FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study programme / specialisation: Computer Science	The *spring* semester, *2023* **Open** / Confidential
Author: Matias Lyngnes Ramsland Roger Bærheim	
Supervisor at UiS: Rui Esteves External supervisor(s): Dan Edvard Halvorsen	
Thesis title: Develop a generic Rules Engine to quality control a CV database	
Credits (ECTS): 20	
Keywords: Quality Control CV, Agile, Scrum, XP, Microservices, .NET, Dapr, Relational Database, Document Database, Business Rules	Pages: 136 Stavanger, *May 15th 2023*

0.1 Abstract

This bachelor's thesis presents a software solution to enhance Bouvet's quality control process for employee CVs. By implementing a generic rule engine with extended functionalities, we identified that 90% of the CVs at Bouvet did not meet the company's business standards. Using Scrum with Extreme Programming as our project management system, we developed a scalable and maintainable pilot, employing Microservices, Event-Driven, and Command and Query Responsibility Segregation architecture. Our pilot allows for future modifications using create, read, update and delete operations. The software solution presented in this thesis can be extended to a production-ready state by implementing an Role-based access control and an API-Gateway. When the event bus project by another group at Bouvet is completed, our implementation will be able to notify employees about their CVs' status, further improving the quality control process. Overall, our results demonstrate the our software solution and project management system in enhancing the quality control of employee CVs at Bouvet.

0.2 Acknowledgements

Firstly, we would express our gratitude towards our academic supervisor, Rui Esteves. His expertise and guidance have been significant in navigating the rigours of this thesis project.

We also wish to acknowledge Bouvet's pivotal role in offering us a platform to conduct our research and compile our thesis. Our sincere appreciation extends to our Bouvet supervisor, Dan Edvard Halvorsen. His valuable insights and professional guidance have been of valuable assistance throughout our work.

Link to source code and documentation on GitHub:

https://github.com/MatiasRamsland01/RulesEngine-Thesis

Contents

0.3 List of Acronyms and Abbreviations

Chapter 1

Introduction

This chapter will present the problem that Bouvet tasked us with solving. Additionally, we will provide an overview of the company and delineate the primary objectives of this thesis.

1.1 Problem

Bouvet has experienced growth in its employee base, which has increased the workload necessary to maintain a high-quality standard across its operations. Given that Bouvet is a consulting firm, all employees must adhere to standards and a certain level of quality in their professional resumes. This is particularly useful to Bouvet's business model, as the firm's success relies heavily on its ability to market its services and appear as appealing as possible to prospective clients. Bouvet's client acquisition process is designed to facilitate problem-solving by pairing clients with qualified consultants. As part of this process, Bouvet provides clients with an overview of potential consultants and their relevant experience. Resumes are a tool for showcasing consultant skills and experience to clients. As such, the quality of consultants' resumes is important in demonstrating Bouvet's level of competence and increasing its chances of securing client contracts.

Every business unit manager at Bouvet must manually read and monitor each employee's Curriculum Vitae (CV), including verifying that the documents meet the required standards and are regularly updated to reflect the employee's latest experiences and completed courses. This process is deemed inefficient, and Bouvet's management has expressed a desire to automate this procedure and free up managers' time.

1.2 Goal

Our research question is: "Can a generic rule engine be developed with extended functionalities to effectively control the quality of a CV database?".

Key goals for our thesis include:

- Develop software to review the CVs of Bouvet's employees to identify any inaccuracies or outdated information.

- Integration of the software with an external notification service to notify applicable employees about necessary modifications or updates to their resumes.

- Include and extend the capabilities of a generic Rules Engine capable of executing rules not just on resumes but also on other data types, as required.

- Be able to change the parameters of the quality control parameters, to support future modifications.

- Facilitating easy maintenance, high scalability, and future enhancements to the application.

1.3 About Bouvet

Bouvet, a Norwegian consulting firm, specializes in delivering solutions to assist businesses in realizing their digital objectives. With a presence across 17 offices in Norway and Sweden, Bouvet employs a workforce of approximately 2,000 workers.

Chapter 2

Background and Theory

Chapter 2 will present the background and theory for relevant technologies, design principles, and project management philosophy and methodologies used in our thesis.

2.1 Project Management Philosophy

Various approaches exist for structuring a project, and selecting an appropriate management philosophy is essential to decision-making, problem-solving, and execution throughout the project. Aligning the project structure with the task's objectives is significant in achieving success and delivering optimal solutions. In this section, we will discuss our chosen philosophy and the underlying principles that guide it.

Agile

Agile prioritizes "individuals and interactions" over rigid processes, emphasizing the importance of human collaboration. It also values "working software" above extensive documentation, streamlining the development process. Furthermore, Agile encourages "customer collaboration" rather than getting bogged down in contract negotiations, fostering a more effective partnership. Lastly, it emphasizes "responding to change" over adhering strictly to a plan, promoting adaptability and flexibility. The Agile philosophy recognizes the significance of individuals and interactions but places greater importance on individuals. (Beck et al., 2001)

Daily brief and concise stand-up meetings, including the software development team, are essential to Agile philosophy. The purpose of the daily meetings meeting is to review and address any issues that may have arisen. (Arun, 2023b)

Regular feedback exchange is another hallmark of Agile development. By continuously soliciting feedback from customers or product owners, Agile teams can ensure they are on the right track and make adjustments if needed. This is called a *continuous feedback loop*. Continuous feedback helps mitigate risks and ensure that the final product meets the customer's needs. (CHERVENKOVA, 2022)

A key component in Agile is "user stories." A user story is a simple, concise description. Consisting of a feature or functionality written from the perspective of the end-user or customer. The motivation behind a user story is to capture the customer's requirements for a particular feature or functionality in a way that is easy to understand and can be translated into actionable development tasks. By focusing on the user's needs, the team can ensure they build a valuable solution for their customers. A typical user story is structured as follows: "As a **user/role**, I want to, so that." For example: "As a customer, I want **to be able to view my purchase history** so that **I can track my purchases.**" (Arun, 2023a)

Agile development emphasizes the importance of continuous improvement. Through regular retrospectives and ongoing learning, Agile teams can identify areas for improvement and make changes to their processes and practices. This iterative approach ensures that teams strive to do better and deliver greater value to their customers. (Francino & Denman, 2023)

To be able to understand easier and follow the Agile approach, the creators of Agile have created 12 principles:

1. "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."

2. "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."

3. "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale."

4. "Business people and developers must work together daily throughout the project."

5. "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done."

6. "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."

7. "Working software is the primary measure of progress."

8. "Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely."

9. ”Continuous attention to technical excellence and good design enhances agility.”

10. ”Simplicity–the art of maximizing the amount of work not done–is essential.”

11. ”The best architectures, requirements, and designs emerge from self-organizing teams.”

12. ”At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.” (Beck et al., 2001)

2.2 Project Management Methodology

Numerous methods and frameworks have emerged in an attempt to adopt the Agile approach, with no definitive right or wrong approach. As such, the selection of management tools that best suit a team can vary depending on its characteristics. This section will present the project management tools we have chosen to employ and provide their underlying theoretical foundations.

Scrum

Scrum is an Agile project management approach which provides a framework for teams to organize and navigate their tasks using a collection of core values, principles, and methodologies. Scrum promotes a culture of learning from experiences. It fosters self-organization as teams tackle issues and encourages them to reflect on their successes and failures to enhance their methods and strategies.(Drumond, n.d.)

Scrum has unique team roles. A Scrum team is small and often not larger than 10 people. In larger projects, Scrum teams are often cross-functional. In addition to developers, the team often includes User Experience (UX) designers, testers/Quality Assurance (QA), and DevOps engineers.(Harris, n.d.)

A Scrum team often consists of 3 different roles within the team. This is the product owner, Scrum master, and the development team(West, n.d.).

The product owner within a project is responsible for being the ”owner” of the product. This role implies comprehending business, customer, and market requirements. This must also be followed by prioritizing the Scrum team’s work.(West, n.d.)

The product owner is responsible for constructing and maintaining the product backlog, ensuring consistent updates and refinements. Further, this person must foster a culture of openness,

trust, and collaboration by forming relationships with the business and development teams. Additionally, this individual must offer the development team directions on prioritizing features that align with the product vision and business strategy to maximize value. Lastly, deciding on product releases strategically, with a preference for more frequent deliveries, the product owner must adhere to deliver value to the business quickly.(West, n.d.)

The product owner must be a single individual to ensure unambiguous guidance to the development team. This is done to avoid any potential for conflicting or confusing directives.(Drumond, n.d.)

The Scrum master is the person within the team responsible for the other team members, making them more effective. This can be done with coaching. Scrum masters also hold daily stand-up meetings and schedule the necessary resources for the team to succeed. Lastly, a Scrum master needs to understand the project goal and the work that is being done by the software team. (Drumond, n.d.)

It is a fundamental premise that members of a Scrum development team possess diverse and distinctive skill sets, which makes it essential to undertake cross-training initiatives, thereby preventing any one individual from becoming a bottleneck in the delivery of work. High-performing Scrum teams can self-organize and approach their projects with a collective and cohesive *we* attitude. All team members actively support and assist each other, ensuring the successful completion of each sprint. A sprint is a specific period in which a given work is supposed to be done. The Scrum team is primarily responsible for planning each sprint, and its members use their historical velocity to forecast the amount of work they can feasibly undertake over the iteration. The iteration's length remains fixed, providing valuable feedback to the development team regarding their estimation and delivery processes. This feedback loop enables the team to enhance the accuracy of their forecasts over time. (Drumond, n.d.)

Extreme Programming

Extreme Programming (XP) is an Agile project management methodology that aims to produce higher-quality code and create a better environment for the team. XP promotes its core values, including effective communication, simplicity, ongoing feedback, courage, and respect. These values help foster a collaborative and productive where team members can work together to deliver high-quality products. (AgileAlliance, 2023)

First, software development is collaborative, and effective communication is essential for a team's success. This relies on knowledge being shared among team members. XP emphasizes using the most appropriate form of communication, which it views as face-to-face communi-

cation, often aided by a whiteboard. (AgileAlliance, 2023)

Second, XP prioritizes simplicity, meaning that the simplest solution, which meets the design and functional requirements, is favoured. Doing so minimizes waste, and the team is only tasked with essential activities to maintain the solution's functionality. (AgileAlliance, 2023)

Third, continuous feedback is an important factor in XP. By regularly receiving feedback on their previous work, teams can identify areas that require improvement and refine their practices accordingly. This approach helps foster simple design, and teams gather feedback on the product's design and implementation to make necessary adjustments to move forward. (AgileAlliance, 2023)

Fourth, XP places a significant value on courage. In this context, courage implies taking action based on XP's principles that prioritize avoiding harm to the team. It takes courage to address organizational issues hindering a team's effectiveness, discontinue ineffective practices, and try new approaches. Additionally, accepting and acting on feedback, especially when it is difficult to receive, requires courage. (AgileAlliance, 2023)

Fifth, XP places a value on respect as the fifth core value. This requires each team member to have respect for one another to provide and accept feedback and collaborate to deliver the best possible product. By fostering a respectful work environment, XP ensures team members feel valued and empowered to share their ideas and opinions openly. (AgileAlliance, 2023)

There are several techniques available for practising XP and adhering to its values:

- **Design** (PAL, 2023) can help eliminate complex dependencies within a system, leading to a more manageable and maintainable solution. Therefore, the effective use of appropriate design principles is crucial for successfully developing and maintaining software systems. (PAL, 2023)

- **Test-Driven Development** is a technique that encourages a different approach to writing code and running tests. Unlike the traditional path of writing code first, then writing tests and finally running tests. TDD involves writing a failing test, running the failing test, developing code to make the test pass, and then repeating the process until all tests pass successfully.(PAL, 2023)

- **Continuous Integration (CI)** is a software development practice that continuously tests newly developed or changed code before adding it to the total solution. By incor-

porating CI, teams can ensure that their code integrates with the existing codebase and avoid costly integration issues down the road .PAL, 2023

- **Pair programming** is a technique where two people work together using a single computer. The idea behind this approach is that two people can work more effectively than one. They can quickly identify potential issues and develop more creative solutions by combining their knowledge and skills. Pair programming helps in the improvement of code quality and also reduces error.(PAL, 2023)

- **CI** is the practice of regularly and automatically building, testing, and integrating code changes into a codebase. CI aims to catch and fix errors early in the development process. (PAL, 2023)

2.3 Architectural Design

In this section, we aim to explore the architectural design theory in the context of modern software systems, specifically focusing on three architectures: Microservices, Event-Driven Architecture (EDA), and Command Query Responsibility Segregation (CQRS). We mention these architectures as they are employed in our pilot project.

Microservices

Microservices architecture structures an application into a collection of loosely coupled and independently deployable services. Each service is responsible for specific functionality and communicates with other services through interfaces, such as a Representational State Transfer (REST) Application Programming Interface (API) or messaging protocol. This modular approach enables teams to develop, deploy, and scale components independently without affecting the entire system. (Richardson, n.d.)

Event-driven architecture

EDA is a software architecture paradigm focusing on the production, detection, and reaction to events. Events are messages or signals generated by components in the system, indicating a change in state or the occurrence of a specific action. In EDA, components interact by producing and consuming events asynchronously rather than through direct method calls or data sharing. Which has the advantage of decoupling of components, scalability, adaptability, and resilience. EDA is often used with a microservices architecture to build responsive and distributed systems. (Nassi, 2022)

Command Query Responsibility Segregation architecture pattern

CQRS is an architectural pattern. It separates the operations that modify the state of a system (commands) from the operations that retrieve data from the system (queries). This separation enables the optimization of each side independently, such as employing different data storage and processing mechanisms. Commands are responsible for updating the system's state, while queries focus on retrieving information from the system without causing any side effects. (EventStore, n.d.)

