Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Optimizing Payment Systems with Microservices and Event-Driven Architecture: The Case of Mollie Platform

---

Author: Yufei Wang

VU: 2756993
UvA: 14170884

1st supervisor:	Adam S.Z. Belloum
daily supervisor:	John Zhao (Mollie B.V.)
2nd reader:	Shashikant Ilager

A thesis submitted in fulfillment of the requirements for

the joint UvA-VU Master of Science degree in Computer Science

December 4, 2024

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

The exponential growth in digital transactions has created a pressing demand for scalable, reliable, and real-time payment platforms. This thesis explores the transition from a monolithic architecture to a microservices-based and event-driven architecture using Mollie, a high-frequency payment platform. By analyzing the challenges of scalability, fault tolerance, and performance in traditional monolithic systems, this research proposes a modular approach applying microservices and asynchronous communication with event-driven architecture. The architecture is designed to enhance operational efficiency, ensure real-time processing, and adapt to the growing demands of diverse payment methods. Tools like DataDog and GCP are used for real-time monitoring and deployment, demonstrating improved reliability and maintainability. Despite advancements, challenges such as event duplication, external API dependencies, and resource overhead are also discussed. The findings contribute valuable insights into scalable and adaptable architecture for digital payment platforms.

Keywords— Payment Platform, Microservices Architecture, Event-Driven Architecture (EDA), Real-Time Monitoring, Scalability, Fault Tolerance

Contents

CONTENTS

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

As the digital development reshapes the financial landscape, it has significantly changed the way payments are processed, leading to an exponential rise in the variety and volume of payment options worldwide. Consumers nowadays demand fast, secure, and flexible payment methods, while merchants seek solutions that are reliable and capable of adapting to evolving market dynamics. Payment Service Providers (PSPs), such as Mollie(1), play a pivotal role in bridging this gap, ensuring seamless transaction experiences for both merchants and consumers. However, integrating diverse payment methods into a unified platform presents critical challenges in scalability, efficiency, and real-time performance (2). As a multi-market payment company, Mollie strives to meet these goals. Serving merchants across every industry and country in Europe, Mollie simplifies payment processing by enabling businesses to accept a wide range of payment methods, including credit cards, direct debit, Paypal, and local options tailored to specific markets. However, challenges exist alongside opportunities. As the platform scales to integrate new payment methods and handle increasing transaction volumes, it must overcome obstacles related to performance, reliability, and flexibility.

Initially based on a monolithic architecture, the Mollie platform shows several limitations, such as tightly coupled components, scalability bottlenecks, and increased risk of failures across services when adding new features. Monolithic systems struggle with performance issues as they grow, exhibit a lack of modularity, and make even minor updates complicated. For rapidly expanding domains like financial technology, where agility and adaptability are key, these limitations are unsustainable. Each new feature or payment integration requires extensive testing and development resources, even risks downtime. Additionally, monolithic systems face challenges in maintaining compliance with dynamic financial regulations and ensuring secure data management as transactions scale globally.

To overcome these challenges, we decide to adopt a microservices-based architecture to decouple functionalities, enhance fault tolerance, and enable more agile development practices. Microservices divide the platform into smaller, autonomous units, each responsible for a specific function, enabling faster development cycles, reduced downtime, and fault isolation. Additionally, an event-driven architecture (EDA) has been introduced to optimize real-time data processing and ensure responsiveness across its distributed systems. Together, these architectural shifts aim to improve Mollie's growth in the future, allowing it to handle increasing transaction volumes while integrating diverse payment methods more quickly and seamlessly.

This thesis focuses on the following research questions:

- How can microservices architecture be effectively applied to enhance the scalability and fault tolerance of payment platforms?
- What role does event-driven architecture play in enhancing real-time processing and decoupling within payment systems?
- How can monitoring and deployment pipelines be implemented to ensure system reliability?

These questions are essential to addressing the above challenges faced by Mollie's payment platform, such as the need for scalability to handle growing transaction volumes against faults to maintain uninterrupted service, and efficient error management to ensure a seamless user experience. Moreover, these issues are not unique to Mollie but are representative of broader challenges faced by PSPs and other high-frequency transaction systems across industries. By delving into these questions, the research not only provides solutions tailored to Mollie but also develops generic methodologies that applicable to large-scale problems involving scalable and real-time transaction processing platforms. Solving these challenges will largely expand the broader for Mollie growth, offering insights into designing scalable, fault-tolerant, and efficient systems in dynamic environments.

This thesis explores the transition from a monolithic to a microservices architecture with event-driven mechanism, using Mollie's payment platform as a case study. The research investigates how architectural changes can enhance scalability, fault tolerance, and performance in a high-frequency transaction environment. By applying tools like Apache Kafka for event streaming and DataDog (3) for real-time monitoring, the study aims to demonstrate practical methodologies for achieving a reliable, efficient, and adaptable platform.

However, designing and developing such scalable system pose significant challenges, including ensuring fault tolerance, maintaining data integrity, and handling real-time transaction processing in a distributed environment. This thesis first delves into these challenges and emerging technologies which can be utilized to address those challenges, and then provides better solutions and improvements to Mollie's existing payment platform. The main objectives of this thesis are to achieve the following features:

- **Scalability:** This thesis aims to employ a microservices framework that decouples core payment services, allowing individual services—such as transaction processing and notifications—to scale independently. This refining enhances the system's ability to meet the demands of high-frequency transactions and provides flexibility for future growth.

- **Deployment:** Deployment has always been a tricky problem for large company with large code base. The pipeline is source-consuming and can be risky, especially in environments with frequent updates. This thesis addresses deployment challenges by implementing orchestration, making sure each microservice can be deployed and updated independently. By utilizing tools like Docker and Kubernetes, this framework supports continuous integration and deployment (CI/CD), reducing downtime and enabling rapid iteration (4).

- **Security:** As data grow exponentially, ensuring data security is crucial in financial technology. This thesis integrates layered security protocols to protect user data, with a focus on meeting standards such as GDPR(5), and other special requirements of payment method partners from different countries. By building compliance and data privacy into the decentralized architecture, the platform is better at protecting data from obtaining by malicious entities.

- **Fault Tolerance:** Fault tolerance is critical to maintaining uninterrupted service in distributed systems. Unexpected incidents could dramatically damage user experience and customer trust. This thesis leverages event-driven architecture (EDA) (6) principles and Apache Kafka for asynchronous communication and load redistribution. By applying real-time monitoring, the system can detect subtle failures before users are aware of them. Moreover, it aims to manage service failures without disrupting core operations, ensuring that payment processing remains resilient under load or partial service interruptions.

- **Performance:** Real-time transaction processing requires optimized performance across all services. Although computational resources can be determining factors, the software efficiency also plays a necessary role here. The thesis focuses on reducing latency and enhancing throughput through efficient data streaming and active monitoring according to microservice architecture. Such optimizations allow for a responsive user experience, even under peak transaction volumes, while ensuring data is processed quickly and efficiently.

- **Maintainability & Extensibility:** As the payment ecosystem continues to develop, more and more payment providers and customers generate higher demands. Thus the platform must be adaptable to new technologies and business requirements. In this paper, we emphasize maintainability through modular design, clear documentation, and automated testing. Additionally, by making each service modular, the system can seamlessly integrate additional payment gateways, analytics tools, or compliance features, supporting ongoing innovation and evolution.

The scope of this research includes comprehensive analysis of the architectural shift, focusing on key aspects such as the design of modular microservices, event-driven communication strategies, and the integration of monitoring and deployment pipelines. It also addresses challenges such as managing distributed systems and maintaining data consistency across services. This thesis contributes valuable insights to the fields of financial technology and software engineering by providing design and implementation process of the transition. It highlights the advantages of microservices and event-driven architectures and offer potential challenges and future directions of the project. Additionally, the research underscores the importance of real-time monitoring and proactive fault management in sustaining high-performance systems. These findings are expected to guide industry practitioners and researchers in addressing similar challenges in digital payment ecosystems, supporting innovation and growth in an increasingly complex financial landscape(7).

Through this exploration, the thesis not only improves Mollie's capabilities to meet present and future demands but also provides a scalable and adaptable architectural model applicable to all large-scale platforms navigating the dynamic and competitive environments.

2

Background

The rapid digitalization of financial services has led to an explosion of payment options, each catering to different markets and customer needs. Payment service providers (PSPs), like Mollie, must integrate a variety of payment methods into their platforms to offer a seamless experience to merchants and consumers. As new payment methods emerge across different regions, the ability to integrate these methods into a unified, scalable, and efficient system has become a core challenge for payment platforms.

2.1 Microservices Architecture

Traditionally, payment platforms were designed using monolithic architectures, where all functionalities—such as payment processing, security, user management, and API integrations—were tightly coupled within a single system. While this design offers simplicity and centralization, it struggles to scale effectively as new payment methods and features are added. Monolithic systems typically face performance bottlenecks, are difficult to maintain, and become increasingly complex over time, making it challenging to integrate new payment methods without disrupting existing services.

To overcome these limitations, many organizations have shifted toward microservices architectures, where functionalities are divided into smaller, independently deployable services. Microservices enable better scalability, fault isolation, and easier maintenance, allowing organizations to integrate new features or services (such as payment methods) without affecting the entire system. As shown in the figure 2.1 , in the context of payment systems, microservices can be highly beneficial by allowing the addition of new payment APIs or methods, like GoCardless, without altering the entire platform's architecture (8). Unlike monolithic architectures, where all components are interconnected and must be deployed

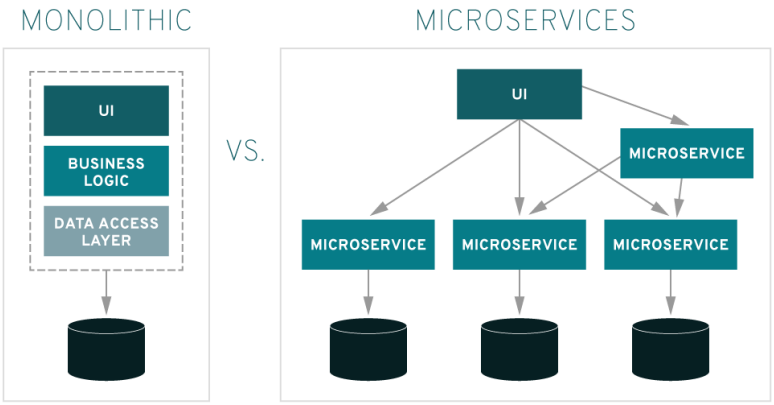

Figure 2.1: Monolithic vs Microservices

as a single unit, microservices are independent services that can be developed, deployed, and scaled separately. This architectural style has gained prominence in recent years, particularly for large-scale platfroms, due to its flexibility, scalability, and fault-tolerant properties. Below are some key Characteristics of Microservices:

- **Modularity:** For each microservice, it encapsulates a specific business functionality, such as payment processing, user management, or notification handling. This modularity ensures that changes or upgrades to one service can be made independently without affecting the entire system.
- **Independent Deployment:** Microservices can be deployed independently, allowing developers to update or revert specific services without redeploying the entire system. This feature is especially useful and powerful when it comes to large payment platforms like Mollie, where issues can be resolved or payment methods can be integrated without disrupting other functionalities and users' experience.
- **Scalability:** One of the core advantages of microservices is horizontal scalability. Individual services can be scaled independently based on the load showed on monitoring applications. For instance, a payment service processing transactions can scale up during high-demand periods (e.g., Black Friday) without impacting other services like reporting or customer management.
- **Polyglot Development:** Microservices architectures allow for the use of different programming languages and databases tailored to the specific needs of each service. This flexibility is crucial in heterogeneous systems where services can be designed

based on their needs for performance, scalability, or security, without significant dependencies.

