

POLITECNICO DI TORINO

Master's Degree in COMPUTER ENGINEERING



Master's Degree Thesis

Developing a microservice solution for automated investment monitoring in Wealth Management

Supervisors

Prof. Alessandro Fiori

Candidate

Rafael Lapetina

Ribeiro Gomes

Company tutors

Luigi D'Onofrio

Fabio Crucini

March 2025

Summary

The thesis focuses on the development of a backend software application built during an internship at Alpien Technologies, to work as a back-office solution for a specialized team at Fideuram Asset Management. The main aim of the application is to automate and optimize the periodic monitoring of discretionary mandates, which denotes an investing strategy whereby a client delegates to a bank the management of their portfolio. This monitoring ensures that investment strategies applied to the customers at the bank remain aligned with the agreed-upon risk profiles.

The important contributions of the project include the automation of a critical monitoring procedure once done manually and subject to inefficiencies and a greater potential for errors. By bringing in cutting-edge technologies like Spring Boot for REST services, gRPC for efficient service communication, Kotlin for reliable backend development, and Docker for containerized deployments, the project improved the accuracy and efficiency of investment compliance processes.

The implementation of the microservices structure, in addition to the CI/CD pipelines guarantees the scalability, maintainability, and continuous improvement of the platform. It should also be noted that the monitoring and management of investment data in one place make for much more ease of access to and use by the working part of this software, thus substantially enhancing the workflow and possibility to maintain investments compliant.

This thesis provides evidence of the successful application of modern software technologies to the financial sector, making the case for the possibilities of automation, the digital transformation and enhancement of the monitoring of investment portfolios. Results reflect the impact of this work, which includes lowering operational workload, enhanced compliance efficiency, high level of testing employed, and ultimately providing a scalable solution for further expanding into other areas of investment management. The project has laid the groundwork for future improvements involving real-time monitoring, further automation, and enhanced data integration capabilities.

Table of Contents

List of Tables	VIII
List of Figures	IX
1 Introduction	1
2 Background concepts	3
2.1 Overview of the Domain	3
2.2 Monolithic vs. Microservices Architectures	4
2.2.1 Overview of Monolithic Architecture	5
2.2.2 Microservices Architecture	5
2.3 Communication Between Microservices	8
2.3.1 REST APIs	8
2.3.2 gRPC APIs	8
2.4 Cloud Computing	9
2.4.1 Cloud Service Models	9
2.4.2 Benefits and Challenges of Cloud Computing	10
2.5 Containerization and Docker	11
2.6 Technological Choices	12
2.6.1 Spring Boot	13
2.6.2 Kotlin	14
2.6.3 PostgreSQL	14

2.6.4	Splunk	15
2.7	Scrum and CI/CD	16
2.7.1	CI/CD Pipelines	18
3	System Architecture	20
3.1	Understanding the old process	20
3.2	Understanding the monitoring process	21
3.3	Project description and objectives	23
3.4	Software design	25
3.4.1	Entities	28
3.5	Back-end Architecture	34
3.5.1	Communication between services	36
3.5.2	External databases	37
3.5.3	Security mechanisms	38
3.5.4	Project Structure	40
3.5.5	Dependency management with Maven	42
3.5.6	Endpoints and protobuf file	45
4	User stories flows	48
4.1	Search investment lines	49
4.2	Import of investments data	55
4.3	Validation of a monitoring and spools	61
5	Testing, Deployment and Monitoring	70
5.1	Testing	70
5.1.1	Unit testing	71
5.1.2	Functional Testing	74
5.1.3	Quality Assurance (QA) test	75
5.2	Deployment	76
5.3	Monitoring	77

6 Conclusion	79
Bibliography	82

List of Tables

3.1	List of packages in the project	40
3.2	List of other gRPC methods and endpoints.	47
4.1	List of allowed filter operations	52

List of Figures

2.1	Monolith vs microservice architecture visualization	6
3.1	Entity-relationship diagram for MiaGP system	29
3.2	System architecture for MiaGP	34
3.3	Typical flow of data within system	41
4.1	Filters in investment line page	50
4.2	Search lines sequence diagram	51
4.3	Ingestion of data sequence diagram	58
4.4	Validation of a monitoring sequence diagram	63
5.1	Code coverage of the application	73
5.2	Backstage tool for deployment	76
5.3	Dashboard in Splunk	78

Chapter 1

Introduction

In the world of banking and wealth management, many institutions have a long history that frequently predates the age of technology. These traditional companies, while offering decades of experience and trust, frequently employ legacy systems and outdated processes. This poses challenges in an increasingly digital world, where agility, compliance, and traceability are critical. For big financial institutions handling substantial client investment amounts, modernizing these systems has become both necessary and a competitive benefit.

In particular, wealth management processes are highly sensitive and mission-critical. Banks must ensure that client investments are not only profitable but also aligned with the client's risk preference, as defined through thorough assessments. Failing to do so may lead to possible financial losses as well as regulatory violations.

In the context of the bank Fideuram, discretionary mandates (Gestioni Patrimoniali) have periodic checks, where clients' portfolios are checked in order to ensure that their investments adhere to the agreed risk profiles. This monitoring, performed every 45 days, involves a detailed analysis of the investment portfolios. When discrepancies are identified, such as when a portfolio is found to be misaligned with the customer's risk profile, the necessary adjustments are made manually, sometimes taking days to complete. As the volume of client portfolios grows, so does the complexity of ensuring compliance with such risk mandates.

In this context, the back-office system developed for Fideuram Asset Management's team is one further step toward digital transformation in a traditional financial setting. It proposes an integrated automated solution to facilitate the monitoring of discretionary mandates with adequacy checks, improving compliance and operational efficiency.

This thesis first studies the fundamentals and moves on to concepts, implementations, and evaluations. Chapter 2 introduces the key groundwork topics: monolithic and microservices architectures, cloud computing, REST and gRPC APIs, and the tech decisions taken that altogether influenced the project. Chapter 3 describes the architecture of the system, providing a detailed view of legacy processes, monitoring workflows, and the design and development of a new back-end system. Chapter 4 shows a selection of user story flows that demonstrate the capabilities of the system, specifying some features developed personally. Chapter 5 relates the testing, deployment, and monitoring strategies employed to ensure quality assurance and reliability concerning the system. Chapter 6 concludes the thesis by summarizing both the results from this project and the overall process of personal development.

Chapter 2

Background concepts

This chapter establishes the fundamental knowledge needed to understand the technical and business context of the project. It starts by explaining key financial concepts, such as wealth management and fiduciary mandates, to apply them when the problem is being addressed. The discussion then shifts to architectural frameworks, communication protocols, cloud computing, and the critical technological decisions that guided the creation of the proposed solution. These core concepts provide the necessary knowledge for the system's design, development, and deployment, which are examined in greater detail in the chapters that follow.

2.1 Overview of the Domain

Firstly, to fully understand the scope of this project, it is crucial to have a grasp of the financial domain in which this work is situated. Below are some key financial concepts that will be referenced throughout this thesis, which will provide context and background for understanding how they are used in the software implementation.

- ***Wealth Management:*** Wealth management is a financial service that provides clients with advice and investment strategies to manage and grow their wealth. It includes financial planning, investment management, tax planning, and retirement planning. In this thesis, wealth management forms the core domain where the development of automated solutions for discretionary mandates plays a significant role. [1]
- ***Fiduciary Mandates:*** Also known as discretionary mandates, they are agreements where a financial institution manages an investment portfolio

on behalf of a client. The financial institution has the authority to make investment decisions that align with the client's goals and risk tolerance. Monitoring these mandates is crucial to ensure compliance with regulatory standards and the agreed-upon risk profile. [2]

- ***MiFID II (Markets in Financial Instruments Directive II)***: MiFID II is a European Union regulation that aims to increase transparency and standardize practices within financial markets. It requires investment firms to provide greater disclosure, protect investors, and ensure that financial services align with clients' risk profiles. Compliance with MiFID II is an essential aspect of the project, as it directly influences how discretionary mandates are managed and monitored. [3] [4]
- ***Risk Profile***: A risk profile defines the level of risk an investor is willing to take, considering their financial goals, investment experience, and overall comfort with risk. Risk profiles are typically determined through a detailed questionnaire, where the bank or other institutions will assign a risk profile to a client, which will determine suitable investment strategies for clients. [1]
- ***Compliance***: Covered in partially in previous points, compliance refers to the adherence to laws, regulations, and industry standards that govern financial activities. This includes ensuring investments are suitable for clients according to their risk profiles. One of the main goals of the system discussed in this thesis is to assure exactly that. [1]

With the main domain topics covered, we can now move on to the technical principles underlying the implementation. In the following sections, we will explore the architectural choices, technologies, and methodologies that form the foundation of the solution, providing a detailed look at the technical components and their roles in achieving the project's objectives.

2.2 Monolithic vs. Microservices Architectures

We are going to start by examining two popular software development architectural paradigms: microservices and monolithic systems. With their own advantages and disadvantages, these architectures offer opposing methods for creating and executing software systems. Since the back-office system's design and development were directly impacted by the choice between these paradigms, understanding them is crucial to understanding the architectural choices taken in this project.

2.2.1 Overview of Monolithic Architecture

A classic method in software development, monolithic architecture is typically used for early-stage projects and small-scale applications where initial complexity is reduced. All of an application's components are merged into a single unit using this software design pattern. A monolithic application usually consists of a single codebase with tightly coupled features and services. Since everything is in one location, this kind of design is frequently simpler to develop at first, making deployment and testing simple. Having all features and logic in one place makes the flow of execution easier to follow and understand. In terms of efficiency, there is no delay or overhead from inter-service communication.

However, monolithic architectures can become difficult to scale and manage as applications get bigger and more sophisticated. Even minor adjustments are complicated and dangerous since updates frequently need redeploying the entire system. Furthermore, scalability is typically vertical, which means that rather than adding more servers, improvements frequently need for hardware upgrades. Because changes made to one part of the system may have unexpected effects on other parts, the close coupling of components leads to a lack of flexibility and makes it more difficult to embrace new technologies.

2.2.2 Microservices Architecture

The microservices architecture can be thought of as an evolution from monolithic systems, breaking an application down into a collection of small, independently deployable, and loosely coupled services. Such an evolution addresses many of the limitations in monolithic systems, from challenges in scaling, through issues of maintaining large codebases, to issues of inflexibility in development cycles. Scaling is usually problematic for most monolithic systems because they depend on vertical scaling, which is expensive and inefficient. Also, in a monolithic architecture, management of a big codebase tends to take more time in development because changes made in one part of the system cascade throughout the application, thus demanding longer test and deployment cycles. [5]

This shift was driven by several factors, including the need to support present-day business requirements for constant changes, updates, and deployments, as well as the desire for improved scalability, flexibility, and efficiency in software development. Microservices are typically divided by business function, with each service responsible for operations related to its specific purpose. We can see a visualization of the difference between microservices and monolith in the figure 2.1.

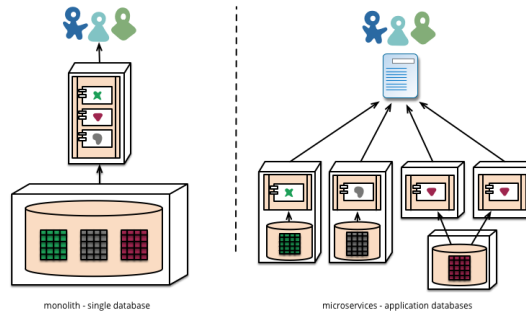


Figure 2.1: Monolith vs microservice architecture visualization

For example, let's consider a complex banking system in which one microservice operates independently from others, managing card payments, while another independently handles all notification processes. Fault tolerance means that if a notification service fails, the card payment service can still be executed without failure. Another pattern here is data autonomy: every microservice has its own database, which means there are no dependencies across services.

The microservices are based upon the key principles such as decentralization of data management, independently deployable units of the architecture, and loose coupling of the different modules. By decomposing any application into smaller modules that can closely interact with each other, microservices do provide enormous benefits in terms of scalability, fault isolation, and flexibility. Some of the key benefits of microservices can be:

- *Scalability:* This makes it possible to scale services autonomously depending on demand, i.e. if a service gets a higher workload, then more computing power or memory can be directed to that service without impacting the rest of the system.
- *Flexibility:* Depending upon the specific needs of each service, other technologies and frameworks can also be used because of the modular design of microservices. That is why new technologies could now be used more easily without disrupting the system as a whole.
- *Agility:* The ability of development teams to work on multiple services at the same time speeds up development time. Less specialized teams of a certain size will be able to design, test and deploy their services on their own, leading to continuous delivery and more rapid iterations.

- *Resilience*: Fault isolation ensures that failure of only part of the system does not bring the entire application down. For example, failure of the payment service would not affect any other service, e.g., for user authentication, and hence the remaining portions of the system are able to function properly. [6]

Although microservices provide clear advantages over monolithic systems, there are still challenges when adopting this approach:

- *Complexity in System Management*: Because microservices are distributed and running in separate processes, it increases the complexity to assure that the system functions cooperatively. Maintaining control over multiple services, each of which has its own underlying infrastructure, database, and dependencies, is challenging and may necessitate complex orchestration tools such as Kubernetes.
- *Transaction Management*: Operations where transactionality is involved turn out to be difficult to manage, due to the lack of consistency and concurrency between different services. As each service can fail in isolation, it is necessary to provide a special structure for each step in a transaction to succeed, either through the Saga pattern or eventual consistency.
- *Inter-Service Communication Overhead*: Microservices involve a lot of inter-service messages which contribute to extra overhead as opposed to functions in monoliths in which function calls are direct. Communication between services may be accomplished via protocols (eg., HTTP or gRPC), however, this introduces network latency and complexity.
- *Infrastructure Management*: Individual microservices could have its own deployment environment, computational resources, and configurations, which results in higher overhead for managing the infrastructure. There are many tools such as Kubernetes, which are commonly employed for the container orchestration, auto scaling, load balancing, and failover, however, they introduce additional levels of complexity. [6]

For large and complicated systems, scalability, flexibility and fault isolation are essential, which makes microservices an appropriate answer. They perform well in a cloud-deployed configuration because in the cloud one obtains the infrastructure to support scalability and distributed service management. However, there are also serious downsides of microservices that cannot be ignored, including infrastructure orchestration, transaction management, and system complexity. For the smaller projects, the microservice paradigm can generate some overhead, which in turn can impact the development significantly, and may justify the extra effort.

2.3 Communication Between Microservices

Usually, when microservices are being employed, they often need to interact with one another in order to perform their specific operations. Communication between these services is critical in order to ensure the system remains scalable, maintainable, efficient and create a seamless integration. This section explores two most used approaches for microservices communication: REST APIs and gRPC APIs.

2.3.1 REST APIs

REST (Representational State Transfer) is a resource-based architecture for building networked applications. RESTful APIs are set with a definition of principles, including statelessness, uniform resource naming and a unified interface. REST makes use of HTTP as the transport protocol and is stateless in nature, i.e., in each communication of a client interaction all relevant information for understanding and processing the communication is included, irrespective from the history of communication. This makes REST excellent for scalability because it makes server management comparatively easy since server state is not required. [7]

HTTP is the existing transport protocol for REST APIs, creating a wide applicability and ease of connectivity between clients and servers. HTTP's verbs GET, POST, PUT and DELETE all correspond well to CRUD operations and make REST a natural choice for the implementation of client-server applications. Simplicity of HTTP, one other strong point of HTTP, also has the consequence that developers are able to debug and work with REST APIs with little to no need for additional tools.

Typical applications of REST are implementing web applications, housing endpoints in mobile applications, and establishing web services with a simple and scalable interface for data exchange.

2.3.2 gRPC APIs

A rather recent solution for inter-process communication is gRPC which implements the Remote Procedure Call model. It is an open-source framework designed with the intention to support high performance and low latency communication. As opposed to REST, gRPC employs a binary serialization method, made possible by using Protocol Buffers (Protobuf) as the interface specification language, which produces the more efficient serialization. Client-server communication is based on HTTP/2, which decreases latency and allows multiplexing, which is perfectly

suitable for microservices that require a reliable, fast communication.

Among gRPC's advantages is performance optimization through binary communication, leading to faster serialization and deserialization than JSON. It also offers type safety, in that client and server have the same "contract" (names of operations, messages, and properties that have been declared explicitly, etc) that are strictly defined in Protocol Buffers preventing the occurrence of runtime error. Through its support of HTTP/2, it facilitates bidirectional streaming in that gRPC can use to carry out client and server streaming, which will enable more sophisticated communication patterns than traditional constituent REST APIs. [8]

Typical applications of gRPC are inter-service communication in distributed systems, real-time streaming capabilities, and applications where response time is important.

2.4 Cloud Computing

Cloud computing refers to delivery of computing resources over the internet. This architectural approach is more popular than ever due to the increasing demand for computing power for specialized tasks. With remote servers managed by cloud providers, such as Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform, users can leave the responsibility of owning and managing hardware to the providers, while still able to scale out or scale up their systems.

A model called pay-as-you-go, as the name suggests, allows users to only be charged of the computing power they used. This becomes attractive when comparing to traditional data-centers, where upfront payments for infrastructure need to be done. Cloud computing usually offers high availability of resources, fault-tolerant platforms, and reduced overhead when it comes to maintenance. Due to this, hosting resources in the cloud is highly beneficial for individuals, developers, and enterprises as a whole. Recently, ideas like hybrid cloud infrastructure and edge computing have expanded cloud solutions.

2.4.1 Cloud Service Models

Cloud computing is generally categorized into three main service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS):

- Infrastructure as a Service (IaaS): Provides users with computing infrastructure, such as virtual machines, storage, and networking. Popular IaaS providers

include AWS EC2 and Google Compute Engine. IaaS allows users to focus on their applications while outsourcing hardware management.

- Platform as a Service (PaaS): Offers a comprehensive platform for developers to build, deploy, and manage applications without worrying about underlying infrastructure. Services like Google App Engine and Microsoft Azure App Services enable rapid development and integration.
- Software as a Service (SaaS): Delivers functional applications over the internet, eliminating the need for local installation or maintenance. Common SaaS examples include Salesforce, Google Workspace, and Microsoft 365, which streamline user access to software. [9]

2.4.2 Benefits and Challenges of Cloud Computing

With respect to other commercial solutions available in the market, cloud computing has found widespread adoption by companies and individuals as a result of its merits. The ability to flexibly increase or decrease capacity available on demand thanks to cloud services allows users to efficiently accommodate transient increases in work load demand. There is also a scaling down of total cost and no need to buy additional hardware in peak hours. Another significant benefit is cost efficiency. In a pay-as-you-go model, the cost paid by the users is only the cost of the used resources, thereby reducing the entry costs and installing IT infrastructure.

Reliability and availability are further strengths of cloud computing. Cloud providers exploit high availability by redundantly replicating information at multiple data center locations, thus reducing the probability that services would be unavailable. Furthermore, flexibility is one of the properties with greatest value, as organizations are able to choose between different service models and deployment options tailored to their own requirements. [10]

However, there are some challenges associated to cloud computing. Security and privacy can be a serious matter of concern since sensitive data is stored and processed on third-party servers, which require adequate encryption and compliance with regulations. Downtime and availability constitute another area of concern which users rely completely upon the integrity of their cloud provider, that the infrastructure of their cloud provider is sufficiently stable on the network level. While there are high hopes for uptime guarantees from the cloud vendors, outages still happen, causing facility delivery failure.

In spite of some difficulties that cloud computing presents, it is still a crucial part of the current IT strategy and provides a highly flexible, scalable, and economical framework for provisioning computing resources.

