing modularity, scalability, and resilience against the inherent complexity of distributed environments.

The architectural choices and patterns reviewed in this chapter were deliberately selected for their direct relevance to the challenges faced by microservice architectures, and for their applicability within this dissertation. For instance, while traditional approaches like 2PC were discussed to illustrate pitfalls in distributed transactions, the adoption of patterns like Sagas are favored to ensure data consistency in highly scalable microservice contexts. Similarly, we examined various microservice patterns as essential tools for building robust and manageable distributed systems.

Ultimately, this discussion highlights the critical role of autonomic principles in managing the complexity of microservice operations, empowering systems to self-regulate and adapt. The autonomic computing paradigm provides systems with self-managing capabilities, enabling them to respond dynamically to changing conditions with minimal human intervention. While all self-managing capabilities are important for autonomic systems, self-optimization remains less explored in the context of microservices. Consequently, this work emphasizes self-optimization and adopts a policy-based planning approach using ECA rules.

3

Planning

3.1 Proposed Solution

As stated in Chapter 1, the core of this work involves the design and implementation of a microservice-based CRM application with built-in autonomic capabilities. The proposed solution is structured around two main stages: the implementation of the CRM itself and its integration with autonomic management features.

3.1.1 CRM Implementation

This first stage focuses on the creation of a functional prototype that has the essential features of a CRM application. We will implement a focused selection of core functionalities in the prototype to effectively highlight the autonomic management features. The core functionalities of the prototype will be:

User management: This feature will comprise actions related to user registration, authentication, access control policies and profiles management.

Customer management: This feature will allow to manage customer information, including contact details and interaction history.

Project management: This feature will allow creating and managing projects, including tasks, deadlines and team assignments.

Communication services: This feature includes emailing and reporting capabilities, enabling users to communicate with customers and team members effectively.

Analytics and Reporting: This feature allows for the generation of reports and dashboards to analyze customer data, project progress and team performance.

Figure 3.1 illustrates the architecture of the CRM application. The core features will be implemented using the Spring Boot framework [75] with Kotlin [38] as the main programming language, taking advantage of Spring's robust ecosystem for building microservices. The CRM backend includes an API gateway, which serves as the entry

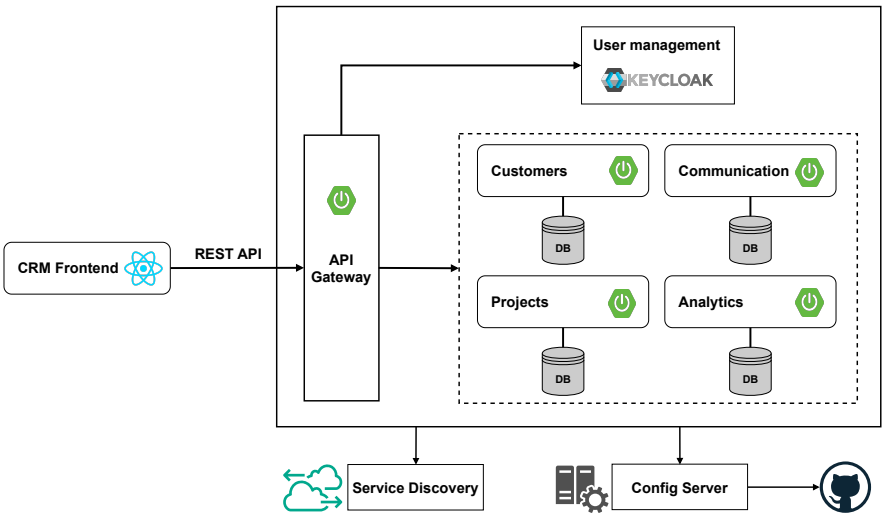


Figure 3.1: CRM Architecture Overview

point for client requests and is responsible for handling authentication and authorization via Keycloak [21] identity and access management system.