2.4 Design Principles, Strategies and Patterns

Employing design principles and patterns enhances an application's code quality, leading to more efficient and robust systems. When addressed appropriately, preferred solutions for frequently encountered problems can result in better design. The difference between design principles and design patterns is that design principles are general guidelines that guide class structure and relationships, while design patterns solve commonly recurring problems using proven solutions. (Mathew, 2021)

Chain of Responsibility - Pattern

The Chain of Responsibility pattern is a behavioural design pattern that facilitates processing a list or chain of diverse requests. It decouples the sender and receiver of a request based on the request type. The main concept of this pattern is that each receiver, or handler, has a reference to another receiver. If a receiver cannot handle a request, it passes it on to the next receiver in the chain until a suitable handler is found. This pattern is particularly useful in situations where multiple objects may need to handle a request, and the specific handling object is uncertain at design time. (Chauhan, 2022)

Publish and Subscribe - Pattern

The publish-subscribe pattern enables an application to send or receive events to or from interested consumers asynchronously without directly coupling the receivers to the senders. The main objective of pub-sub messaging is to provide information from one component to another as events occur. Using asynchronous messaging, the sender and receiver are decoupled, and the sender avoids being blocked while waiting for a response. (Microsoft, n.d.)

Mediator - Pattern

The mediator pattern is a behavioural design pattern to reduce tangled dependencies between objects. This pattern curtails direct interactions between objects, compelling them to collaborate through a mediator object. The mediator pattern provides communication by introducing

a central control point, simplifying object relationships, and enhancing maintainability within complex systems. (Refactoring.Guru, n.d.-c)

Factory Method - Pattern

The factory method, a virtual constructor, is a creational design pattern that establishes an interface for object creation within a superclass while granting subclasses the ability to modify the type of objects being instantiated. The Factory Method pattern proposes substituting direct object construction calls (employing the `new` operator) with calls to a dedicated factory method. Objects generated by a factory method may be referred to as products. This approach enables a single point of creation logic, making developers change the logic in one space rather than throughout the codebase. (Refactoring.Guru, n.d.-b)

Operation Result - Pattern

The operation result pattern is a design pattern that involves categorizing the results of a function into three buckets: success, failure, and exception. This helps identify the possibility of failures in the contract of a function. For instance, we embed the result of an API call, which could be a success or a failure, into an Operation Result object. By doing this, the caller of the API knows what the outcome of the operation was and can act on it accordingly. For instance, if the API call fails due to a validation error, the caller can present the error message to the user.(Cummings, 2018)

Data Transfer Object - Pattern

The Data Transfer Object (DTO) is a design pattern used in software engineering, predominantly in systems that involve significant data exchange across layers or components. The primary goal of the DTO is to bundle multiple data items into a single object to reduce the number of methods calls, thus improving communication efficiency, particularly in distributed systems where these calls can be expensive. It is a data structure with no business logic but can transport data between processes. It is used in remote interfaces, web services, and when data needs to be transferred over a network or between different parts of an application. (Baeldung, 2022)

Options - Pattern

The options pattern is a software design pattern commonly used in .NET applications to manage configuration settings. The pattern provides a convenient way to map JavaScript Object Notation (JSON) configuration files to configuration objects in the codebase. This enables developers to manage and change application settings without modifying the code.

The options pattern provides a flexible way to manage configuration settings at runtime without restarting the application. (Larkin & Anderson, 2022)

Dependency Injection - Pattern

Dependency Injection (DI) is a design pattern that separates the creation of an object from its dependencies, which a separate component, such as a container, can manage. DI can be implemented using techniques like constructor, property, or method injection, which pass the dependencies as arguments or properties. (Larkin et al., 2023)

API versioning

API versioning is a strategy employed in software development to make changes or enhancements to an API without disrupting existing API users. As APIs evolve, changes can introduce incompatibilities which may break applications that rely on the previous behaviours. To manage this, developers use versioning. (Kleier, 2020)

API versioning can be implemented in several ways, each with its characteristics. One prevalent method is Uniform Resource Identifier (URI) versioning, which embeds the version number directly within the URI, as seen in *https://api.example.com/v1/users*. Another used approach is Query Parameter Versioning. Where it is included in the URI, like in *https://api.example.com/users?version=1*. On the other hand, Header Versioning opts for a cleaner Uniform Resource Locator (URL) by incorporating the version number in the HTTP request header, making it less visible but maintaining a streamlined URI.(Kleier, 2020)

API Contracts

An aspect of creating an API is defining the API contract. The API contract is a formal agreement between the provider of the API and the consumer of the API that outlines what the API will do and how it will behave. This includes the functions that the API will expose, the parameters that are required, and the expected responses. It helps ensure that the API is consistent and reliable for its consumers. By defining a clear API contract, developers can reduce misunderstandings and ensure that their applications will work together as expected. Additionally, a well-defined API contract can make it easier to maintain and update the API over time. (Lane, 2021)

SOLID - Principle

SOLID is a set of principles commonly used in object-oriented software design. The SOLID principles help developers understand the need for software architecture and design patterns.

The main objective of SOLID is to create understandable, readable, and testable code that multiple developers can collaboratively work on. (Erinç, 2020)

The SOLID acronym represents the following:

- **Single Responsibility Principle (SRP)** asserts that a class should focus on one task and have only one reason for the modification. This means that if a class serves as a data container with fields characterizing the entity, it should only change when the data model is altered. The SRP improves the maintainability of the code and makes it easier to understand.(Erinç, 2020)

- **Open-Closed Principle** suggests that classes should be closed for modification and extension. This implies that classes should be able to incorporate new functionalities without altering the existing code. When modifying already-written code, we introduce the risk of creating bugs. Adding new functionality to a class without modification can be achieved using abstract classes or interfaces.(Erinç, 2020)

- **Liskov Substitution Principle** asserts that a derived class object must be capable of substituting base class objects without disrupting the overall integrity of the program. In other words, if class A is a subclass of class B, then an object of class B should be replaceable by an object of class A without changing the desirable properties of the program.(Erinç, 2020)

- **Interface Segregation Principle** advises keeping interfaces separate. A client should not have to implement more than necessary. This is achieved by creating many client-specific interfaces instead of large ones. Using large interfaces could lead to clients implementing more functions than they need.(Erinç, 2020)

- **Dependency Inversion Principle** implies that classes should not depend on concrete classes and functions but rather on abstract classes and interfaces. By adhering to this principle, classes become more open to extension, which aligns with the Open-Closed Principle that encourages software entities to be open for extension but closed for modification. (Erinç, 2020)

2.5 Security Standards and Practices

OWASP Top 10

Security is an important aspect of our pilot. Recognizing the evolving landscape of cybersecurity threats and the need for protection measures, we aim to utilize the Open Web Application Security Project (OWASP) Top 10 to address potential vulnerabilities.

The OWASP Top 10 serves as a guide for developers and web application security specialists. It encapsulates a widely agreed-upon list of the most severe security threats faced by web applications.(OWASP, 2021). Following the OWASP Top 10 guidelines is a strategic decision to deliver a solution that minimizes security risks.

Secret Key Store

The secret key store provides a secure and centralized location for storing sensitive information, such as passwords, API keys, and other secrets for the application's security functionality(Dapr, n.d.-c).

2.6 Technologies and Libraries

There are technological decisions to be made when developing an application. Normally, there are many technologies and libraries to choose from to address specific needs. Although a certain library is functional presently, it might not be so. As such, selecting technologies from sources known for maintaining and updating their libraries is important.

Examples of technology decisions to be made:

- Which framework to use for the backend and/or front end?
- Should the application be containerized? If so, which container service to use?
- How should the security of the application be managed?
- What type of database to use for storage?

Next, we will present the technologies used in the experiment.

Backend

The scope of the project was developing a backend system. A front end was not encompassed in the scope of this thesis, as the primary goal was to establish a backend capable of integration with other services via APIs.

Our backend system was developed using the .NET 7 framework, which supports creating applications across multiple Operating System (OS), employing the C# programming language. We integrated various frameworks, each selected based on its ability to address specific tasks and requirements.

Rules Engine

A Rules Engine is a system that manages rules, typically defined as "if-then" conditional statements. For instance, a rule might be: "If condition A is met, then perform action B.

These rules, which guide an organization's operations, establish what actions are allowed or disallowed. When a dataset is evaluated through a business rules management system using these pre-set conditions, it generates a true or false outcome based on whether the input data aligns with the specified rule. (Nowak, 2023)

Containerization

The process of packaging software code, along with all its essential elements, such as libraries, frameworks, and dependencies in their isolated containers, is called containerization. This approach to running the application ensures that the software can run in any environment, regardless of the infrastructure or the environment's OS. (RedHat, 2021)

GitHub

GitHub is a collaboration platform and version control system that enhances teamwork in software development projects. It offers a collection of features that streamline the development process, such as:

- **Issues** aiding in monitoring bugs, enhancements, or tasks within a repository, promoting communication among team members about the project's progress(GitHub, n.d.-a).

- **Pull Requests** enables developers to propose and discuss changes before merging them into the main codebase, encouraging collaboration and code quality(GitHub, n.d.-d).

- **Milestones** assisting in organizing and tracking issues and pull requests, simplifying

the planning and management of project timelines and progress(GitHub, n.d.-b).

- **Workflows** automating the software development process by enabling CI and Continuous Deployment (CD) pipelines, ensuring that code changes are built and tested before it is implemented in the codebase(GitHub, n.d.-e).

- **GitHub Projects** that resembles a spreadsheet and can integrate with issues and pull requests, facilitating task planning and tracking. Users can establish and customize different views, visualize progress using adjustable charts, and incorporate custom fields to handle team-specific data. (GitHub, n.d.-c)

NuGet

The package manager for the .NET software development framework is called NuGet and is used to distribute and manage reusable code libraries for developers. NuGet provides a centralized package repository called the NuGet Gallery, which all package consumers and authors utilize. (nuget, n.d.)

MongoDB

MongoDB is an open-source Not only Structured Query Language (NoSQL) document database. Unlike relational databases, which use a tabular structure, MongoDB uses a document-oriented format and stores data as key-value pairs. The relationships between data are embedded within the document itself. This creates a dynamic schema, allowing the data structure to be changed without modifying the database. (Gillis, 2023).

Microsoft SQL server

An Structured Query Language (SQL) Server database comprises an assortment of tables which holds a distinct collection of structured data. These tables consist of rows — often called records or tuples — and columns, alternatively known as attributes. Every column within a table is purposefully designed to house a specific kind of information, such as dates, names, monetary values, or numerical data. (Microsoft, 2023c)

Entity Frameworks

Entity Framework (EF) is an open-source Object-Relational Mapping framework for .NET applications. It bridges object-oriented programming and relational databases, enabling developers to work with data using .NET objects, eliminating the need for the data-access code developers must write. (Microsoft, 2022c)

EF abstracts the underlying database system's details, enabling developers to use the same code to interact with different databases, such as SQL Server, MySQL, or Oracle. The developer defines the model (the data structure), and EF creates the database that matches this model. (Microsoft, 2023b)

Furthermore, EF tracks changes made to these objects, translate object manipulations into equivalent SQL queries and executes them against the database when the application chooses to save these changes. This provides developers with a higher level of abstraction when dealing with databases and data manipulations, thus facilitating the development process. (Microsoft, 2021a)

Dapr

Distributed Application Runtime (Dapr) provides APIs, often called "sidecars", facilitating microservice communication. It handles tasks such as managing secrets, encryption, service discovery, integration of message brokers, and monitoring. The functionalities are component blocks and can be used as needed. This component model also decouples technology choices, making combining different tools or programming languages practical. Dapr is applicable wherever an application runs, meaning it can be hosted on a Virtual Machine (VM), Kubernetes, or deployed in the cloud. (Dapr, n.d.-a)

Google Remote Procedure Call

Google Remote Procedure Call (gRPC) represents a contemporary, high-efficiency framework that reimagines the traditional Remote Procedure Call (RPC) protocol. On the application layer, gRPC smooths the communication process between clients and backend services. (Microsoft, 2023d)

Docker

Docker is a development platform that utilizes images and containers to help developers build, run, and share applications. A container is where the program runs in isolation on a host machine. A container is derived from an image, encompassing all the necessary dependencies, configurations, binaries, scripts, and other elements required for the container to operate. Metadata, default commands, and environment variables are also preserved within the image. Utilizing Docker for application development alleviates concerns about a program functioning on one machine but not another, as all components are contained within the image. (Docker, n.d.)

RabbitMQ

RabbitMQ is an open-source message broker. A message broker accepts and forwards messages between services. RabbitMQ supports asynchronous messaging and handles message queuing, different messaging protocols, and delivery acknowledgement. The broker can be deployed with Kubernetes or Docker, and messaging can be used across programming languages. RabbitMQ has various tools and plugins, including monitoring and management.(Rabbitmq, n.d.)

Zipkin

Zipkin is a distributed tracing system that assists in gathering timing data to debug or diagnose service architectures. With Zipkin, we can query on service, operation name, tags, and duration. Else, we can trace the ID in a log file and jump directly to it. Zipkin shows data such as time spent in service and whether an operation failed or not.(Zipkin, n.d.)

Seq

Seq is a platform for detecting and diagnosing problems in applications and microservices. It provides real-time search and analysis.(seq, n.d.)

FluentValidation

FluentValidation is a validation library for .NET applications that enables developers to build and enforce validation rules.(FluentValidation, n.d.)