2.5 Containerization and Docker

Before the advent of containerization, software deployment typically relied on setting up applications on physical servers or using virtual machines. In this case software deployment involved the configuration of the environment on each of the servers manually, ensuring that all dependencies, libraries and configuration existed. Nowadays, the empirical process is typically error-prone, time-consuming, and resulted in non-homogeneous environments, creating the infamous "it works on my machine" phenomenon.

Since the introduction of virtual machines (VMs), it was now possible to encapsulate an entire OS plus an application into a single, portable unit, with the benefit of increased isolation and addressing compatibility problems. Nevertheless, VMs are a resource-eater, because each VM needs a complete OS (and consequently high overhead and low efficiency).

As an alternative to these traditional approaches the containerized approach provides us with a lightweight, mobile tool to develop applications. Containers exploit the host operating system kernel to consume less system resources than VMs while providing a suitable isolation and a stable environment.

Containerization is the process of bundling and compressing an application and all of its dependencies (e.g., libraries, settings, files) into a portable, lightweight container. Containers guarantee that an application is always running under any kind of computing environment with all necessary dependencies provided. Compared to Virtual Machines, in which the hardware is virtualized and a complete operating system is included, the containers share the host OS kernel and are containerized only for the application and its dependencies. This results in lighter, faster, and more resource-efficient deployments. [11]

Docker is one of the well-known tools for containerization that gives the developers and system administrators the possibility to create, deploy, and manage containers with ease. While working with Docker, a developer writes code locally, packages it into a container with all its dependencies, and trusts that it will run on every system running Docker.

Moreover, the container environment is defined in Dockerfiles, including the base image, application code, dependencies, and configurations needed to create the container image that can be run as a container on any compatible Docker host. In addition, it has tools such as Docker Compose for defining and running multi-container Docker applications to make the management of complex deployments of multiple interconnected services much easier. [12]

Some of the most important benefits of containerization, especially with Docker,

make it attractive as a solution for the modern software development and deployment domain.

- *Portability*: With the fact that Docker supports containers, they can run on any system that supports Docker, making them highly portable across different cloud providers, on-premises servers, or developer machines.
- *Scalability*: With its lightweight properties, multiple instances of a container could be deployed to deal with cases of high load to scale an application horizontally.
- *Consistency Across Environments*: By having all dependencies bundled inside a container, developers can be sure the application will behave consistently in that environment, ending the chances of encountering the "it works on my machine" problem.
- *Resource Efficiency*: Containers share the host OS kernel, which means that overhead is lowered compared with traditional VMs; hence, it provides an environment where you get density for applications on the same physical hardware.

However, containerization also presents challenges:

- *Security Concerns*: Since containers share the host OS kernel, the vulnerabilities of that kernel can compromise every container running on that host. Proper isolation and application of security patches are needed. [11]
- *Complex Orchestration*: While Docker simplifies the creation and management of individual containers, deploying and managing containers at scale can be challenging. This is where container orchestration tools like Kubernetes come into play, taking care of scaling, load balancing, and failover.

Containerization with Docker has become a fundamental element of modern DevOps practices, providing an efficient and scalable way to manage applications. Despite some challenges, its benefits make it a valuable technology for ensuring smooth deployments and reliable software performance across different environments.

2.6 Technological Choices

In this section, we will go over each of the technologies used in this project, discussing the reasons behind their selection and how they contribute to the final solution,

while also exploring possible alternatives, evaluating their potential benefits and explaining why they were not chosen for this specific implementation. This analysis provides insights into the decision-making process and the trade-offs considered during the development of the system.

2.6.1 Spring Boot

Spring Boot was selected as a key component to work on the backend of this project. Spring Boot provides an efficient and simplified approach for creating production-ready applications, making it really attractive for rapid prototyping and development. Unlike classic frameworks that often require a lot of boilerplate configuration code, Spring Boot uses auto-configuration and embedded servers that help developers get going quickly, not slowed down by the common setup complexities.

One of the main features here with Spring Boot is how it brings together the best of Spring architecture. It follows a layered architecture, separating concerns between the Presentation, Business, Persistence, and Integration layers. Such an approach makes the application more organized in nature and easy to maintain, especially when they grow with more lines of code and complexity. Nevertheless, with embedded servers such as Tomcat and Jetty, Spring Boot manages to simplify further the deployability of applications that become entirely self-contained and easily usable.

Another major feature is how Spring Boot employs some core concepts from the broader Spring Framework. For instance, Dependency Injection (DI) loosely couples components by injecting their dependencies instead of hard-coding them. This makes the code more modular and, hence, highly testable. Inversion of Control gives the lifecycle and dependencies of such objects to the Spring container, this way reducing the complexity faced by the developer. To address cross-cutting concerns like logging and security, Spring AOP works in conjunction with the aspect-oriented programming, which keeps the main business logic clean. [13]

Spring Boot's seamless integration of Spring MVC provides one extra layer of abstraction above the servlet API, thus allowing for easier building of web applications and REST APIs. This permits the developer to concentrate on the definition of routes/handlers without much concern over all the internal complexities that it encompasses. These features may justify Spring Boot's claim to being the future in building scalable applications that are more easy to maintain.

Node.js would be an example of some alternatives to this technology. That said, Spring might be a better choice because it works great when it comes to CPU-bound operations, along with the other enterprise-level guarantees coming from its rich

ecosystem. Things like transactional management, security, and scaling all sit comfortably in the built-in feature set of Spring Boot, making it a more attractive proposal for complex project requirements.

2.6.2 Kotlin

Kotlin was selected as the programming language to operate alongside Spring Boot, as it has modern syntax and utilizes capabilities of the JVM, making it appealing for back end development. Out of the box, Kotlin is somewhat less verbose than Java, allowing exposition of clearer and more readable code. This should increase productivity and diminish the incidence of errors.

Null safety features in Kotlin are a general way around `NullPointerException`s—a pain point for Java developers. With Kotlin, null references are checked statically at compile time, helping to avoid runtime crashes and making the codebase more resilient. Kotlin also supports functional programming to allow developers to write very expressive and efficient code with higher-order functions, and lambdas being two of the many functional constructs. [14]

Another reason to choose Kotlin over Java or Go is that it's fully interoperable with Java, meaning you can use all kinds of things from the vast Java libraries and toolset without question. While Golang allows for a few things with simplicity and does concurrency really well, it lacks a few object-oriented and functional features that Kotlin provides. For an enterprise project that will benefit from JVM tooling, Kotlin's extensive feature set, along with its integrative capability, provided a better fit.

2.6.3 PostgreSQL

PostgreSQL was selected due to its overall compatibility with the requirements of this project, such as consistency, reliability, and scalability. The system is open-source and is a relational database meant to provide strong consistency and availability, fitting statuses for those requiring ACID transactions and integrity of information. This adherence to consistency leads to an excellent fit for applications that need accurate data.

It also integrates well with Spring Boot through Spring Data JPA, presenting an easy way to perform database operations. It uses object-relational mapping (ORM), allowing the developers to create, read, update, and delete entities without having to create large SQL statements. This adds to more efficient data handling and reduction of boilerplate code for these operations.

Another reason for selecting PostgreSQL is its versatility. It helps incorporate sophisticated features like indexing, JSONB data types, and SQL standard compliance, which offers great flexibility for varied use cases. Summarily, it is an efficient option with reliable results when it comes to executing more complex queries for data requirements of this project. Also, being open-source has quite a lot of support from the community, which sporadically makes for timely solving of issues and keeping operational costs low.

Choosing the right database depends on specific needs of projects, and there are many possible alternatives available in the market to PostgreSQL, such as MySQL and MongoDB. MySQL is easy to use, fast, along with easy installations and simpler transactional needs. However, PostgreSQL outmatches MySQL with some relatively richer features, making it the obvious choice and advantageous for handling advanced queries with JSON data, for example. On the contrary, being a NoSQL database, MongoDB offers a great deal of flexibility in managing schema-less data and doing justice to large volumes of unstructured data, however where clear-structured relations exist and are required by the data with ACID-compliant transactions is where it is opined the strengths of PostgreSQL arise.

2.6.4 Splunk

Monitoring the performance of applications is essential in today's complex technological environment. By gathering, analyzing logs, and metrics, teams can observe in real-time how apps function. This anticipatory approach allows issue identification and resolution before escalation into major issues. In this area, one could not ask for a more powerful software than Splunk, which allows in depth search into how the system behaves.

A centralized repository for logging and metric-tracking is invaluable when some unusual event occurs. Aggregating data from different sources permits Splunk troubleshooting to be highly streamlined. Instead of manually going through pages of logs, teams can easily search and filter for the information they need to identify the root cause of win issues more efficiently.

Furthermore, Splunk offers custom dashboards to enable the conversion of raw data into structured actionable intelligence. Important information can now be represented in a format that is very easy to read with trends, anomalies, and key performance indicators saving precious time in accessing repetitive sets of information. It not only helps the technical teams in diagnosing issues, but it also allows all the stakeholders to gain a comprehensive view of the general health of the whole system.

2.7 Scrum and CI/CD

To provide support for the constantly evolving environment of software and requirements engineering, flexible contracts, changes in technologies, and the complexity of products, new work methodologies also need to evolve together. The traditional project management methodologies have started to significantly fail. Let's take Waterfall for example, a methodology that assumes that there must be a well defined start-to-finish plan, where each phase can be completed without coming back to the previous one. That is perhaps an indication that this approach is not well suited for today's constant change of requirements.

Scrum emerged in response to traditional methodologies like Waterfall, that struggles to survive the test of time on the basis of constant requirements changing. Scrum, a framework within the Agile methodology, is used to manage complex projects and software development. This particular process supports changing requirements utilizing iterative development, wherein the product is built in a series of cycles called sprints. In this process, teams can respond to the changes quickly while incrementally adding value. Scrum's visibility began to grow from the 2010s as organizations sought independence and flexibility to adapt to shifting goals in their development pipeline.

Today, Scrum is one of the most widely used methodologies in the Software Engineering field, as it creates a defined yet adaptable structure for teamwork that supports constant development. For example, during the retrospective held at the end of each sprint, teams reflect on their processes and identify opportunities for improvements; thus, they support a culture of continuous improvement. Since Scrum fosters visibility, examination, and flexibility throughout the development cycle, it has become a preferred choice by many development teams.

As described in the official scrum guide[15], scrum framework is built around three key components: Sprint Team, Sprint Events, and Sprint Artifacts. These components create a complete and coherent way of working that provide not only a dynamic and constant collaboration between team members, but also efficiency in processes and delivering value in a fast and iterative way.

Sprint Team

The Scrum Team is composed of three main roles:

- *Product Owner*: The Product Owner is responsible for maximizing the value of the product and for the management of the product backlog. On the product owner's side, everything is done to ensure that the scrum team spends time

delivering things that offer the maximum value to stakeholders and help the team organize product objectives and priorities.

- *Scrum Master*: The Scrum Master is the facilitator of the Scrum process and that the team adheres to Scrum practices and principles. He delivers remove impediments to progress, creates a conducive environment for working, and trains team members in self-management and cross-functionality.
- *Developers*: The developers are all the people giving their contributions to deliver projects with potential increments in each of the sprints; they plan the work for the sprint while the onus is on them to satisfy the definition of done.

Sprint Events

Scrum makes use of a number of events to provide regular opportunities for inspection and adaptation. Each event is a formal opportunity for reviewing progress and making changes if necessary:

- *The Sprint*: It is the base, time-bound unit of development, generally lasting from one to four weeks. It is a container for the other Scrum events and creates a stable rhythm for the team. During the sprint, changes are not to be made that will hinder the delivery of the sprint goal.
- *Sprint Planning*: Each sprint begins with Sprint Planning. In this event, the team together decides what work is to be possessed and how it is to be done. The Sprint Goal is defined in this meeting, which provides a shared goal for the sprint.
- *Daily Scrum*: It is a brief daily meeting, usually held at a fixed time, which is usually called the daily stand-up. The developers discuss what they did the previous day, what they plan to do today, and what obstacles they might face. It allows the team to synchronize their activities and be transparent.
- *Sprint Review*: At the end of any sprint, a sprint review is held by the whole team to inspect the Increment and obtain feedback from the stakeholders. This meeting serves as an opportunity to inspect what was done in the sprint and adapt the Product Backlog as necessary.
- *Sprint Retrospective*: Following the sprint review, the retrospective is done to reflect on the sprint process. This is an opportunity to evaluate what went well, what did not and examine ways for gradual improvements for the team in the next sprints.

Sprint Artifacts

Scrum defines three main artifacts that represent work or value:

- *Product Backlog*: The Product Backlog is a product artifact that is an ordered list of the necessary features that are still to be developed for preparing a product. The customer and the development team both contribute and agree on this list since it is the foundation of any activity. It acts as the team's work plan in a way, since the team uses this plan as the basis for their Scrum tools. It is the single source of requirements and is constantly updated as new insights are gained.
- *Sprint Backlog*: The Sprint Backlog is a list of tasks drawn from the product backlog and selected for a sprint. To break down the tasks, the team first estimate the amount of work in a relative unit of time. In the middle stage of the sprint they are visually emptied and enables the team to focus more on achieving the sprint goal and completing their activities. It is owned by the developers and communicates the team's work during the sprint.
- *Increment*: An Increment is developed by adding all the Product Backlog items completed during a sprint combined with the value that is obtained from the previous increments, where each of them must meet the definition of done.

2.7.1 CI/CD Pipelines

To achieve a fast and automated workflow, the project makes use of Continuous Integration (CI) and Continuous Deployment/Delivery (CD) pipelines using tools like Jenkins and GitHub CI. These pipelines streamlined the development process from code commits to deployment. These are important steps of delivering products fast, which usually are overlooked because these processes are running on the background.

Continuous Integration (CI) is the practice of frequent integration of changes into the main branch of the codebase. Each change is automatically built and tested to catch the errors and conflicts early. Use of Jenkins and GitHub CIs made automated builds, unit and integration tests easier, and Sonar's analysis, run whenever a pull request hits our main branch. This method of doing things made it easier for us to keep the most recent copy of the codebase in a stable state and thus be able to work on further development.

Continuous Deployment/Delivery (CD) deploys the changes that passed CI and the decision is made automatically, the changes are deployed to either the

staging or production environments, whichever is the deployment strategy. Jenkins and GitHub Infrastructure were used for automated deployments, which not only minimized manual mistakes but also sped up the release of new features. It was through Continuous Delivery that we assured the automatic insertion of the code into the production environment was viable, from which it was then just a matter of selecting the right time for the updates to be released.

Implementing CI/CD pipelines provided several advantages:

- *Reduced Manual Effort*: Automating testing, building, and deployment minimizes manual intervention and the chance of human error and saves time.
- *Faster Feedback Loop*: Automated processes provides quick feedback to developers, allowing them to identify and fix issues promptly.
- *Improved Software Quality*: Continuous testing and integration helps detecting regressions or potential issues early, leading to more reliable releases.

Chapter 3

System Architecture

This chapter provides an in-depth study of the design and structure that has been evolved for this project. It discusses the current process and monitoring workflow and identifies major issues and inefficiencies that have been addressed in the new system. It describes the objectives of the project, then presents architecture of the system and explains constituent parts and the way they relate to each other. Backend design is discussed thoroughly, including how it communicates, its integration with external databases, security, and the structure. Understanding the system architecture clarifies how technology selection and design decisions were made to make compliance easier and operations more efficient.

3.1 Understanding the old process

The project itself presented across this work is a complete revamp and modernization of a legacy process where investments were monitored through running manual procedures requested on demand, with the use of email for official communication and tracing of activities and excel spreadsheets for analysis of investments. Before the implementation of the system described in this thesis, the constant manual labor, semi-automatized processes and the lack of a centralized way to manage these investment lines made the old process error prone, which asked for the creation a dedicated application.

The old process relied on several manually triggered SQL Server procedures which would update support tables to reflect the updated picture of the customers investments. After aligning with recent data, the investment monitoring would then be generated, which is a static picture of clients investments that are inadequate.

Following the availability of the data present in the generated report, it was communicated to the wealth management (WM) team that the data was ready. The WM team is responsible of analyzing the data, managing investment lines and requesting the communication of other dedicated teams that take actions on these investments. The majority of communication and interaction between different parties associated with this process was made via email, which although is convenient and effective, it lacks on traceability and auditing.

The way the WM team would analyze the data and manage the investment lines was through excel files. With around 10 to 15 thousand lines in each excel, examining thoroughly the data was not straightforward. Excel itself is a very complete tool and its uses widespread, filters, advanced formulas and mature data handling can definitely create a useful tool. However, this application was used more like a database, which stored the investment lines in a structured data, leading to inefficiencies in deeper analysis.

3.2 Understanding the monitoring process

The newly implemented system, known as MiaGP (Integrated Adequacy Monitoring for Wealth Management, or in Italian *Monitoraggio Integrato di Adeguatezza per le Gestioni Patrimoniali*), as developed with the purpose to address some of the most basic gaps that existed in the previous system in place for slightly less than one decade. MiaGP aggregates all the investment line data into a single repository where the WM team and other stakeholders can then thoroughly analyze this information and manage it satisfactorily. The benefit of this centralized improvement is both on availability and accuracy, which is critical in decision-making. Scheduling the transmission of official notices triggered by defined conditions will also automate decision communication and ensure uniform and timely communication among all concerned parties.

Before going into detail about architecture, implementation, and system components, background and system needs must be understood.

MiaGP incorporates all the logic of the previous process, as the end goal remains the same: to perform periodic monitoring of inadequate investment lines. The new system, however, introduces convenience and enriches the process with yet more features. But what in the first place is meant by the "inadequate investment line" and how does the system know about it?

In wealth management, a private banker (PB) is responsible for making investment decisions on behalf of clients, who in turn decide the investment strategy they want the PB to follow. When there is a misalignment between the client's risk

expectations and the actual risk of the investments, the investment is considered inadequate. This misalignment can happen under several conditions: the client's MiFID questionnaire may be expired, the client may lack a defined risk profile, or the PB may have chosen products that carry a higher risk than the client prefers. Such situations can have significant financial implications, making the correct identification of inadequate lines essential.

These are what we call inadequate investment lines, named as such because there is a snapshot of the client's respective investment instead of an actual investment; these get captured into the monitoring process. Monitoring is done roughly every 45 days, giving a snapshot of pooled investments on all clients, to gauge if they comply and what actions ought to be taken. Also, different from this monitoring, MiaGP routinely watches these inadequate lines every week and assesses investment lines that seem inadequate between monitoring dates so that teams can take a first look for possible flaws in coming records.

New investments that appear during monitoring are tracked by associating each investment with its contract, which enables the system to assign it a status. This way, repeated instances of inadequacy from previous monitoring cycles can be clearly identified. These statuses are called cycle statuses and can take values of R1, R2, R3, R4, RX, or FC. When an investment line first appears as inadequate, it is typically assigned the status R1, and the status progresses to R4 if the inadequacy persists in subsequent monitoring cycles. The status FC (which stands for "out-of-cycle" or *fuori ciclo* in Italian) indicates lines that, for specific reasons, are excluded from the regular R cycle.

The out-of-cycle status is generally applied when exceptions are made to the usual logic that would consider an investment inadequate and require corrective actions. Examples of such exceptions might include minor deviations in the percentage of risk that would classify a line as inadequate, or investments that do not meet the minimum required investment threshold.