As illustrated in Figure 3.1, the CRM application will be designed as a collection of microservices, with each service dedicated to a distinct domain or functionality. Adopting the database per service pattern [17], every microservice will operate with its own dedicated database instance. This design allows each service to choose the most suitable database technology for its operational needs, providing flexibility that ensures optimal data modeling and performance across the varied requirements of the CRM system.

Inter-service communication will be tailored to the business context, employing either synchronous or asynchronous protocols. For operations that require immediate response and coordination, synchronous communication will be implemented using REST [14] or gRPC [27], enabling direct exchanges between services. In scenarios where decoupling and scalability are prioritized, asynchronous communication will be achieved through Kafka [43], supporting event-driven workflows and robust message handling.

The backend will provide a comprehensive RESTful API, serving as the integration point for the web-based user interface. The frontend will be delivered as a modern Single-page application (SPA) built with the React framework [60], ensuring a responsive and interactive user experience that seamlessly connects to the underlying microservices architecture.

3.1.2 ATOM

The integration of the autonomic capabilities will reside in a centralized service, architecturally independent of the CRM application, which we refer to as *ATOM*. This dedicated component consumes metrics and telemetry data collected by the monitoring infrastructure to orchestrate adaptive actions across the microservices environment, as conceptually illustrated in Figure 3.2.

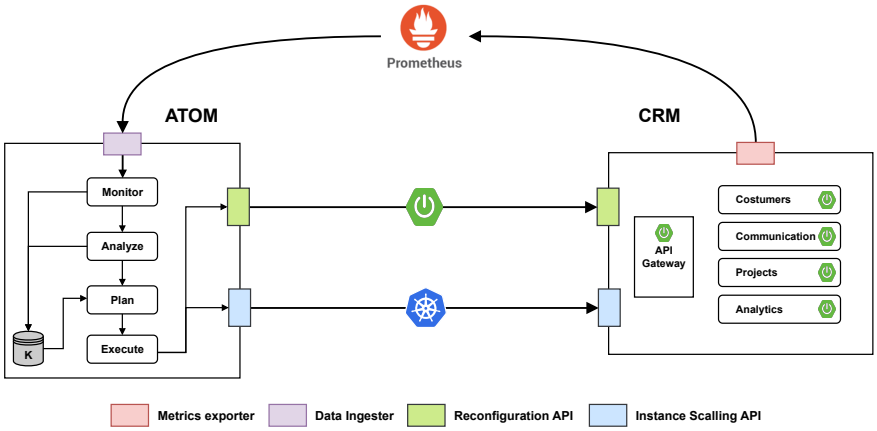


Figure 3.2: Conceptual Overview of CRM with Autonomic Management

The initial phase, also called *monitor* phase, involves establishing observability for the system, enabling the continuous collection of operational metrics and telemetry data from both the CRM application and its runtime environment. To achieve comprehensive observability, the system leverages a Spring Boot feature called Actuator, which provides production-ready endpoints for monitoring and managing the application. The actuator when combined with Micrometer [6], which is the core library to instrument the application code, enables the collection of detailed metrics from within the Spring Boot services with minimal overhead. These collected metrics can include Java Virtual Machine performance indicators, database query performance and custom application-specific metrics. In addition, distributed tracing is implemented using the OpenTelemetry protocol [22], which captures end-to-end request flows and service interactions across the microservices architecture. The system also collects infrastructure-level metrics directly from the Kubernetes environment. All collected metrics and traces will be exported to Prometheus [23], which serves as backend time-series database for storage, querying, and visualization of the collected data. For visualization, Grafana [45] will be used to create dashboards that provide insights into the system's performance and health. This combination ensures robust, real-time monitoring across both the application and its orchestration platform, forming the foundation for subsequent analysis and autonomic adaptation.