Hosted Service

Hosted services in C# are a part of the .NET Core framework, providing a mechanism to run a background task or a long-running service within the same process as its hosting application. Essentially, a hosted service is a class that implements the `IHostedService` interface. This interface provides two methods: a start and a stop method, called when the application starts and stops.(Microsoft, 2023a)

Hosted services are typically used for background tasks that do not require user interaction, like data processing, monitoring, timers, or messaging. For instance, hosted service can be used in web applications to offload a time-consuming task from a web request to ensure the request completes quickly while the task continues.(Microsoft, 2023a)

MediatR

The MediatR package is an open-source library that implements the mediator pattern in .NET applications. The mediator pattern helps decouple application components by facilitating communication through a mediator object. MediatR enables developers to invoke request and notification handlers, which can be executed synchronously or asynchronously.(Bogard, n.d.)

xUnit.net

xUnit.net is an open-source unit testing framework for .NET applications. It provides an intuitive way to write, manage, and execute tests. The framework aims to ensure the quality and reliability of software applications. xUnit.net supports a variety of testing styles, including parameterized tests, data-driven tests, and test fixtures. (Frühauff, 2022)

mitmproxy

mitmproxy is an open-source interactive Hypertext Transfer Protocol Secure (HTTPS) proxy. mitmproxy is used to intercept, replay, inspect, and modify Hypertext Transfer Protocol (HTTP) traffic. It is also used for web traffic using Websockets or Secure Sockets Layer (SSL)/Transport Layer Security (TLS)-protected protocols such as HTTPS.(mitmproxy, n.d.-b)

mitmproxy, short for man-in-the-middle proxy, mimics the server to the client and vice versa while sitting in the middle, decoding traffic from both sides. mitmproxy contains a trusted Certificate Authority (CA) implementation that generates interception certificates dynamically. Users manually register mitmproxy as a trusted CA on the device to ensure the client trusts these certificates. (mitmproxy, n.d.-a)

Python Requests package

Python Requests is an HTTP operation library designed for Python(Foundation, n.d.).

mitmproxy2swagger

mitmproxy2swagger is a command-line program written in Python that uses the *flow* file created by mitmproxy to create an OpenAPI 3.0 specification in YAML Ain't Markup Language (YAML) format. Using this together with mitmproxy reverse-engineers an API. (mitmproxy2swagger, 2023)

JQ

JQ is an MIT-licensed tool for working with structured data such as JSON. JQ can use math, RegEx, and restructuring of objects. JQ takes an input and produces an output using a filter. Filters can be combined, piped into another filter for further output processing, or collected the output of a filter into an array. (JQ, n.d.)

Busybox

BusyBox is a small executable incorporating tiny versions of many common UNIX utilities. It offers replacements for most utilities in GNU file utils, shell utils, and similar packages. BusyBox is designed with size optimization and limited resources in mind. (BusyBox, n.d.)

NSwag

NSwag is a Swagger/OpenAPI toolchain used for ASP.NET Core, .NET, and TypeScript. The toolchain is written in C# and can generate openAPI specifications from ASP.NET Web API controllers or create HTTP clients from an OpenAPI specification. The toolchain can be used with a Windows Graphical User Interface (GUI), called NSwagStudio, or from a command line.(Nswag, n.d.)

Prometheus

Prometheus is a time series database used for monitoring and alerting systems. It records metrics with corresponding time stamps and optional key-value pairs known as labels. This time series database is designed to handle large amounts of time-stamped data and provide access to the data for analysis and visualization.(Prometheus, n.d.-a)

Grafana

Grafana illustrates and monitors metrics regardless of their storage location. It offers the ability to generate and disseminate visually appealing dashboards among a team, promoting a culture that values and relies on data. (GrafanaLabs, n.d.)

Chapter 3

Methodology

Bouvet faces challenges as managers manually verify employee CVs for accuracy and quality - a time-consuming and inefficient process. To streamline this procedure and free up managers' time, Bouvet seeks to develop a pilot that applies business rules to CVs, ensuring they meet company standards. By notifying employees of substandard CVs, the pilot aims to reduce errors and enhance Bouvet's appeal to potential clients. Our team is tasked with creating this pilot to quality control the CVs.

3.1 Preparatory Work

In the initial stages of our thesis, we explored various project management philosophies, including Agile and Waterfall.

The Waterfall method operates on a sequential process in which one phase must be completed before transitioning to the next. This approach necessitates that all project requirements be defined at the outset. Any changes to these requirements become challenging once the project progresses.(Whyatt, 2023)

For instance, if Bouvet's requirements evolve during development, it is not a case of adjusting the ongoing work. Instead, any work performed based on the initial requirements may need to be discarded. The project might have to revert to an earlier stage to accommodate new requirements, lengthening delivery times. Given these limitations, we found Waterfall less adaptable to change and less suitable for our project, especially considering that the requirements may shift during development.

Our assessment suggested that Agile aligned with our project's objectives, given its emphasis on swift product delivery, responsiveness to change, and promoting a productive client-team relationship. In our opinion, the rapid product delivery aspect of Agile was key, as it will enable us to react promptly to Bouvet's feedback, with each sprint resulting in a shippable product increment. This allows for continuous delivery by us and immediate feedback from

Bouvet. The iterative process enables us to make changes and improvements at each stage, ensuring the product evolves to meet Bouvet's expectations.

Following Agile, we needed a methodology encompassing this philosophy. However, there are different frameworks within Agile, each with its unique approach to managing projects. We studied the Agile methodologies Lean, Scrum and XP.

Lean concentrates on waste reduction and process optimization to yield an efficient product. This waste-minimizing process is done in each code iteration, increasing development time. (Kooijman, n.d.)

Given our limited schedule, optimising features and removing unnecessary code could hinder our ability to deliver the project on time, as the experiment goal was to develop a pilot application. Consequently, we decided the Agile frameworks Scrum and XP better suited our project's needs.

As a team, we appreciated Scrum's predefined roles and actions, which provided a clear framework for managing our project. This framework helped us prioritize tasks and on delivering value to Bouvet while aligning with our objectives and working towards a shared goal.

Further, we valued Scrum's flexibility and adaptability to change. We believe Scrum methodology enabled us to respond quickly to changing requirements and priorities, which is necessary if the outline of the project is subject to change.

Additionally, Scrum methodology facilitated regular progress reviews and retrospectives, enabling us to reflect on our performance, identify improvement areas, and adjust our approach as needed. We believe this feedback loop promoted continuous improvement and improved outcomes for the project and us.

Using XP, we appreciated its emphasis on coding and testing. XP's continuous feedback and communication principle fostered regular code reviews and pair programming sessions. This enabled us to consistently evaluate our work, identify areas for enhancement, and refine our strategies as needed. In our opinion, this iterative feedback mechanism encouraged continuous learning. It helped us maintain code quality and standards while focusing on our objectives and working towards a shared goal for Bouvet.

Evaluating Scrum and XP, we realized that combining the best practices of both Scrum and XP would be most effective for our project. We believed in a hybrid approach emphasising

teamwork, communication, adaptability, and code quality. We used Scrum's framework to manage the project, including its roles and methods.

We adopted XP's core practices, including pair programming, automated testing, and CI to meet Bouvet's code quality standards.

Combining the practices of Scrum and XP, we had the right tools to effectively manage our project, deliver a solution that meets Bouvet's standards, and maintain a healthy relationship with Bouvet. Utilizing our hybrid approach, we leveraged the strengths of both frameworks while minimizing their weaknesses. This approach set the foundation for our project's path toward integrating the pilot.

3.2 Planning Phase

Our initial meeting with Bouvet in early December 2022 was in the product owner role. We aimed to identify Bouvet's problems and requirements regarding its CV quality process. After conducting an analysis, we identified the following requirements that the proposed solution must meet:

- A generic Rules Engine capable of executing business rules.
- A service that can detect CV changes and store a timeline of the CVs.
- The ability for users to perform Create, Read, Update, and Delete (CRUD) operations on business rules.
- A service that can integrate with another service to notify an employee when their CV violates business rules.
- A library of custom types that other employees within the Bouvet organization can use.
- A generic service that can locate a website's API endpoints and, from these endpoints, create an HTTP client.
- Pilot that facilitates easy maintenance, high scalability, and allows for future enhancements to the application.

We split the requirements into smaller, independently developable features to manage the

project's scope. These features were grouped into epics, which represented larger categories of functionality. Our epics included creating the services *Bouvet.CV.Service*, *Bouvet.RuleEngine.Service*, *Bouvet.Notification.Service* and also the utilities *mitmproxy2client*, *Fields* library, and the *Generic Rules Engine*.

Once we had defined the epics, we began breaking them down into smaller, more manageable features, enabling them to be developed independently. From there, we further refined the features into user stories, each representing a specific user interaction or scenario. Lastly, we broke the user stories into specific tasks, outlining the necessary steps to complete them.

As part of our methodology process, we identified and extracted functional requirements from Bouvet's list and considered non-functional requirements.

Non-functional requirements are the characteristics or qualities a software system must possess to meet performance, reliability, security, and other criteria. The non-functional requirements were defined, documented, and prioritized alongside the functional requirements to ensure that the software system met Bouvet's intended functionalities.

3.3 Features and User Stories

In the following section, we will present some features we identified from the requirements in the preceding section and discuss how they were further refined to meet the project's objectives.

The structure of the presented features will be as follows:

- Features
  - User Stories
    - Tasks
- It should be possible to execute a workflow.
  - As a user, I want a system that can run a workflow to ensure a given standard on the CV.
    - Create workflow class.

        * Create rules class.

        * Crease generic Rules Engine that can execute workflows with the given input.

    - As a user, I want to execute workflow on any input because the Rules Engine might be needed for multiple purposes.

        * Expand Rules Engine to run on any given input.

- It should be able to fetch data from websites dynamically.

    - As a user, I want to get data even if a website changes its structure because website structure may change in the future.

        * Use a proxy service to store the response from an API.

        * Use a script that sends requests to a given website through the proxy.

        * Use a program to extract the open API 3.0 specification from the response.

        * Create a script that extracts all the components in the open API 3.0specification.

        * Use a program that generates an HTTP client from the open API 3.0 specification.

        * Automate all of the given steps in a containerized manner.

- It should be able to persist data.

    - As a user, I want to store my CV in a database because I want historical data on my CV.

        * Create document database (MongoDB).

        * Connect Dapr state store to database.

        * Create functionality to store multiple CV's with the same key in the database(

key=email, value=[]listOfCvs).

* Create functionality to save CVs to the database if CVs do not exist or CVs that exist have been updated.

- As a user, I want to see if and when a CV had changed because I want to know if an employee changes the CV and when it was changed.

  * Create functionality to see if a CV has been changed regarding the previous CV (See delta between two CVs).

- As a user, I want to store the rules in a database for easier access, such that I do not need to write new rules every time.

  * Create a relational database for storing rules(SQL).

  * Create tables for the database (Rules, Workflows, and RuledataField) and the relationships between them.

  * Connect database to SQL server using Dapr secret store.

  * Connect database context to Microsoft EF.

The features, user stories, and tasks mentioned above represent some of the features created throughout the software development process. As the project progressed and we discussed more with Bouvet, we better understood their needs. We then refined the requirements and added more features, user stories, and tasks.

3.4 Scrum

In embracing the Agile project management strategy, our team employed the Scrum methodology to foster a versatile and cooperative development process. Our work was structured into sprints, which are predefined, time-limited periods. We planned two sprints: The first aimed to deliver the Minimal Viable Product (MVP) by March, while the second focused on completing the pilot by May.

At the start of each sprint, we held sprint planning meetings where we reviewed the backlog of requirements, user stories, and tasks. We estimated the effort required for each item

and selected a prioritized set of items to work on during the sprint. These items were then broken down into smaller tasks and distributed among us. Sprints are closely tied to the Agile philosophy Manifesto principle of "Working software over comprehensive documentation."(Beck et al., 2001). By defining predetermined, time-limited periods for development, sprints allowed us to deliver working software in small increments rather than spending time on comprehensive documentation or planning. We prioritized the delivery of working software as the primary measure of progress rather than focusing on completing documentation or other non-essential tasks. Sprints also enabled us to iterate quickly and receive feedback on our work and make necessary adjustments and improvements.

As the two of us worked close, we found that formal daily stand-up meetings were unnecessary. Instead, we had constant communication and collaboration, working together in an agile and flexible manner. However, when working remotely, we found that stand-up meetings held on *Teams* were beneficial for keeping us up-to-date on progress and discussing any issues that arose. Our custom approach to daily stand-ups is consistent with the Agile Manifesto principle of prioritizing "Individuals and interactions over processes and tools."(Beck et al., 2001), and, in our view, this approach promoted effective communication and collaboration among us.

In addition to daily stand-up meetings, we regularly met with Bouvet to discuss requirements and gather feedback. These interactions were important to understand Bouvet's evolving needs and incorporate changes into our sprints as necessary. For example, during the project, we identified the need for a generic system to locate HTTP endpoints on a website. This required us to adapt our approach and incorporate this new feature into our development plan.

After each sprint, we conducted retrospective meetings, reflecting on the sprint's outcomes, and identified successes and areas for improvement. We discussed what went well, what could have been done differently, and how to optimize our processes for the next sprint.

Since our team consisted of only two members, we did not strictly assign specific Scrum roles to each other. For instance, during meetings with the Bouvet, we both took on the responsibilities of the Product Owner, gathering requirements and discussing priorities. Similarly, during our meetings, we acted as both the Scrum Master and the developer. This approach suited our team, allowing us to adapt to our project's evolving needs. Utilizing this approach, we collaborated, shared responsibilities, and made decisions as a team. This led us to leverage our strengths to ensure project progress. While we did not adhere strictly to traditional Scrum roles, we followed the Scrum principles and adapted them to our dynamics.