- **Fault Isolation:** Because microservices operate independently, failures in one service do not necessarily cause the entire system to fail. This fault isolation ensures better system resilience, which is critical in payment platforms where transaction processing must be always reliable and continuous.

The integration of multiple payment methods into a unified platform like Mollie requires an architecture that can support flexibility and scalability. Traditional monolithic systems, where all features are tightly coupled into one huge codebase, often struggle to meet these demands due to their inherent limitations. First, they are difficult in scaling. In a monolithic architecture, scaling requires duplicating the entire system, even if only one component requires additional resources. This leads to inefficiencies and increased operational costs which can significantly affect the performance as the system grows larger and larger. Besides, modifying or adding new functionality in a monolithic system can require extensive changes to the entire codebase, increasing the risk of introducing bugs or downtime during deployment. This can be particularly troublesome in the dynamic fintech domain. There are large volumes of data and backend processing services are involved for each customer. For continuous deployment, it is more challenging in monolithic architectures. The entire system must be tested and deployed as a single unit, even for small code changes. Microservices allow for quicker development cycles, enabling faster adaptation and require less copmutational resources to market and regulatory changes. On the other, a microservices-based architecture supports the modularity and independence, which perfectly resolve the problems faced by traditional large-scale systems. By breaking down different functionalities into smaller, self-contained services, each service can be developed, deployed, and scaled independently. It improves agility and reduces complexity. This approach decreases the dependencies between teams and allow them to choose the best tools and technologies for each service, optimizing the system performance. Moreover, microservices enable faster, more frequent updates through independent deployments. Organizations are able to adapt to new requirements and make the teamwork more efficient.

While microservices offer numerous advantages, there are still some certain challenges have to be considered and addressed, from design, monitoring, to testing (9). Managing numerous independent services requires sophisticated collaboration and monitoring tools to ensure smooth operation. Payment platforms must implement robust infrastructure to

manage communication and load balancing across microservices. Moreover, in a microservices architecture, services often need to communicate with each other. This introduces complexity. During development, developers have to start different mircoservices to make sure the changes work for the communications between them. It also means ensuring reliable, secure, and low-latency communication is crucial. Event-driven communication, which will be discussed in the following section, is one solution to this challenge. Another chalenge is the data consistency. Achieving data consistency across distributed services can be complex, especially there are some data shared through different platforms. Payment platforms often deal with real-time data, and ensuring consistency between various payment services requires careful design, particularly when adhering to regulatory standards.

2.2 Event-Driven System

Event-driven system is an architectural pattern in which system components communicate by producing and consuming events or inputs that trigger the system to respond. In this type of system, different components communicate through events, and the system reacts to those events as they occur. In this kind of systems, services respond to events, which are signals indicating that a change has occurred, such as the completion of a payment transaction, a user action, or a webhook. Event-driven architectures (EDA) offer significant advantages in systems that require real-time processing and high scalability, making them particularly well-suited for payment platforms that handle numerous transactions.

2.2.1 Key Features

As the name indicates, event-driven systems are based on events. An event represents a change in the system's state. For example, a successful payment authorization, from the partner like Ideal, would generate an event that can trigger downstream processes or services. Those entities generate or publish events are called event producers. For example, in the context of integrating a payment method, the event producers can be both the payment solution provider and the partners who engage in the payment and transaction process and generate events for each transaction status. Event consumers, on the other hand, are those services or components listen for and respond to events. Their positions can be reversed. Once an event is produced, it is consumed by one or more services responsible for handling the event, such as updating payment status, sending notifications, or triggering follow-up actions like refunds or chargebacks.

For large-scale payment platforms that need to process thousands of transactions across multiple markets and payment methods, an event-driven system provides several critical advantages:

- **Asynchronous Communication:** Event-driven architectures allow components to communicate asynchronously, meaning they can continue processing other tasks without waiting for a single response. This reduces bottlenecks and improves system efficiency, which is important in a high-performance environment like payment processing (**?**).
- **Scalability:** The asynchronous communication feature enables event-driven systems to achieve scalability (**?**). As transaction volume increases, event producers can continue generating events while consumers scale horizontally to process those events. This enables the systems to reduce latency and enhance the overall throughput by handling multiple events concurrently.
- **Real-Time Processing:** EDA is ideal for systems that require real-time data handling. In payment systems, real-time transaction processing is critical to ensure fast responses to customer actions. EDA enables payment platforms to process and confirm transactions instantly, providing a seamless user experience for both merchants and consumers.
- **Decoupling of Services:** EDA decouples event producers and consumers, allowing each service to evolve independently. This is particularly beneficial for payment platforms integrating multiple payment methods. For instance, the integration of GoCardless does not necessitate changes in other payment-related services on Mollie's platform.

2.2.2 Apache Kafka

Nowadays, event-driven systems are usually powered by technologies like Apache Kafka (10). Apache Kafka is a distributed event streaming platform that plays a central role in implementing event-driven architectures. It was originally introduced by LinkedIn and later open-sourced, Kafka has become a popular choice for event-driven systems, especially in domains like financial services where real-time data processing is crucial.

Kafka decouples communication between services through the use of producers and consumers. Producers send event data to Kafka, and consumers retrieve that data by subscribing to specific Kafka topics. Kafka's simple yet powerful client library enables producers

and consumers to interact with Kafka clusters via transparent communication APIs. One of Kafka's key strengths lies in its ecosystem, which includes additional tools such as Kafka Streams and Kafka Connect, extending Kafka's functionality to meet various data processing and integration needs. These tools enable powerful stream processing capabilities and seamless integration with other data systems, making Kafka a versatile solution for real-time data pipelines (11).

Kafka's event storage is structured as an append-only log, and consumers can retrieve events using offsets, which track the position of the event within the partition. This allows for reliable, ordered event processing. Each Kafka cluster is composed of Kafka brokers, with each broker holding one or more partitions of a topic. Kafka ensures redundancy and fault tolerance through topic replication. This allows Kafka to tolerate up to n-1 broker failures (where n is the number of replicas) without losing data or compromising message delivery. In comparison to other message brokers, such as RabbitMQ, Kafka excels in handling high-throughput workloads with shorter message payloads, which is beneficial for platforms, especially for payment systems where many small transactions are processed rapidly.

Kafka is crucial for managing real-time payment events. When a payment is initiated, Kafka ensures that the event is distributed to all necessary services, enabling independent processing of transactions, notifications etc. By adopting Apache Kafka and EDA, we can ensure seamless integration of payment methods like Ideal while maintaining scalability, responsiveness, and resilience. This approach allows fintech companies to decouple services, respond to real-time events efficiently, and meet the diverse and evolving needs of merchants and consumers in a dynamic payment domain.

2.3 Real-Time Data Monitoring and System Performance

As reliability and efficiency of transactions is essential in payment systems, real-time data monitoring and system performance management play a critical role in maintaining the stability, security, and responsiveness. With all kinds of payment methods integrated into one platform, monitoring becomes even more important to manage increased complexity and traffic. It enables payment platforms to proactively identify and address issues before they impact users. By continuously tracking key metrics such as transaction latency, error rates, and resource utilization, the system can detect anomalies early, enabling quick intervention to minimize downtime or transaction failures.

In payment processing, the most important issues are high—transaction delays or failures can lead to significant financial losses, compliance issues, or customer losses (12). Therefore, an effective monitoring solution must provide comprehensive visibility into all parts of the system, from API requests to database interactions. Modern monitoring tools like Datadog, Prometheus, and Grafana provides concurrent insights into the health of the system by tracking key metrics such as transaction throughput, latency, resource utilization, and error rates. This real-time visibility is essential for:

- **Centralized Monitoring**: Aggregating logs and performance data from all microservices in a centralized dashboard helps identify different causes of issues instantly. This is important when dealing with distributed architectures like microservices and event-driven architectures.

- **Real-Time Alerts and Notifications:** By setting up alerts based on predefined thresholds (e.g., transaction latency), the platform can notify engineers immediately when something goes wrong, reducing response times. Whenever an unexpected event occurs in the application, the error is logged to an error reporting service for debugging.

- **Performance Metrics Tracking:** Monitoring critical metrics such as transaction response times, request throughput, database performance, and server resource usage ensures that the platform remains within acceptable performance bounds. Moreover, Real-time insights into resource utilization enable dynamic scaling of services. This is particularly useful in microservices architectures, where different services can scale independently based on current transaction loads.

Regarding the monitoring tools, there are many platforms can be used to address the problem.

- **Datadog:** A cloud-scale monitoring and analytics platform, Datadog offers real-time insights into the performance of microservices. It integrates seamlessly with cloud infrastructure and supports automatic scaling and alerting based on custom-defined performance indicators. Datadog's modular pricing allows users to select the specific monitoring capabilities they need, providing flexibility for organizations of any size, especially those managing microservices and dynamic cloud infrastructures.

- **Dynatrace:** A software intelligence platform focused on full-stack monitoring with AI-powered automation and root-cause analysis. It offers automatic discovery and real-time visibility across applications, infrastructure, and user experience. Dynatrace's AI engine, Davis, automates the detection of performance issues and correlates them to root causes, making it ideal for complex cloud-native environments. It is particularly suited for enterprises needing deep observability, automated troubleshooting, and end-to-end tracing across hybrid and cloud infrastructures.
- **Prometheus:** An open-source monitoring solution, Prometheus specializes in time-series data collection and real-time alerting. It can be integrated with services to collect performance data in real time, making it ideal for monitoring event-driven architectures. Prometheus stores data locally and provides flexible alerting, making it a go-to choice for infrastructure and application performance monitoring, especially in Kubernetes environments. It integrates well with visualization tools like Grafana and is widely adopted for monitoring microservices and dynamic infrastructures.

Real-time data monitoring is an indispensable part for payment platforms, ensuring that they can handle the growing demands of merchants and consumers. By leveraging the advantages of real-time monitoring, fintech companies can meet SLAs, optimize resource utilization, and provide reliable and efficient service to users.

This chapter introduces the primary concepts applied to the new architecture, including microservices architecture, event-driven systems, and real-time monitoring. The integration of these technologies can form a cohesive strategy to address the critical challenges of scalability, fault tolerance, and system responsiveness. Microservices architecture enables independent scaling and development of services, reducing bottlenecks and increasing modularity. Event-driven system, powered by tools like Apache Kafka, decouples service interactions and enable asynchronous, real-time communication for high-throughput scenarios. Meanwhile, real-time monitoring tools, as we use Datadog for Mollie Platform, offer essential visibility into system health, facilitating proactive error management and performance optimization. Together, these concepts lay the foundation for the scalable and reliable payment platform. The following chapters will discuss further on how they are applied and implemented.