The *analyze* and *plan* phases are tightly integrated within ATOM. During the *analyze* phase, the system processes the collected metrics and telemetry data to evaluate the application's current state against established performance thresholds and operational policies. Building on these insights, the *plan* phase is responsible for devising appropriate reconfiguration strategies. Here, the planning logic is primarily policy-based, enabling the system to autonomously select adaptation actions. For instance, the system can adjust the number of running instances for specific microservices based on real-time workload metrics. While the current implementation emphasizes policy-based planning for clarity and demonstrability, the architecture is intentionally designed to be extensible. This

allows for the future integration of advanced AI or Machine Learning techniques, which could enable predictive analytics, dynamic policy generation, and more sophisticated adaptive behaviors as the system evolves, thereby reducing even further the need for manual intervention. The planned system adjustments are put into effect by applying the chosen adaptation strategies directly to the running environment. These interventions can be categorized into two main types:

Configuration Parameter Adjustment: The system enables dynamic updates to configuration parameters for individual microservices or groups of services. These adjustments may involve changing operational specific application parameters such as timeouts, number of retries, pooling intervals or other settings that can better align with current workload and performance demands. Configuration changes are managed in a centralized manner and automatically distributed to the relevant services. Once notified of an update, each service retrieves the latest configuration and applies the new settings immediately, without requiring a restart.

Service Instance Scaling: The system autonomously manages the number of running instances for specific microservices by interfacing directly with the Kubernetes API. This enables both scaling up and down in response to real-time workload or performance metrics. For demonstration purposes, this approach deliberately bypasses Kubernetes built-in Horizontal Pod Autoscaler (HPA), allowing the custom decision logic of the autonomic manager to be showcased in controlling the scaling process.

3.2 Evaluation

The evaluation of the proposed solution will be rely on large-scale emulation. The main objective is to assess the performance and resilience of the CRM application under autonomic management by ATOM compared to its baseline operation without autonomic capabilities. All experiments will take place in a controlled emulation environment that deploys both the CRM application and ATOM on a Kubernetes cluster to closely replicate a production environment.

The evaluation will employ a comprehensive testing methodology that reflects realistic operational scenarios for the CRM application, ensuring rigorous assessment under a wide range of conditions. JMeter [20] will be used to orchestrate various types of tests, including load tests to measure performance under high number of concurrent users activity, stress tests to determine system limits, spike tests to observe behavior during sudden surges in traffic, and volume tests to assess handling of large data sets.

During each experiment, the system's adaptive responses, including dynamic scaling of service instances and real-time configuration changes, will be carefully observed to evaluate the effectiveness of the autonomic management features. The monitoring infrastructure will continuously gather performance metrics such as service throughput, request

latency, error rates, and resource utilization, providing a rich dataset for analysis. This approach provides detailed insight into both application and infrastructure performance, supporting a thorough assessment of the autonomic management capabilities under varying workloads and helping to ensure the reliability and scalability of the proposed solution in real-world deployment scenarios.

4

Work Plan

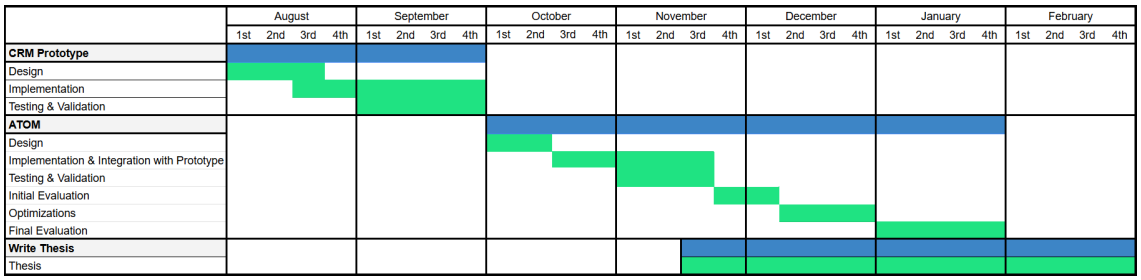


Figure 4.1: Gant chart of the expected work plan

In Figure 4.1, we show the expected work plan for the dissertation. The proposed work plan begins with the development of a CRM prototype. This phase starts with the design of the system, where the architecture and its key components are defined. Once the design is established, the implementation phase follows, focusing on building the individual microservices and core functionality required by the CRM system. After the prototype is implemented, it will be tested to ensure that it meets the functional requirements and performs as expected.