3.5 Extreme Programming

As part of our Agile software development approach, we embraced the principles of XP to foster collaboration, quality, and efficiency in our development process. One key XP practice extensively used was pair programming. We worked in pairs, with one person ”driving” (writing code) and the other person ”observing” (reviewing code and providing feedback). In our view, This practice promoted collaboration, enhanced code quality, and improved efficiency in our development process.

In addition to pair programming, we employed Test-Driven Development (TDD) for some features. With TDD, we followed a ”red-green-refactor” approach, where we wrote tests for the desired functionality before writing the corresponding code.

”Red-green-refactor” is a mantra often used in TDD, a software development approach that emphasizes writing tests before writing the code. ”red” refers to writing a failing test, ”green” refers to writing code, and only the code, to make the test pass, and ”refactor” refers to improving the code without changing its behaviour (Khine, 2019).

In contrast, for other features, we followed a more traditional approach of writing tests after the code was implemented to compare and contrast the effectiveness of different development methodologies. This made us reflect on what worked best for our project. We will reflect upon our findings in section 5.

Furthermore, to ensure the quality and reliability of our software, we implemented a testing strategy that included automated testing frameworks and tools to execute tests as part of our continuous integration pipeline. As part of our CI/CD process, we leveraged GitHub Actions, a workflow automation tool, to automatically build, test, and deploy our software.

We configured GitHub Actions to run our unit and integration tests on every pull request. Code changes were tested and bugs could be noticed within our code before merging it with the existing codebase. This allowed us to achieve continuous delivery of the product. In our view, this approach facilitated the early detection of issues and improved collaboration.

3.6 Version Control

We decided to use GitHub as our primary version control system. GitHub provides us with a platform to collaborate on our code. One of the key benefits of using GitHub is the ability to create issues. This allowed us to track and manage tasks that needed to be completed. This

feature was useful to us, enabling prioritizing tasks and allocating resources accordingly.

GitHub has built-in support for workflows. These workflows allowed us to automate our build and test processes, ensuring that our code met the required quality standards before it was merged into the main branch. In addition, we implemented a code review process using GitHub pull requests. This process ensured that every code change went through another team member's review before being accepted into the main codebase. Using this process, we believe we maintained a high code quality and identified and addressed potential issues or bugs before the code was deployed.

To facilitate our project methodology, we leveraged the use of the GitHub Projects Kanban board in our software development process. The Kanban board visualized our project's workflow, allowing us to organize and track our tasks, user stories, and features. We organized that every card heading was a feature, and the description contained the user stories with their respective tasks. We created different columns on the Kanban board to reflect the various stages of our development process, such as "New," "Backlog," "Ready," "Priority," "In Progress," "In Review," and "Done." Figure 3.1 shows the Kanban Board we used.

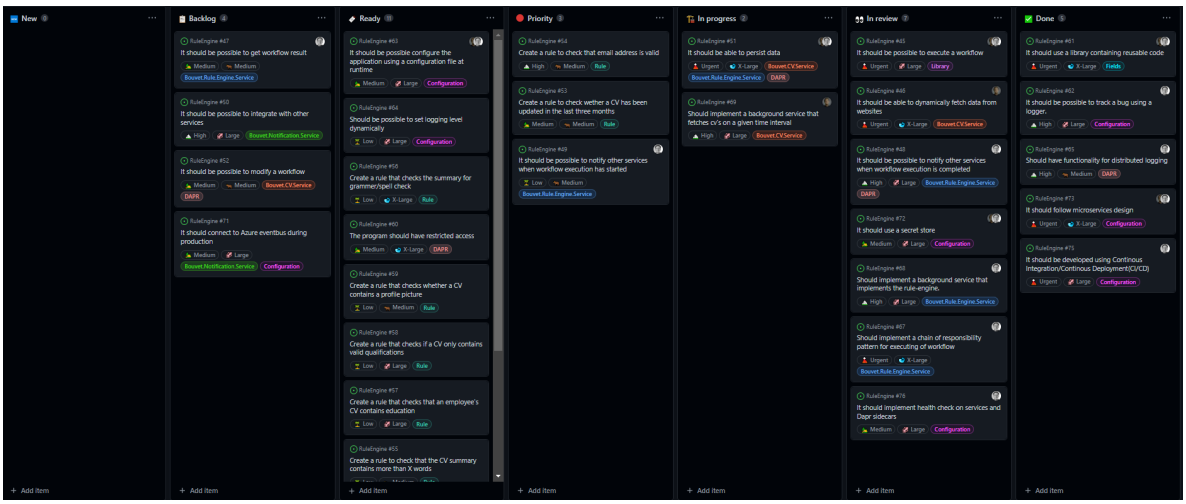

Figure 3.1: GitHub Projects Kanban Board illustrating the project management process.

This made it intuitive to move the cards across columns to reflect the progress of the tasks. For instance, when a task was assigned to a team member, and they started working on it, the card would be moved to the "In Progress" column. Once the task was completed, it would be moved to the "Review" column for peer review and then to the "Completed" column once it passed the review and was done. The Kanban board provided, in our view, a clear and visual overview of the status of each task, allowing us to identify any bottlenecks or delays in the

workflow.

Using the GitHub Projects Kanban board as our project management approach, we utilized custom tables to display the features and their respective service/technology dependencies. These tables visually represented how each feature related to different services or technologies, enabling us to track and prioritize development efforts based on these dependencies. This helped us understand the scope and complexity of each feature in the context of the larger project.

Further, the GitHub Projects Kanban board was also leveraged to manage our sprints and milestones. We used two milestones as our MVP and "Finished Product" sprints and created custom columns on the Kanban board to represent these milestones. We organized our tasks and features based on their priority and timeline and tracked their progress visually.

In addition to the benefits mentioned above, incorporating size estimations for each feature on the GitHub Projects Kanban board provided a valuable reference point for us. We believe it better aligned our expectations and created a more collaborative environment by establishing a shared understanding of each task's relative size and complexity.

Furthermore, by regularly reevaluating and updating the size estimations based on our evolving knowledge and experience, we accurately represented the work required for each task. This dynamic sizing approach helped us adapt to changes and challenges more effectively, which in our view, resulted in a more efficient and agile development process.

Additionally, the GitHub Projects Kanban set priority levels for each feature or task, identifying and picking the most important features to work on. This feature helped us manage our resources efficiently and ensure that the most critical features were addressed first. Hence our Kanban card would include the relevant feature, the user stories with their respective tasks, the milestone, status, size, and priority. Figure 3.2 displays one such card detail.

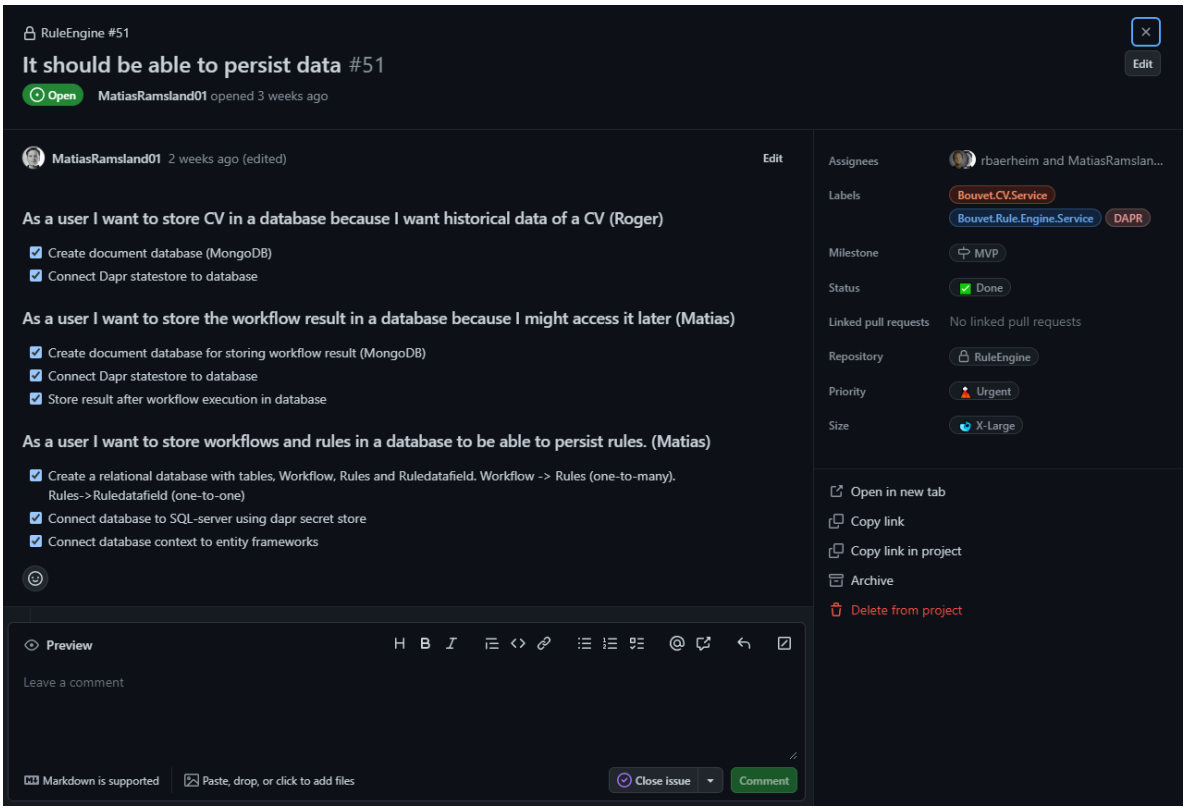

Figure 3.2: Example of a Kanban card in our GitHub Projects, detailing specific task requirements.

In addition to managing the tasks, the Kanban board facilitated communication and collaboration. We can leave comments on the Kanban cards, provide updates, ask questions, or provide feedback. This helped streamline communication as it ensured we were on the same page in regard to the status of the tasks and the project's progress.

Lastly, we decided to use GitHub Projects for our project management approach because of its integration with our existing workflow on GitHub. Since our software development project was hosted on GitHub, we created issues, connected them to pull requests, branches, and commits, and tracked their progress on the Kanban board. This gave us an overview of the entire development process, from planning to implementation to review and deployment, all in one place. The ability to link issues to specific code changes and track their status on the Kanban board, we believe, helped us manage our development workflow and ensure that all tasks were completed on time. This integration with GitHub made GitHub Projects a natural choice for our pilot.

Chapter 4

Experiment

This section will overview our software solution experiment designed to address Bouvet's needs for quality control CVs. We will discuss the architecture, design, various libraries created and utilized in the development process, and microservices implementation for our solution.

4.1 Architecture

We believe it is essential to establish a software architecture before commencing the work. Software architecture encompasses the layout and organization of a software system, comprising its components, interconnections, and core principles. By formulating a software architecture from the outset, we laid the groundwork for our development efforts, enabling us to create a scalable, maintainable, and efficient software solution, as mandated by Bouvet. This section will expound on the planning and rationale behind our software architecture, which was the bedrock for our development work. This foundation made us focus on our design strategy and the development of features and functionalities.

Architectural Approach

As part of our software development project, we evaluated various architectural patterns to determine the most suitable approach. This included monolith architecture, microservice architecture, EDA, layered architecture, and CQRS architecture. We built our application by integrating a microservice architecture, using an EDA for communication between the microservices, and implementing the CQRS pattern within each service.

From our point of view, while layered architecture and monolithic systems possess certain merits, they also exhibit limitations, such as tight couplings, that do not align with our project objectives.

Medium.com writes: A layered architecture typically adheres to a hierarchical organization with layers like presentation, business logic, and data access. This structure can result in tight coupling between components. Although this approach promotes a clear separation of

concerns, it may hinder flexibility and adaptability, as changes in one layer often necessitate alterations in others.(Kaseb, 2022)

In our opinion, monolithic architecture presents a set of challenges. First, monolithic architecture can lead to a tightly coupled system that becomes progressively harder to maintain and scale as it expands. Second, monolithic systems have reduced resilience to failures, as a single point of failure can jeopardize the entire system's functionality.

Consequently, we believe a monolithic architecture does not align with our architectural goal of creating a maintainable system capable of accommodating future enhancements but rather opting for a microservice architecture.

Our system was divided into three major domains, each serving as a distinct service within our microservices architecture and each with a specific function. Each domain was designed to be independent within our chosen microservices architecture. In our opinion, this approach aligned with our goal of creating a distributed microservice application.

1. The first domain, *Bouvet.CV.Service* was designed for CV management. Its main function was to retrieve and manage CV data obtained from CV Partner. A feature of this service was its ability to detect changes in a CV. Upon identifying such changes, the Bouvet.CV.Service would send the updated CV and the associated persisted workflow directly to the rules engine service for processing. The workflow accommodated changes, as Bouvet were able to perform CRUD operations.

2. Following the *Bouvet.CV.Service*, we established the Bouvet.Rule.Engine.Service. This domain's principal function was to host a generic rules engine, a component that executed the supplied workflow based on the input received. While the primary execution command source was the Bouvet.CV.Service, the design of the Bouvet.Rule.Engine.Service was such that it could also accept execution requests from other external services.

3. The final domain in our system was the *Bouvet.Notification.Service*. This service was tasked with analyzing the results provided by Bouvet.Rule.Engine.Service. Upon analysis, if it was determined that an employee's CV had violated any of the rules set in the workflow, the Bouvet.Notification.Service would proceed to notify the respective employee within Bouvet.