3

Related Work

The growing complexity of payment systems and the need for scalable, fault-tolerant solutions have driven significant research into backend architecture designs, event-driven systems, and modularity through microservices. This section reviews relevant studies and highlights their contributions, limitations, and applicability to the research problem.

3.1 Backend API Design for Cashless Transactions

Adam et al.(13) developed a REST API-based backend server to support cashless payment systems for small-scale retail communities. Their design included critical features such as user registration, balance management, and transaction processing etc. The system utilized Node.js for server-side logic and MongoDB for data management. A token-based authentication mechanism (JWT) was implemented to ensure secure API interactions. Their system aims to enable seamless integration across diverse client platforms via a centralized backend, based on the flexibility of REST APIs for inter-service communication. They tested their backend system under high API traffic, simulating 100 concurrent requests per second, achieving a 76.92% success rate across 13 features. Transaction-related features showed lower reliability—45% for reducing buyer balances and 65% for adding seller balances—highlighting scalability issues and inefficiencies in database query handling under heavy loads.

While REST APIs offered flexibility for small-scale systems, their stateless nature required frequent database interactions, introducing latency and limiting scalability. These issues become critical in high-frequency payment environments, where thousands of transactions per second demand real-time processing (14). This research addresses these challenges by designing a backend optimized for high-frequency data processing. Instead of

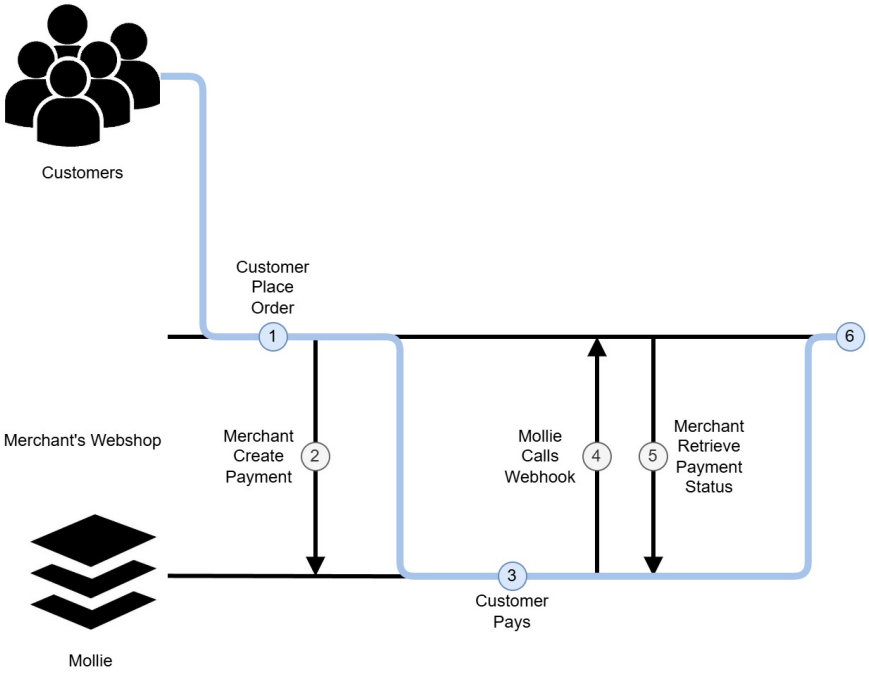

Figure 3.1: API Flow at Mollie

relying solely on REST APIs, the system integrates event-driven communication (discussed later) to reduce database dependencies, enhance throughput, and improve fault tolerance. By extending Adam et al.'s foundational work to a large-scale platform, this study offers a robust backend architecture tailored to the demands of modern payment systems.

Another study by Aué et al.(15) on faults in web API integration within a large-scale payment system, analyzing over 2.43 million API error responses from Adyen's payment platform. The study categorized API integration faults into 11 general causes. For example, invalid or missing user input, insufficient permissions, and internal errors. They found that most faults stemmed from invalid or missing data, with third-party integration issues also matter. The study concluded that API consumers often rely on official documentation, but they face different challenges due to inadequate guidance on error handling and recovery processes.

While their study focused on diagnosing faults in existing APIs, this research addresses these issues by a more robust and fault-tolerant API design for payment platforms. Using an event-driven architecture, the system proactively minimizes errors by decoupling services and validating API requests before processing. Feature flag is also introduced to meet special requirements of different payment methods and merchants. Some payment

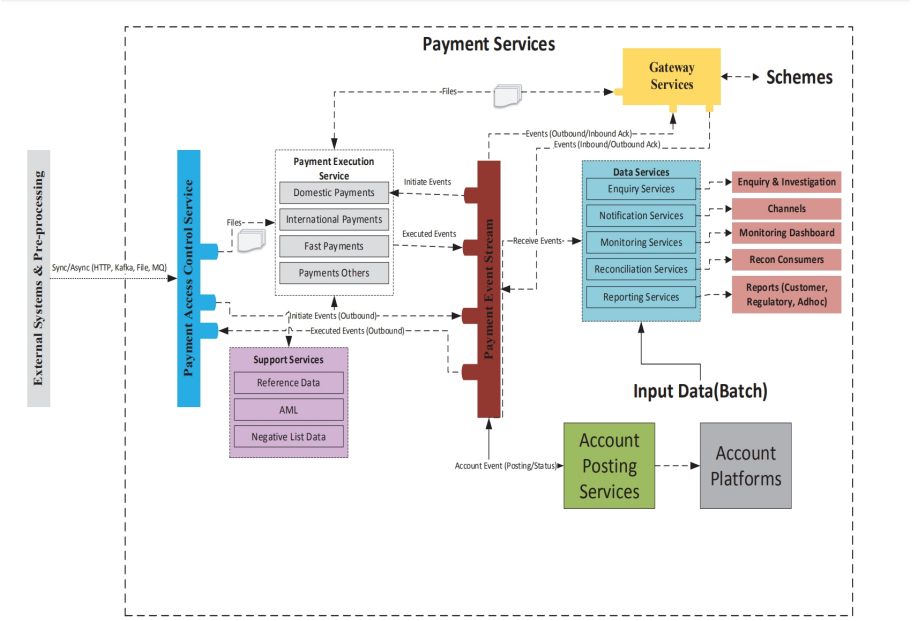

Figure 3.2: Event-driven Architecture for Payment Services by Vangala et al.

methods require more information, such as billing and shipping address when creating a payment, for instance. Real-time monitoring tools further enhance fault detection and provide immediate alerts, enabling rapid error resolution and reducing downtime. This approach ensures scalable, reliable API integration for high-frequency payment systems, overcoming the limitations of traditional REST-based designs.

3.2 Event-Driven and Streaming Architectures

Event-driven and streaming architectures have become essential in financial services due to their ability to handle real-time data flows and enhance system efficiency. Vangala et al.(16) investigate Apache Kafka's role in managing high-throughput transactions. Not only did it highlight Kafka's ability to decouple services, but it also reduced latency and enhances fault tolerance. Their study shows that Kafka's event-streaming model enables each service to process transaction events independently as shown in figure 3.2, improving resilience by eliminating synchronous dependencies that could otherwise cause bottlenecks.

Building on Vangala et al.'s findings, this thesis leverages Kafka within Mollie's payment integration to handle transaction events across diverse payment providers. By adopting a Kafka-based event-streaming approach, the integration can achieve low latency and high scalability, providing a flexible foundation for handling real-time transactions. Expanding

on this, Vyas et al.(17) evaluate Apache Kafka's performance in real-time data streaming. Their study highlights Kafka's efficiency in handling large data volumes with minimal memory utilization by leveraging disk-based message storage. They also explore critical performance metrics like throughput and latency to demonstrating how configuration parameters like partitioning and polling intervals influence system performance.

Despite its benefits, however, the application of EDAs in high-volume financial transaction systems also presents challenges, including event duplication, out-of-sequence events, and load balancing, as mentioned in the paper. During development, similar issues were observed, particularly when repeated event notifications caused redundant operations, leading to false alerts and reduced efficiency. To solve the conflicts, idempotency strategies (unique event identifiers and time-stamping) are introduced.

Real-time monitoring and distributed tracing, as recommended by both teams, play an important role in ensuring system performance. In this thesis, monitoring tools are deployed to track performance metrics such as latency, throughput, and error rates. These metrics provide actionable insights for identifying and resolving bottlenecks, ensuring stable transaction handling. Such proactive measures not only enhance system reliability but also offer a seamless payment experience, meeting the rigorous demands of financial services.

3.3 Monolithic & Microservice Architecture

In recent years, microservices have become a popular architectural approach for organizations transitioning from traditional monolithic systems. A case study (18) on redesigning a real estate sector integration platform from a Service-Oriented Architecture (SOA) to microservices highlights key benefits from this shift. The transition allowed independent scaling of services, which enables resource allocation to specific bottlenecks without affecting the entire platform at the same time. Deployment cycles became much shorter, with updates applied to individual services without resulting in the system-wide downtime. Additionally, the queue-based communication inherent to microservices improved fault tolerance by isolating failures to specific services, preventing widespread disruptions. Enhanced resource management and the ability to use specific technologies for different services further increased scalability and operational efficiency.

However, the study showed transition to microservices also presents challenges, including the complexity of managing inter-service dependencies and asynchronous communication flows. The study proposed using service mesh technologies to streamline service interactions and distributed tracing tools for improved observability and troubleshooting.

Building on these findings, microservices are particularly advantageous for high-frequency transaction environments in fintech. Event-driven communication, in turn, reduces service coupling and improves fault isolation. This modular design makes sure the integration of new payment methods or onboarding functionalities without destabilizing the system. By separating database interactions within a dedicated microservice, as demonstrated in the study, transaction consistency is maintained while simplifying data access and management. This approach keeps the flexibility and resilience, making them an ideal choice for dynamic, scalability-driven domains.

Besides, such a architecture can significantly improve the security and privacy of data. Li(19) presents a robust design framework for electronic payment systems tailored to financial institutions, emphasizing stringent security protocols, scalability, and transaction speed—core requirements in high-stakes environments like finance. The study discusses several design principles including multi-layered access control, data encryption, and end-to-end security for compliance with regulatory standards like PSD2 and GDPR. The research outlines the high-level security demands that must be met in any payment system, which provides invaluable insights for our study. However, it remains focused on traditional payment processing models and lacks emphasis on modularity.

In contrast, this thesis builds on their security framework by employing a microservices architecture specifically for integrating different sevices within Mollie's ecosystem. This approach leverages microservices to isolate sensitive financial functions, ensuring that the security protocols around each module align with compliance needs while supporting real-time processing. Furthermore, this research introduces event-driven architecture (EDA) as an additional layer to handle asynchronous transactions, providing both resilience and enhanced security in real-time processing environments.