Following the implementation of the CRM prototype, attention shifts to the development of *ATOM*. The work on *ATOM* starts with its design, where the self-management logic, monitoring components, and decision-making strategies are outlined. Once the design is complete, the implementation phase begins, during which *ATOM* is developed and integrated with the existing CRM prototype to enable runtime adaptability and control.

After integration, the system undergoes testing and validation to ensure correct functionality and interaction between *ATOM* and the CRM prototype. Initial evaluation of the integrated system is then conducted to assess its effectiveness. Based on the outcomes, optimization activities are carried out to refine performance and improve autonomic behavior. A final evaluation follows to confirm that the system meets its intended goals.

The writing of the document will overlap some of the final work stages, with the last month being reserved for its conclusion.

Bibliography

[1] Amazon. *Horizontal scaling*. URL: https://wa.aws.amazon.com/wellarchitected/2020-07-02T19-33-23/wat.concept.horizontal-scaling.en.html (cit. on p. 7).

[2] P. Arcaini, E. Riccobene, and P. Scandurra. "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation". In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2015, pp. 13–23. DOI: 10.1109/SEAMS.2015.10 (cit. on p. 25).

[3] K. Barde. "Modular Monoliths: Revolutionizing Software Architecture for Efficient Payment Systems in Fintech". In: *International Journal of Computer Trends and Technology* 71 (2023-09), pp. 20–27. DOI: 10.14445/22312803/IJCTT-V71I10P103 (cit. on p. 7).

[4] D. Box et al. *SOAP: Simple Object Access Protocol (SOAP) 1.1*. https://www.w3.org/TR/2000/NOTE-SOAP-20000508/. W3C Note. 2000 (cit. on p. 11).

[5] E. Brewer. "Towards robust distributed systems". In: 2000-07. DOI: 10.1145/343477.343502 (cit. on p. 13).

[6] Broadcom. *Micrometer*. URL: https://micrometer.io/ (cit. on p. 30).

[7] A. Bucchiarone et al. "A MAPE-K Approach to Autonomic Microservices". In: 2022-03, pp. 100–103. DOI: 10.1109/ICSA-C54293.2022.00025 (cit. on pp. 25, 26).

[8] S. Cusimano. *Service Discovery in Microservices*. 2025-03. URL: https://www.baeldung.com/cs/service-discovery-microservices (cit. on p. 20).

[9] I. Docker. *Docker*. URL: https://www.docker.com/ (cit. on p. 9).

[10] V. Dudas. *Understanding The Application Of CRM Systems In Business*. 2024-10. URL: https://www.forbes.com/councils/forbestechcouncil/2024/10/22/grow-faster-and-smarter-with-data-driven-business-decisions/ (cit. on p. 3).

[11] E. Evans. *Domain-Driven Design. Tackling Complexity in the Heart of Software*. 1st ed. Addison-Wesley Professional, 2003-08. Chap. 14. ISBN: 9780321125217 (cit. on p. 8).

[12] Facebook. *GraphQL*. https://spec.graphql.org/. 2015 (cit. on p. 11).

[13] D. Faustino et al. "Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation". In: *Performance Evaluation* 164 (2024-03), p. 102411. DOI: 10.1016/j.peva.2024.102411 (cit. on pp. 7, 8).

[14] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Ph.D. thesis. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (cit. on pp. 11, 29).

[15] B. Foote and J. Yoder. "Big Ball of Mud". In: (1997-08). URL: https://s3.amazonaws.com/systemsandpapers/papers/bigballofmud.pdf (cit. on p. 7).