As investment lines progress through the R statuses, specific actions and communications are undertaken. Initially, the PB is notified of the inadequacy and given an opportunity to take corrective action (R1). Subsequently, the client receives an official notification regarding the situation (R2), followed by further communications as needed. When a line reaches R4, a crucial action is triggered: a notification is sent to an automated system that forcefully change the client's investment to a more conservative option. This procedure is known as CLU (office line change or *cambio linea d'ufficio*).

3.3 Project description and objectives

Now that we have the general information of the system in place, we can discuss the detailed requirements of the system and the implementation aspects. It is important that the transition is perfectly transparent and barely noticeable in case we want to improve the existing process, so ensuring the change is done smoothly and with the least disturbance is crucial. One very important requirement is that the system be auto-adapting, which will make it capable of seamlessly integrating with the operations that are already ongoing. Such benefits may also include the anticipated outcomes like a fully-automated data processing system, an improvement in communication patterns, and a higher level of analysis efficiency as well as the addition of new elements that will be useful for the teams using the application.

The project's aims are not only about copy-pasting the current legacy process' functions, the main idea of the project is to create a more robust and sustainable infrastructure that leads to better decision-making. The following high-level requirements have been established to meet the above-mentioned goals: effective data acquisition, proper monitoring, and validation and enabling the user to exercise control over the calendar management operation.

In greater detail, the three different categories that the functionalities can be split are:

- *Calendar Management:* This is the most intuitive part of MiaGP, since it is the final user interacting with the front-end to manage monitoring runs. Amongst these are creation and deletion of monitoring runs, validation or invalidation of particular monitoring sessions, view of investment lines, analyzing trends as well as updating the status of investment lines. This calendar management function has a key role in overseeing the investment adequacy process and offers a fair amount of flexibility for adapting to changing scheduling needs. For example, invalidating or re-running a monitoring session allows the team to respond to data quality issues.
- *Data Ingestion:* Data ingestion is at the center of the application. It is responsible for collecting and processing data on the investments of the clients. This module's primary responsibility is the combination of data from disparate data sources and their transformation and consolidation into a single dataset that can be analyzed for inadequacies. We are dealing with sensitive personal data of clients, therefore, accuracy and precision are of utmost importance.
- *Validation and Spools:* Validation is the end result of an official monitoring run. When the WM team validates the monitoring, it indicates that the data has proven to be accurate, and the system is ready for further actions, notification

delivery, or CLU communication. The term "spools" which was adopted during development, is used here to refer to the automation workflows that handle such actions. Spools guarantee that, once the data is validated, the required actions are undertaken immediately and precisely with no manual intervention whatsoever. The relevant level of automation improves operational efficiency significantly and reduces chances of human errors.

The back-end system has been built using the Spring Boot framework, with Kotlin as the programming language. In this chapter, we will detail the design choices made, including the identification of entities and the construction of the relational model for the PostgreSQL database.

Learning Objectives

The project's learning objectives are closely linked to gaining practical experience in the design and implementation of a sophisticated software system. The specific learning outcomes are as follows:

- *Designing a microservice:* This project includes the design of one microservice following best practices and established design patterns. The application was not designed with the thought of being used by other microservices, since it focuses on a very specific task. However, that said, it mainly relies on the microservice architecture to be able to communicate with other services in a robust way while performing its tasks.

Deepening technological skills with Spring Boot and Kotlin: The combination of Spring Boot and Kotlin is expected to afford this opportunity for deepening technological skills that are being applied well in the current tech world. The ultimate goal is to be able to fully extract the capacities of both the technologies by developing clean, performant, and high-quality services.

- *Using State-of-the-Art Tools and Libraries:* During the development period, several new tools and libraries are used to extend the functionality of the system. Use of these tools not only improves technical skills but also helps build a more flexible and robust system.
- *Working within a Scrum team:* The project is developed within a Scrum-like environment, therefore underscoring continuous delivery and collaboration. Experience is gained in iterative development, backlog prioritizing, and stakeholder involvement, which are all essential to accomplishing what is required for successful completion of this project. A major objective is to learn how to break down complex requirements into manageable tasks, prioritize work

based on business value, and consistently deliver incremental improvements to stakeholders.

3.4 Software design

To provide the level of trust in any application through which client investment may be handled, there were many individuals involved, more so towards the bigger projects. At the beginning of the project, the responsibilities of the products manager and the software architect included finding out high-level business and product requirements for the legacy system, as well as defining new functionalities. For smaller projects, it may not be unusual to have software engineers take an active part in gathering requirements; in this case, it became quite an efficient way to let developers focus on user stories and the actual implementation of functionalities that had been agreed upon with the stakeholders.

Once functional requirements are collected and broken down into manageable parts, they are translated into user stories. User stories define smaller, actionable bits of the entire system that can be separately worked on, with criteria of acceptance determining what is expected to result. While initial requirements can frequently be determined through early discussions with stakeholders, putting the final user story together is often a process that can take some considerable time. As mentioned in the section 2.7, the product owner is responsible for breaking down the larger features into different user stories, and this process is improved with time during the refinement sessions, where the technical team collaborates in discussing the implementation process in order to ensure that both business and technical objectives are aligned.

From the three categories defined in the previous section, the final user stories below are organized and assigned accordingly. This follows the traditional user story format: "AS a ... I WANT to ... SO THAT ...," which helps identify the actor, the action, and the business value.

According to the three types of user stories in the previous section, the final user stories are organized and assigned as follows. This follows the traditional user story format: "AS a ... I WANT to ... SO THAT ...," which helps identify the actor, the action, and the business value.

- *Calendar management:*

1. AS a WM team user

- I WANT to create a new official monitoring ¹
SO THAT I can create a picture of clients inadequate investment data on the chosen date
2. AS a user
I WANT to see all the official monitoring runs
SO THAT I can see have a complete list of them
 3. AS the system ²
I WANT to create automatically a weekly monitoring
SO THAT users can have it consistently generated on the same day each week.
 4. AS a user
I WANT to see the past four weekly monitoring runs
SO THAT I can review previous weeks' monitoring results.
 5. AS a user
I WANT to filter the official monitoring runs by date or year
SO THAT I can obtain precise results.
 6. AS a user
I WANT to check all the investment lines in a monitoring
SO THAT I have a complete picture of the inadequate lines
 7. AS a user
I WANT to see the monitoring trends
SO THAT I can compare the number of inadequate lines across previous runs
 8. AS a user
I WANT to download in an excel all the investment lines of a monitoring
SO THAT I can share it with others or conduct my own analysis
 9. AS a user
I WANT to filter lines with errors
SO THAT I can understand the reasons behind their inadequacy
 10. AS a user
I WANT to apply multiple filters to investment lines
SO THAT I can refine my search and find relevant data

¹We are using the word monitoring as the process conducted at a specific date to review client investment lines for compliance with predefined adequacy criteria

²Although using the system as an actor is an open debate, there are situations where the system itself has to act as an agent, which ultimately has to result in business value added to the end user

11. AS a WM team user
I WANT to delete an official monitoring set for the future
SO THAT no data will be imported on that specified date
12. AS a user
I WANT to validate a monitoring SO THAT spools can be generated by the system for automated actions

- *Data Ingestion:*

1. AS the system
I WANT to import client investment lines that do not meet adequacy criteria
SO THAT appropriate actions can be initiated to fix the inadequacies
2. AS the system
I WANT to send an email to users when the data is available SO THAT they can proceed with the analysis

- *Validation and Spools:*

1. AS the system
I WANT to communicate with private bankers
SO THAT they are informed about their clients' inadequate lines and can take corrective action
2. AS the system
I WANT to send a communication to customers about their reoccurring inadequate investment lines
SO THAT they are aware of the issue
3. AS the system
I WANT to create and upload the CLU (forced change of investment) file to Google Cloud Store
SO THAT clients' investments can be regularized
4. AS the system
I WANT to send communications to another office specifically about investment lines of deceased clients
SO THAT the heirs of the clients can be communicated about the inadequate investments
5. AS the system
I WANT to send communication to the compliance office
SO THAT they can be updated about inadequate investments

6. AS the system

I WANT to send to a another office all personalized lines ³

SO THAT appropriate specialized management can be undertaken

7. AS the system

I WANT to send communication to another office for inadequate lines where the client is a minor

SO THAT communication can be directed to the parents or responsible guardians

As evident from the detailed user stories, the number of user stories for calendar management is greater compared to the other categories. However, taking a deeper look of the tasks within each category, it becomes clear that the complexity of data ingestion and validation is higher than that of calendar management. While the calendar management mostly concentrates on CRUD operations, the data ingestion and validation require completing tasks that are risk-prone, such as importing data accurately, coordinating communication among different parties, creating customized files based on recipients, creating fail-safe mechanisms that provide reliability, etc.

3.4.1 Entities

The entity-relationship diagram (ERD) presented below provides a visual representation of the core tables and relationships that define the system's data model. Thinking of entities and how they connect together will establish a good picture of the structure of the database so that every relevant data point is represented, and relationships are right. The ERD allows a better understanding of how data flows through the system. As the relations are mapped, the design of the database can be made properly aligned with the functional requirements of a system, eventually paving the way for design efficiency and easy maintenance.

³a personalized line in this context refers to a special investment solution due to its high value or unique nature

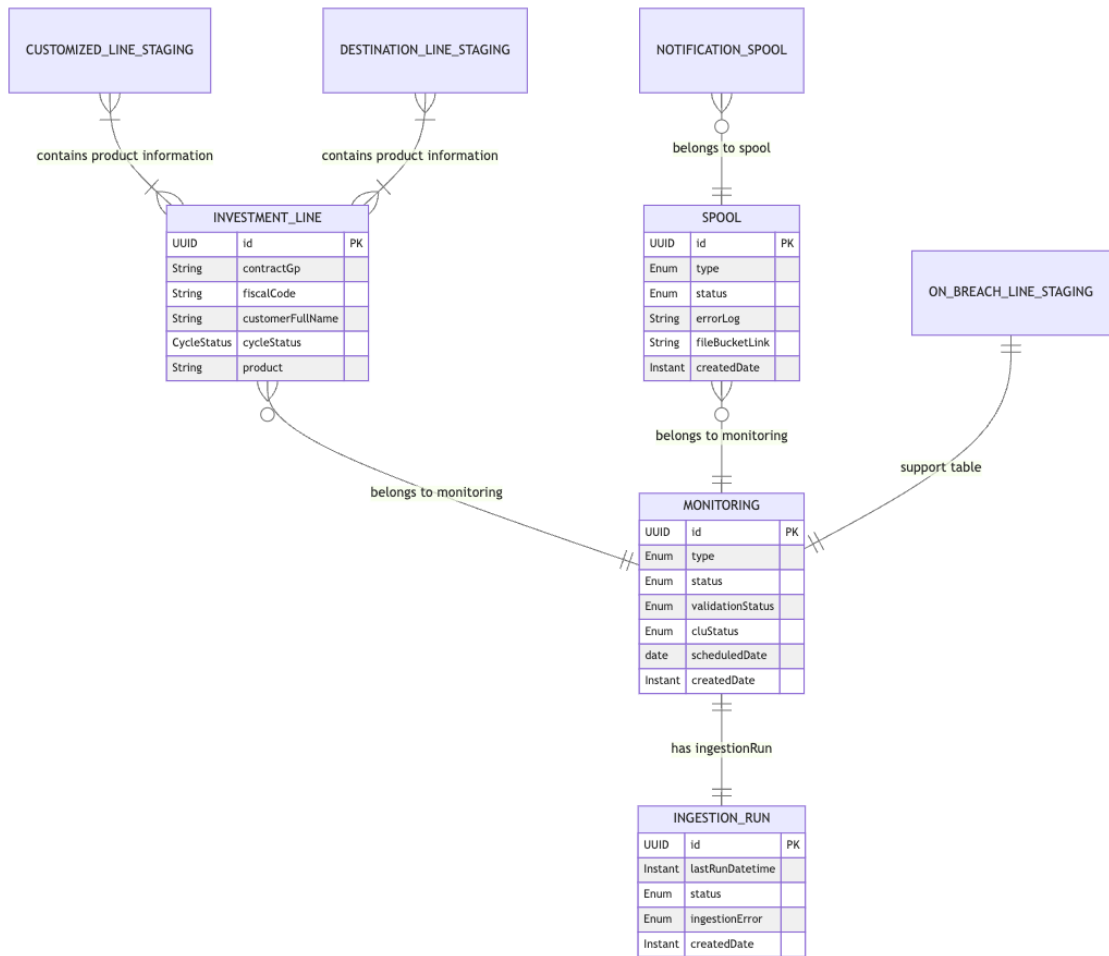


Figure 3.1: Entity-relationship diagram for MiaGP system

As seen in figure 3.1, the tables and relationships in this project are relatively straightforward. Nevertheless, to achieve this clarity, the data several times had to be refined to focus on what was absolutely necessary, with discussions and sessions with the team architect and backend developers to create alignment with system specification and scalability targets.

The first step is to create a representation for the MONITORING entity, as it is central to the system’s primary functionality. For each of the entities mentioned in this case, the one that possessed certain significant characteristics or attributes representing it has been highlighted. Now let us discuss the main fields and the information they possess:

- ***scheduledDate***: Defines the date when the official monitoring should occur

or when data for inadequate investment lines should be imported.

- ***type***: Enumerated field used to distinguish between official and weekly monitorings, as they are handled differently.
- ***validationStatus***: Indicates whether a monitoring has been validated; upon validation, the spools are generated.
- ***cluStatus***: Represents the status of the CLU process, which is arguably the most critical step involving forced changes in investments.

The INVESTMENT_LINE entity holds information about inadequate investments. Most fields from this entity are excluded from the diagram as they are specific to the wealth management context and mainly informational. These fields may include details about the private banker, investment product, capital invested, etc. Key fields include:

- ***gpCodLink***: Serves as a unique identifier for investment lines recurring in monitoring cycles. It is generated from contract details and is used to identify specific investments a client holds.
- ***contractGp***: Serves as the basis for constructing gpCodLink.
- ***fiscalCode***: Represents the customer's fiscal code (codice fiscale).
- ***profile***: A value between 0 and 4 that describes the client's risk profile. A value of 0 indicates an expired profile (expired MiFID questionnaire), while 1 represents the lowest risk and 4 indicates the highest risk.
- ***varMax***: Represents the maximum allowable percentage of risk associated with the client's risk profile.
- ***varPtf***: Indicates the percentage of risk present in the client's current portfolio.
- ***exclusionOrException***: As detailed in section 3.2, certain investments are placed out-of-cycle due to exceptions, meaning they are excluded from the regular monitoring cycle.
- ***cycleStatus***: Describes the recurrence status of a line, taking values from R0 to R4, RX, and FC, as explained in section 3.2

The INGESTION_RUN entity has a one-to-one relationship with MONITORING, as it represents the ingestion or import of data on a given date. The ingestion

process is managed through a scheduled Spring job that runs at specific times of the day to import monitoring data. The ingestion process must be reliable and capable of recovering from failures, such as unexpected errors or pod crashes. The fields in this entity are:

- ***lastRunDateTime***: Records the last date and time when ingestion was performed. It is a field used to ensure reliability, as it is used in a fail recovery logic.
- ***status***: Indicates whether the ingestion was successful or encountered an issue. Possible values include `READY_FOR_INGESTION`, `IN_PROGRESS`, `ERROR`, and `COMPLETED`.
- ***ingestionError***: Provides debugging information to help determine the cause of any errors during ingestion.

The `ON_BREACH_LINE_STAGING` table serves as a support table similar to `INVESTMENT_LINE` and will carry temporary data during the ingestion process. Interim results obtained from data import queries use this table, which will in turn be manipulated to populate the `INVESTMENT_LINE` table.

The `CUSTOMIZED_LINE_STAGING` and `DESTINATION_LINE_STAGING` tables store information about investment products used by specific processing flows.

Having established a basic understanding of the entities and how their relationships, we now get into detailing how those entities are represented and treated in the system via the Spring Framework and Spring Data JPA.

Spring Data JPA offers a powerful abstraction layer development framework for dealing with PostgreSQL databases. This means that, with simple annotations, we get a good representation of the entities in the system and their relationship definitions. The abstraction layers that Spring Data creates have an additional advantage of simplifying the connection to the database so that one doesn't have to deal with much boilerplate coding traditionally entailed in persistence solutions. Be it PostgreSQL in this case or any other RDBMS, most of the complexities associated with this are abstracted away by Spring, allowing the developer to concentrate more on a fluent, Kotlin-style of development. Features such as entity relationships, cascading, and automatic auditing (to keep track of the creation/modification dates) can be easily configured with annotations so as to promote the cleanliness and consistency of a domain model while ensuring data persistence and integrity.

In order to not repeat ourselves by showing how all the entities are implemented, let's look at how the monitoring is done.

Listing 3.1: Example of an implemented entity

```

1  @EntityListeners(AuditingEntityListener::class)
2  @Entity(name = "mme_monitoring")
3  @SQLDelete(sql = "UPDATE mme_monitoring SET status = '$DELETED' WHERE id=?"
4  )
5  @Where(clause = "status <> '$DELETED' OR status IS NULL")
6  class MonitoringEntity(
7      @Id @GeneratedValue val id: UUID? = null,
8      @Column(name = "type") @Enumerated(EnumType.STRING) var type:
9      MonitoringType? = null,
10     @Column(name = "status") @Enumerated(EnumType.STRING) var status:
11     MonitoringStatus? = null,
12     @Column(name = "validation_status") @Enumerated(EnumType.STRING) var
13     validationStatus: ValidationStatus? = null,
14     @Column(name = "clu_status") @Enumerated(EnumType.STRING) var cluStatus
15     : CLUStatus? = null,
16     @Column(name = "scheduled_date") var scheduledDate: LocalDate? = null,
17     @Column(name = "latest_data_avail_on") var latestDataAvailableOn:
18     LocalDate? = null,
19     @OneToOne(cascade = [CascadeType.PERSIST, CascadeType.MERGE])
20     @JoinColumn(
21         name = "ingestion_run_id",
22         nullable = true
23     ) var ingestionRun: IngestionRunEntity? = null,
24     @Column(name = "created_date") @CreatedDate var createdDate: Instant? =
25     null,
26     @Column(name = "last_modified_date") @LastModifiedDate var
27     lastModifiedDate: Instant? = null,
28 )

```

We can see some recurring patterns in this code as well as in the declaration of entities in general. The main feature here is the use of annotations to declare particular behaviors and configurations, recognized, and then processed by Spring. These annotations enable short and readable entity definitions, allowing Spring to manage complex tasks such as database mappings, auditing, cascading, and custom behaviors. Below is a detailed explanation of each annotation used.

- *@EntityListeners*: This annotation is used to attach one or more entity listeners to an entity. Here, we use `AuditingEntityListener` to automatically update auditing fields, such as `createdDate` and `lastModifiedDate`.
- *@Entity*: Marks the class as an entity to be mapped to a table in the database. The optional parameter `name` is used to specify the name of the table if it is different from the class name.
- *@SQLDelete*: This annotation adds some more customization to the deletion operations. Instead of deleting records from the table, we execute an SQL

query to set the `status` field to `'DELETED'`. This mechanism implemented allows soft deletes whereby deleted rows still remain in the table. When querying, these deleted rows may be filtered out.