3.4 Reflection

The previous studies above have addressed individual aspects of backend design, event-driven systems, and microservices. However, our research combines these elements to propose a holistic solution for the high-frequency transaction system. Based on asynchronous event handling and modularizing services, this work bridges gaps in scalability, fault isolation, and real-time performance. Unlike prior works, which primarily focus on diagnosing existing issues, this research proactively designs and implements a system that mitigates common challenges such as event duplication and inter-service dependencies.

3. RELATED WORK

This thesis contributes by providing a scalable and reliable architecture specifically designed for the increasing demands of payment platforms. It addresses not only the technical challenges but also operational issues, such as ease of deployment and maintainability. This study offers a comprehensive framework applicable not just to payment platforms but to any large-scale system requiring robust performance and scalability.

4

Design

In this chapter, we provide a high-level overview of the system architecture, detailing the previous monolithic structure and the proposed modular approach. We focus on the transition from a monolithic architecture to Independent Autonomous Services (IAS), highlighting the benefits in scalability, fault tolerance, and operational efficiency. These services communicate through event streams and APIs, ensuring data consistency, efficient processing, and fault isolation.

4.1 Previous Architecture

The previous architecture was based on a monolithic structure, where various functionalities, such as payment processing, onboarding, risk assessment, and accounting, were tightly integrated within a single codebase and database environment. While this setup provided a straightforward deployment model and simplified some aspects of data sharing across modules.

As shown in the figure 4.1, all the sevices and functionalities were integrated into the monolithic architecture, which causes low latency and operational complexity. As transaction volumes increased, it was difficult for the monolithic architecture to handle the huge data input/output efficiently. Scaling required adding resources to the entire system, rather than to individual services. This vertical scaling approach was costly and inefficient. It failed to balance resources based on specific components. For example, payment processor requires more computational resources than other services. However, it did not provide enough resources which caused long response time. Besides, the monolithic architecture lacked isolation between services. A failure in one component could damage and bring down the entire system. For instance, if the accounting service experienced a error

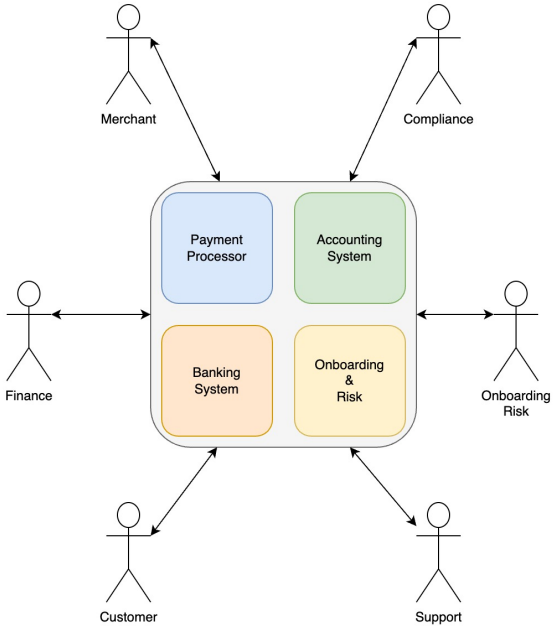

Figure 4.1: Monolithic Architecture of Mollie Platform

while a merchant onboarding, it could potentially influence the whole system, disrupting other critical services, such as payment processing. This can lead to a high risk to service reliability.

For developers, updating or deploying new features required redeploying the entire monolithic application. It costed approximately 40 minutes to run the pipeline on GitLab at first. Even minor code errors forced developers to wait for the lengthy deployment pipeline to identify issues, slowing down the development cycle. It also means development teams had to work in a shared codebase, which created bottlenecks in the development process. When I first joined the company, I spent almost three months to get familiar with the code base because it was too huge to understand every parts including unit test and integration tests. Furthermore, as all services shared a common database and environment, it was challenging to implement security protocols effectively. Sensitive data used by the banking system or risk management was accessible across modules since everyone can use the same encrypted token, increasing the risk of data exposure. The monolithic structure complicated efforts to meet compliance standards, and it was difficult to isolate and secure specific data flows.

In conclusion, while the monolithic architecture initially provided a simple and centralized approach when first creating the platform, it became struggling to handle growing

needs as time passes. The low fault tolerance, lack of scalability, and security limitations draw the needs for a more modular, scalable, and resilient architecture. Although transitioning is complex, adopting a microservices-based structure, or Independent Autonomous Services (IAS), would provide a more robust foundation for the company's growth.

4.2 Architecture Design

This section describes a high-level architecture for Mollie platform. The focus is on the large components and the interaction between them. The proposed solution contains both the point on the horizon and a strategic way to get there. The primary objective is to deal with the growing number of transactions and to limit incidents. As the number of users grows, we face both external and internal challenges have to be addressed. For the external factors, there are more and more transactions and higher demands of transaction processing per second and peak usage of our platform. Addtionally, high-volume customers expecting a higher level of quality and stability. On the other hand, the growing engineering workforce that needs to work independently and autonomously for internal teams. Furthermore, The desire to lower the complexity and coupling of the current code-base in order to speed up time to competency for new engineers as we face a fast growth phase, as well as assisting the current teams to be able to move more quickly and with less risk. Thus the new architecture has to match requirements that remove the current constraints and provide a stable foundation for future development. The following high-level functional requirements must be met:

4.2.1 Primary Requirements

- Component Resilience: Individual components should be able to fail without impacting critical business processes such as payment processing.
- Fault Tolerance: The system should be designed to handle failures gracefully while payment processing.
- Scalability: The architecture should support horizontal scaling to handle increased transaction volumes predictably.
- Availability: The system should span multiple availability zones and support failover across regions.

- Governance Friendly: The architecture should apply solutions that simplify the governance of systems and infrastructure.
- Tool Standardization: Minimize the use of redundant tools with similar functionality.
- Security: Solutions should enable the creation of secure components by default. The system should provide tools for engineers and DevOps that expose security defects early in the development lifecycle.

4.2.2 Secondary Goals

- Modularity: A modular structure enables higher predictability and faster delivery of new functionality.
- Cloud-Native Design: Leverage cloud-native solutions to enhance scalability and reliability.
- Simplified Debugging: The architecture should minimize time spent on production debugging.
- Service Boundaries: Clear boundaries between services allow consumers to remain unaffected by internal changes.
- Standard Solutions: Favor industry-standard solutions over custom-built ones.
- Pattern Consistency: Prefer repeatable patterns over one-off implementations.

4.2.3 Proposed Architecture

Following the above rules and goals, the proposed architecture is shown as figure 4.2, by splitting up the monolith, services are divided into areas that provide the most value to the customers. These areas of value are influenced by Mollie's operational domain (online payments) and the services provided to merchants. Payment processing is at the core of these services. Additional tools support the Onboarding & Risk, allowing them to assess risks and manage merchant payment capabilities. The architecture also separates accounting, which underpins secondary business processes, from the primary payment services.

By identifying these key areas and their dependencies, we can define the distinct boundaries. The diagram illustrates a high-level overview, showing how the huge monolithic platform is segmented into smaller, logical units with clear interaction boundaries. Each of these units, or containers, represents an independent service. Containers are self-contained,

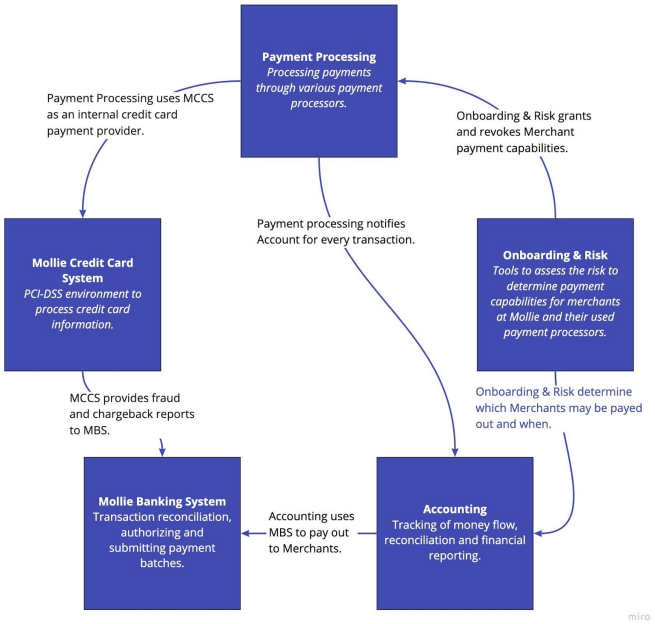

Figure 4.2: Proposed Architecture of Mollie Platform

with their own resources, and communicate with each other. Each service and functionality is exposed externally through a load balancer.

Inter-service communication is achieved through event streams and APIs. Each service has its own resources, such as databases and storage. Communication between services follows a transactional protocol (e.g., HTTP). Asynchronous data is transferred by event streams, which allow a publish-subscribe model for communication. Rather than integrating via shared database access, event streams enable decoupled communication. Messages within the system are immutable, and data cannot be queried directly. This structure enhances decoupling and enables flexible scalability.

The event stream mechanism will be a shared resource for all the contexts. Access to the information is organized around streams. Streams can be represented as topics, queues, or exchanges. Event streams are used for communication, they are not job queues. All communication contained within the events streams should conform to a common format. The implementation will be chosen based on the highest common denominator. The mechanism must be horizontally scalable and have user-based permissions. The goal of using this common format is governance. Therefore, the permissions should be managed in a configuration-as-code way, ensuring we can audit it and get a deterministic overview of the access of information. We would like to create a set of tools ensuring components

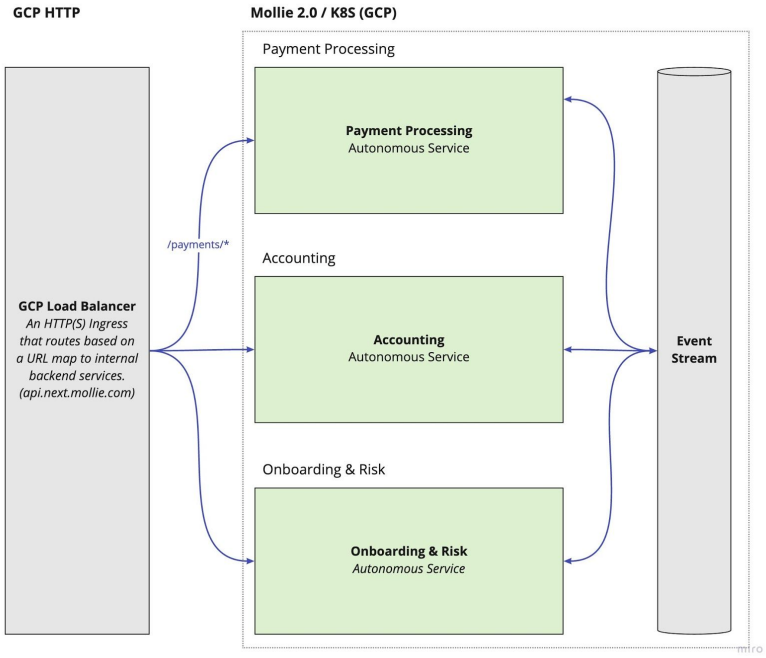

Figure 4.3: GCP-Based Service Architecture for Payment Processing

are secure and compliant by default. Resource isolation, standardized communication, and access control all play a big part to achieve that.