[16] N. Ford et al. *Software Architecture: The Hard Parts. Modern Trade-Off Analyses for Distributed Architectures*. 1st ed. O'Reilly Media, 2021-10. Chap. 6. ISBN: 9781492086895 (cit. on p. 6).

[17] N. Ford et al. *Software Architecture: The Hard Parts. Modern Trade-Off Analyses for Distributed Architectures*. 1st ed. O'Reilly Media, 2021-10. Chap. 7. ISBN: 9781492086895 (cit. on pp. 9, 29).

[18] N. Ford et al. *Software Architecture: The Hard Parts. Modern Trade-Off Analyses for Distributed Architectures*. 1st ed. O'Reilly Media, 2021-10. Chap. 8. ISBN: 9781492086895 (cit. on p. 12).

[19] N. Ford et al. *Software Architecture: The Hard Parts. Modern Trade-Off Analyses for Distributed Architectures*. 1st ed. O'Reilly Media, 2021-10. Chap. 2. ISBN: 9781492086895 (cit. on p. 12).

[20] A. S. Foundation. *Apache JMeter*. URL: https://jmeter.apache.org/ (cit. on p. 31).

[21] C. N. C. Foundation. *Keycloak*. URL: https://www.keycloak.org/ (cit. on p. 29).

[22] C. N. C. Foundation. *OpenTelemetry*. URL: https://opentelemetry.io/ (cit. on p. 30).

[23] C. N. C. Foundation. *Prometheus*. URL: https://prometheus.io/ (cit. on p. 30).

[24] A. Fox et al. "Cluster-Based Scalable Network Services". In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, 1997, pp. 78–91 (cit. on p. 15).

[25] H. Garcia-Molina and K. Salem. "Sagas". In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. SIGMOD '87. San Francisco, California, USA: Association for Computing Machinery, 1987, pp. 249–259. ISBN: 0897912365. DOI: 10.1145/38713.38742. URL: https://doi.org/10.1145/38713.38742 (cit. on pp. 17, 18).

[26] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (2002-06), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601 (cit. on p. 13).

[27] Google. *gRPC: A high performance, open-source universal RPC framework*. https://grpc.io/. 2016 (cit. on pp. 11, 29).

[28] J. Gray. "Notes on Data Base Operating Systems". In: *Operating Systems, An Advanced Course*. Berlin, Heidelberg: Springer-Verlag, 1978, pp. 393–481. ISBN: 3540087559. DOI: 10.5555/647433.723863 (cit. on p. 15).

[29] A. W. Group. *AMQP, Protocol specification*. 2008. URL: https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf (cit. on p. 11).

[30] T. Haerder and A. Reuter. "Principles of transaction-oriented database recovery". In: *ACM Computing Surveys (CSUR)* 15.4 (1983), pp. 287–317 (cit. on p. 14).

[31] P. Horn. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. Tech. rep. IBM Corporation, 2001 (cit. on p. 24).

[32] HubSpot. *Hubspot*. URL: https://www.hubspot.com/ (cit. on p. 5).

[33] M. C. Huebscher and J. A. McCann. "A survey of autonomic computing—degrees, models, and applications". In: *ACM Comput. Surv.* 40.3 (2008-08). ISSN: 0360-0300. DOI: 10.1145/1380584.1380585. URL: https://doi.org/10.1145/1380584.1380585 (cit. on pp. 25–27).

[34] IBM. "An Architectural Blueprint for Autonomic Computing". In: (2005-06) (cit. on pp. 2, 25, 26).

[35] IBM. *Microservices*. 2021-09. URL: https://www.ibm.com/think/topics/microservices (cit. on p. 2).

[36] M. R. Intellect. *Cloud Microservices Market Set to Reach USD 22.1 Billion by 2031, Driven by Digital Transformation and Scalability Needs*. 2024-12. URL: https://www.prnewswire.com/news-releases/cloud-microservices-market-set-to-reach-usd-22-1-billion-by-2031--driven-by-digital-transformation-and-scalability-needs---market-research-intellect-302327537.html (cit. on p. 2).