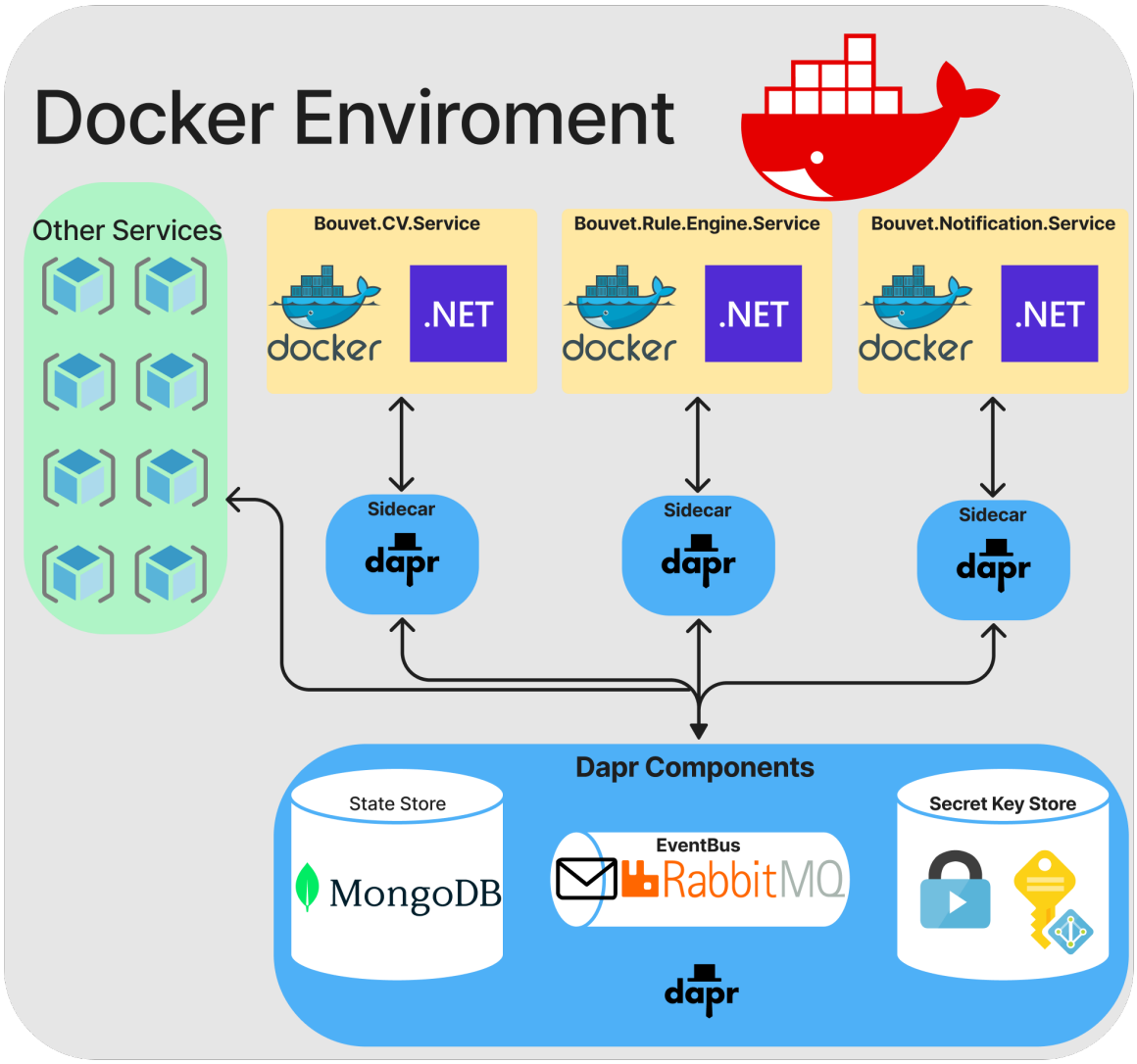

Figure 4.1: Microservices Architecture of our system.

The system architecture of our pilot project encompasses three primary services, as depicted in Figure 4.1. In addition, the diagram displays the communication middleware Dapr, which we will elaborate on in section 4.1.1. Our services also interact with various microservices, such as databases, logging, and metrics. We chose microservice architecture because it supported scalability. We had services with varying load differences, and in our opinion, it was important that our system could handle these differences efficiently. For instance, our rule engine service was designed to be used by other external applications, which could lead to spikes in demand at unpredictable times. By breaking down the system into smaller, independently deployable services, we scale each service independently, enabling us to handle these load differences. This

approach also made it easier to reuse our rule engine service from other external applications, adding value to our pilot.

Furthermore, we believe that microservices modular and decoupled architecture simplifies both system maintenance and enhancement processes. Each service can be developed, updated and deployed autonomously using microservices without affecting other system components. This made us respond to evolving specifications and refine the system to meet changing requirements. This enabled us to deliver a more agile service aligned with our methodology.

In addition to scalability, the flexibility of microservices architecture was also a key consideration. With microservices, we selected the most suitable tools, technologies, and frameworks for each service based on its requirements. We believe this resulted in a more flexible and adaptable system that can evolve with changing technology landscapes.

We implemented an EDA to establish a common communication framework for the microservices. We believed this to be an optimal selection because of its inherent capacity to promote loose coupling and offer scalability. One of the key features that made our services manage events is an event bus, enabling services to process these events when available, such as asynchronous. This setup ensures continuity in service operations, as tasks can proceed without waiting for responses from other services but instead rely on subsequent events. In contrast, not choosing to use an EDA, we would need to align our pilot to a more traditional, synchronous communication model. This approach, in our opinion, would lead to bottlenecks in our system, especially under high demand, as services can get tied up waiting for responses, resulting in an overall decrease in system performance and responsiveness.

In addition, we have also implemented an internal architectural strategy for each service. We decided to adopt CQRS. This differs from traditional request handling, where a single component manages commands and queries. This decision was motivated by several factors. First, CQRS fosters a clear separation of concerns and, in our opinion, promotes a codebase that is easier to maintain and understand. Second, CQRS provided separate strategies for commands and queries, optimizing them for each type of operation.

Recognizing Bouvet's business expectations and anticipation for future growth, we understood the importance of a system capable of managing increased workloads and adapting to changing requirements. We believe integrating microservices, EDA, and CQRS was the optimal solution for our pilot project. While the pilot did not explicitly demand such architecture, our solution is, in our view, prepared to scale horizontally to accommodate increased demand and be open to new functionality, should Bouvet require it. We realised that integrating microservices and

EDA necessitated additional time and effort, and we believed the long-term benefits would surpass the initial investment.

4.1.1 Facilitating Microservice Communication with Dapr

Utilizing the microservice architecture, we acknowledged the necessity for a middleware solution for communication between our microservices while adhering to our architectural goal. After evaluating various technologies, such as Dapr and Istio, we chose Dapr as our communication middleware.

We believe that selecting Dapr as the communication middleware for our experiment was the optimal decision due to its increased flexibility and compatibility and its ability to facilitate easy communication between our services.

Consequently, we concluded that Istio's service mesh solution, with its features, added complexity and overhead that did not correspond with our project's needs (Istio, n.d.). Instead, Dapr, a more simplified and accessible alternative, was chosen. This decision followed our XP methodology, which emphasizes employing the simplest approach that fulfils a client's requirements rather than opting for complex paths that can hinder progress and efficiency.

The Dapr sidecars handled cross-cutting concerns like service discovery, routing, and communication protocols. This abstraction provided a consistent and unified programming model for microservices, regardless of the specific communication mechanism or technology involved. This approach introduced, in our opinion, a reduction in overhead and complexity, a benefit when creating a distributed microservice application.

Reflecting on our choice of Dapr, a key factor was its support for gRPC. According to the Dapr documentation, gRPC outperforms regular HTTP regarding communication speed, approximately seven times faster (Fernando, n.d.). The prospect of enhanced performance was a factor in our decision to utilize Dapr.

The selection of Dapr was also influenced by its pluggable component model. This feature allowed selection and integration of components most relevant to the project's needs. Furthermore, the ability to replace or modify components through configuration changes without the need for code alterations made Dapr, in our opinion, a flexible and adaptable microservices infrastructure.

Figure 4.2 showcases our microservices architecture, with Dapr as our communication middleware. It leverages Dapr's sidecar pattern for communication and employs RabbitMQ as

the message broker to support our EDA. In the illustration, each microservice is represented by a rectangular box and a hexagonal box representing the corresponding Dapr sidecar.

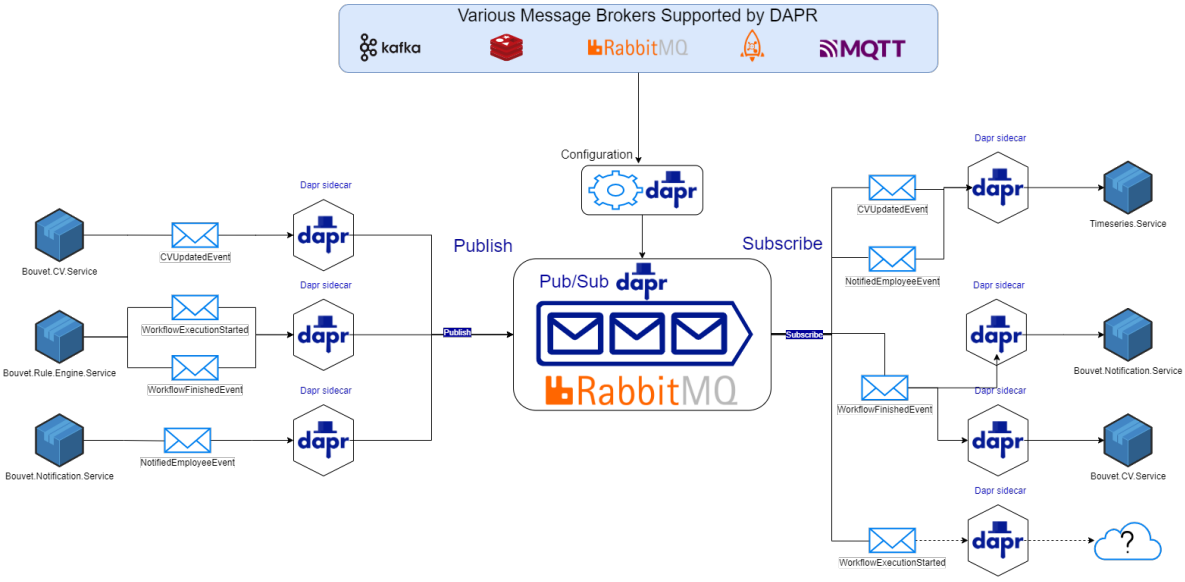

Figure 4.2: Dapr pub/sub mechanism in our microservices architecture, facilitating service-to-service communication.

Also presented in Figure 4.2, our services publish events, as presented by arrows pointing from the microservices to the sidecar, which in turn points to the RabbitMQ message broker. The events were notifications, updates, and any other message that must be communicated across microservices. There is only one instance of the event being published. Still, multiple subscribers, represented by arrows pointing from the RabbitMQ message broker to the microservices, can choose to subscribe to it.

This approach enables microservices to subscribe to events specifically suited to their requirements and interests, which, as we see it, promotes adaptability and minimal interdependence. This decoupled strategy enables microservices to evolve autonomously, as they can subscribe or unsubscribe from events without impacting other microservices within the system.

4.1.2 Dapr with Docker Compose

Dapr was implemented alongside Docker Compose for our microservices-based application. In our opinion, this decision proved advantageous as we defined and configured our application's services and dependencies as containers.

Adopting Docker with Dapr as a containerized solution was beneficial because it deployed and managed our microservices-based applications. Docker achieved this by providing an environment across development, testing, and production stages, which in our opinion, shortened development time and simplified the management of our application's distributed nature.

Implementation

Each service in our application featured its own *Dockerfile*, configured to expose port 80 and execute the service within a Docker container. These Dockerfiles were devised to package our services and their dependencies into portable containers.

The Dockerfiles start as the basic ASP.NET runtime image. It then copies the project files and dependencies into the container and runs a `dotnet restore` command to restore the NuGet packages for the solution. Subsequently, the entire source code is copied into the container. The Dockerfile builds and publishes the project before establishing the container's entry point, which serves as the microservice's main point of entry.

Additionally, we created a Docker Compose file that brought all the Dockerfiles together, along with Dapr sidecar containers and the other necessary containers. The services encompass RabbitMQ, Redis, SEQ, Zipkin, SQL-Server, MongoDB, Dapr placement, Mongo Express, Prometheus, Graphana, WebStatus, Bouvet-CV-Service, Bouvet-Rule-Engine-Service, Bouvet-Notification-Service, and their corresponding Dapr sidecars. The total of 19 running containers using Docker Desktop is portrayed in figure 4.3.