In summary, the proposed architecture is based on Independent Autonomous Services (IAS) and microservices. These are self-contained units aligned with Mollie's value streams, providing a scalable, resilient, and modular structure that supports future growth and adaptability.

4.3 Optimizing Event Dispatching

In the current architecture, each domain provides event streams to allow asynchronous integration across services. We distinguish between public and private events using a marker interface, yet there remains some ambiguity about which delivery channels (topics, exchanges, or queues) are intended for general or internal consumption. The existing setup caused significant delays (up to 80 seconds) for downstream systems to consume events, which is unacceptable for new use cases that require near real-time processing, particularly for payment-related events. This delay in dispatch compromises the effectiveness of public events in use cases that demand minimal latency. To clarify and streamline this distinction,

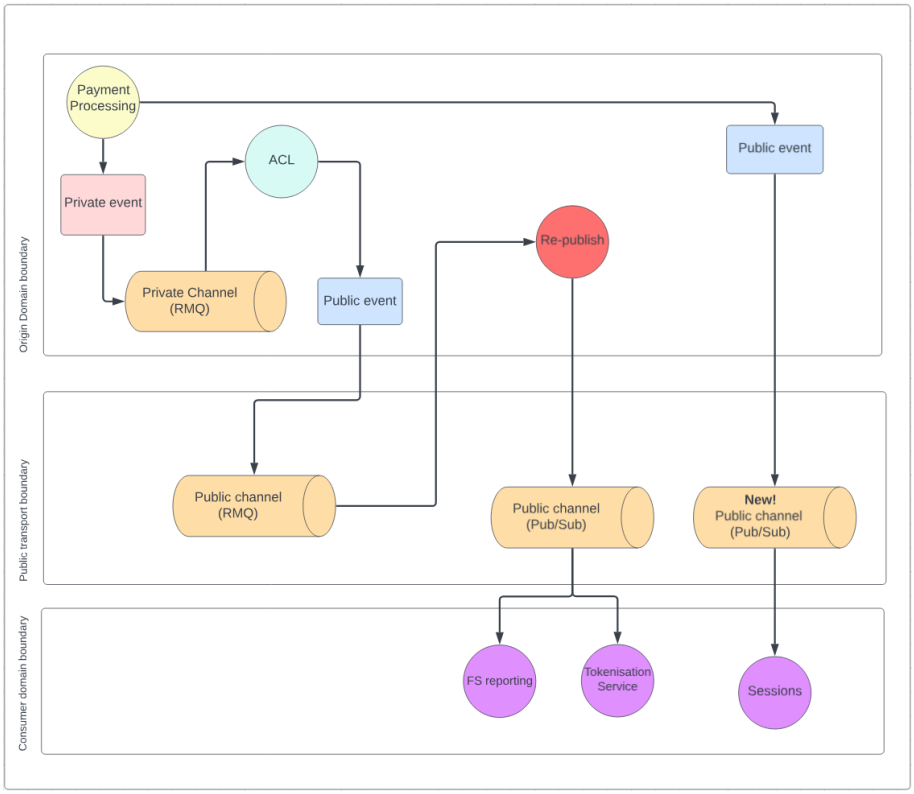

Figure 4.4: Optimized Event Dispatching Architecture

this RFC proposes a standardized convention to explicitly differentiate between public and private channels.

To address the delay, a direct publishing mechanism for the event-dispatching system is introduced. This would involve adding a secondary outbox table to handle direct publishing as part of the business processes that trigger these events. Leveraging data already generated during payment processing and transaction workflows, public events will be directly dispatched to new public channels using this outbox setup. During the migration, we will continue the existing process for converting private events to public ones and redistributing public events from RMQ to Pub/Sub, minimizing disruption while gradually adopting the new setup.

For implementation, a new outbox table is created to log public events. This table will be used to dispatch events directly to designated public channels, bypassing the main RMQ-based dispatch path, thus eliminating delays in event consumption for downstream systems. Besides, new public channels are introduced specifically for near real-time events. Downstream consumers would gradually adapt to these channels, ensuring a smooth tran-

sition with minimal impact. To reduce risks, both the existing RMQ-to-Pub/Sub dispatch and the new direct dispatch still work with each other. This provides a fallback while ensuring reliability for consumers who are yet to transition to the new channels.

5

Implementation

In response to the complex challenges presented by the previous architecture, we continuously adopted a modular, decoupled architecture to simplify the development (integration of new payment methods for our team) and optimize system scalability. This implementation section outlines the steps taken to decouple core components, design a plugin-based architecture, and integrate new payment methods in a straight forward way. Key technologies and methodologies—including the gateway, microservices, and plugin integration layers—were employed to ensure robustness and flexibility in Mollie's ecosystem.

5.1 Decoupling the Architecture

Because the previous architecture was so huge and complicated, we constantly tried to decouple the coupled monolithic structure that previously governed Mollie's platform without affecting current functionalities. The existing monolithic system required extensive resources to scale or update, making it costly and time-consuming. Moving to a modular, decoupled architecture allows each component to function independently, reducing technical debt and enhancing scalability. The following shows the decoupling process followed in three steps.

5.1.1 Move By Namespace

Two distinct main services we usually involve in, Payments and Onboarding, were identified as core modules essential for financial transaction management and user access control. Each service was designed as an autonomous module capable of future transition to a fully-fledged microservice if required by scaling demands. The Payment service manages

all aspects of financial transactions, including API interactions for refunds and webhook handling. In contrast, the Onboarding service focuses on user authentication.

Code from the application/classes directory is moved into a dedicated folder for each domain. Ownership of code is clarified, code that has co-ownership would need to be untangled. During this stage there will still be cross-domain coupling. Code that uses code from another domain is still allowed at this stage. We unlock quantifying domain entanglement by counting the cross-domain imports and class usage. There are four rules here:

- Move all code from the application/classes directory to domain-specific namespaces (e.g., Payments, Onboarding).
- Business logic related code has to be moved to its respective domain-specific namespace to ensure that domain-specific logic remained encapsulated.
- Database-dependent code was placed exclusively within domain-specific namespaces to eliminate cross-domain database access.
- Foundational functionalities essential across domains were migrated to an overarching application namespace.

5.1.2 Eliminate Cross-Domain Direct Database Access

The next step aims to remove direct database access across domains, transitioning each domain's data dependencies to an independent storage system. By separating databases, each service could maintain its own database without cross-domain interference, which is crucial for scaling and security.

The goal is to reach zero direct database access between domains. Progress can be tracked on GCP that monitored database dependencies. These metrics, visualized on a dashboard. It allows real-time monitoring and ensured that each service adhered strictly to its domain's boundaries. This step builds the foundation for a fully modular system where each domain can be scaled up and operated independently without direct cross-references.

5.1.3 Reimplement integration points over the network

With code and database being divided into different modules, it is time to reimplement all integration points using network-based APIs or event streams. This transition enables real-time communication between services while ensuring decoupling, as integration points

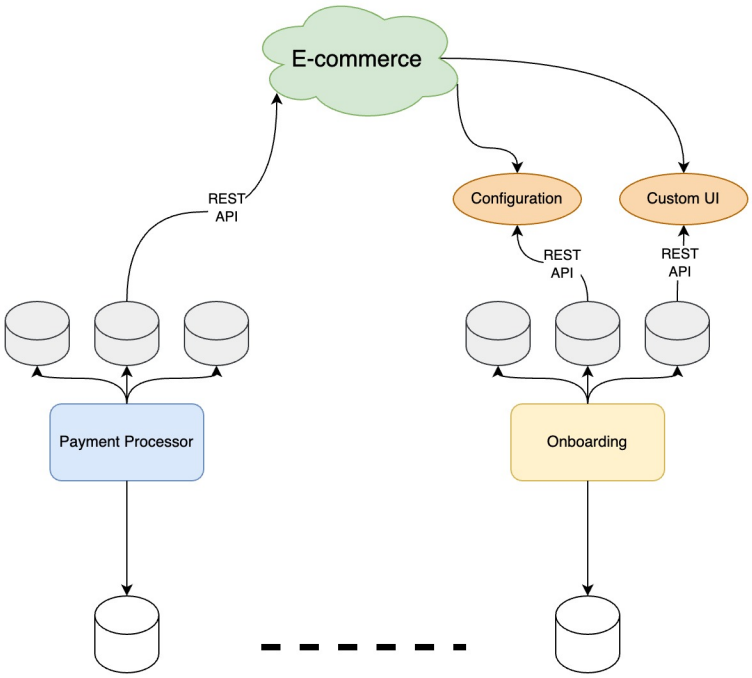

Figure 5.1: E-commerce Gateway

no longer involves direct calls across domain boundaries. By leveraging API endpoints or message queues for interactions, integration becomes scalable and location-independent. It potentially supports migration to cloud-based services such as GCP in the long run. The API or event-driven structure allowed each domain to communicate asynchronously, providing resilience and flexibility.

In order to make sure the successful transition, the tracking progress is necessary. The effectiveness of this transition requires monitoring for zero violations of direct integration between sub-domains. This ensures that each domain operates as a self-contained unit, ready for independent scaling and deployment.

5.1.4 Gateway and Workflow Management

To manage and coordinate workflows across decoupled services, the gateway plays an important role. This gateway processes high-level business events into actionable Mollie workflows. It perfectly abstracts the underlies logic and enabling seamless handoff between services.

For instance, if a payment event triggers a split-payment workflow, the gateway spots

this need and initiates the appropriate steps. This centralization of workflow management simplifies service interactions. As long as the gateway, as shown in figure 5.1, handles them in advance, the services do not need to know the specific implementation details of each workflow. The gateway enhances modularity by serving as an abstraction layer, which allows for flexible workflow modifications without direct changes to core components.

5.1.5 Event-Driven Architecture

To support real-time interactions between decoupled components, we choose event-driven architecture (EDA) as the solution. Using Apache Kafka for event management, EDA facilitates asynchronous communication. It minimizes dependencies between services at the same time. By applying Kafka for inter-service communication, we ensure each service operates independently. When one action produces or received, then it consums or generates an event as the intermediate without impacting the entire system.

5.2 Partner Integration

A main part I work on in this project is the maintenance of current payment methods and integration of new payment methods into the platform. The decoupled architecture provides support for streamlined integration of payment methods. The previous integration process required significant code changes and manual configuration, leading to extended development timelines. Under the new architecture, each payment method is treated as a plugin module, encapsulating all platform-specific requirements. Each team can collaborate with consistent APIs, focusing on their own domains. This makes the integration of new payment methods more efficient and ensures consistency across integrations.