[37] Y. Izrailevsky, S. Vlaovic, and R. Meshenberg. *Completing the Netflix Cloud Migration*. 2016-02. URL: https://about.netflix.com/en/news/completing-the-netflix-cloud-migration (cit. on p. 2).

[38] JetBrains. *Kotlin*. URL: https://kotlinlang.org/ (cit. on p. 28).

[39] Jophin. *How Microservices are Revolutionizing the IT Landscape?* 2023-11. URL: https://www.fortunesoftit.com/how-microservices-are-revolutionizing-the-it/ (cit. on p. 2).

[40] J. Kephart. "Research challenges of autonomic computing". In: 2005-01, pp. 15–22. DOI: 10.1109/ICSE.2005.1553533 (cit. on pp. 25, 26).

[41] J. Kephart and D. Chess. "The Vision Of Autonomic Computing". In: *Computer* 36 (2003-02), pp. 41–50. DOI: 10.1109/MC.2003.1160055 (cit. on pp. 24–26).

[42] M. Kleppmann. *Designing Data-Intensive Applications. The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. 1st ed. O'Reilly Media, 2017-03. Chap. 9. ISBN: 9781491903100 (cit. on p. 14).

[43] J. Kreps, N. Narkhede, and J. Rao. "Kafka: A Distributed Messaging System for Log Processing". In: 2011. URL: https://notes.stephenholiday.com/Kafka.pdf (cit. on pp. 12, 29).

[44] Kubernetes. *Kubernetes Documentation*. URL: https://kubernetes.io/docs/concepts/overview/ (cit. on p. 2).

[45] G. Labs. *Grafana*. URL: https://grafana.com/ (cit. on p. 30).

[46] L. Lamport. *Distributed Systems*. 1987-05. URL: https://lamport.azurewebsites.net/pubs/distributed-system.txt (cit. on p. 8).

[47] J. Lewis and M. Fowler. *Microservices*. 2014-05. URL: https://martinfowler.com/articles/microservices.html (cit. on p. 8).

[48] M. Loukides and S. Swoyer. *Microservices Adoption in 2020*. 2020-07. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/ (cit. on p. 2).

[49] D. McCarthy and U. Dayal. "The architecture of an active database management system". In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. Portland, Oregon, USA: Association for Computing Machinery, 1989, pp. 215–224. ISBN: 0897913175. DOI: 10.1145/67544.66946. URL: https://doi.org/10.1145/67544.66946 (cit. on p. 26).

[50] S. Newman. *Building Microservices. Designing Fine-Grained Systems*. 2nd ed. O'Reilly Media, 2021-08. Chap. 1. ISBN: 9781492034025 (cit. on pp. 1, 8, 9).

[51] S. Newman. *Building Microservices. Designing Fine-Grained Systems*. 2nd ed. O'Reilly Media, 2021-08. Chap. 10. ISBN: 9781492034025 (cit. on p. 24).

[52] M. T. Nygard. *Release It! Designing and Deploying Production-Ready Software*. 1st ed. Pragmatic Bookshelf, 2018-01. Chap. 5. ISBN: 9781680502398 (cit. on pp. 22, 23).

[53] O'Reilly. *Survey reveals the opportunities and realities of microservices*. 2018-12. URL: https://www.oreilly.com/content/survey-reveals-the-opportunities-and-realities-of-microservices/ (cit. on p. 2).

[54] Oracle. *Oracle CRM*. URL: https://www.oracle.com/oce/dc/assets/CONT8595828BECB04D7D827D2517E1A269DC/native/siebel-crm-cloud-native-architecture-lpd400077424-10.pdf (cit. on p. 5).

[55] Oracle. *What is CRM? The complete CRM guide*. URL: https://www.oracle.com/dk/cx/what-is-crm/ (cit. on pp. 3, 5).