- *@Where*: The annotation specifies an SQL where clause that is automatically added to every query involving the annotated entity. Here the `@Where` clause ensures that records with a `status` of anything other than `'DELETED'` are returned in results-excluding soft deleted records.
- *@Id*: Marks the field as the primary key of the entity. Each instance of the entity will have a unique identifier.
- *@GeneratedValue*: Specifies that the primary key value is automatically generated. In this example, a UUID is generated to ensure that each record has a globally unique identifier.
- *@Column*: Used to map a field to a column in the table. The optional parameter `name` allows us to explicitly define the column name in the database, which may differ from the entity's field name.
- *@Enumerated*: This annotation is used for fields that are enumerations. It specifies how the enumeration should be persisted. Here, `EnumType.STRING` is used, which means that the enum values are stored as their corresponding string representations, ensuring better readability of the database values.
- *@OneToOne*: Defines a one-to-one relationship between entities. In this case, `MonitoringEntity` has a one-to-one association with `IngestionRunEntity`. The `cascade` attribute defines the cascade behavior, which allows the associated `IngestionRunEntity` to be created or updated automatically when changes are made to `MonitoringEntity`.
- *@JoinColumn*: This annotation is used to specify the foreign key column that maps the relationship. In this case, the column `ingestion_run_id` serves as the link to the `IngestionRunEntity` associated with each monitoring entity. The `nullable = true` parameter indicates that the relationship can be optional.
- *@CreateDate*: This annotation marks the field as the creation timestamp of the entity. Spring Data JPA will automatically populate this field with the current timestamp when the entity is first persisted.
- *@LastModifiedDate*: Similar to `@CreateDate`, this annotation is used to track modifications to the entity. Whenever the entity is updated, Spring Data JPA automatically updates this field with the current timestamp.

With an understanding of how the entities and tables were designed in this application, we now turn toward the architectural design. The use of Spring Data JPA became a major contributor in simplifying data persistence implementation which gave us ample space to concentrate on core business logic with little regard to boilerplate database operations.

3.5 Back-end Architecture

Also discussed in the prior sections is the microservice architecture on which the back-end system is developed. The core microservice used to manage and import investment lines is dependent on many other services to provide functionalities such as authorization and sending notifications. This section will describe the various designs, dependencies, and external connectivity of the system. User stories will be converted into sequence diagrams and a brief review of the code of the system will occur, highlighting important design patterns and allowing for diverse functionalities to meet various system requirements.

Let us take a look into the system architecture and its components with the help of figure 3.2:

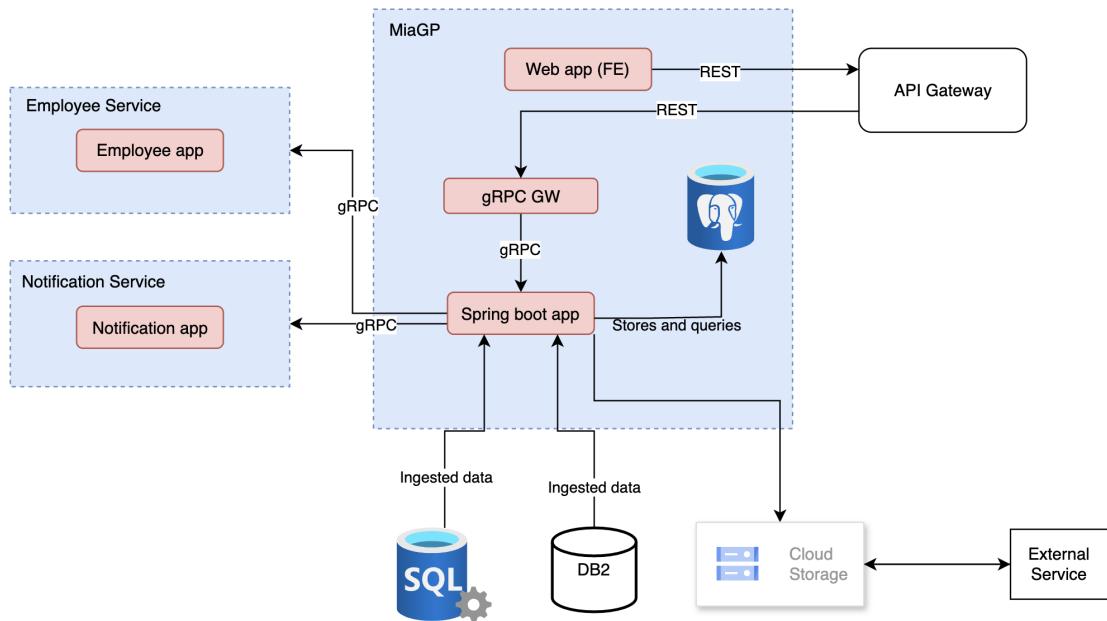


Figure 3.2: System architecture for MiaGP

At first glance, the architecture may seem to contain a few microservices.

However, this has become increasingly difficult due to multiple external dependencies that may add considerable complexity. Using external services can soon become problematic in the performance of certain reliable systems if that issue is not entirely managed. The architecture demonstrates two microservices that use gRPC for communication, as well as two external databases, which the MiaGP microservice accesses directly for importing data on the inadequate investment lines. We'll discuss those in-depth in subsequent paragraphs.

The MiaGP system is developed upon making full use of the Spring Boot and Kotlin stack to provide a complete back-end solution. It exposes gRPC methods while defining associated HTTP verbs and URIs for each endpoint. The front-end application uses HTTP to communicate with the back-end, first with the general-purpose gateway that will, in turn, forward the requests to the respective application gateway. The application API gateway and the specific gateway were both written in Go, where the first is responsible for acting as a single point of entry for all applications and the second for translating HTTP requests to gRPC binary objects and then the gRPC responses to HTTP⁴.

The use of Google Cloud Storage (GCS) is a core part of the internal functioning of the entire system. The system must upload a custom file to a GCS bucket through an automated process, particularly the spool responsible for sending the CLU. An external service picks this custom file up to process investment lines that need to be forcefully switched to a more conservative option.

Besides using SaaS cloud computing services such as Google Cloud Storage (GCS), we also use Platform as a Service (PaaS) to deploy our microservices on Google Cloud Platform (GCP). This CI/CD process is quite efficient, applying progressive CI/CD (continuous integration and delivery) best practices. Integration with GCP is easily achieved by means of tools such as GitHub Actions which execute pipelines in pull requests and merges in order to enforce quality of the code, build the project, run tests and set up automated deployments to GCP.

In order to maintain code quality metrics, SonarQube scans are inserted into pipelines, which officially share feedback on code quality. Moreover, Backstage, an open-source platform for developer portal development, plays a critical role in keeping track of the various environments, versioning, and allowing for one-click deployment. This centralized visibility greatly simplifies the deployment process.

All these processes are supported by a dedicated platform team that ensures the infrastructure is stable and efficient, allowing developers to focus on building

⁴The gateways are out-of-scope for this thesis and will not be covered, since they are already implemented and widely used by the company

features and delivering value quickly.

3.5.1 Communication between services

Among the many things dealing with interconnected systems is information flow. Well, gRPC would usually be the right way for communication between microservices because it combines efficiency and low latency with the benefits of automatic object generation. We will see, since it's implemented using gRPC, developers benefit from using Protocol Buffers (protobuf), which allow them to encapsulate the data definition structures, automatically generating the objects used as Data Transfer Objects (DTOs) and used by the application. This saves lots of boilerplate code while speeding up development time since the same protobuf definitions are used in different services.

True, this introduces some degree of coupling between the API layer and the service layer, but ultimately it reduces redundancy and helps to keep services consistent. Debated among developers, this style is used in this project. The strongly typed aspect of gRPC means that any discrepancies between the expected message structure and the actual data that gets sent will raise compile-time issues that make it far less likely for there to be issues at runtime, with far less edge-case handling in the code.

In addition, the outputs generated can be easily incorporated in any other service making use of the same gRPC endpoints, thereby providing a unified, efficient data workflow throughout the system. This reusability is particularly helpful when the additional functionality is offered, as it makes both the development and integration of new services (i.e., the notification and employee services) easily achievable while the message formats are maintained as similar across microservices.

While gRPC is efficient inter-microservice communication, REST remains the better choice for client-facing APIs, especially web clients, due to its simplicity and wide compatibility. Built on top of HTTP, REST allows integration with any front-end systems and external applications. Here, we have an architecture that uses gateways to bridge REST and gRPC, effectively translating RESTful requests into gRPC calls.

The use of gateway allows us to keep client interactions simple, simultaneously hiding the complexity of the workings of gRPC from the clients. This approach decouples the client API from the underlying microservice implementation which makes it easier to change; switching the internal services doesn't involve altering the external interface. It centralizes security, rate limiting, and authentication behind a single access point while keeping internal services protected.

Using the gateway pattern allows us to achieve a kind of balance where we still enjoy the simplicity of REST, even though we sacrifice some performance in order to gain type safety for inter-microservice communications through gRPC. Achieving this level of setup can indeed offer better efficiency, scalability, and separation between the concerned parties.

3.5.2 External databases

If we were to strictly adhering to the microservice architecture principles, one might argue that relying on external databases as direct data sources is violating the separation of concerns-and in this case they would be right. However, real-world system design often involves making trade-offs to balance practicality, complexity, and performance needs. In our system, we have two external database connections, one SQLServer and another DB2 instance, and they are crucial for ingesting the approximately 10,000 to 15,000 investment lines for every monitoring. By accessing these databases directly, we are able to query tables that hold client, investments, risk profile and products information and aggregate all this data to filter out the results we are looking for: identifying inadequate investments.

Integrating these external connections directly in the service makes for some advantages. The first is that the entire ingestion and aggregation logic can be held in one unified place, easing integration, and hence minimizing over-complexity in dealing with the architecture. One might have to dedicate two new microservices, for instance, to each external database. Having the logic tied to one service will cut down on the need to distribute transactions across different services, which can be challenging to manage, especially when massive sums of data are involved. This means that we will not impose the extra burden of managing multiple services for retrieval and duplication of code and data, making the whole process more efficient and manageable.

That said, there are some drawbacks to this approach. The major problem here is the lack of proper separation of concerns-directly accessing external databases means we lose control over those databases. Therefore, any changes to the structure of these databases could have a direct impact on our system. In this case, since we are talking about main databases of a vital service like a bank, we would not expect unannounced sudden changes that do not give us an opportunity for appropriately planned actions. Nevertheless, the aforementioned adds a sense of fragility to the architecture, as any external databases on which our service relies may evolve independently of our application. Tight coupling to these data sources limits our flexibility, therefore decreasing our ability to respond to change.

Ultimately, in answering the question of: why to directly query external

databases-it became a simpler solution with fewer moving parts, although retaining its efficiency during data ingestion. While this does not closely adhere to the recommended microservice principles, it remains a genuine real-world compromise between architectural purity and practical need.

3.5.3 Security mechanisms

When building applications that handle sensitive data, security must be a priority. Banking information, such as client personal data, investment details and portfolio information are protected under GDPR, which emphasizes the critical importance of securing it. Since the application is deployed in the bank's environments and MiaGP is a back-office tool, employees have to use VPN to access it. This is an indirect security procedure that adds an extra layer of protection, and it comes for free since there is no direct implementation being done.

Although the front-end is beyond the scope of this thesis, discussing its login procedures helps us have a better end to end understanding. The login procedure done on the FE relies on the bank's authentication system. During this process, the FE receives a JWT token generated by the authentication server and stores it as a cookie. This JWT token contains information about the user, for example their user id, authorizations and roles, and is included in all requests sent to the back-end. As shown previously in the 3.5, all the front-end requests first pass through a general API gateway, that already is capable of performing authentication checks. In this way, unauthenticated requests attempting to reach protected back-end resources are intercepted, and an error is returned.

This is a widely recognized architectural pattern where the gateway filters out requests that do not meet the required security levels. Having this centralizes the authentication logic in one single place, which allows for considerable one-time effort when implementing microservices architecture. However, authorization checks are required at the back end. Since MiaGP is a backend application, employees from different departments log in through the bank's authentication system, but only certain users should possess the required administrative privileges to carry out certain actions. Granular validation is carried out on an endpoint-by-endpoint basis to have this level of security achieved.

This process of verification can be supplied by Spring via the use of `@PreAuthorize` and `@Secured` annotations, which intercept method calls and enforce security rules in accordance with Aspect-Oriented Programming within Kotlin. However, in addition to checking the user's role from the JWT, it is necessary to invoke a request to the employee microservice for further confirmation. This is done by

having a custom annotation pointing out a join point⁵ and having custom logic to coincide with it.

Listing 3.2: Annotation declaration

```
1 @Target(AnnotationTarget.FUNCTION)
2 annotation class Authorize(val value: Array<String>)
```

Using the `annotation class` keywords, we define the custom annotation. After that, we create an aspect that applies the authorization logic to any method annotated with `@Authorize`, as outlined in the listing 3.2:

Listing 3.3: Annotation declaration

```
1 @Aspect
2 @Component
3 internal class AuthorizationAspect(
4     private val jwtExtractor: JwtExtractor,
5     private val authorizationService: AuthorizationService
6 ) {
7
8     @Around("@annotation(authorize)")
9     fun authorize(joinPoint: ProceedingJoinPoint, authorize: Authorize): Any? =
10         AuthorizationContextHolder.use {
11             val roles = authorize.value.toList()
12
13             val claims = jwtExtractor.getJwtClaim() ?: throw unauthorizedException("JWT
14                 not found in Context")
15             authorizationService.authorize(roles, claims)
16
17             joinPoint.proceed()
18         }
19 }
```

In this example 3.3, the `AuthorizationAspect` class defines an aspect that intercepts any method annotated with `@Authorize`. It extracts the roles specified by the annotation, validates the JWT claims, and invokes the authorization service to confirm the user's permissions before allowing the original method execution to proceed.

In summary, a strong security strategy is required to have multiple layers of verification and authentication at the gateway level and granular authorization checks for each action within the back-end. Although some of these security

⁵According to (<https://docs.spring.io/springframework/docs/2.0.x/reference/aop.html>), a join point is a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

measures implement company-provided libraries or the gateway handles them, it is important to highlight how the microservice achieves security and guarantees data integrity and compliance.

3.5.4 Project Structure

When implementing the project requirements and actually writing the code, adhering to patterns and design choices when starting the project is important to have a clean code-base, create less tech debt and ultimately deliver a better quality product. To understand how the project is structured, listed below is the table 3.1 containing all the packages under `src/main`. There are also the tests packages `functional` and `test` that replicate the structure under `main`.

Package name	Description
channel	Contains outbound and inbound communication points, such as the gRPC server, REST controller, outgoing gRPC clients, and scheduled jobs.
config	Spring Bean declarations, such as configurations for the databases, GCP bucket, exception handlers, access to application properties, and others.
exceptions	Declaration of custom exceptions that override general ones.
extensions	Custom handlers, gRPC to entity mappers, and vice versa.
filtering	Dedicated folder for dynamic and general filtering support.
mappers	Used to map DTO objects to outgoing objects.
persistence	Organized in sub-packages, the entities and repositories for each database are declared.
service	Custom handlers, gRPC to entity mappers, and vice versa.
util	Utility classes and objects reused across the code.
validation	Custom classes used to validate gRPC requests against constraints.

Table 3.1: List of packages in the project

After the small description of the package structure of the project, and what each module does, the next step will be to look at how data travels through the system, both for REST and gRPC requests. Knowing the flow that information takes inside the system is essential so that every component can make sure it communicates to one another correctly, keeping a maximum separation between layers in a clean way.

The steps involved in this interaction often follow the structure of the Model-View-Controller pattern, which has a well-defined structured approach in handling incoming requests, processing data returned, and returning appropriate responses. Whether through a REST endpoint or through a gRPC server, data usually takes

the same path. It starts with receiving a request at the controller or channel layer down to the final interaction with the persistence layer. Each layer of the system plays a specific role in this process, always trying to keep a separation of concern, in order to create a cleaner code and less complex architecture.

Let's look now at how the system is designed to handle these flows—so, from the path which data takes from its point of entry right through to storage and back again if needed, as visualized in Figure 3.3.

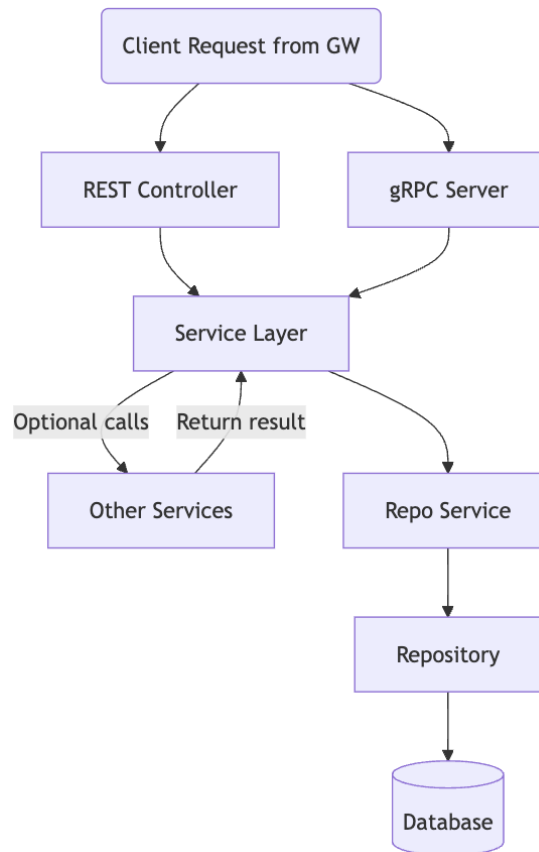


Figure 3.3: Typical flow of data within system

In this architecture, the request flows through different layers, each fulfilling a specific role:

- **gRPC Server:** This serves as an entry point of incoming client requests. Its main function is to delegate the gRPC generated object to the service layer.
- **Service Layer:** First, it performs validation on the request object. The service layer executes the business logic and orchestrates all actions required

in order to satisfy the request. In this layer, data persistence is agnostic and the emphasis is on processing business use cases without caring about the way data is stored and retrieved.

- **Other Services:** In some cases, the service layer may need to communicate with other services (most of the time within the same microservice) to complete its operations.
- **Repo Service:** The repository service is in front of the service layer and of the repository. Its prime function is to separate the service layer from the implementation of the repository to be used. That is, the service layer is not required to manage the data persistence, so that the service layer remains more readily adaptable in the face of changes in the way data are accessed or stored.
- **Repository:** This layer is responsible for database interactions. These with the help of tools such as JPA (Java Persistence API) abstract data operations, specifically enabling the creation, reading, update, and deletion of data records. It guarantees that the communication with the database is always standard and properly contained.
- **Database:** Finally, queries are performed on the PostgreSQL database by Spring JPA.

Organizing in this way, the main objective is to maintain the separation of concerns principles. Each layer will be used to deal with a different one of the application concerns. The Repo Service is actually the bridge needed in order to keep the service layer decoupled from the actual data access implementation.

This modularity is quite useful in the microservice architecture, because the adjustment or change of such modules becomes easy. If the change requires database technology changes or data access logic modifications, then only the Repo Service or Repository Layer needs to change and not the business logic of the Services Layer, for instance. This flexibility makes the system easier to maintain and scale over time.

3.5.5 Dependency management with Maven

As a popular build automation tool in Java-based applications, Maven is often used in managing the code dependencies or libraries in Java-based projects, and there is an important role played by Maven in synchronizing builds and standardizing modules. With an XML-based project object model definition (POM) a notion

of Maven manages development processes in a consistent and predictable manner by providing a definition for both dependency and plugin management, as well as build process steps.

Relative to Gradle, one of the most widely used build tools, Maven features a more simple type of build tool that is based on convention over configuration. While Gradle is powerful, offering flexible scripting with Groovy or Kotlin DSL and faster build speeds, Maven's more declarative nature makes it a preferred choice for projects that value simplicity and a clear standard. With the enterprise-class nature of this project, Maven was the preference as its standard, established workflow provides a consistent, inherently predictable, configuration especially in the context of a multi-person team, where reproducible design is paramount.