5.2.1 Plugin-Based Architecture

Each supported e-commerce platform (e.g., Shopify, Lightspeed, Wix), as the upstream of the whole process, operates via a plugin module (20). This module serves as an intermediary between Mollie's internal systems and the specific e-commerce platform. The plugin module structure follows an 80/20 model, where 80% of the functionality is shared across integrations, with the remaining 20% tailored to individual platform requirements. This approach enables Mollie's internal systems to communicate efficiently with external platforms. It not only isolates platform-specific logic within each plugin, but it also allows for consistent performance across different environments. Within this architecture, external platforms can communicate with Mollie platform seamlessly, without changing any API

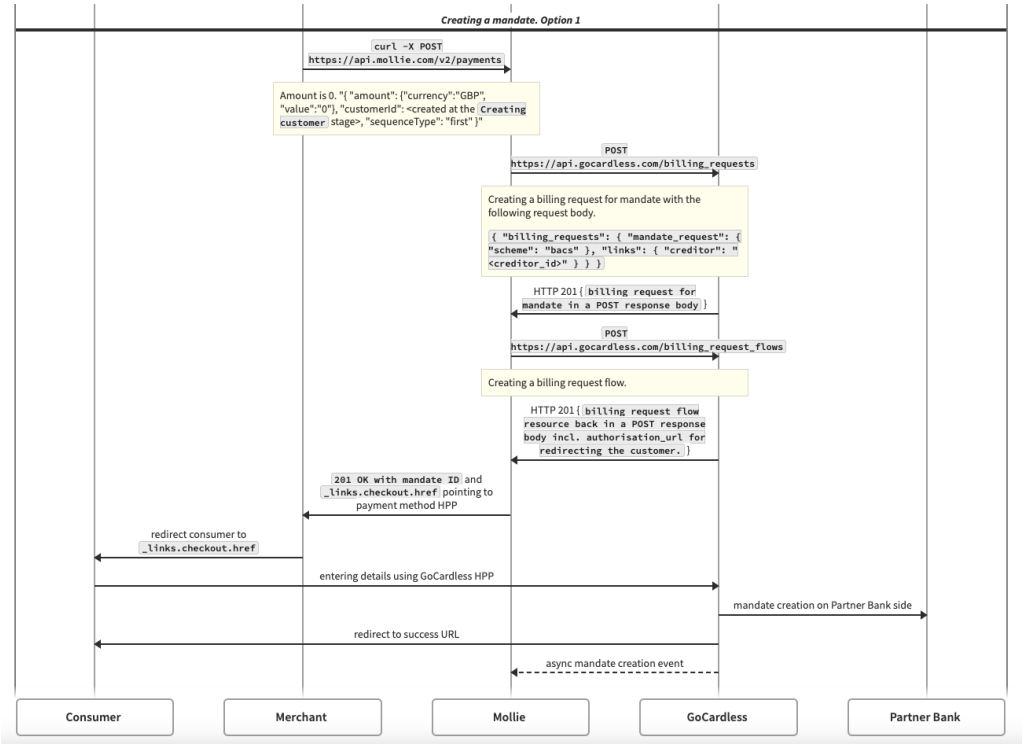

Figure 5.2: Mandate Creation Workflow - GoCardless

requirements through the transition. Each plugin module manages platform-specific configurations internally, thereby decoupling the setup requirements from Mollie's core systems. This structure minimizes the need for custom adaptations in Mollie's primary codebase.

This architecture supports that each plugin module to independently connect to custom services via REST APIs, offering flexible customization for diverse e-commerce platform requirements. Furthermore, it speeds up the process of new external platform but also provides a scalable, modular framework that aligns with Mollie's strategic goal of creating a resilient e-commerce integration system.

5.2.2 Workflow-Based Payment Method Integration

Payment methods, on the other hand, play as the downstream for us, handling the following payment and transcations on a more bank-related foundation. As for the development process, once a new payment method is identified, a workflow is initiated within the gateway to manage the payment lifecycle, ensuring that it flows smoothly from initial authorization to settlement and any subsequent refund processing. This process is driven by interactions with the Payment Service and event-driven components. The integration of new payment

methods follows a clear and structured workflow. Our team work on the payment service and onboarding service to support various steps within the workflow, providing standardized handling for tasks such as creating and processing payments/mandates, managing recurring payments, and handling refunds and chargebacks. However, different payment methods may have different requirements when it comes to the payment details. I take the workflow stages of integration of a UK-based payment method, GoCardless, as an example to illustrate here 5.1.

Beside the general steps mentioned above, there are still several other possible stages need to be done, such as settlement, reconciliation, and notification services, which ensure that all funds are correctly transferred and provide customized services according to rules or policies of different countries. For GoCardless, it is mandatory to send notification for direct debits to customers due to the UK bank regulations.

5.3 Monitoring & Continuous Deployment

To support the high availability and reliability required in payment domain, monitoring and deployment processes are necessary for multiple services in the new architecture.

5.3.1 Automating Deployment and Testing

A continuous integration and deployment (CI/CD) pipeline is widely used for platform deployment and testing (21). When the architecture is established, the deployment and automated testing for each service can be processed independently. The modularity of the new architecture supports a rolling release strategy. Each module is able to be updated without impacting other services. This pipeline accelerates the development cycle and minimizes downtime, improving the user experience and facilitating faster response to market demands.

Every merge request (MR) into the master branch triggers an automatic deployment pipeline. This enables traceability for each release, with commit identifiers make sure it possible to rollback in emergencies. Feature toggles are used to control the release of specific features to targeted users or groups, enhancing deployment flexibility and risk management.

Besides, every service has logging, monitoring, and alerting configured, as we will discuss in the following part. Datadog and other tools track critical metrics such as error rates and response times, with alerts set for any unexpected failure in the pipeline. This ensures rapid response to issues during deployment.

Stage	Description
Onboarding Merchant	Onboarding merchants can be manually or automatically, including the setup of necessary credentials and configurations for payment processing. Besides, there is a payment method activation check based on the merchants' information and the payment method company's requirements.
Creating Mandates	For methods requiring mandate-based authorization (such as direct debits), the workflow starts by creating mandates to collect payments and the following payments can be deducted on the same mandates. As shown in 5.2
Payment Processing	Once the mandate is set, the payment can be created and processed using Mollie's Payment API. This involves making an authorization request, capturing the payment, and performing any necessary fraud checks.
Recurring Payments	The system must support recurring payments if mandates are allowed, particularly for subscription-based services or memberships. This involves creating recurring payment instructions, managing payment schedules, and handling automatic renewals.
Webhook Handling	Webhooks play an important role in the communication with external systems. To process webhooks, the system is abole to receive incoming events such as payment authorization, mandate activation, or other updates related to payment transactions (including payments, mandates, and activation). These events trigger follow-up actions to update transaction status, notify users, or initiate refunds and chargebacks.
Refund and Chargeback Processing	Refunds and chargebacks are handled through specific workflows that react to events such as customer disputes or payment errors, ensuring the correct funds are returned to customers or deducted from merchants. What should be noticed is that refund is more generic than chargeback. Chargeback is not always supported by all payment methods.

Table 5.1: Payment Method Integration Stages

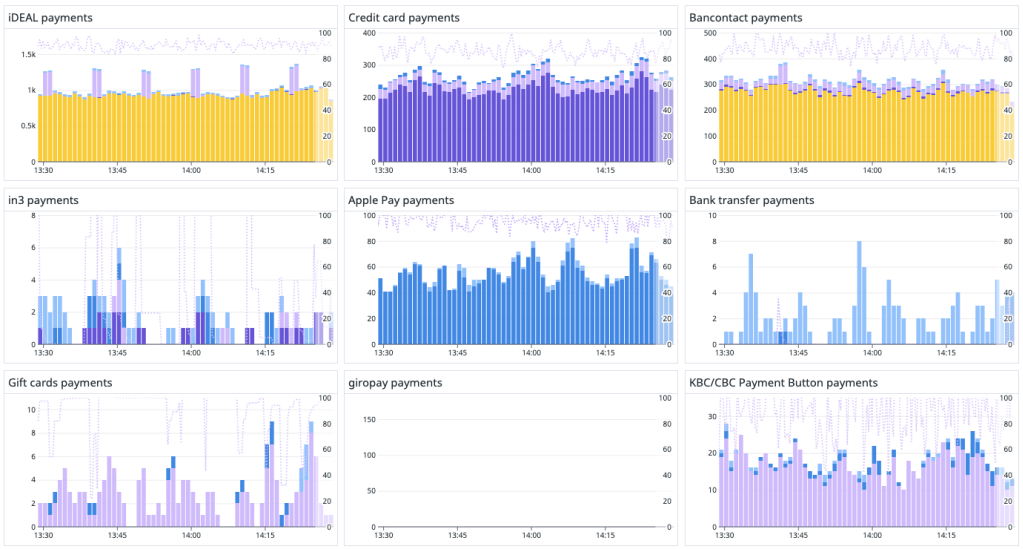

Figure 5.3: DataDog Dashboard - Payment Methods Overview

5.3.2 Real-Time Monitoring

To ensure the robustness, performance, and reliability of all services within Mollie's platform, real-time monitoring tool, DataDog, has been utilized across both core systems and plugins. Datadog serves as the primary monitoring tool at Mollie, where key performance metrics, such as transaction latency, error rates, and system health, are visualized on centralized dashboards. These dashboards allow for rapid issue detection and resolution, which provides a comprehensive view of operational performance across different payment methods and integration points.

With the centralized dashboards, we can obtain insights into real-time operations and the health of various services. First and most important, the real-Time operations overview. There is a dashboard displays live data on payment volumes and processing times across different payment methods as in figure 5.3, allowing quick visibility into any operational fluctuations. Moreover, it tracks authorization rates, declines, and errors for every payment, enabling proactive troubleshooting at the scheme or acquirer level. Additionally, there are dashboards monitor the performance and reliability of command operations, essential for orchestrating asynchronous events across the platform. These dashboards serve as crucial tools for Mollie's engineering teams, allowing us to monitor operational health, react to unexpected failures timely.

Based on Kubernetes, we deploy services expose Prometheus metrics (22), which are scraped by the Datadog Agent. Kubernetes pod annotations are used to specify which

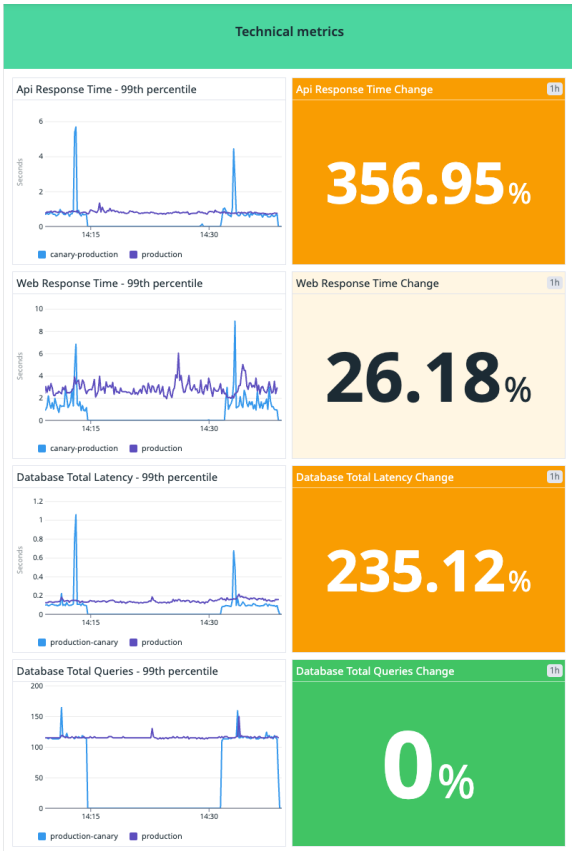

Figure 5.4: DataDog Dashboard - Technical Metrics

metrics to collect, allowing selective and efficient monitoring. Some key metrics used by Mollie platform include: 1. response time: measures latency for critical API calls (e.g., payments, authorizations) to track performance and spot slowdowns, as shown in figure 5.4. 2. error rates: tracks the number of errors encountered in service interactions, which indicates potential issues in services or external dependencies. 3. transaction volume: provides insights into transaction load by counting payment or authentication events, allowing Mollie to monitor system usage and scale accordingly. 4. computational resource: measures CPU, memory, and other resource usage across services to ensure efficient resource allocation.