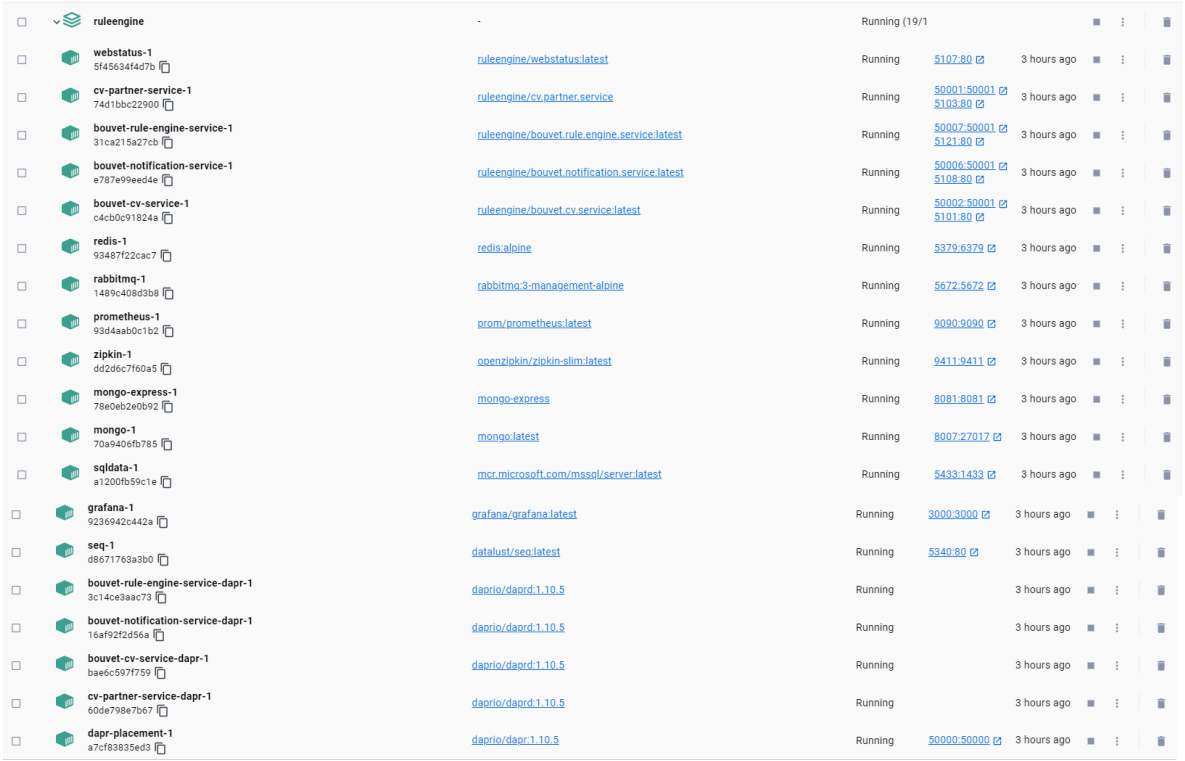

Figure 4.3: Docker Desktop interface running our system's Docker containers.

Each service is defined with its image, either pulled from Docker Hub or built using a Dockerfile in the specified context. Dependencies between services are established using the *depends_on* command, ensuring that dependent services start before those that rely on them. Volumes are defined for data persistence across container restarts. The Docker Compose file offers a method for defining the entire application stack, facilitating the process of starting all the services together for local development or testing purposes using the command `docker compose up` in the command line. During the experiment, this design made it, in our opinion, straightforward to add another service when necessary.

To use configure Dapr, we structured Dapr configuration files into two folders: one for components and the other for configuration. The components folder facilitated the addition and configuration of Dapr components, such as state stores, secret stores, and message brokers. The configuration folder consisted of Dapr's configuration files, such as placement and tracing settings, enabling customization of Dapr's behaviour per our application's requirements.

We faced challenges while implementing Docker Compose in conjunction with Dapr. One challenge pertained to the management of ports. Dapr services often necessitate the exposure

of ports to facilitate inter-service communication. Coordinating and managing port mappings for multiple services can be complex and laborious, particularly when multiple containers operate concurrently on the same host machine.

We addressed this issue by creating an overview of the container ports, as illustrated in Figure 4.4. This was necessary, considering the potential for up to 38 distinct port numbers in total. By documenting these ports, we believe the application's network communication was better organized and easily managed.

Service	Public Port	Private Port	Private GRPC port	Public GRPC port
bouvet.cv.service	5101	80	50002	50001
bouvet.notification.service	5108	80	50006	50001
bouvet.rule.service	5121	80	50007	50001
Webstatus	5107	80		
RabbitMQ	5672	5672		
Redis	5379	6379		
Seq	5340	80		
Zipkin	9411	9411		
Grafana	3000	3000		
Grafana	9090	9090		
SQL server	5433	1433		
MongoDB	8007	27017		
Mongo express	8081	8081		
Dapr Placement			50000	50000
cv.partner.service (our mock and testing service)	5103	80	50005	50001

Figure 4.4: Overview of Docker container service ports.

Another challenge we encountered was dependency management. Dapr services rely on other services, such as databases or message brokers, necessitating proper orchestration within Docker Compose. While the built-in `depends_on` command in Docker Compose verifies if a container is operational, it does not ascertain whether the database has been initialized, presenting a potential obstacle. Ensuring that all dependencies are correctly initiated and accessible before launching Dapr services can be complex, given that Docker Compose initiates all services concurrently by default. To overcome this challenge, we implemented strategies such as customizing the startup sequence and incorporating health checks to ensure the dependencies were prepared before Dapr services commenced.

Lastly, troubleshooting and debugging were more complex when using Docker Compose with Dapr. With multiple containers running in a distributed environment, identifying and resolving issues can require comprehending the interactions between Dapr, Docker Compose, and the services running in containers. To address this challenge, we implemented logging and metrics services like Zipkin and Seq. This will be further discussed in section 4.5.

4.1.3 Command Query Responsibility Segregation

CQRS architecture ensures that each endpoint in our API performs a single task - fetching data (queries), commanding data (commands), or triggering actions.

We believe that adopting the CQRS provides several advantages, including enhanced scalability and separation of concerns. However, it also presents challenges, such as ensuring data consistency and increasing development efforts, which we addressed.

Implementation

Figure 4.5 demonstrates how we separated the various commands and queries using a single database. This visualization emphasizes the code paths for queries and commands and how they interact with the underlying database.

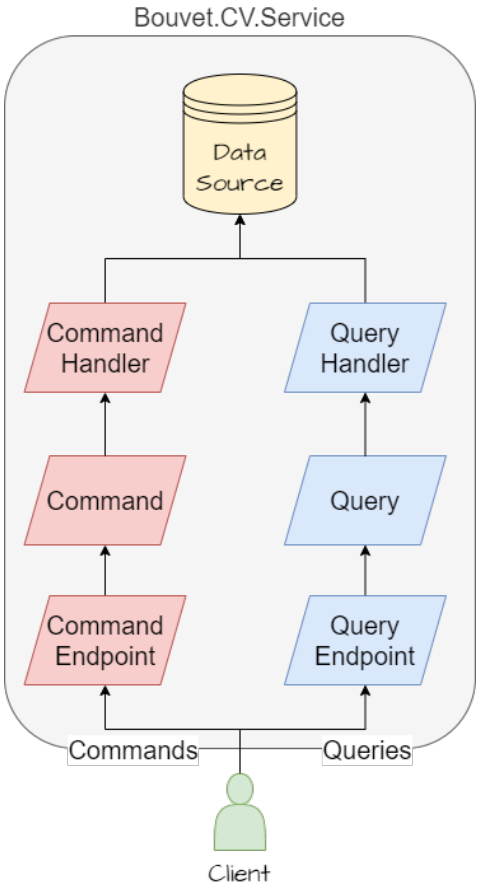

Figure 4.5: Overview of our CQRS architectural pattern approach.

To implement this approach, we organized our project where each microservice had a folder structure that included a folder named *domain*. Within this folder, we created two sub-folders: *commands* and *queries*, which contained the CQRS functionalities.

```
[HttpGet("getrule")]
    public async Task<OperationResult<RuleDTO>> GetRule(string ruleName) {
      var result = await _mediator.Send(new GetRuleQuery(FieldsFactory.StringField(ruleName,
    _stringValidator)));
      Response.StatusCode = result.HttpStatusCode;
      return result;
    }
```

Code Sample 4.1: CQRS query to get a rule

As demonstrated by code sample 4.1, we implemented the CQRS architecture while emphasizing the importance of HTTP methods and status codes. In our opinion, this approach ensured that requests were processed correctly and consistently. We used HTTP methods to

differentiate between commands and queries, employing GET for queries and POST, PUT, and DELETE for commands.

Furthermore, we utilized the HTTP status codes in our CQRS system. We returned status codes when executing commands to indicate their success or failure. Similarly, for queries, we provided status codes to inform the client about the system's state and the status of their request. Following this approach, we believe our API adhered to the REST standard even when implementing the CQRS architecture and utilizing HTTP methods and status codes which maintain a consistent and efficient system.

In our opinion, an important implementation element involved managing commands and queries asynchronously rather than synchronously. This approach enabled us to separate the processing of commands from the response, leading to enhanced performance and responsiveness in our system as we handle multiple commands and queries concurrently.

In implementing the CQRS architecture, we utilized a single database for write and read operations instead of two separate databases. There were several reasons for this decision:

First, since our system's requirements do not require extensive read and write operations, we decided that one database is enough and it avoids additional infrastructure complexity introduced by separate databases.

Second, a single database minimized the potential for data inconsistencies and reduced the complexity of data synchronization between multiple databases.

Based on our understanding, CQRS provided benefits in separating concerns and scalability but also posed certain challenges. The presence of separate code paths for queries and commands required us to implement and maintain two sets of logic, increasing the overall development effort. Testing also became more intricate as query and command paths must be tested along their interactions with the underlying data stores and business logic.

4.2 Design Principles, Strategies and Patterns

This section provides an overview of our common design principles, strategies and patterns adopted across our various services to ensure a consistent and cohesive approach to our development efforts. We aim to ensure that the resulting codebase meets the standards Bouvet expects.

Design patterns offer a standardized approach to addressing common design challenges, enabling us to leverage proven solutions to recurring problems instead of ”reinventing the wheel”. Using design strategies provided guidelines and principles helped us make decisions about code structure.

Although design patterns present advantages, it is key to recognize and address their potential drawbacks. Criticisms include the risks of over-engineering - using patterns where simpler code would be sufficient. Moreover, developers often adhere to patterns without modifying them to suit the needs of their projects. (Refactoring.Guru, n.d.-a)

4.2.1 Data Transfer Object - Pattern

Our main reason for using DTO in our microservice application is to have a defined contract between the services regarding what data they expect. This separation of concerns, in our opinion, optimized the data format and reduced unnecessary data transfer. For example, we transfer only the necessary data to another service rather than sending the entire data model.

In our view, another advantage of using DTOs in our microservice application was improved scalability. As each service operates independently, we optimize the data transfer format for each service's specific use case, allowing us to scale each service as needed. We constructed the DTOs as shown in Code Sample 4.2.

```
1 public record WorkflowDTO {
2     public BWorkflow Workflow { get; set; }
3     public WorkflowDTO(BWorkflow workflow) {
4       Workflow = workflow;
5     }
6   }
```

Code Sample 4.2: WorkflowDTO

In our opinion, using DTOs to transfer data between the different microservices was an optimal choice in our solution as it enhanced the scalability and maintainability of our system, enabling each service to operate independently and facilitating the addition or modification of services with minimal impact on the rest of the implementation.

4.2.2 Mediator - Pattern

The mediator pattern was, in our opinion, the most beneficial design pattern implemented and was facilitated by the MediatR NuGet package. The mediator design pattern proved managing requests and events while enabling a publish-subscribe pattern in our application.

MediatR served as a mediator between objects, eliminating the need for these objects to interact with each other directly, reducing the complexity of their relationships. By encapsulating how these objects interacted and cooperated, MediatR minimized tangled dependencies and promoted loose coupling, which, in our view, enhanced the maintainability and flexibility of our codebase. This proved beneficial in this evolving project, where the ability to adapt to new requirements efficiently was a goal based on our Agile methodology.

4.2.3 Operation Result - Pattern

In addition to using DTOs, we also implemented the OperationResult design pattern in our microservice application.

One example of using the OperationResult design pattern in our microservice application was in the query to get the workflow result. When a user/service attempted to get the workflow, we used the OperationResult as shown in code sample 4.3 to represent the outcome of the operation. If the workflow was successfully fetched, we returned an OperationResult object with a success message and relevant data, such as the workflow. If the workflow was unsuccessfully fetched, we returned an OperationResult object with an error message and applicable error data.

```
public interface IOperationResult {
    IEnumerable<Error> Errors { get; set; }
    bool Success { get; set; }
    string Message { get; set; }
    int HttpStatusCode { get; set; }
  }
  public interface IOperationResult<T> : IOperationResult {
    T Value { get; set; }
  }
  public class OperationResult<T> : IOperationResult<T> {
    public IEnumerable<Error> Errors { get; set; } = new Error[0];
    public T Value { get; set; } = default!;
    public bool Success { get; set; }
    public string Message { get; set; } = " ";
    public int HttpStatusCode { get; set; }
  }
```

Code Sample 4.3: OperationResult class and its associated interfaces.

Implementing the OperationResult design pattern in our microservice application we used a generic type which allows us to use the same object for various inputs. This approach was used to return rules, workflows, and other inputs.

Using the OperationResult design pattern provided several benefits for our microservice application. In our opinion, representing the outcome of an operation in a standardized way, we ensured that our APIs communicated with each other consistently. Additionally, by including any relevant data or error messages in the OperationResult object, we could provide clients with feedback about their request's success or failure.

While the OperationResult design pattern offers advantages to our microservice application, it is essential to acknowledge the challenge it introduced. We believe it increased system complexity due to the added abstraction layer and may present maintenance and comprehension difficulties for those unfamiliar with the codebase.

After evaluation, we determined that implementing the OperationResult design pattern benefitted our microservice application. Standardizing the representation of operation outcomes improved, in our view, the consistency of our APIs, resulting in better communication between microservices. Additionally, we provide clients with feedback about their requests' success or failure.