The architecture of the project is built on multi-module POM files. A project POM, like the one located in the root directory of the project, acts as the main configuration node by managing several modules, each having its own POM. This enables each module to have its own, separate dependencies and configurations, without losing the controlled influence of the project-level POM. Architecture with this type of modularity also makes dependence management easier and enables a more appropriate separation of concerns. For example, in the project, the `api`, `bom`, `service` modules shall have a clearly defined role with specific/independent dependencies.

Using Maven repositories is an effective way of integrating external utility code in your own project. It starts by downloading libraries from Maven Central, a public repository where thousands of open-source Java libraries are hosted. However, often within a corporate context, companies may not wish to use a public repository for distributing libraries for internal projects., but rather find a way to manage proprietary libraries securely and privately. In such cases, companies can host their own Maven repositories using services like GitHub Packages for example, which ensures that sensitive libraries are kept private and not exposed on the internet.

In this project, internal libraries, such as the gRPC authorization library shown in the subsection 3.5.3, are hosted on a private GitHub repository. This setup allows us to add internal dependencies directly to the POM files, just like we would with any public library, making the integration seamless across multiple projects.

The following code 3.4 is an example of the project-level POM of the system, showing how we define properties, modules, and repository configurations. Notice how each module (`api`, `bom`, and `service`) is included under the section, so their POM files are linked under the same parent. Starting with `<profiles>` tag we define specific repository configurations for hosting dependencies privately on GitHub:

Listing 3.4: Project level POM

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
2   /2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.
4     org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>com.fid.financial.be.miagp.monitoring.engine</groupId>
8   <artifactId>fidit-be-miagp-monitoring-engine</artifactId>
9   <version>1.0.38-SNAPSHOT</version>
10  <packaging>pom</packaging>
11
12  <name>MiaGP Monitoring Engine</name>
13  <description>MiaGP Monitoring Engine</description>
14
15  <modules>
16    <module>api</module>
17    <module>bom</module>
18    <module>service</module>
19  </modules>
20
21  <profiles>
22    <profile>
23      <id>alpian</id>
24      <distributionManagement>
25        <repository>
26          <id>alpian-github</id>
27          <name>GitHub Alpian Apache Maven Packages</name>
28          <url>https://maven.pkg.github.com/alpian-technologies/alpian-packages
29        </url>
30      </repository>
31      <snapshotRepository>
32        <id>alpian-github</id>
33        <name>GitHub Alpian Apache Maven Packages</name>
34        <url>https://maven.pkg.github.com/alpian-technologies/alpian-packages
35      </url>
36    </snapshotRepository>
37    </distributionManagement>
38  </profile>
39 </profiles>
40 </project>
```

This project POM provides a centralized management point for dependencies and plugins. It provides very competent workflows in building, integration, and upload or download of libraries, hence allowing the smooth functioning between different environments. The repository configurations under allow easy and secure management of company-specific packages, thus providing flexibility during development and deployment phases.

3.5.6 Endpoints and protobuf file

For each user story collected from section 3.4, we can identify the endpoints that need to be defined. A user story and an endpoint are not always in a direct one-to-one relationship; however, we can figure out which user stories should create new endpoints and which can be grouped together into one single endpoint. For example, here is a user story: "I want to create a new official monitoring." This automatically calls for a POST endpoint which will create the relevant data. Another example is: a story "I want to get all investment lines" together with, "I want to apply multiple filters to investment lines," suggests that a single endpoint that can receive the filtering options can return the investment lines.

While these are referred to as endpoints and the front-end sends HTTP requests, it is important to remember that there is a middleware layer in between. The gateway will act as the one which takes the HTTP request that it gets, translate it into the corresponding gRPC method, and forward that to the Spring Boot app. In the API module of the project, we have a protobuf file with methods, requests, and response objects defined. The dependencies have to be added for this file to work and be capable of generating objects that can be used in Kotlin code, and these are included in the POM file of the API module.

So, we go in and take a look at the protobuf file to get a detailed understanding of how the API is structured and how communication works, as shown in 3.5. Here, definitions of methods that are working for gRPC, along with the request and response messages, are defined as DTO objects through the service.

Listing 3.5: Protobuf file

```
1 service MiagpMonitoringService {
2
3   rpc GetMonitoring(GetMonitoringRequest) returns (Monitoring) {
4     option (google.api.http) = {
5       get: "/investments-monitoring/monitorings/{monitoring_id}"
6     };
7   }
8
9   message GetMonitoringRequest {
10    string monitoring_id = 1;
11  }
12
13  message Monitoring {
14    string id = 1;
15    string scheduled_date = 2;
16    MonitoringStatus status = 3;
17    int32 notes_count = 4;
18    string latest_data_available_on = 5;
19    MonitoringType type = 6;
```

```
20 |     optional int32 trends_warnings_count = 7;
21 |     ValidationStatus validation_status = 8;
22 |     CluStatus clu_status = 9; /* aka Office Line Change */
23 | }
24 | }
```

First, we declare the service `MiagpMonitoringService`, which includes all gRPC methods related to the monitoring functionality. The declaration of RPC methods is straightforward and begins with the keyword `rpc` followed by the method name, in this case, `GetMonitoring`. Following the method name, the request object, or message, `GetMonitoringRequest` is specified in parentheses. Conventionally, the word "request" is included in the request object and "response" in the return object to make identification clearer. After the request object, the returns keyword is used, followed by the response message, in this case, `Monitoring`.

The `GetMonitoring` method also has an option for HTTP mapping, specified with `(google.api.http)`. This facilitates calling the gRPC method as a REST endpoint, making it callable directly from the Spring boot application. Specifically, this situation maps the `GetMonitoring` method to an HTTP GET on the endpoint `/investments-monitoring/monitorings/{monitoring_id}`, where `{monitoring_id}` is a path parameter corresponding to the ID of the monitoring entity.

One might wonder: why do we need a gateway at all as a middle layer if we can just declare the HTTP endpoints for our gRPC methods? The HTTP endpoint is primarily for debugging and it is not designed to be an interface you would directly work on while in production. For heavy lifting, we still rely on gRPC because it's more efficient with a binary protocol that means better performance and lower latency than traditional REST APIs.

Using Google's `protobuf` library or any other similar library, the same gRPC method is available via both an HTTP endpoint and the gRPC protocol through the use of HTTP annotations. Examples of these annotations are `(google.api.http)`, which define mapping between HTTP endpoint and the gRPC method. The gateway basically listens for HTTP requests and translates them to gRPC calls to service the related service method. This way, the service can keep the core backend logic in one place but at the same time allow flexibility in clients to interact with it.

In order to stay concise and still have visibility for all endpoints, below the table 3.2 lists all gRPC methods and endpoints. The `(prefix)` represents `/investments-monitoring/monitorings`.

Table 3.2: List of other gRPC methods and endpoints.

gRPC Method	HTTP Verb	Endpoint
GetMonitoringTrends	GET	(prefix)/{monitoring_id}/trends
SearchMonitorings	POST	(prefix):search
SearchInvestmentLines	POST	(prefix)/{monitoring_id}/investment-lines:search
CreateMonitoring	POST	/investments-monitoring/monitorings
DeleteMonitoring	DELETE	(prefix)/{monitoring_id}
UpdateInvestmentLinesStatus	PUT	(prefix)/{monitoring_id}/investment-lines/status
ValidateMonitoring	PUT	(prefix):validate
CancelMonitoring	PUT	(prefix):cancel
SearchInvestmentLinesOption	GET	(prefix)/{monitoring_id}/investment-lines/search-options
UpdateMonitoringValidateLinesOption	PUT	(prefix)/{monitoring_id}/investment-lines/validation-options

Chapter 4

User stories flows

Up to this point, we covered the business requirements, followed by high-level exploration of the software architecture, identification of the main entities, discussion about how the microservice communicates to other services, and addressing security features, dependency management, and endpoint declaration. Now we will discuss the operation of bringing together all the components in a coherent manner by walking through the complete implementation of a few user stories.

The first step consists of sequence diagrams, which always provide an overview of how different components of a system interact with each other. This would assist us in visualizing the moving parts and flow of data across systems. Following that, we would go through the code logic addressing the user stories. Furthermore, the screenshots will aid in bridging the gap between the backend logic and end-user experience as they would depict what the user experience with the system's functions looks like, thus making it easier to understand how the user experience is facilitated by the backend implementations.

The user stories covered will include one from each of the identified categories: calendar management, data ingestion, and validation and spools. We will start off with different flows concerning investment line search, loading investment line data, then validation, and finally will finish with emailing the Personal Bankers. These flows were selected based on their importance and degree of impact within the overall system in comparison to CRUD operations, such as creating and deleting monitoring runs. Also, these particular user stories represent cases where I was present for the entire procedure from those conversations first held with the product team to the technical implementation. The level of detail presented in these user stories reflects the depth of my involvement in their implementation and my comprehensive understanding of the system.

4.1 Search investment lines

That gives us a user story within the calendar management domain: the users use a front-end interface to search for investment lines, using various filters to allow users to analyze the current monitoring data and action on individual investment lines as required. The filters allows users to specify what they want to look for, meaning they can focus on specific aspects of investment lines most relevant to their current analysis.

Table 3.2 indicates that the corresponding gRPC method is `SearchInvestmentLines` with a POST request done from the front end, unlike the standard GET request. This design is done so that different filter and sorting details may exist within the body of the request. A POST request thus enables a more defined type of data transmission, with complexity arising in further filter requirements potentially needing more fields and possibly more queries.

Let's take a look at the following JSON example of the request object to see how dynamic filtering with paginated results is supported.

Listing 4.1: Example JSON request

```
1  {
2    "monitoring_id": "1d6a6fc9-5291-4e18-b287-69f83f10fd58",
3    "page": 0,
4    "page_size": 20,
5    "filters": [
6      {
7        "field": "fiscal_code",
8        "operation": "LIKE",
9        "value": "PDMR"
10     },
11     {
12       "field": "profile",
13       "operation": "EQUAL",
14       "value": "0"
15     }
16   ],
17   "sort": [
18     {
19       "field": "var_max",
20       "direction": "DESC"
21     }
22   ]
23   "custom_search": "John"
```

}

The search investment lines endpoint is used as soon as the user enters the investment lines page, initially filtering by monitoring ID to present only relevant data. The user can then add additional filters, which support a set of predefined columns and operations. These additional filters might include attributes such as cycle status, profile, customer type and product, allowing for a granular view of the investment lines. Users can also sort by specific columns to bring the most critical information to the top. Also users have the option to adjust the page size, which means they can decide how much data they want to view at a time.

In order to have a better picture of how the filter may be employed by the users, below in figure 4.1 is a screenshot of the web application with the filters tab open, where there are the supported filters listed.

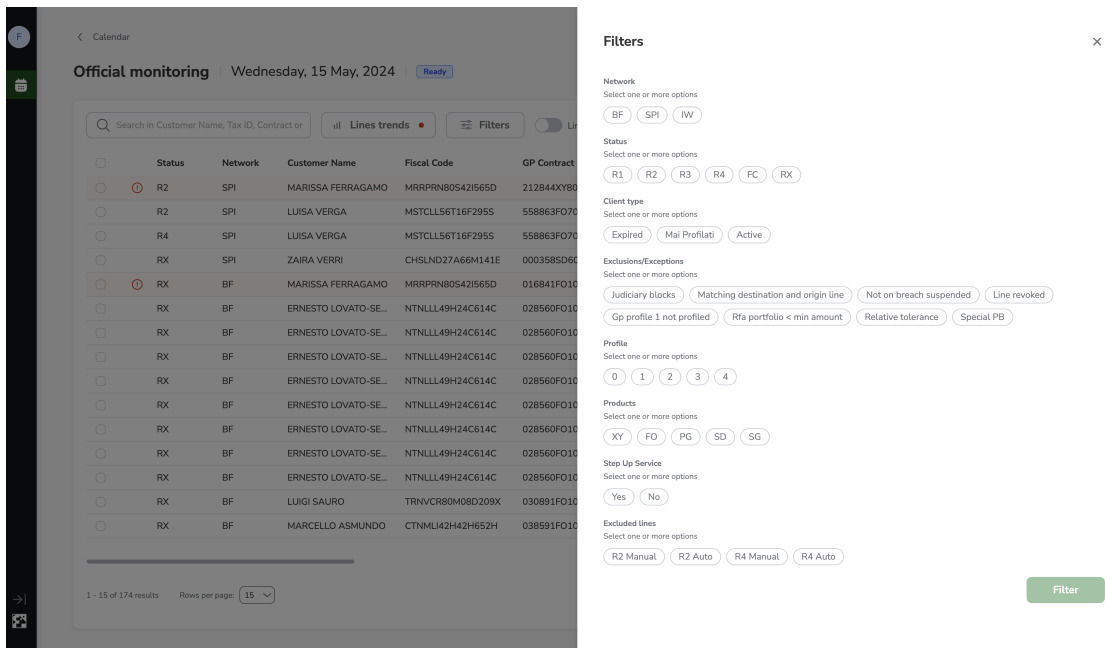


Figure 4.1: Filters in investment line page

The fields and their possible values are provided by the front-end, allowing requests to be constructed based on the user's selections. It is important to note that the customer data displayed is mocked (i.e. fake) and used strictly within the development environment. Another filtering option available to users is the search bar on the left side of the screenshot, which allows searching by customer name,

fiscal code, contract number or mandate. By leveraging this search the value the user types in the search bar is sent in the `custom_search` field.

With this clearer understanding of how the filters function from the end user's perspective, the following sequence diagram 4.2 illustrates the end-to-end flow when a user applies filtering on the investment line page.

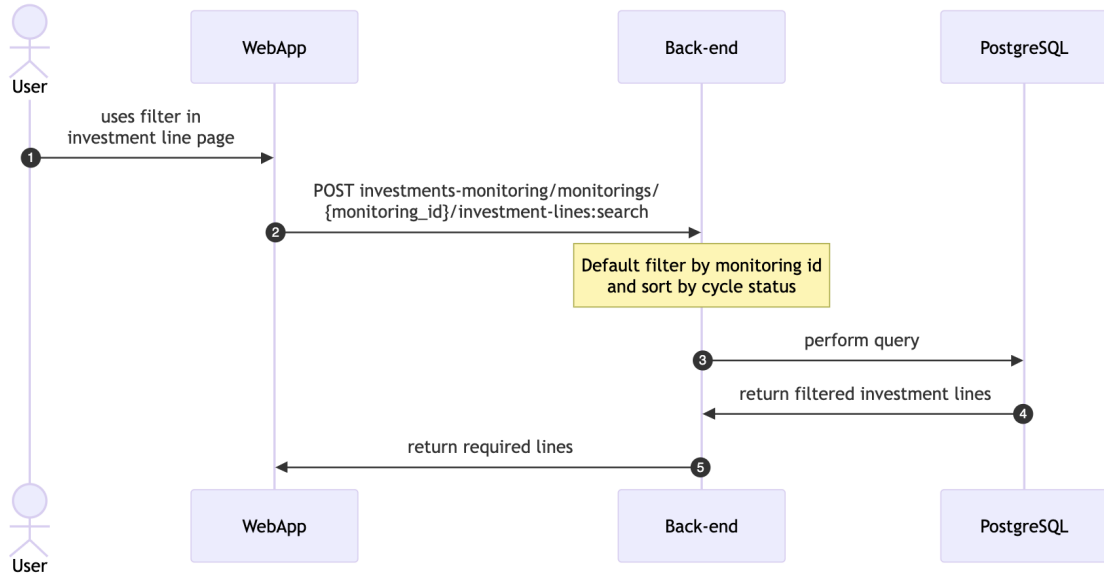


Figure 4.2: Search lines sequence diagram

Although this POST endpoint allows for more dynamic filtering, such as accepting the field, operation, and value from the request, extra care must be taken with request validation to avoid exposing unwanted database operations or attempts to exploit the filtering functionality.

First, when the request arrives at the Spring application, basic validation is performed on several fields. The `monitoring_id` is verified to be a valid UUID, while the page number and page size are checked to ensure they are positive integers. This validation is managed using a custom annotation `@ValidSearchInvestmentLinesRequest` that uses jakarta/javax ContraintValidator.

Next, we proceed to validate the filters with a more thorough check. Specifically, we verify if the `field` provided by the client is a valid property of the `InvestmentLine` entity and confirm that it is declared as a valid filter column. Only explicitly defined columns can be filtered, which adds an important layer of security between the client and the database, this way preventing the client from having full control over query parameters.

Following this, we validate whether the provided operation is also an approved type. Each column is restricted to a specific set of operations, as outlined in table 4.1. Then, we have to parse the value, which is declared as a string in the proto file. This means that for whatever data type is the column we want to filter, the request always arrives as string, therefore these values need to be converted into the appropriate type.

Table 4.1: List of allowed filter operations

Operation	String in operation field
Equals	EQUAL
Not Equals	NOT_EQUAL
In	IN
Greater Than	GT
Greater Than Or Equals	GTE
Less Than	LT
Less Than Or Equals	LTE
Is Null	IS_NULL
Is Not Null	NOT_NULL

The sorting functionality follows a similar strategy as filtering. The direction field is straightforward, users can sort by either ascending (ASC) or descending (DESC) order. The validation for the sort field works in the same manner as the filter field, comparing against a set of pre-approved columns to ensure only allowed columns are sorted.

Now, let's examine the code in 4.2 to see how this logic is implemented in practice. We'll begin with the gRPC Server layer, which will only be shown for this user story, as the logic is essentially the same across similar implementations.

Listing 4.2: gRPC Server class

```

1 @GrpcService
2 class GrpcServer(
3     private val investmentLineService: InvestmentLineService,
4     /* ... other services */
5 ) :
6     MiagpMonitoringServiceGrpc.MiagpMonitoringServiceImplBase() {
7
8     /* ... other fuctions */
9
10    @Authorize(["SO", "SA", "COM", "AR"])
11    override fun searchInvestmentLines(
12        request: SearchInvestmentLinesRequest,
13        responseObserver: StreamObserver<SearchInvestmentLinesResponse>
14    ) {

```

```

15     responseObserver.onNext(investmentLineService.searchInvestmentLines(request
16     ))
16     responseObserver.onCompleted()
17 }
18
19 }

```

A number of elements compounded the given group and plan to be discussed further. First, the `@GrpcService` annotation would come from `gRPC-starter` dependency, responsible for the `gRPC` server's setup or installation. The derived class `MiagpMonitoringServiceGrpc` is an auto-generated interface that contains all the defined methods in the proto file so that there is no ambiguity between the implementation of the service and the definition. We note that the methods in this class are marked with the `override` annotation-which means that these implement methods declared in the base interface.

Here, the `@Authorize` annotation is for role-based authorization, allowing given roles-`SO`, `SA`, `COM`, `AR`-to access this endpoint. Thus, it serves to provide a layer of security, as explained in Section 3.5.3.

The `onNext` method is used to send the response generated by the `investmentLineService.searchInvestmentLines(request)` call back to the client. After the response is successfully sent, `onCompleted` is called to indicate that no more responses will be sent, completing the `gRPC` call lifecycle.

Now, in the listing 4.3, we will move into the Investment Line Service, which the `grpc` server delegates to.

Listing 4.3: Search investment lines service