Overall, this real-time monitoring framework ensures that Mollie maintains high service quality and meets performance standards. The modular architecture helps manage different service and spot and resolve potential errors more efficiently.

6

Evaluation

This chapter aims to evaluate the performance, reliability, and scalability of the proposed microservices and event-driven architecture, using real-time metrics from Datadog (23). By examining key metrics, such as API response times, database latency, and error alarming, this section validates the architecture's effectiveness in supporting high-frequency payment processing. The findings also highlight areas for both performance and computational resource improvement, contributing to the system's ongoing optimization.

6.0.1 API Response Time

To begin with, the current architecture has contributed significantly to improved API performance (24), as shown in the Datadog response time dashboard for multiple APIs 6.1.

We take one of the detailed API response time dashboards as an example. The updated response time graph for the "Get Method Selection" API indicates significant improvement compared to the previous data, as shown in figure 6.2 and figure 6.3. Currently, the 99th percentile response time is stabilizing between 600-700 milliseconds, the 95th percentile around 350-450 milliseconds, and the 50th percentile approximately 200-250 milliseconds.

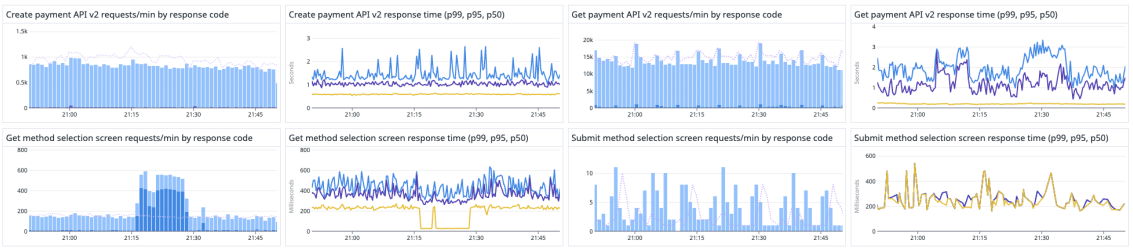

Figure 6.1: API Response Time Overview

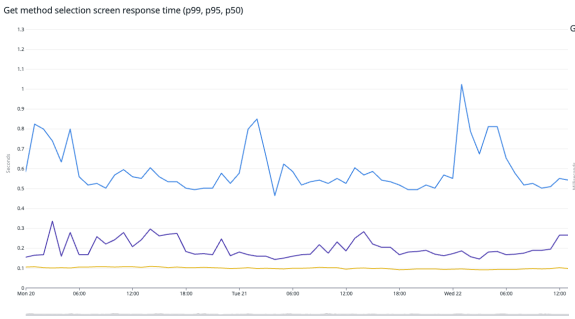

Figure 6.2: Previous Response Time for Get Method Selection API

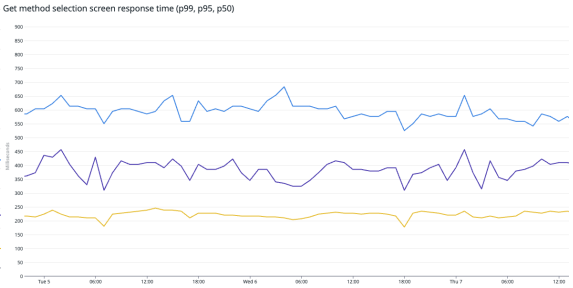

Figure 6.3: Current Response Time for Get Method Selection API

This is a substantial reduction from prior peaks of 1 second and above in the earlier data, suggesting that recent optimizations have been effective.

The improvements illustrate that the system has effectively mitigated earlier bottlenecks that were impacting performance. The improvements are considered coming from three aspects. First, because of the decoupled architecture, frequently accessed data can be cached, reducing the load on backend services. Second, distributing traffic more evenly across API instances improves stability and load balance. Furthermore, the optimization of database and queries reduce data retrieval times and directly contribute to improved API response. This evaluation and improvement demonstrate the architecture's resilience and adaptability in handling transaction volumes efficiently. However, continued monitoring is recommended to ensure these improvements hold under varying load conditions.

6.1 Database Performace

This section shows how performance changes from database level. Evaluating database performance is crucial in understanding the efficiency and reliability of the system, especially in a high-frequency transaction platform. Key aspects of database performance include query latency, resource allocation, and workload management (25). Efficient handling of these elements ensures that the system can support real-time processing with minimal delays, which is essential for maintaining a responsive user experience.

Figure 6.4 and 6.5 show the slot allocation of underlying database. In the previous data, slot allocation showed high variability, with peaks up to 1.5k slots during certain periods. This fluctuation suggested inefficiencies in query management, potentially leading to delays and increased query latency during high-demand periods. Recent dashboard indicates a more stable and reduced slot allocation pattern, with peaks now barely reaching around

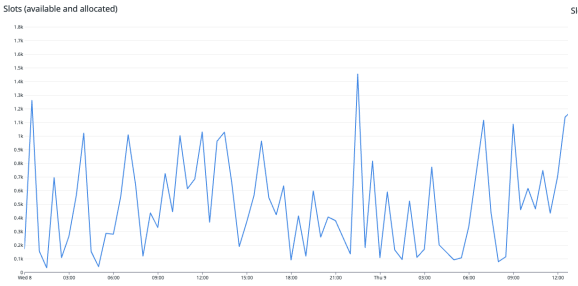

Figure 6.4: Previous Database Slot Allocation

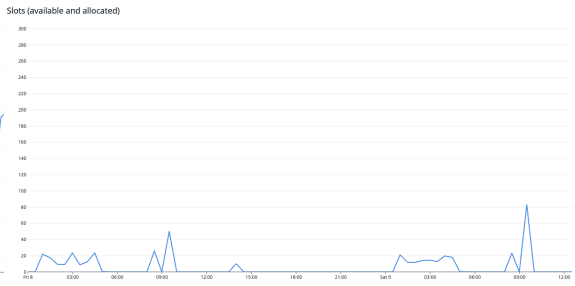

Figure 6.5: Current Database Slot Allocation

250 slots. This change illustrates that improvements in workload distribution and query optimization have reduced the need for excessive resource allocation. As a result, the query latency is lower. More stable slot usage has contributed to faster query processing, which, in turn, supports efficient API response times. The more consistent slot allocation prevents resource waste and allows the database to handle peak loads more predictably.

6.2 Error Handling and Alerting in Event-Driven Architecture

Although overall platform performance has been improved, there are still some unexpected errors caused by code issues or incorrect operations. Therefore, it is important to catch those errors instantly, especially in high-stakes environments like a payment platform. In order to make timely remedies like revets or quick fixes, real-time error detection is essential.

To achieve this, we use the Slack notification service, integrated with Datadog, allowing the team to receive immediate alerts for critical errors. Figure 6.6 shows how Datadog bot alerts teams of a HTTP 500 error for Payment Method Activation API. The Slack alert suggests it is potentially caused by unhandled exceptions or server-side issues. In an event-driven architecture, error handling is critical to maintain smooth communication between services.

To reduce these errors, the error handling mechanisms by implementing detailed logging also help resolve root causes faster. Furthermore, setting up more modular alerts in Datadog would allow for early detection of errors without excessive false positives. It largely improves fault tolerance at long run. Besides, key metrics such as load time, error rates, and resource utilization were used to identify performance bottlenecks and improve system

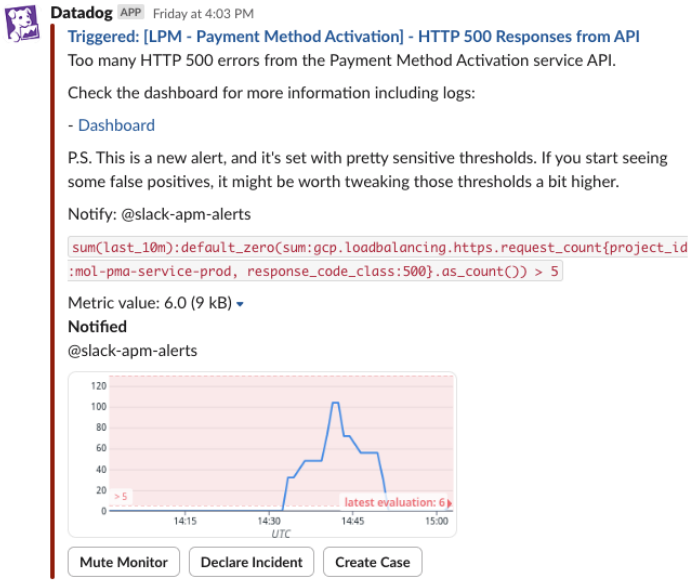

Figure 6.6: Datadog Slack Alert for HTTP 500 Error in Payment Method Activation API

reliability. These insights validate the applicability of the monitoring in real-world scenarios and align with our findings on integrating hybrid monitoring strategies for enhanced performance (26).

7

Discussion

This chapter provides a comprehensive discussion of the project's key reflections, addresses the challenges & limitations, and outlines potential directions for future development and enhancement of the platform.

7.1 Reflections

The transition from monolithic to microservices and event-driven architecture has significantly enhanced the Mollie payment platform's performance, scalability, and resilience. By decoupling the huge system, this project succeeded in independent scaling and greater fault tolerance. The adoption of an event-driven approach with Apache Kafka further enables real-time and asynchronous data processing, essential in handling high transaction volumes efficiently. The results underscore the advantages of microservices. Because of the greater modularity, it improves fault isolation and reduces development cycles. This modular structure allows the platform to deploy more easily and respond quickly to evolving market demands. Meanwhile, the event-driven system enhances data flow between services, which also lower the latency and maintain system responsiveness.

With the help of the real-time monitoring with tool DataDog, it has been invaluable for providing deep insights into system performance and supporting proactive error management. Engineers are able to spot the error or failure within a short time, helping the platform to maintain high availability and minimize downtime. These architectural enhancements have proven effective in fulfilling the project's core goals of substantially boosting reliability and supporting Mollie's sustained growth and future scalability.