4.2.4 API versioning - Strategy

We decided to use API versioning design strategy in our microservice architecture to ensure that our APIs remained consistent and reliable over time. In our opinion, API versioning is an important aspect of our microservice architecture, allowing us to maintain consistency and reliability in our APIs while promoting backward compatibility and modularity. This approach allowed us to add new features or capabilities to our APIs in a controlled and incremental manner, reducing the risk of introducing errors or inconsistencies.

In our opinion, one of the key benefits of API versioning is backward compatibility, meaning that clients can continue to use the same API version even as new versions are released. This helps ensure clients are not "stranded" or forced to make application updates. Additionally, API versioning enables us to communicate changes and updates to our clients in a clear and organized way, reducing confusion and errors.

Another benefit of API versioning, in our judgment, is the increased modularity and flexibility it provides to our codebase. We maintained modularity and flexibility in our codebase by enabling us to change our APIs without impacting other application parts. This approach made developing and testing new features or capabilities easier, enabling us to do so in isolation from other system parts. Additionally, this aligns with our microservices approach of loose coupling between services.

```
1 [Route("api/v0/")]
```

```
2 [ApiController]
3 public class AddRuleController : ControllerBase {
```

Code Sample 4.4: API versioning

Our implementation of versioning the API is shown in Code Sample 4.4. As our microservice application is currently in the development phase, we decided to use the API versioning convention of V0 instead of the standard naming convention of V1 for the first stable release. Using this convention ensures, in our opinion, consistency and clarity in our API versioning, making it easier for clients to understand and adapt to any changes as the application evolves. If the application reaches a stable release, the API version should be updated to V1, following industry standards and best practices for API versioning.

4.2.5 API Contracts - Strategy

We have incorporated API contracts as a design strategy for events and commands in our pilot. However, we refrained from implementing contracts for queries due to the nature of HTTP GET requests, which are used for data retrieval and do not contain a body to transmit contracts. Instead, we utilize the URL to convey necessary information.

In our implementation, we utilized the *record* type instead of a traditional class for our API contracts for commands and events.

Records create immutable reference types, which help prevent bugs related to mutable states. Additionally, records provide a built-in value-based equality implementation. (Microsoft, 2022d)

We eliminated the need for an empty default constructor, ensuring that all parameters must be provided to create a valid object. In our view, this design choice prevents nullable values and guarantees that all instances of the contracts will always have the necessary data.

Reflecting on our experience, we believe API contracts facilitate communication between components and services, enabling consistent behaviour across the entire system. Adhering to API contracts ensures that each component and service follows the expected interactions.

4.2.6 SOLID - Principle

During our experiment, we implemented the SOLID principles of software design to the best of our abilities. These principles, namely, SRP, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion, were our guiding tenets throughout the development

process. The complexity of the problem domain, time constraints, and conflicting requirements occasionally hindered our full compliance with these principles. An example was our struggle with SRP in using EF. The SRP emphasize that a class should have only one reason to change. In our case, our EF entities often found themselves responsible for data and navigation properties, thus handling their data and relationships with other entities. In our opinion, this multi-responsibility nature of our entities can be seen as a violation of the SRP. Therefore, we had to strike a balance between strictly adhering to the SOLID principles and managing the practical constraints of our project.

4.3 Utilities

This section will discuss the various utilities we developed to support our microservices-based application, specifically designed for quality control of CVs. We will delve into these utilities' design, implementation, and integration within our application, demonstrating how they enhance our software solution.

4.3.1 Generic Endpoint HTTP Client Generator

This section will explore the creation and application of our generic HTTP endpoint client generator, which we have dubbed *mitmproxy2client*. We will discuss the reasons behind the need for this tool and its potential benefits for Bouvet. Furthermore, we will examine the tool's design and infrastructure before delving into the implementation process and the challenges encountered. We will also demonstrate how mitmproxy2client was utilized to reverse-engineer CV Partner's API and how it can be leveraged for future projects at Bouvet. The five tools we will use are mitmproxy, Python with the Request package, mitmproxy2swagger, a script written in JavaScript, and an HTTP client generator called NSwag.

Challenge and Rationale

To control Bouvet CVs effectively, we must fetch them from where they are stored. Bouvet uses CV Partner as their collaborator for keeping their employees CVs. Regrettably, the OpenAPI documentation from CV Partner was outdated during the experiment. As a result, we reached out to them, requesting an updated version of the documentation. They replied that they did not have an OpenAPI specification available at the moment but were planning to implement it in the future. We presented our findings to Bouvet. They now perceived the development of an application capable of automatically reverse engineering CV Partner's API using endpoints as a more efficient approach to generating our own OpenAPI specification. This approach had to be carried out to fully utilize the available resources and enhance the functionality of our project, all while respecting CV Partner's property.

The HTTP Client Generator tool was implemented to ensure adaptability and keep in sync with CV Partner API changes. Moreover, this approach facilitated a generic solution allowing Bouvet to leverage this technology for other projects, expanding its potential applications and promoting reusability within the organization.

A challenge in developing our application's generic endpoint HTTP client generator was faced with two viable options for generating an HTTP client from an OpenApi Specification: NSwag and Swagger OpenAPI tools.

Both of these tools proved efficient in generating clients. To compare the two, we evaluated the generated code, associated documentation, and the overall complexity of utilizing each tool.

Upon examination, we found both tools to be user-friendly. However, the Swagger tool generated a more substantial number of files, creating a separate file for each JSON property. In contrast, NSwag generated a single, larger file encompassing all JSON properties and deserialization. Even though dividing code into multiple files may enhance readability and maintainability, we believe it introduces unnecessary complexity to our application. Moreover, the Swagger tool employed the Newtonsoft package, while NSwag allowed the selection between Newtonsoft and Microsoft's System.Text.Json package.

In our research, we compared the default JsonSerializer behaviour of System.Text.Json and Newtonsoft.Json to determine the most suitable library for our application. The default JsonSerializer behaviour in System.Text.Json and Newtonsoft.Json differs in several aspects.

System.Text.Json prioritizes strictness, deterministic behaviour, execution time, and security, while Newtonsoft.Json is more flexible by default. (Microsoft, 2022a)

Key differences include:

1. **Case sensitivity**: Newtonsoft.Json does case-insensitive property name matching during deserialization, while System.Text.Json is case-sensitive, providing better performance. (Microsoft, 2022a)

2. **Character escaping**: System.Text.Json escapes more characters by default for enhanced security against Cross-Site Scripting (XSS) and information-disclosure attacks. (Microsoft, 2022a)

3. **Comments and trailing commas**: Newtonsoft.Json ignores comments and trailing commas in JSON during deserialization, while System.Text.Json throws exceptions due to their exclusion from the "RFC 8259"(T. Bray, 2017) specification. (Microsoft, 2022a)

4. **Converter registration precedence**: System.Text.Json and Newtonsoft.Json have different registration precedence for custom converters, with System.Text.Json prioritizing run-time changes over design-time choices.(Microsoft, 2022a)

5. **Maximum depth**: Both libraries have a default maximum depth limit of 64. ASP.NET Core sets the default maximum depth limit to 32 when using System.Text.Json indirectly. (Microsoft, 2022a)

These differences in design and functionality were factors in our decision-making process.

A comparison study by Medium.com(Streng, 2022) corroborates the claims made in the Microsoft documentation, asserting that System.Text.Json outperforms Newtonsoft.Json in terms of speed. The study compares Newtonsoft.Json and System.Text.Json by evaluating their deserialization and serialization performance. Given that the main emphasis of this segment of our program is the deserialization of numerous small data objects — transforming JSON objects into C# objects — we will primarily focus on this aspect of the performance comparison. Comparison data and graph from Medium.com are depicted in Figure 4.6 showing 10000 objects deserialized using the two different tools.

Method	Count	Mean	Ratio	Allocated	Alloc Ratio
NewtonsoftDeserializeMuchData	10000	15.577 ms	1.00	35.54 MB	1.00
MicrosoftDeserializeMuchData	10000	7.916 ms	0.51	4.8 MB	0.14

DeserializeMuchData.md hosted with ❤ by GitHub view raw

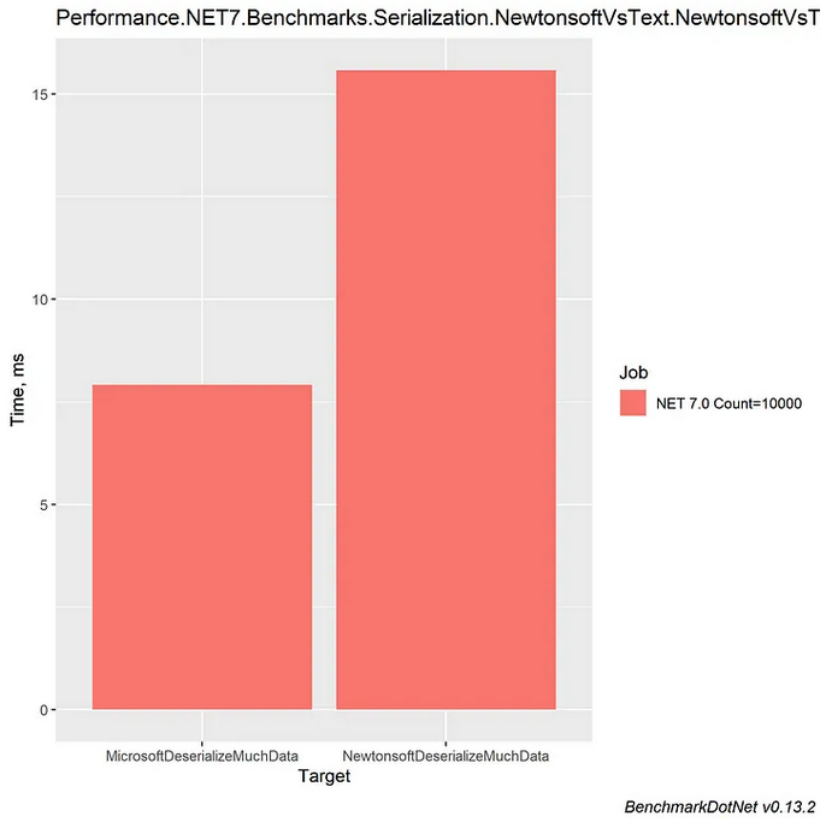

Figure 4.6: Microsoft's framework, System.Text.Json vs Newtonsoft.Json. (Streng, 2022)

The comparison reveals that System.Text.Json is nearly twice as fast as Newtonsoft.Json in processing time, and the memory consumption of System.Text.Json is almost seven times lower than that of Newtonsoft.Json.

Considering our security goal of utilizing up-to-date and well-maintained packages as discussed in section 4.6.4, along with those native C#, we decided to employ NSwag for generating the

HTTP client, the System.Text.Json package. Our decision was further supported by the following advantages:

1. NSwag produced less code for equivalent functionality.

2. System.Text.Json C# package provided faster deserialization.

3. Stricter deserialization approach with System.Text.Json for greater control.

Design

The tool's design uses a modular configuration, making it, in our opinion, easy to use and configure for fast and effective utilization because the users were able to configure parameters such as the base URL, authentication tokens (if required), request headers, and timeout settings. This provided flexibility and tailored the tool to different APIs and use cases.

mitmproxy2client was also designed with a microservices architecture, allowing future updates using other services containing different frameworks or libraries. In our view, this modular design ensured the tool was scalable and adaptable.

Infrastructure The infrastructure of our tool comprises a range of interconnected components, parts, and modules that collaborate. The infrastructure is summarized in the following list and Figure 4.7:

Configuration Management: A file to handle essential configurations such as the base URL, authentication tokens, request headers, and timeout settings, allowing the tool to adapt to different APIs and use cases.

HTTP request and response capture: A component used to save, intercept, inspect, modify, and replay traffic between a server and a client. The component enables us to capture HTTP and HTTPS requests and responses, including headers, query parameters, and request/response bodies.

Networking layer: A core module responsible for establishing connections, sending HTTP requests, and receiving server responses. The module simplifies creating and sending HTTP requests, supporting various methods (GET, POST, PUT, DELETE, etc.). Further, the module processes and parses HTTP responses, extracting relevant information such as status codes, headers, and response bodies.

OpenApi Specification Generation: A component that automates OpenAPI documentation generation based on observed network traffic. It is an addition to the toolset for API analysis, documentation, and integration tasks.

Data Formatting and Manipulation: A central element that simplifies the process of designing and documenting APIs by automating the creation of standardized and modular components within the OpenAPI specification. The element will generate schema definitions for various data types, including objects, arrays, and primitives.

Http client generation: The part that generates client code from OpenApi Specifications. It offers configuration options to tailor the generated code and documentation to our requirements.