```

1 @Service
2 @Validated
3 class InvestmentLineService(
4     private val monitoringRepoService: MonitoringRepoService,
5     private val investmentLineRepoService: InvestmentLineRepoService,
6     private val querySpecConverter: LinesQuerySpecConverter,
7 ) {
8
9     @Transactional(readOnly = true)
10    fun searchInvestmentLines(@ValidSearchInvestmentLinesRequest request:
11        SearchInvestmentLinesRequest): SearchInvestmentLinesResponse {
12        val monitoringId = request.monitoringId
13        logger.info { "searching investment lines for monitoring identified by
14            $monitoringId" }
15
16        val monitoring = monitoringRepoService.findByIdOrThrow(UUID.fromString(
17            monitoringId))

```

```

15     val customSearchFilterSpec = querySpecConverter.handleCustomSearch(request.
16     searchValue)
17     val filterSpec: List<FilterSpec> =
18         listOf(FilterSpec(InvestmentLineEntity_.MONITORING, EqualOp, monitoring))
19         .plus(querySpecConverter.toFilterSpecs(request.filtersList))
20         .plus(customSearchFilterSpec?.let { listOf(it) } ?: emptyList())
21
22     val typeSort = when (monitoring.type) {
23         MonitoringType.UNOFFICIAL_WEEKLY -> SortSpec(
24             InvestmentLineEntity_.LATEST_OM_STATUS_ENTITY + "." +
25             CycleStatusEntity_.ORDINAL,
26             Direction.ASC
27         )
28         else -> SortSpec(InvestmentLineEntity_.CYCLE_STATUS_ENTITY + "." +
29             CycleStatusEntity_.ORDINAL, Direction.ASC)
30     }
31
32     val sortSpec = listOf(typeSort, SortSpec(InvestmentLineEntity_.CONTRACT_GP,
33     Direction.ASC))
34     .plus(querySpecConverter.toSortSpecs(request.sortList))
35
36     val pageResult = investmentLineRepoService.search(request.page, request.
37     pageSize, filterSpec, sortSpec)
38
39     logger.info { "found ${pageResult.totalElements} matching the provided
40     filter" }
41     return pageResult.toGrpc()
42 }

```

The `InvestmentLineService` class annotated with `@Service` is responsible for executing the core business functionalities associated with the search of investment lines. The annotation indicates that the class can be said to provide the service functionality within the Spring context, and therefore it is eligible for component scanning, as well as dependency injection.

The `@Validated` annotation, when applied, allows for method-level validation of the parameters. The service checks the request parameter when `searchInvestmentLines` gets invoked by the application, in order to ensure the incoming request towards that method meets the respective constraints.

While behind the scenes all the validation logic, creation of JPA specifications, and repository methods are invoked, the use of this service gives a glimpse into how the service delegates work down to specialized components. Here follows a step-by-step explanation of what is being done in this service method.

1. Validate the Request: The request object is annotated with

`@ValidSearchInvestmentLinesRequest`, which automatically applies validation logic for the incoming request before entering the actual method.

2. **Throw exception if monitoring does not exist:** The service calls `monitoringRepoService.findByIdOrThrow()` to fetch the `monitoring`. This line checks out whether in the repository the monitoring entity with the provided ID exists. If it does not, an exception is thrown, ensuring no further operations are performed on an invalid or nonexistent entity.
3. **Handle custom search filter:** The method `handleCustomSearch()` is used to create a `FilterSpec` object for the `custom_search` value coming from the request. This `FilterSpec` is a custom object that will be used to construct the JPA specification to be used in the query. This is handled independently because this search is an OR in the fields and regular filters are being joined with an AND condition.
4. **Convert query specifications:** The methods `toFilterSpecs()` and `toSortSpecs()` act to convert the filter and sort lists from the request to JPA specifications that may be interpreted by the corresponding repository.
5. **Default filter and sorting logic:** The service sets the default filter by ID and allows for the scoping of the result set to only those ones related to the specific monitoring query. The other part is that there are some default sorting mechanisms one of which will be based on the `cycle status` by ascending order. In fact, for a weekly monitoring process cycle status does not exist, and consequently another field that illustrates the latest official monitoring status is used for ordering.
6. **Performing query and return paged results:** The `investmentLineRepoService.search()` method performs the actual query, using the provided filter and sorting specifications. This query returns a `pageResult` containing the results of the search, along with pagination information (e.g., page size and number). The method then returns these results in the gRPC response format using `pageResult.toGrpc()`.

4.2 Import of investments data

The ingestion of investment line data for each official monitoring, as well as for weekly runs, is essential for the correct functioning of the system. As previously mentioned, the system relies on external database connections to import the necessary data.

To automate this process, Spring schedulers are used. They allow for the execution of recurring tasks based on predefined rules. The `@Scheduled` annotation, allows us to configure methods to run at fixed intervals, by defining cron expressions. These schedulers provide several conveniences of Spring in creating, configuration, managing easily scheduled tasks.

The data ingestion process begins with an external database query, and obtaining all relevant investment data for the client with respect to risk will be initiated with a first-level query against the SQL Server database. This query will look for investment lines on whatever are termed inadequate lines based on client risk profile and portfolio risk level. The actual information retrieved concerning the client's current investments will identify those that no longer satisfy the risk expectations set forth by the client. The specific query is not presented here, given the highly specific nature to that domain and the nonproprietary nature of the database being used.

The staging table is that table where data extracted from the initial query are loaded before an additional process takes place. The purpose of a staging table is essentially allowing to import data into another one place. In this case, after running through a series of transformations and processes, this would be the table representing the `InvestmentLine` entity. To assist in managing application pod memory load and to avoid spikes in resource utilization, processing is done in batches of 500 records at a time.

Next, additional information about each investment, such as the product details and its destination line (a comparable and lower-risk investment option that may be used in cases where forced changes are required), is gathered by querying the other external database, the DB2 instance.

Shedlock is employed as a kind of lock to ensure that only one instance of the ingestion process is executed at a time in the scheduling of this scheduler. Shedlock is used to ensure that duplicate processing is not done across the different running instances with regard to scheduled jobs and that overlapping executions are avoided, thus working toward making sure that conflicts or inconsistencies do not arise in cloud environments where multiple instances of the service may be being run. Shedlock can also auto-reboot itself in the event a lock runs for more than a specified period of time, and auto-reboot the application in the case that a pod crashes during an ingestion process.

Listing 4.4: Search investment lines service

```
1 @Component
2 class IngestionScheduler(val scheduledIngestionRunner: ScheduledIngestionRunner
3     ) {
```

```

4 | @Scheduled(cron = "\${custom.scheduler.ingestion.cron}")
5 | @SchedulerLock(name = "ingestion-trigger", lockAtMostFor = "\${custom.
   |   shedlock.ingestion.lock-at-most-for}")
6 | fun scheduleIngestion() {
7 |     logger.info { "IngestionScheduler: scheduleIngestion" }
8 |     scheduledIngestionRunner.runIngestion()
9 |     logger.info { "IngestionScheduler: scheduleIngestion - done" }
10 | }
11 | }

```

The above piece of code 4.4 declares the scheduler component along with the Shedlock lock, in order to take full benefit from its mechanisms. A custom cron expression and the lock time limit are defined in the `application.yml` and are accessed at runtime. This approach of declaring in a properties file is advantageous, especially when dealing with different environments or when behavior needs to be adjusted based on certain conditions. For instance, the cron expression in production is specified as `0 0 8,15 * * MON-FRI`, meaning the job runs at 8:00 and 15:00 on weekdays. This scheduler class is solely responsible for declaring the scheduling mechanism, which subsequently delegates the actual processing to another method containing the logic.

Before the data is imported for any run, several preparatory steps need to be undertaken. Although periodic triggers are set for multiple days every week, the ingestion process is best if it happens only once per monitoring. What follows is the list of steps taken before reaching this ingestion service:

- *Fetch monitoring:* First, an official monitoring record with the status `SCHEDULED` is searched for in the database. If none are found, a weekly monitoring is created.
- *Check Monitoring date:* The scheduler is triggered at various times throughout the week, but it should only be able to handle ingestion if the monitoring scheduled date's day is the same or before the current day. For example, if the scheduler is triggered on Monday, it sees that the scheduled date is on Wednesday, and it exits early so that ingestion is not started.
- *Check IngestionRun Status:* If the monitoring has an `IngestionRun` status set to `COMPLETED`, the function exits early to avoid redundant ingestion. Otherwise, the execution continues as expected.
- *Data Ready Flag:* Before starting with the import of data from an external database, a check is made to ensure that the data ingestion is ready. This is done through a "data ready" flag confirming that the automatic procedures have already gone through, and the data is currently up-to-date.

To provide a clearer visualization of the information discussed above and how this part of the system operates, Figure 4.3 illustrates the various components involved and their interactions in importing customers' inadequate investment lines.

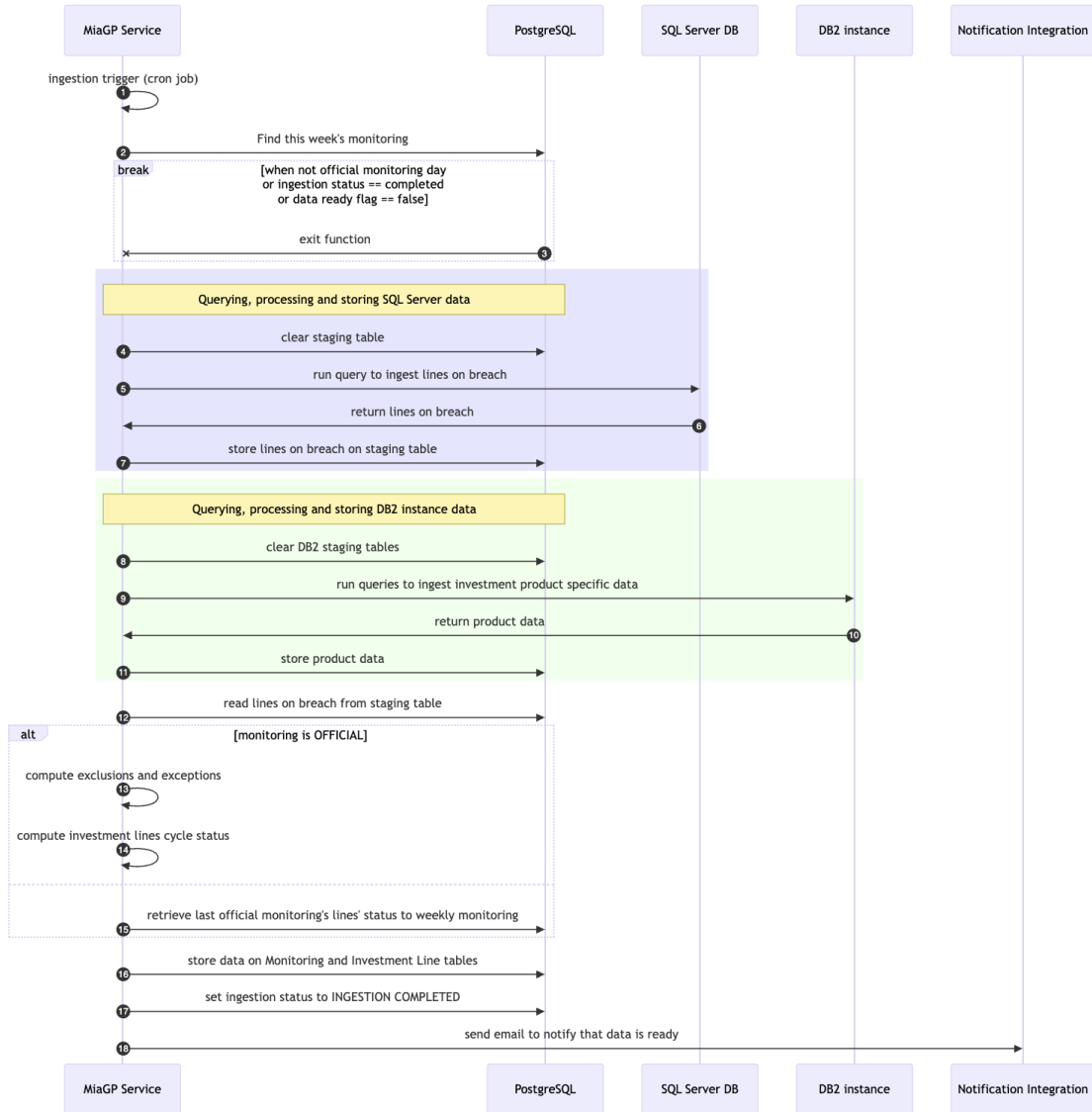


Figure 4.3: Ingestion of data sequence diagram

Once these preliminary steps are complete, the process proceeds to the `IngestionService`. The `triggerIngestion` method, as its name suggests, handles the querying of the external database, gathering data, and processing it to create the appropriate entities in the system's database.

Listing 4.5: Search investment lines service

```
1 @Service
2 class IngestionService(
3     private val localDateNowSupplier: () -> LocalDate,
4     private val onBreachLinesLoader: OnBreachLoaderService,
5     private val napDataLoader: NapDataLoaderService,
6     private val monitoringRepoService: MonitoringRepoService,
7     private val onBreachStagingTransformer: OnBreachStagingTransformer,
8     private val notificationService: NotificationService,
9     private val ingestionDoneEmailMapper: IngestionDoneEmailMapper,
10    private val investmentLineRepoService: InvestmentLineRepoService,
11 ) {
12
13    fun triggerIngestion(currentMonitoring: MonitoringEntity) {
14
15        // 1. Read all the lines which are on breach.
16        onBreachLinesLoader.loadInStaging()
17
18        // 2. Update destination and custom lines (from NAP).
19        val now = localDateNowSupplier()
20        napDataLoader.loadInStaging(now)
21
22        // 3. Find previous monitoring
23        val previousMonitoring = monitoringRepoService.
24        findPreviousOfficialMonitoring(currentMonitoring).also {
25            it?.also { logger.info { "$LOG_PREFIX Previous monitoring is ${it.id} -
26            with scheduled date ${it.scheduledDate}" } }
27            ?: logger.info { "$LOG_PREFIX Previous monitoring cannot be found" }
28        }
29
30        // 4. Read all lines on breach from a staging table and store them as
31        investment lines.
32        onBreachStagingTransformer.toInvestmentLines(currentMonitoring,
33        previousMonitoring)
34
35        // 5. Setting the status of the monitoring to READY.
36        currentMonitoring.status = MonitoringStatus.READY
37        currentMonitoring.latestDataAvailableOn = now
38        monitoringRepoService.save(currentMonitoring)
39
40        // 6. Apply automatic R4 exclusions if applicable
41        markLinesAsR4OfficeChangeAutomaticExcluded(currentMonitoring,
42        previousMonitoring)
43
44        // 7. Notify SO team by e-mail.
45        notifyIngestionCompleted(currentMonitoring)
46
47        logger.info("$LOG_PREFIX Ingestion completed")
48    }
49 }
```

Analyzing 4.5, the `triggerIngestion` function contains comments to provide an overview of each step being executed, allowing us to understand what each call to a service or method accomplishes. Let's investigate these steps in greater detail:

1. *Access external database and import lines:* This step consists of the accessing of the external SQL Server database utilizing a custom-built JPA repository `OnBreachLinesRepository`. A native query is executed through that repository to retrieve investment lines that would be tagged as improper and imported into a staging table. This will allow replacing the old data from the previous monitoring with the new data.
2. *Update DB2 database:* Inputs from an external database, specifically a DB2 instance that retrieves other sorts of information, such as product information and destination lines for a secondary external DB2 instance, will be gathered in the next step. This information will subsequently reside in the PostgreSQL database for use in pending ingestion steps.
3. *Find previous monitoring:* The previous monitoring is retrieved in order to check the recurrence of inadequate investments and track the progression to subsequent cycle statuses.
4. *Create investment line entities:* The data previously loaded into staging tables is now transformed and mapped to the `InvestmentLine` entity. At this stage, specific attributes, such as cycle status and custom validation options, are generated for each investment line.
5. *Adjust monitoring status:* This step involves updating information related to the current monitoring run. The status is updated to reflect that data ingestion is complete and that the monitoring is ready for the next steps. Additionally, the latest ingestion date is set to the current date.
6. *Mark lines as R4 excluded:* This logic pertains to the Office Line Change (CLU) process. If an investment line reaches cycle status R4—which indicates that it has been flagged as inadequate for four consecutive times—automatic corrective action is taken through the spool. However, if a line marked with R4 status during an official monitoring disappears during a subsequent weekly monitoring, it is marked as "excluded from forced change." This ensures that lines no longer considered inadequate are not subject to forced modification.
7. *Notify ingestion completion:* Once all data has been processed, the monitoring is finalized with the investment line data fully loaded. At this point, if it is

an official monitoring, an email is sent to the relevant team to notify them of the successful ingestion. The email is sent using a separate microservice responsible for managing email communications.

After all the steps in the function are completed, the process concludes, and no further code is executed. The imported data becomes immediately available in the web application, with the monitoring status set to `READY`. The investment lines page is populated with the newly ingested data, making it ready for analysis and further actions.

4.3 Validation of a monitoring and spools

As briefly introduced in section 3.3, the validation marks the culmination of the official monitoring process. This action is explicitly triggered by a user after the investment lines have been imported, analyzed, and necessary actions have been performed—such as adjusting the cycle status of specific lines. Validation signifies that no further manual analysis is required, and all automated actions are executed.

The most difficult issue to address with this approach concerned the design of a system, not only capable of processing spools as a generic class of objects with common characteristics like lifecycle management, execution status, and scheduling, and capable of delivering the specialized behavior needed for every single spool class. Since the spools are clearly defined and follow distinct logic, a factory-based approach was chosen. By defining a general interface that encapsulates the core operations of a spool, it became possible to create specialized implementations for the needs of each type. This approach will be explained more in depth as we advance in this section.

In order to understand better what are exactly the spools and what they do, below is a detailed breakdown of the spools and their respective functions:

- *R2 Letter*: Investment lines with a cycle status of R2 are grouped into an txt file and sent to a designated office. This office processes the data to generate physical letters, which are dispatched to customers. The letters inform customers that their investments are deemed inadequate and encourage them to contact their private banker to address and resolve the issue.
- *Email to PB (Private Banker)*: Investment lines that are "in cycle"—specifically those with statuses R1, R2, R3, or R4—are aggregated into an Excel file and sent directly to the responsible private banker. This ensures that the key stakeholder, who has the authority to take corrective actions, is notified and can intervene to resolve the inadequacies.

- *Build Office Line Change file (CLU)*: Theoretically, investment lines that have appeared for four continuous cycles (R4) will trigger a forced change to a more conservative investment option. Unlike other spools, this process does not run after the monitoring finished checks. Instead, it waits until the next weekly monitoring run has been completed. This allows the system sufficient time to exclude any lines that the private banker might have resolved meanwhile. So at this point, the remaining lines not yet resolved are tagged as R4, and a custom file containing the required investment data is created and uploaded to a Google Cloud Storage bucket for further action.

Send email to specific offices within the bank:

Apart from the key spools already discussed, there are five more email spools, designed for the different departments or offices. Each spool is customized to fit distinct departmental requirements, creating targeted filters for what goes into the reports. This ensures that the reports will only contain data that is relevant to the office's mandate.

For instance, one spool contains investment lines related to investors who have passed away, with such investments being blocked through unique block codes to prevent further transactions. The department receives an Excel file containing those lines flagged for action. The flagged lines are then worked upon, and the department can communicate with the heirs of the investors to manage the investments appropriately.

Before going in the code, the image 4.4 gives us a high level understanding of how the validation of a monitoring was be done. It aligns with the factory-based implementation as earlier mentioned, with a "loop" that processes each spool type differently. When outside of this loop we are at an upper level of the code.

The sequence diagram 4.4 illustrates the structured approach used in the monitoring validation process, aligning with the factory-based implementation mentioned earlier. The validation follows a uniform sequence, where the system identifies pending tasks (spools) and processes them in a loop. Each spool type is handled differently, and the overall flow remains consistent, this way we benefit from the factory approach that dynamically selects the appropriate processing logic based on the spool category.

The sequence diagram 4.4 for monitoring validation shows a structured approach to the process of validation in tune with the factored implementation already exposed. The validation, therefore, follows a uniform sequence, whereby the system senses its pending work (spools) and treats those spools in a loop. Each spool type is treated individually; the overall flow, however, remains consistent, therefore, offering a virtue of the factory approach, dynamically selecting the appropriate

processing logic based on the spool category.

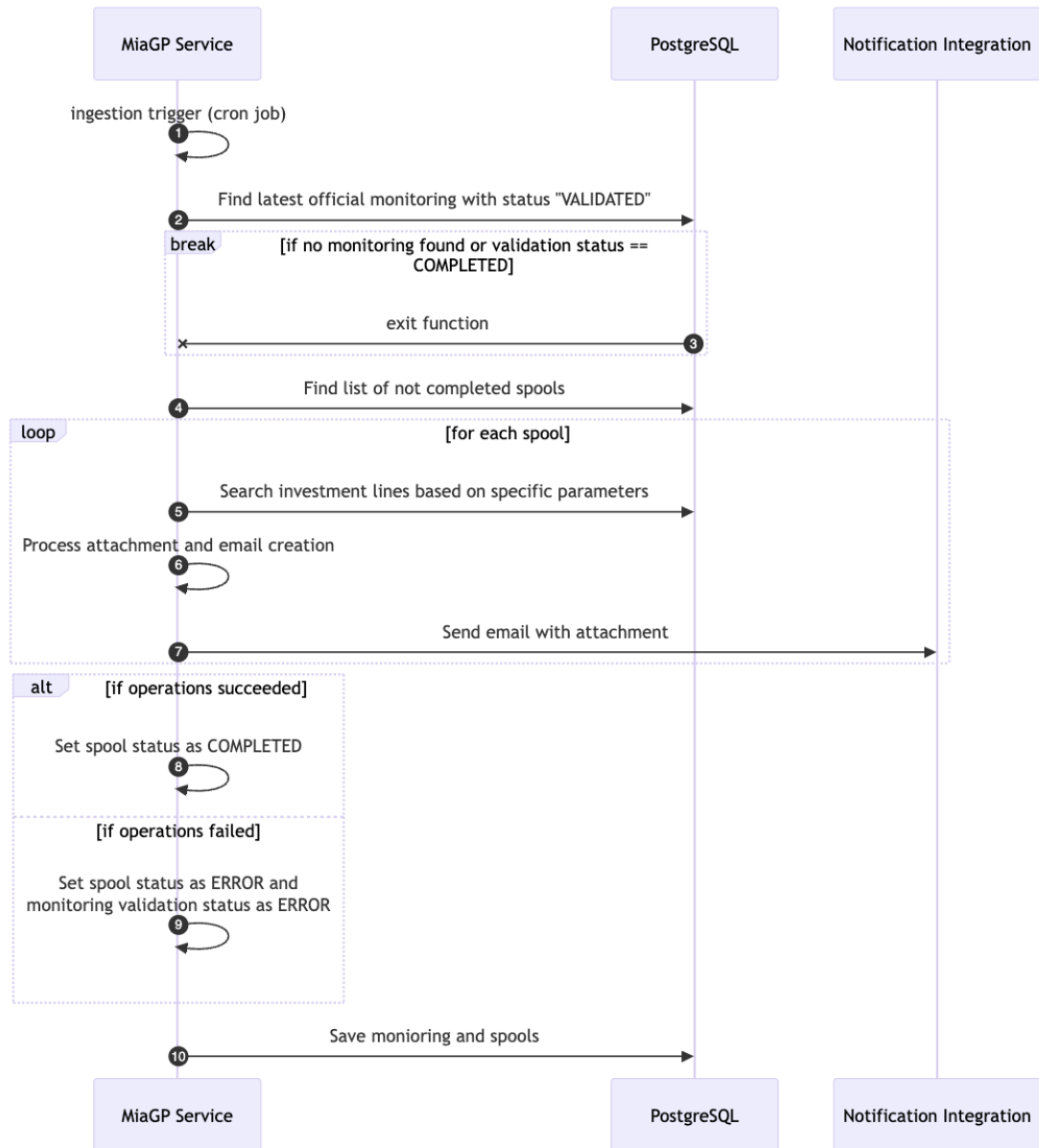


Figure 4.4: Validation of a monitoring sequence diagram

Similar to the ingestion process, validation also uses Spring’s scheduler to ensure a retryable mechanism. A job is scheduled to run every 30 minutes to check for any official monitoring runs with a status of `VALIDATED`. Once identified, the job triggers the processing of these runs. Along this process, the `ValidationStatus`

field inside the monitoring entity keeps track of the state of the validation process. It will allow both reliability and automatic retries for transient failures, assuring good unto the integrity and completeness nature of the validation workflow.

Listing 4.6: Validation run service