7.2 Challenges and Limitations

Even though the architectural improvements have brought many benefits, several challenges and limitations still remain. A lot of adaptions have been implemented to decouple the architecture without disrupting the functionalities, yet certain issues persist (27). One key issue is event duplication and out-of-sequence processing. Duplicate events result in redundant triggers, affecting downstream systems, while out-of-sequence events may lead to inconsistencies in workflows. Although we tried to apply different anti-duplication strategies and timestamp-based ordering, these solutions sometimes are not enough, which produces new errors and we need to fix. Besides, it adds computational overhead and increases complexity. Another technical weakness is the real-time performance. The payment system requires real-time nature which needs API response times and event handling latencies to stay within strict thresholds. Certain external dependencies, such as third-party APIs, occasionally fail to meet these requirements, causing bottlenecks in workflows. Achieving consistent latency remains an ongoing challenge. In other saying, one delayed response can cause the following communications more delayed due to the increasing internal API communications.

On the other hand, it also brings operational challenges such as the resource overhead. The modular character of microservices demands dedicated resources for each service, including isolated databases, independent deployment pipelines, and monitoring configurations. This setup brings huge benefits for scalability and fault isolation, but it also increases the overall resource footprint and operational costs meanwhile. In other words, although modular approaches simplify scaling and flexibility, integrating multiple payment gateways can introduce complexity in handling diverse workflows (28). Additionally, transitioning from a monolithic to a microservices architecture requires substantial training for engineering teams. While modularity enables teams to specialize in specific areas, they are not able to understand how the system work as a whole. Instead, understanding new paradigms such as event-driven communication and distributed database management may cause a hard time learning.

In summary, while the platform has made significant strides in addressing its legacy limitations, these challenges highlight areas that require ongoing attention. Addressing these limitations will be crucial to sustaining growth, improving operational efficiency, and meeting the increasing demands of users and partners.

7.3 Future Works

Based on the limitations discussed above, several areas for improvement and future directions are proposed to further enhance the platform's performance, scalability, and adaptability.

- **Advanced Event Management:** Develop more efficient and lightweight algorithms for event deduplication to reduce computational overhead. Leveraging unique event classifier or implementing distributed consensus protocols can help mitigate duplication and out-of-sequence issues without sacrificing performance. For events with higher priority, mechanisms can be applied to prioritize high-priority events in the processing queue to ensure critical workflows maintain sub-second latency even under peak load conditions. As suggested by Petrov et al. (29), future advancements in event processing can benefit significantly from the integration of artificial intelligence and adaptive load balancing.

- **Enhanced Developer Tools:** The staging environment cannot support some online APIs, which causes much trouble doing after-deployment testing. Building tools to simulate complex inter-service interactions in a test environment would help developers identify and resolve issues more quickly. Besides, comprehensive documentation is crucial for a company like Mollie to grow at this point. For all microservices and workflows, detailed and centralized documentation should be used to facilitate faster onboarding for new engineers and improve cross-team collaboration.

- **Real-Time Performance:** Because when we try to investigate into an specific transaction failure or error, it is difficult to spot the related logs on GCP or DataDog. Therefore, it is necessary to detect the system's real-time performance with detailed information and logs which can help solve the issues more efficiently. Moreover, to optimize the API gateway to handle high traffic with reduced latency, we may try some cutting-edge technologies like HTTP/3 and adaptive rate limiting. Retry mechanisms and local caching for third-party API calls will also be considered to minimize the impact of external delays on internal workflows.

8

Conclusion

This paper illustrates the transition from a monolithic architecture to a modular microservices and event-driven architecture in the context of a high-frequency payment platform at Mollie. By employing a modular approach, the research addresses key challenges of scalability, fault tolerance, and operational efficiency, making sure the system is suitable for real-time, high-volume transaction processing. Through the design and implementation of an optimized architecture, this work demonstrates how asynchronous communication and modularization enhance performance while reducing latency. Real-time monitoring tools like DataDog, implemented with Google Cloud Platform (30), further facilitate proactive error management, contributing to the system's reliability and maintainability.

The findings first demonstrate that adopting a microservices architecture successfully decoupled tightly integrated functionalities. This transition facilitated independent scaling and enhanced modularity. The performance results of DataDog confirm that the microservices architecture effectively addressed the challenges of scalability and fault tolerance. Second, event-driven architecture (EDA) also played a significant role in optimizing real-time processing and decoupling services. By leveraging Apache Kafka for asynchronous communication and event streaming, the platform reduced latency, improved responsiveness, and minimized dependencies between services. Furthermore, the integration of real-time monitoring tools like DataDog instantly reflected system's health and greatly enhanced system reliability. Real-time alerting mechanisms allowed for proactive issue resolution, while independent deployment pipelines enabled seamless updates without service interruptions, which addressed the third research question.

Despite the significant improvements achieved, this research highlights areas requiring ongoing attention, such as event duplication, limited error logs collection, and external API dependency bottlenecks. The challenges of resource overhead and the complexity of

transitioning to a microservices model also underscore the need for continued refinement (31). Looking forward, the findings of this thesis proposes a good solution for innovations in payment platform architecture. Advanced event management, enhanced developer tooling, and optimized real-time performance monitoring stand out as key areas for ongoing exploration. These improvements offers valuable insights for both academia and industry in the evolving financial technology landscape.

References

[1] Mollie B.V. **Official Website**. `https://www.mollie.com/`. Accessed: 2024-09-03. 1

[2] Aathira S Nair and P Kannan. **Digital Payment Methods: Challenges And Opportunities**. *Journal of Namibian Studies: History Politics Culture*, **37**:367–376, 2023. 1

[3] Inc. Datadog. **Datadog: Monitoring and Security Platform for Cloud Applications**. Online Resource, 2024. Accessed: 2024-11-30. 2

[4] Aparna Naik, Janhavi Choudhari, Vedanti Pawar, and Sanjay Shitole. **Building an edtech platform using microservices and docker**. In *2021 IEEE Pune Section International Conference (PuneCon)*, pages 1–6. IEEE, 2021. 3

[5] Paul Voigt and Axel Von dem Bussche. **The eu general data protection regulation (gdpr)**. *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, **10**(3152676):10–5555, 2017. 3

[6] Brenda M Michelson. **Event-driven architecture overview**. *Patricia Seybold Group*, **2**(12):10–1571, 2006. 3

[7] Jungho Kang. **Mobile payment in Fintech environment: trends, security challenges, and services**. *Human-centric Computing and Information sciences*, **8**(1):32, 2018. 4

[8] Saulo Soares De Toledo, Antonio Martini, Agata Przybyszewska, and Dag IK Sjøberg. **Architectural technical debt in microservices: a case study in a large company**. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 78–87. IEEE, 2019. 5

[9] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. **Design, monitoring, and testing of microservices systems: The practitioners' perspective**. *Journal of Systems and Software*, **182**:111061, 2021. 7

[10] Nishant Garg. *Apache kafka*. Packt Publishing Birmingham, UK, 2013. 9

[11] Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: the definitive guide: real-time data and stream processing at scale*. " O'Reilly Media, Inc.", 2017. 10

[12] Henk Grent, Aleksei Akimov, and Maurício Aniche. **Automatically identifying parameter constraints in complex web APIs: A case study at adyen**. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 71–80. IEEE, 2021. 11

[13] Bob Maulana Adam, Adnan Rachmat Anom Besari, and Mochamad Mobed Bachtiar. **Backend server system design based on REST API for cashless payment system on retail community**. In *2019 International Electronics Symposium (IES)*, pages 208–213. IEEE, 2019. 13

[14] Ersin Ünsal, Bilgehan Öztekin, Murat Çavuş, and Suat Özdemir. **Building a fintech ecosystem: Design and development of a fintech API gateway**. In *2020 international symposium on networks, computers and communications (ISNCC)*, pages 1–5. IEEE, 2020. 13

[15] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. **An exploratory study on faults in web API integration in a large-scale payment company**. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 13–22, 2018. 14

[16] Sreenivas Rao Vangala, Bharath Kasimani, and Ravi Kiran Mallidi. **Microservices Event Driven and Streaming Architectural Approach for Payments and Trade Settlement Services**. In *2022 2nd International Conference on Intelligent Technologies (CONIT)*, pages 1–6. IEEE, 2022. 15

[17] Shubham Vyas, Rajesh Kumar Tyagi, Charu Jain, and Shashank Sahu. **Performance evaluation of apache kafka–a modern platform for real time data streaming**. In *2022 2nd International Conference on Innovative Practices in Technology and Management (ICIPTM)*, **2**, pages 465–470. IEEE, 2022. 16

[18] Ervin Djogic, Samir Ribic, and Dzenana Donko. **Monolithic to microservices redesign of event driven integration platform**. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1411–1414. IEEE, 2018. 16

[19] Shujing Li. **Research on the design of electronic payment system of financial company**. In *2021 2nd International Conference on E-Commerce and Internet Technology (ECIT)*, pages 91–94. IEEE, 2021. 17

[20] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. **Extraction and evolution of architectural variability models in plugin-based systems**. *Software & Systems Modeling*, **13**:1367–1394, 2014. 30

[21] Charanjot Singh, Nikita Seth Gaba, Manjot Kaur, and Bhavleen Kaur. **Comparison of different CI/CD tools integrated with cloud platform**. In *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 7–12. IEEE, 2019. 32

[22] Walid Maalej and Martin P Robillard. **Patterns of knowledge in API reference documentation**. *IEEE Transactions on software Engineering*, **39**(9):1264–1282, 2013. 34

[23] Xian Jun Hong, Hyun Sik Yang, and Young Han Kim. **Performance analysis of RESTful API and RabbitMQ for microservice web application**. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 257–259. IEEE, 2018. 37

[24] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. **A catalogue of inter-parameter dependencies in RESTful web APIs**. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17*, pages 399–414. Springer, 2019. 37

[25] Stavros Christodoulakis. **Implications of certain assumptions in database performance evauation**. *ACM Transactions on Database Systems (TODS)*, **9**(2):163–186, 1984. 38

[26] Linh Pham. **Real user monitoring for internal web application**. 2020. 40

[27] Konrad Gos and Wojciech Zabierowski. **The comparison of microservice and monolithic architecture**. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020. 42

[28] Muchsin Hisyam and Ida Bagus Kerthyayana Manuaba. **Integration Model of Multiple Payment Gateways for Online Split Payment Scenario**. In *2022 International Conference on Information Management and Technology (ICIMTech)*, pages 122–126. IEEE, 2022. 42

[29] YUREVNA AVKSENTIEVA and VLADIMIROVICH BRYUKHANOV. **Current issues and methods of event processing in systems with event-driven architecture**. *Journal of Theoretical and Applied Information Technology*, **99**(9), 2021. 43

[30] Inc. Google. **Google Cloud Platform: Scalable Infrastructure and Services for Modern Applications**. Online Resource, 2024. Accessed: 2024-11-30. 45

[31] Chien-Chang Liu, Chien-Chang Huang, Chia-Wei Tseng, Yao-Tsung Yang, and Li-Der Chou. **Service resource management in edge computing based on microservices**. In *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pages 388–392. IEEE, 2019. 46

Appendix