[56] A. Owen and N. Lous. "Integrating Self-Healing Mechanisms in Microservices Architecture Using Machine Learning Techniques". In: (2022-10) (cit. on p. 2).

[57] M. Parashar and S. Hariri. "Autonomic Computing: An Overview". In: vol. 3566. 2004-01, pp. 257–269. ISBN: 978-3-540-27884-9. DOI: 10.1007/11527800_20 (cit. on p. 24).

[58] R. pillai and R. Student. "A Scalable Metadata-Driven Architecture for Dynamic Workflow Automation in Multi-Tenant CRM Platforms". In: 6 (2025-05), pp. 42–48 (cit. on p. 5).

[59] Pipedrive. *Pipedrive*. URL: https://www.pipedrive.com/ (cit. on p. 5).

[60] M. Platforms. *React*. URL: https://react.dev/ (cit. on p. 29).

[61] M. Richards and N. Ford. *Fundamentals of Software Architecture. An Engineering Approach*. 1st ed. O'Reilly Media, 2020-01. Chap. 9. ISBN: 9781492043454 (cit. on pp. 1, 6).

[62] M. Richards and N. Ford. *Fundamentals of Software Architecture. An Engineering Approach*. 1st ed. O'Reilly Media, 2020-01. Chap. 16. ISBN: 9781492043454 (cit. on p. 1).

[63] M. Richards and N. Ford. *Fundamentals of Software Architecture. An Engineering Approach*. 1st ed. O'Reilly Media, 2020-01. Chap. 17. ISBN: 9781492043454 (cit. on p. 12).

[64] C. Richardson. *Microservices Patterns*. 1st ed. Manning Publications, 2018-10. Chap. 4. ISBN: 9781617294549 (cit. on pp. 13, 16, 18).

[65] C. Richardson. *Microservices Patterns*. 1st ed. Manning Publications, 2018-10. Chap. 3. ISBN: 9781617294549 (cit. on pp. 19, 20).

[66] C. Richardson. *Microservices Patterns*. 1st ed. Manning Publications, 2018-10. Chap. 11. ISBN: 9781617294549 (cit. on pp. 21, 23).

[67] C. Richardson. *Microservices Patterns*. 1st ed. Manning Publications, 2018-10. Chap. 8. ISBN: 9781617294549 (cit. on p. 21).

[68] A. Rud. *Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience*. 2019-07. URL: https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/ (cit. on p. 2).

[69] Salesforce. *Salesforce*. URL: https://www.salesforce.com/ (cit. on p. 5).

[70] D. Singh. "AI and Cloud Integration: Transforming CRM Practices". In: *INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING TECHNOLOGY* 15 (2024-11), pp. 197–210. DOI: 10.5281/zenodo.14055235 (cit. on p. 6).

[71] M. van Steen and A. S. Tanenbaum. *Distributed Systems*. 4th ed. 2023-01. Chap. 1. ISBN: 9789081540636 (cit. on p. 8).

[72] M. van Steen and A. S. Tanenbaum. *Distributed Systems*. 4th ed. 2023-01. Chap. 4. ISBN: 9789081540636 (cit. on p. 11).

[73] M. van Steen and A. S. Tanenbaum. *Distributed Systems*. 4th ed. 2023-01. Chap. 2. ISBN: 9789081540636 (cit. on p. 11).

[74] M. van Steen and A. S. Tanenbaum. *Distributed Systems*. 4th ed. 2023-01. Chap. 7. ISBN: 9789081540636 (cit. on p. 14).

[75] V. Tanzu. *Spring Boot*. URL: https://spring.io/projects/spring-boot (cit. on p. 28).

[76] W. Vogels. "Eventually Consistent: Building reliable distributed systems at a world-wide scale demands trade-offs?between consistency and availability." In: *Queue* 6.6 (2008-10), pp. 14–19. ISSN: 1542-7730. DOI: 10.1145/1466443.1466448. URL: https://doi.org/10.1145/1466443.1466448 (cit. on p. 14).