```

1 @Service
2 class ScheduledValidationRunner(
3     private val monitoringRepoService: MonitoringRepoService,
4     private val validationRunHandler: ValidationRunHandler
5 ) {
6     fun runValidation() {
7         // get latest monitoring with type OFFICIAL_CONSULTANCY
8         val monitoring = monitoringRepoService.findLatestMonitoringByTypeAndStatus(
9             MonitoringType.OFFICIAL_CONSULTANCY,
10            MonitoringStatus.VALIDATED
11        )
12
13        if (monitoring == null) {
14            logger.info { "No monitoring to run the validation on" }
15            return
16        }
17
18        when (monitoring.validationStatus) {
19
20            // proceed with the monitoring validation
21            ValidationStatus.STARTED,
22            ValidationStatus.ERROR,
23            ValidationStatus.IN_PROGRESS -> {
24                validationRunHandler.handleValidationRun(monitoring)
25            }
26
27            // skip the validation for the following status
28            ValidationStatus.NOT_STARTED,
29            ValidationStatus.COMPLETED -> {
30                return
31            }
32
33            // unexpected scenario
34            null -> logger.warn("Unexpected value for ValidationStatus")
35        }
36    }
37 }
38 }

```

The `runValidation` function presented above in 4.6 is called when the scheduler runs every 30 minutes and is responsible of finding the current week's official monitoring with `VALIDATED` status and check if it needs to run the validation process based on the `ValidationStatus`. As seen in the `when` branch, there are

three statuses that allow the validation to proceed, otherwise the validation process stops. This is followed by another call to another function `handleValidationRun` that either creates the spools in the database if the validation is running for the first time and then continues to the code below that processes the individual spools.

Since all spools are treated the same and just have different implementations, the `SpoolProcessorService` uses a factory approach, that decides which component to use based on the `SpoolType`. The components are all classes that implement the same abstract class, this way they can be treated as the same and call the same method for all the classes that have different implementations. (explain better the factory approach).

Since all spools share the same processing flow but differ in their specific logic, the `SpoolProcessorService` leverages a factory design pattern to decouple the decision-making process for selecting the correct implementation. This approach allows the service to remain agnostic of the underlying spool logic, creating a cleaner and more maintainable code. The factory determines which component to use based on the `SpoolType`, retrieving a specific strategy that implements the abstract class or interface `SpoolProcessingStrategy`. Each concrete class encapsulates the logic required for a particular spool type and implements the common method from the inherited class. As a result, all components can be treated uniformly, enabling the same methods to be called regardless of the spool implementation.

Listing 4.7: Process spools service

```

1 @Service
2 class SpoolProcessorService(
3     private val spoolRepoService: SpoolRepoService,
4     private val monitoringRepoService: MonitoringRepoService,
5     private val spoolProcessingStrategyFactory: SpoolProcessingStrategyFactory,
6 ) {
7
8     @Loggable
9     fun processSpools(monitored: MonitoringEntity) {
10         // retrieve spools for the given monitoring ID that are not in COMPLETED
11         status
12         val spools = monitored.id?.let { spoolRepoService.
13             findByIdAndMonitoringIdAndStatusNotCompleted(it) }
14
15         val completedSpools = spools?.map { spool ->
16             // get the processing strategy for the spool type
17             val strategy = spoolProcessingStrategyFactory.getStrategyForSpoolType(
18                 spool.type)
19
20             strategy.evaluateAndProcessAbort(spool)
21
22             if (spool.status != SpoolStatus.ABORTED) {

```

```
20     // run the processing strategy, handling success and failure cases
21     processSingleSpool(spool, monitoring, strategy)
22 }
23
24     spoolRepoService.save(spool)
25 } ?: emptyList()
26
27 // determine the validation status based on the status of completed spools
28 val validationStatus = if (completedSpools.any { it.status == SpoolStatus.
29 ERROR }) {
30     ValidationStatus.ERROR
31 } else if (completedSpools.any { it.status == SpoolStatus.NOT_STARTED }) {
32     ValidationStatus.IN_PROGRESS
33 } else {
34     ValidationStatus.COMPLETED
35 }
36
37 // Spools may update the monitoring, therefore we retrieve the latest
38 // version from DB
39 val updatedMonitoring = monitoringRepoService.findByIdOrThrow(monitoring.id
40 !!)
41 updatedMonitoring.validationStatus = validationStatus
42 monitoringRepoService.save(updatedMonitoring)
43 }
44
45 private fun processSingleSpool(
46     spool: SpoolEntity,
47     monitoring: MonitoringEntity,
48     strategy: SpoolProcessingStrategy
49 ) {
50     runCatching {
51         strategy.process(spool)
52     }.onSuccess {
53         spool.status = SpoolStatus.COMPLETED
54     }.onFailure { ex ->
55         spool.status = when (ex) {
56             // CLU expected to be run only after one week, so handle custom
57             // exception
58             is PostponeSpoolException -> {
59                 SpoolStatus.NOT_STARTED
60             }
61             else -> {
62                 SpoolStatus.ERROR
63             }
64         }
65         spool.errorLog = ex.message
66     }
67 }
```

```
65 |  
66 | }
```

The process, presented above in the listing 4.7, begins by querying the database for spool entities related to the current monitoring entity. Only spools that are not yet marked as `COMPLETED` are retrieved for further processing. The `SpoolType` determines which strategy will be selected through the factory. The factory works as follows:

1. It exposes a method `getStrategyForSpoolType`, which accepts the spool type as input.
2. Internally, it maps each spool type to its corresponding implementation of `SpoolProcessingStrategy`, which contains the specific processing logic for that type.
3. The components implementing the strategy are Spring beans, making it easier for the factory to inject and manage them automatically.

The status of the monitoring's validation is recalculated again and based on the spools status after the spool processing. If errors occurred on any spool, validation will be set to `ERROR`. If any spool is in a `NOT_STARTED` status, validation will get `IN_PROGRESS` status. Otherwise, this will be marked as `COMPLETED`. The overall assessment of the monitoring thus indicates a successful running of the spool process.

Now, we may take up one of the implementations of the strategy, which is the *Send R2 Letter* spool, as shown in listing 4.8 below. This implementation is much simpler in comparison to other spools that may encounter a far more complicated set of rules such as *Send Email to PB* or *CLU* spools and shares some similarities with other spools that are responsible for sending emails to certain departments. The major actions undertaken by this spool are as follows.

- Query the investment lines: The implementation leverages the specification-based function discussed in the `Search Investment Lines` story. This approach allows the creation of dynamic and flexible queries without having to manually declare each one in the JPA repository. By reusing this functionality, the spool retrieves only the lines marked as R2 for the current monitoring.
- Generate the TXT file: A dedicated service is responsible for generating a structured TXT file that contains the filtered investment lines. This service ensures that the file format adheres to the requirements of the receiving

department, including details such as line identifiers, client information, and investment statuses.

- Send notification: The system then uses the `Notification` microservice to send an email to the relevant parties, attaching the generated file. This ensures that the lines are delivered to the appropriate department for further action, such as notifying customers about their inadequate investments.

Listing 4.8: R2 letter spool strategy

```

1 @Component
2 class ProcessR2LetterStrategy(
3     val investmentLineRepoService: InvestmentLineRepoService,
4     val rowTextGeneratorService: RowTextGeneratorService,
5     val notificationService: NotificationService,
6     val notificationMapper: R2LetterNotificationMapper
7 ) : SpoolProcessingStrategy() {
8
9
10  override fun process(spool: SpoolEntity) {
11      // Retrieve the investment lines to be included in the message.
12      val lines = investmentLineRepoService.search(
13          pageSize = Int.MAX_VALUE,
14          filterSpecs = listOf(
15              FilterSpec(InvestmentLineEntity_.MONITORING, EqualOp, spool.monitoring
16              !!),
17              FilterSpec(InvestmentLineEntity_.CYCLE_STATUS, EqualOp, CycleStatus.R2)
18              ,
19              FilterSpec(InvestmentLineEntity_.VALIDATE_OPTION, NotEqualOp,
20              R2_LETTER_EXCLUDED_AUTO),
21              FilterSpec(InvestmentLineEntity_.VALIDATE_OPTION, NotEqualOp,
22              R2_LETTER_EXCLUDED_MANUAL),
23          ),
24          sortSpecs = listOf(
25              SortSpec(InvestmentLineEntity_.NETWORK, Direction.DESC),
26              SortSpec(InvestmentLineEntity_.PB_CODE, Direction.DESC),
27          )
28      ).toList()
29
30      // Generate the attachment.
31      val attachment = rowTextGeneratorService.generate(lines,
32      R2LetterFieldExtractors.allFields).toByteArray()
33
34      // Send notification.
35      notificationService.sendNotification(notificationMapper.map(spool.
36      monitoring, attachments = arrayOf(attachment)))
37  }

```

33 |
34 | }

In this section, we discussed the stories from among the three main categories: calendar management, data ingestion, and validation processing. By conducting an in-depth examination of each of the user stories beginning from the requirement to implementation in code, we were able to understand the system architecture comprehensively, along with the rationale for some key design decisions made and the use of various design patterns. These were insights into patterns that included aspects of factory design, modular service orchestration, and specification-based query.

Even though we could not cover all the functions, these selected stories offered a representative overview of the core operations within the system. These examples showed how the API endpoints are structured, and also combined validation into the workflow, as well as how reusable components are utilized to avoid redundancy while enhancing application-wide consistency. This overview indirectly breaks down how a similar process is applied otherwise to other functionalities—for example, other API endpoints and validation jobs.

Chapter 5

Testing, Deployment and Monitoring

This chapter discusses practices and processes that ensure the reliability, scalability, and maintainability of the system and starts with discussions of the testing strategies adopted, how they validate the correctness and performance of the system by unit testing, functional testing, and quality assurance (QA). The deployment process describes the next chapter in which the transition of the system from development into production is performed with a high level of minimal downtime and robustness. Towards the end of the chapter, the discussion of monitoring strategies will be applied with a sense of how the health, performance, and compliance of the system are observed constantly to ensure that things stayed in position for long-term stability and operational excellence. Together, these will provide a full view of how the system is verified, deployed, and maintained in a production environment.

5.1 Testing

In any software development lifecycle, testing plays an essential role in ensuring the reliability, performance, and correctness of the application. For MiaGP, a robust testing strategy was employed, covering unit testing, functional testing, and Quality Assurance (QA) testing, each serving specific objectives in validating the system's behavior.

5.1.1 Unit testing

Unit testing emphasizes testing the components in isolation, which are typically the smallest pieces of the application, generally one function or method. The primary concern is thus to test every significant edge case wherein some code outcomes may be potentially incorrect. These tests aim for 100% code coverage, meaning all branches, conditions, and statements are executed during testing.

Mocking of any interaction with dependencies is done to allow the test to focus on the logic contained only within the unit under investigation. Mocking libraries, such as Mockito, are widely used to simulate external calls or to stream in any mock data to keep the unit test as limited in scope as possible. Unit tests are fast, lightweight, and live in the Continuous Integration pipeline.

Here, the example below 5.1 demonstrates a unit test for the `searchInvestmentLines` method from the service class introduced in `searchInvestmentLines`. The test mocks the dependencies, calls the method to be tested, and asserts the expected result.

Listing 5.1: Search investment lines success unit test

```
1 @ExtendWith(MockitoExtension::class)
2 class InvestmentLineServiceTest {
3
4     @Mock lateinit var monitoringRepoService: MonitoringRepoService
5     @Mock lateinit var investmentLineRepoService: InvestmentLineRepoService
6     @Mock lateinit var querySpecConverter: LinesQuerySpecConverter
7
8     @InjectMocks lateinit var underTest: InvestmentLineService
9
10    @Nested
11    inner class SearchInvestmentLines {
12
13        @Test
14        fun `should succeed`() {
15            // given
16            val monitoringId = UUID.randomUUID().toString()
17            val page = 111
18            val pageSize = 222
19            val filter = FilterInfo.newBuilder().setField("product").setOperation("
EQUAL").setValue("XY")
20            val sort = SortingInfo.newBuilder().setField("var_max").setDirection("ASC
")
21            val request =
22                SearchInvestmentLinesRequest.newBuilder()
23                    .setPage(page)
24                    .setPageSize(pageSize)
25                    .setMonitoringId(monitoringId)
```

```
26     .addFilters(filter)
27     .addSort(sort)
28     .build()
29     val foundMonitoring = MonitoringEntity(
30         id = UUID.fromString(monitoredId),
31         type = MonitoringType.UNOFFICIAL_WEEKLY
32     )
33
34     // some hidden variables
35
36     given { monitoredRepoService.findByIdOrThrow(any()) } willReturn {
37         foundMonitoring }
38
39     val pageMock = mock<Page<InvestmentLineEntity>>()
40     given { pageMock.number } willReturn { page }
41
42     // some hidden mocks
43
44     given { investmentLineRepoService.search(any(), any(), any(), any()) }
45     willReturn { pageMock }
46
47     // when
48     val actual = underTest.searchInvestmentLines(request)
49
50     // then
51     then(monitoredRepoService).should().findByIdOrThrow(UUID.fromString(
52         monitoredId))
53
54     // some hidden checks
55
56     assertThat(actual.page).isEqualTo(page)
57     assertThat(actual.pageSize).isEqualTo(pageSize)
58     assertThat(actual.totalCount).isEqualTo(0)
59     assertThat(actual.resultsList).isEmpty()
60 }
61 }
```

There are some aspects of this unit test worth highlighting, since it follows this pattern across all project:

- Dependency Mocking: All dependencies declared in the service are mocked using Mockito's `@Mock` annotation. This ensures the unit test remains focused on the service logic itself, isolating it from external dependencies such as repositories or converters.
- Test Structure: The test follows the standard “given-when-then” pattern

- Given: Mocks return specific values to simulate external dependencies.
- When: The function under test is called.
- Then: Assertions are made to validate the expected results and ensure external methods were called as intended.
- Assertions: the test includes both:
 - Assertions on the return values to ensure the service produces the expected output.
 - Verifications that external methods were called with the correct arguments, ensuring proper interactions between components.

Although unit testing are important for catching edge cases and ensuring consistency and reliability of the code, achieving a high level of coverage may be time-consuming to implement because it requires careful accounting for every conceivable branch and scenario within the code.

Element ^	Class, %	Method, %	Line, %	Branch, %
com	87% (634/725)	38% (1529/3930)	36% (5483/15190)	28% (1625/5622)
fid.financial.be.miaqp.monitoring.engine	88% (629/712)	42% (1510/3593)	39% (5390/13670)	32% (1605/4971)
channel	62% (10/16)	72% (27/37)	84% (77/91)	60% (6/10)
config	97% (36/37)	91% (57/62)	87% (117/134)	25% (1/4)
exceptions	100% (9/9)	100% (9/9)	85% (12/14)	50% (1/2)
extensions	100% (5/5)	100% (26/26)	97% (169/173)	94% (178/189)
filtering	93% (28/30)	97% (67/69)	97% (138/142)	95% (71/74)
grpc.v1	93% (107/114)	25% (676/2674)	23% (2512/10661)	15% (586/3685)
mappers	100% (13/13)	92% (23/25)	94% (90/95)	83% (10/12)
persistence	64% (53/82)	81% (100/123)	93% (364/391)	63% (29/46)
service	90% (332/368)	91% (447/488)	96% (1583/1638)	71% (537/750)
util	100% (17/17)	100% (39/39)	99% (146/147)	91% (148/161)
validation	94% (18/19)	97% (38/39)	99% (181/182)	100% (38/38)
App	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
AppKt	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)

Figure 5.1: Code coverage of the application

The figure 5.1 above showcases the test coverage for MiaGP, demonstrating an 87% coverage for classes, which is a strong indicator of comprehensive testing. However, the method coverage appears lower at 38%, primarily due to the inclusion of auto-generated gRPC code in the analysis by Jacoco, the test coverage tool. These auto-generated files contain minimal tests, decreasing the results. When excluding this folder from the analysis, we observe that out of 1256 methods, 853 are covered by tests, increasing the method coverage to a more accurate and satisfactory 67%. Similarly, for lines and branches, excluding the generated code adjusts the coverage to 65% and 53% respectively. These numbers reflect a more realistic assessment and indicate the test coverage for the project was considerably complete.

5.1.2 Functional Testing

Functional testing evaluates the complete application by testing whole functionalities from end to end. Unlike unit tests, these consider the underlying in a black box and call and validate the outcomes over inputs and expected output. Tests of this type for MiaGP may confirm that any data ingested into an instance of the application set up in a controlled fashion will be correct within the resulting database.

Functional testing usually relies on the usage of test containers, which execute a full Spring application for each test class, together with an initialized database and make sure that everything behaves as it should in a near-to-production environment. Because of the higher computational costs and execution time, there are usually fewer functional tests compared to unit tests. Such tests are most useful when the assurance of proper work of some complex workflow—under realistic conditions, for instance—should be provided. A good example is data ingestion combined with its validation.

For the gRPC functional tests, the gRPC methods are tested as if they were being invoked by an external client. This is achieved by starting the application in a test container and using the generated gRPC stubs to create a client-like interaction. Using these stubs, the test function can build and send requests to the gRPC server and read responses, thus creating an effect of real usage. The method guarantees that the methods under test can operate closely to the production environment.

The dependencies may also be mocked during functional tests to ensure control over the behavior of these components that could impact the test reliability. Let us consider this example, where the clock bean is being mocked to return a fixed value. With this technique, time-sensitive logic methods are insulated from arguments concerning time apart from example-based testing. After invoking the gRPC method, the outcome is guaranteed to be checked against the expectations.

Beans can also be mocked during functional tests so that dependencies can be controlled in case they might affect the reliability of the test. Let us consider the code below 5.2, where the clock bean is being mocked to return a fixed value. This technique is useful for methods that depend on time-sensitive logic because it ensures consistent and repeatable test outcomes. After invoking the gRPC method, the outcome is guaranteed to be checked against the expectations.

Listing 5.2: Search investment lines success unit test

```
1 @SpringBootTest
2 @Testcontainers
3 class MonitoringServiceIT(
```

```
4  @Autowired private val underTest: MonitoringService,
5  @Autowired private val repoService: MonitoringRepoService,
6  ) {
7
8  @Nested
9  inner class CreateMonitoring {
10     @Sql("/db/data/delete_all.sql")
11     @Test
12     fun `should succeed`() {
13         // given
14         val now = Instant.parse("2023-11-23T14:00:00.0000Z")
15         Mockito.`when`(clockCET.instant()).thenReturn(now)
16         Mockito.`when`(clockCET.zone).thenReturn(ZoneId.of("CET"))
17         val requestDate = "2024-12-06"
18
19         val request = CreateMonitoringRequest
20             .newBuilder()
21             .setMonitoringType(MonitoringType.MONITORING_TYPE_OFFICIAL_ADEQUACY)
22             .setScheduledDate(requestDate)
23             .build()
24
25         // when
26         val actual = underTest.createMonitoring(request)
27         assertThrows<ApplicationException> { underTest.createMonitoring(request)
28     }
29
30     // then
31     assertThat(actual).isNotNull
32     assertThat(actual.id).isNotNull
33     assertThat(actual.type).isEqualTo(MonitoringType.
34     MONITORING_TYPE_OFFICIAL_ADEQUACY)
35     assertThat(actual.scheduledDate).isEqualTo(requestDate)
36     assertThat(actual.validationStatus).isEqualTo(ValidationStatus.
37     VALIDATION_STATUS_NOT_STARTED)
38     }
39 }
```

5.1.3 Quality Assurance (QA) test

QA testing is the last stage before deployment to further environments and is usually done by project managers or a dedicated QA team. This is where true end-to-end testing is done, involving the whole application, including the web application and how it talks with the backend. Unlike functional tests, QA testing checks the user experience, ensuring the behavior of the frontend is proper and easy to use, along with the correctness of the backend responses.

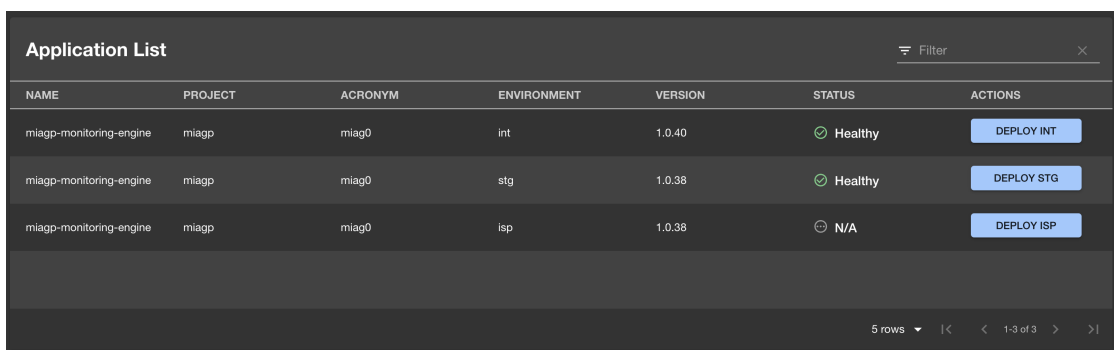
QA testing is highly instrumental in the identification of issues that may not come to the surface in isolated or automated test environments, such as inconsistent user interfaces, performance bottlenecks, or unexpected edge cases in the application's workflow. This phase mostly includes user acceptance testing, whereby the stakeholders act like real users in order to validate the application against business requirements.

5.2 Deployment

The primary purpose of each deployment is to provide seamless transitions from various environments relating to quality, stability, and reliability. The CI/CD pipeline supported by GitHub Actions is at the basis of that transition. Such pipelines have automated features that run builds and tests while publishing Docker images into Artifactory in order to ensure that the whole deployment flow remains steady and functional.

On top of it, deployments between the different environments are further simplified by using open-source framework called Backstage. This tool, set up by the platform team, provides a centralized way to manage services that simplify deployment processes. More importantly, by making it easy to deal with this component, the visibility of the versions and various deployment statuses to upgrade to the next level becomes effortless with only a click away.

Meanwhile, versioning plays a significant role in this workflow. Each build becomes versioned for traceability, so if something goes wrong during or after deployment, rollbacks could be made easy. Version control organizes the history of the deployments and guarantees that only a specific version of the application runs in the various environments.



NAME	PROJECT	ACRONYM	ENVIRONMENT	VERSION	STATUS	ACTIONS
miagp-monitoring-engine	miagp	miag0	int	1.0.40	Healthy	DEPLOY INT
miagp-monitoring-engine	miagp	miag0	stg	1.0.38	Healthy	DEPLOY STG
miagp-monitoring-engine	miagp	miag0	isp	1.0.38	N/A	DEPLOY ISP

Figure 5.2: Backstage tool for deployment

From the image 5.2, we can see the entire process of deployment divided into three stages: development, staging normally referred to as UAT, and production. Each stage has a different purpose that guarantees features are well tested before they go to production.

- *Development*: This is the first stage at which new features and bug fixes are integrated into the application. It serves as a first interaction point with the frontend and gives quick feedback on how changes made in the backend align with the needs of the frontend. Automated unit tests and functional tests will also be run to catch any regressions early.
- *Staging-UAT*: The staging environment is a small version of the production environment. It is mainly used for User Acceptance Testing. Different stakeholders, like project managers or clients, interact with an application to validate that new features cover business requirements. A great deal of functional and QA testing is performed in this environment because the system should behave as expected.
- *Production*: This is the final stage where the application is made to face the real world. No feature makes it to production without being put through a series of unforgiving tests in the stages described above. The focus of this stage is stability, scalability, and availability, as any issues here directly impact the end users.

5.3 Monitoring

Monitoring the application while it is in production is critical in order track the health of the application. Issues can also be resolved much faster with the help of performance insights-based real-time data on system performance, user behavior, and potential issues.

When properly implemented, monitoring can ensure that any failures associated either with slow speeds, lost transactions, or misbehavior in the system are quickly identified and rectified. This would also help in determining usage patterns of a system, which could be useful in making informed decisions regarding scaling or the enhancement of features.

Specific custom logs declared in the code of MiaGP feed into a fully featured monitoring dashboard in Splunk. Such logs capture important events and metrics, like the statuses of data ingestion, validation processes, or spool executions. All this information, after rolling up, gets visualized in the dashboard for easy access to the

activity overview of the application and, thus, enables developers and stakeholders to track its performance or diagnose issues efficiently.

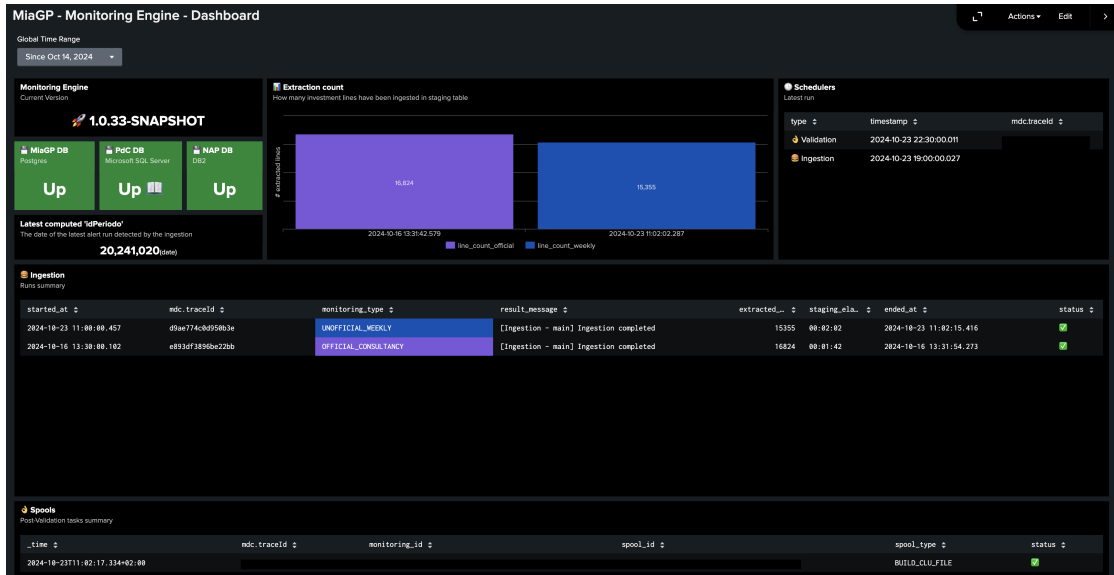


Figure 5.3: Dashboard in Splunk

From the figure 5.3, some critical information about the current status and operation of the system is shown: The dashboard summarizes the status and availability of the databases, with metrics related to the ingestion process, such as the count of investment lines ingested for both the weekly and official monitoring runs. Also present is the execution history of the last schedulers executed, showing their timestamps, and a status display of spools, including their type, time and outcome. The comprehensive visualization thus provides an intuitive interface from which key aspects of the system can be monitored and quickly troubleshoot when required.

Chapter 6

Conclusion

In conclusion, this project shows a great deal of improvement over its predecessor legacy systems and already proved to greatly increase the productivity of the dedicated teams currently using this tool. By understanding the old processes and their limitations, it was possible to design and implement an application that not only kept all previously existing critical functionalities-such as the automated identification of thousands of inadequate investment lines based on client risk profiles-but, even better, to improve them by way of automation, scalability, and user experience.

User feedback confirms the impact of these improvements. Automating email communication and developing the automatic investment line change (CLU) feature has eliminated repetitive manual tasks which used consumable time, improving time savings of approximately 80% in the data ingestion process and 90% during line changes. The wealth management team has also improved in efficiency thanks to advanced filtering, automatic data ingestion, and validation functionalities in the workflows. The system has become so central to all daily activity that team members would have to reschedule monitoring activities altogether in case the system was down, indicative of its essential role.

The adoption of modern technologies such as Kotlin, Spring Boot, microservice architecture, gRPC APIs, and CI/CD pipelines has enabled it to build a strong, scalable, and efficient application. The system has been tested to be reliable and effective in production, processing millions of investment lines of all the clients from outside databases and fetching, on average, 15,000 inadequate investment lines within approximately two minutes. This is achievable through parsing, converting, and enriching raw data into valuable, structured information for users. In comparison with the previous manual process that used to take about two full working days due to data dependencies and manual updates, this is a significant

improvement in terms of efficiency and speed. The first live monitoring cycle was completed following extensive testing in UAT, validating the system's ability to enable critical compliance processes.

According to user feedback, the new system has a number of notable advantages. First, the users appreciated the automatic validation phase, which allows the system to send relevant email notifications to the respective responsible offices in the bank. This feature reduces the burden of manual communication by the user. Second, the users appreciated the automatic submission of CLU files, which allow changes on investment lines to be implemented more efficiently than before. Users rated the ease of only using the new system compared to the old process an average of 9 out of 10, which indicates an improvement in overall usability.

Also, users reported increased confidence in the accuracy of the output data and reliability of the data, especially found it easy to extract subsets of R4 cases and exclude R4 cases for certain approvals. The underlying external databases have not changed, but automating the processes of data extraction and automatically excluding some investment lines with cycle status R4 from the CLU has decreased the possibility of human error and inconsistencies when working manually.

The system has become essential itself in daily work, as users noted that if the system stopped being available they would be disrupted, with one user commenting that if the system was unavailable for a short time, monitoring activities would be postponed until the issue was addressed. If the system was unavailable for a long time, the previous manual processes would need to be resumed, but this would require careful consideration to revisit all tasks that the automated the system could handle.

One possible shortcoming that can be thought of the system, that does not have to do with its design or the application itself, is relying on multiple external components, such as the SQL Server database or the DB2 instance, that are not controlled directly by who implemented it. Instead, these are databases used organization wide that can evolve by themselves, such as column or table names changing, discontinued automatic procedures, etc. Usually this is done with previous communication where it allows systems that rely on them to adapt. However, this means that the system requires possibly some attention in order to remain working with no occurrences or faults.

However, from a user perspective, MiaGP allows them to work with greater agility and confidence. Features such as advanced filtering, sorting, automatic ingestion of data and validation mechanisms simplify complex workflows, while automated communication spools ensure prompt dispatch of messages to clients and stakeholders such as private bankers.

The project and internship, in terms of personal development, are invaluable opportunities for applying the concepts learned in the university under real-world situations. Getting the technical skills in software engineering practices and microservice architecture, and coding ability is something very significant, but even more valuable are the lessons in understanding the corporate environment. The exposure to value delivery, deadlines, and working with diverse teams and stakeholders was very rewarding and excellent preparation for future professional challenges.

Bibliography

- [1] Michael Hudson. *Fund Managers: The Complete Guide*. XYZ Publishing, 2019, pp. 209–216. ISBN: 978-1-119-51534-0 (cit. on pp. 3, 4).
- [2] Paul B. Miller and Andrew S. Gold. «Fiduciary Governance». In: *William Mary Law Review* 57.2 (2015). Accessed: October 22, 2024, pp. 513–585. URL: <https://scholarship.law.wm.edu/wmlr/vol57/iss2/4> (cit. on p. 4).
- [3] Will Kenton. *Markets in Financial Instruments Directive (MiFID) Definition*. Accessed: October 21, 2024. 2023. URL: <https://www.investopedia.com/terms/m/mifid.asp> (cit. on p. 4).
- [4] *Investment services and regulated markets*. Accessed: October 16, 2024. 2024. URL: https://finance.ec.europa.eu/capital-markets-union-and-financial-markets/financial-markets/securities-markets/investment-services-and-regulated-markets_en (cit. on p. 4).
- [5] Martin Fowler and James Lewis. *Microservices: A Definition of This New Architectural Term*. Accessed: November 1, 2024. 2014. URL: <https://martinfowler.com/articles/microservices.html> (cit. on p. 5).
- [6] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O’Reilly Media, 2021. ISBN: 1492034029 (cit. on p. 7).
- [7] Roy T. Fielding. «Architectural Styles and the Design of Network-based Software Architectures». PhD thesis. University of California, Irvine, 2000 (cit. on p. 8).
- [8] Nithin Pawar and Suraj Pitre. *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Microservices and the Cloud*. O’Reilly Media, 2020 (cit. on p. 9).
- [9] Amazon Web Services. *What is cloud computing?* Accessed: November 14, 2024. URL: https://aws.amazon.com/what-is-cloud-computing/?nc2=h_ql_le_int_cc (cit. on p. 10).

- [10] Vincent Ugwueze. «Cloud Native Application Development: Best Practices and Challenges». In: *International Journal of Research Publication and Reviews* 5 (Dec. 2024), pp. 2399–2412. DOI: 10.55248/gengpi.5.1224.3533 (cit. on p. 10).
- [11] Junzo Watada, Arunava Roy, Raturaj Kadikar, Hoang Pham, and Bing Xu. «Emerging Trends, Techniques and Open Issues of Containerization: A Review». In: *IEEE Access* 7 (2019), pp. 152443–152472. DOI: 10.1109/ACCESS.2019.2945930 (cit. on pp. 11, 12).
- [12] Docker Documentation. *What is docker?* Accessed: November 23, 2024. URL: <https://docs.docker.com/get-started/docker-overview/> (cit. on p. 11).
- [13] Adam L. Davis. *Spring Quick Reference Guide: A Pocket Handbook for Spring Framework, Spring Boot, and More*. Accessed: November 29, 2024. Apress, 2021. ISBN: 978-1-4842-6143-9. DOI: 10.1007/978-1-4842-6144-6 (cit. on p. 13).
- [14] Kotlin Documentation. *Kotlin Overview*. Accessed: November 29, 2024. URL: <https://kotlinlang.org/spec/introduction.html> (cit. on p. 14).
- [15] Ken Schwaber and Jeff Sutherland. *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*. Accessed: January 23, 2025. 2020. URL: <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf> (cit. on p. 16).