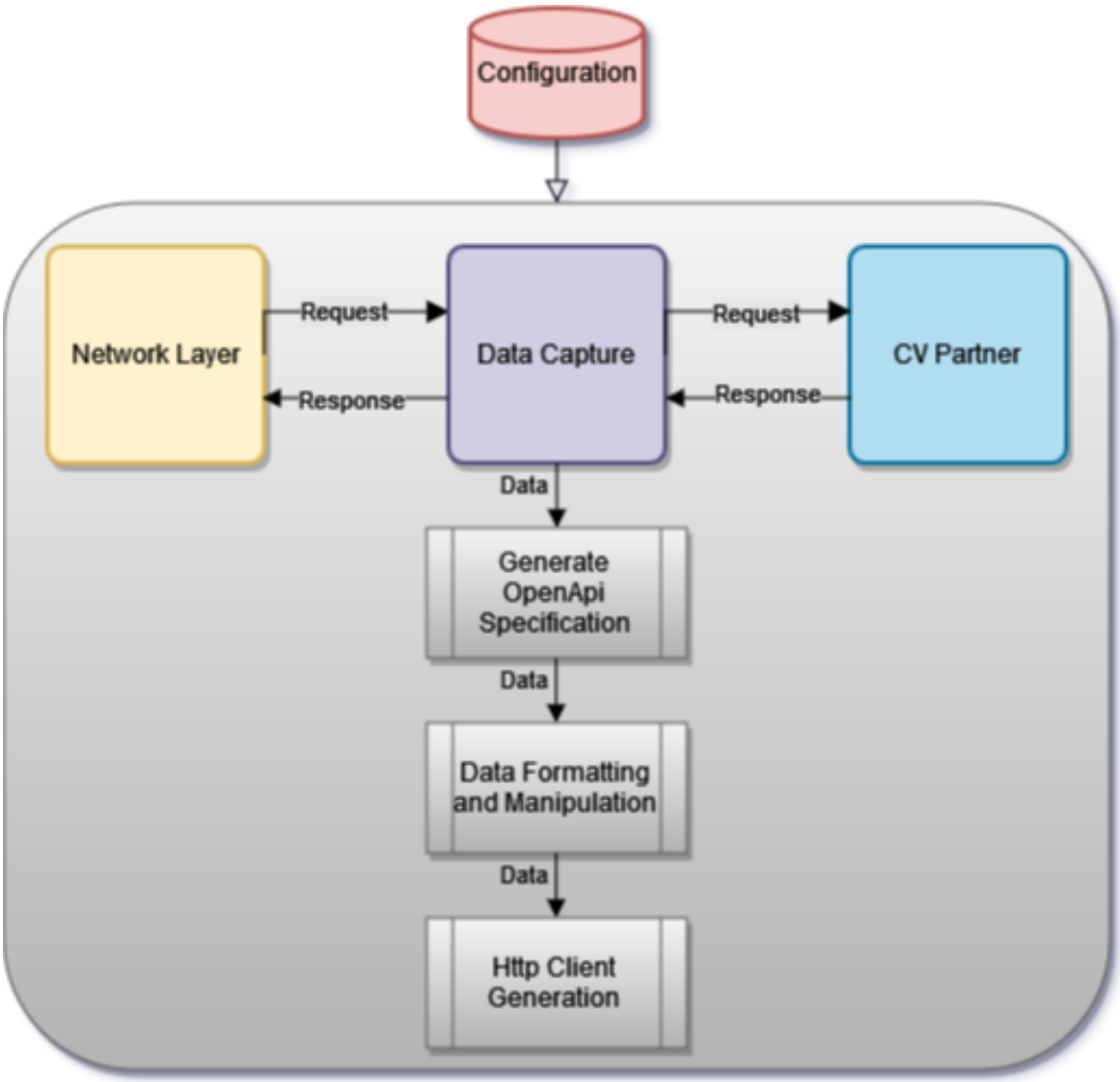

Figure 4.7: Interaction of components in the HTTP client generation tool.

Implementation

As the tool combines five individual services, we decided with Bouvet to run the tool in a Docker containerized environment. We believe the containerized environment ensures universal operability, allowing the program to run on any computer or OS. By leveraging the functionality of Docker Compose using a compose.yaml file, multiple containers are able to operate in unison.

A dedicated, user-defined *bridge*-type network between the Docker containers named ”mitmproxy2clientnetwork” was established to facilitate communication between the containers. We used a user-defined network instead of the default network that Docker installs on every host.

The user-defined network ensures better network isolation, allowing attaching and detaching containers on the fly and exposing all ports to each other without needing to publish the port using the `-p` or `--publish flag` (Inc., n.d.-b).

To facilitate data sharing between our containers, we used docker volumes. In our opinion, implementing shared volumes has resulted in an organized system by promoting resource utilization and efficient data management, enabling containers to store and access commonly used files.

HTTP request and response capture

The container containing the HTTP request and response will start when we run the composed containers. The proxy capturing web traffic is mitmproxy which will need to start first to gather the necessary data for further processing.

To containerize mitmproxy, we use a start-up script and the official mitmproxy Docker image from Docker Hub. The script sets configurations read from *.env*. The .env file is the environment variable file setting variables in the containerized environment. Every container running in mitmproxy2client contains a start-up script and common settings. As the settings are similar across the different services, the start-up script will only be explained here, even though all services contain such a script.

To save files for backup purposes, the script checks the configuration. For example, If `SAVE_OLD_FLOW` is set to `true`, the previously saved *flow* file containing the captured web traffic is saved. Code Sample 4.5 shows part of mitmproxy start-up script. Our reasoning for creating back-ups is further elaborated at the end of this section.

```
1 if [ -f "$SHARED_FOLDER_PATH""flow" ]; then
2     if [ "$SAVE_OLD_FLOW"=true ]; then
3         DATE=$(date +'%b%d%Y%X')
4         echo SAVE_OLD_FLOW=${SAVE_OLD_FLOW} - Saving old flow to ${BACKUP_FOLDER_PATH}
5         mv ${SHARED_FOLDER_PATH}flow ${BACKUP_FOLDER_PATH}flow${DATE}
6     else
7         echo SAVE_OLD_FLOW=${SAVE_OLD_FLOW} - Old flow file is deleted..
8         rm ${SHARED_FOLDER_PATH}flow
9     fi
```

```
10 fi
```

Code Sample 4.5: mitmproxy Start-up script

The conventional method of downloading mitmproxy CA certificates through a browser described in the mitmproxy theory section does not apply to our implementation. As mitmproxy operates without a User Interface (UI) within our containerized environment, we share the certificates between containers using Docker volumes to adapt to this setup.

Networking layer

After mitmproxy started recording web traffic, a container running a Python Requests script was started. Python Requests will send requests to CV Partner, which are intercepted and recorded by the mitmproxy container. Figure 4.8 from the mitmproxy documentation shows the interaction between mitmproxy, the client, and the server. Python Requests is the client, while CV Partner is the server in our implementation. The implementation uses the *.env* file for storing authorization cookies, tokens, and other confidential data, this information is not hard-coded, which we believe reduces the risk of accidental exposure or unauthorized access.

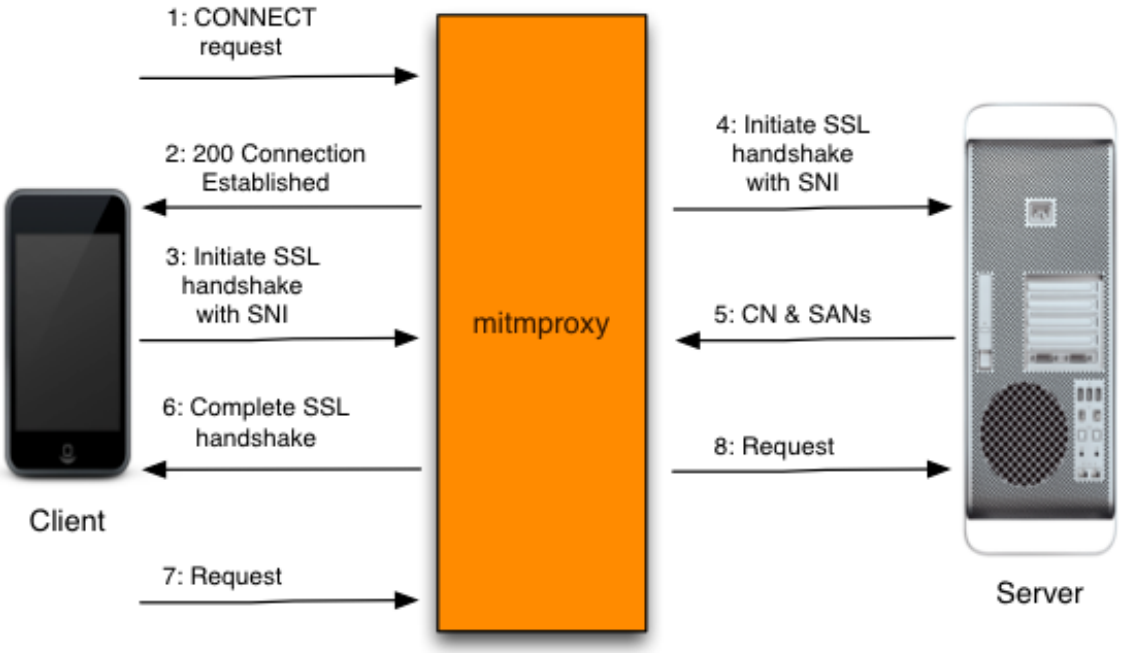

Figure 4.8: Requests made from client to server through mitmproxy. Each arrow indicates the flow of information.

(mitmproxy, n.d.-a)

When running the services together, we faced a challenge: Python Requests sent requests before mitmproxy generated its CA certificates. Consequently, the server denied the requests.

Initially, we addressed this issue by forcing Requests to wait for a set duration before starting using a sleep function. This approach worked but adversely impacted time performance, causing the program to wait longer than necessary.

Our new approach involved controlling the startup and shutdown order of the containers using the `depends_on` functionality of Docker with the `service_started` condition.

We identified a problem with this approach as well. Although mitmproxy had started running before we initiated the requests, the CA certificates had not been generated.

We found that during initialization, Docker Compose does not pause for a container to reach the ”ready” state; instead, it proceeds once the container is actively running, which is also stated in the Docker documentation (Inc., n.d.-a).

This implied that even if the proxy component of mitmproxy was running, it did not guarantee the generation of the CA certificates.

In response to this problem, we utilized the condition `service_healthy` which verifies that a predefined criterion within a service had been met before initiating another container for the `depends_on` functionality.

We discovered that the folder (/.mitmproxy) containing the certificates would not be created until the certificates were generated. Consequently, the solution was to verify the folder's existence before starting the Requests service.

mitmproxy generates various certificates for different web browsers and OS. One of the main objectives of our implementation is to ensure flexibility. Adding the folder containing all certificates to the check enables the service to be run on different systems.

We set the volume using the code illustrated in Code Sample 4.6 to enable the containers to access CA certificates in the shared volume. This instructs the MitmProxy container to share its */root/.mitmproxy/* folder as the volume named *ca-certs*.

```
1 MitmProxy:
2   volumes:
```

```
3     - ca-certs:/root/.mitmproxy/
```

Code Sample 4.6: mitmproxy container's */root/.mitmproxy/* folder shared as the volume *ca-certs*.

Folders have different names inside each container but still map to the same shared volume. If the shared volume specified in the Docker Compose file does not exist, it is automatically created, as is the case for each container volume, like the */certs/* folder in the Python Requests container. By sharing the generated CA certificates with the Requests container, Python Requests can send encrypted messages using HTTPS.

To shut down mitmproxy after the requests are finished, we send a request from the Python Requests container to the */stopmitmdump* endpoint in the mitmproxy container over the Docker network. Upon receiving this request, the proxy container terminates using a kill script.

Python Requests stops as all the requests have been executed, and the captured data is stored in a flow file. The flow file is saved in the shared volume. Figure 4.9 shows the sequence of running requests through mitmproxy to CV Partner.

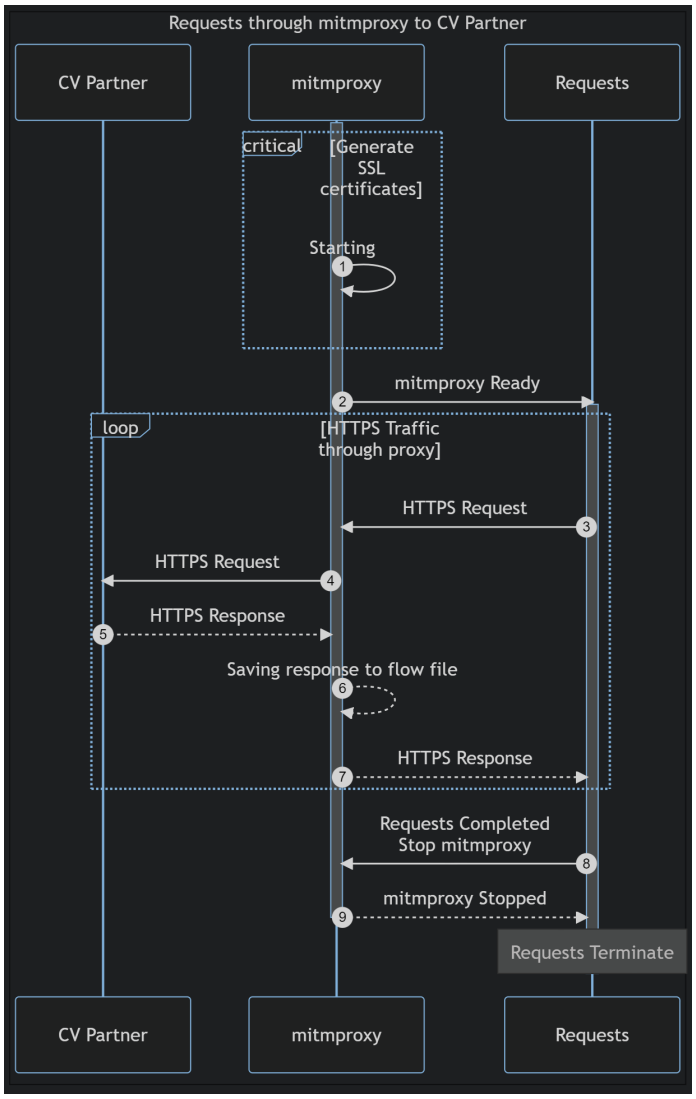

Figure 4.9: Communication between mitmproxy, Python Requests, and CV Partner website.

OpenApi Specification Generation

Upon completion of Python Requests, the traffic is saved to the flow file for use by the tool mitmproxy2swagger. mitmproxy2swagger is initiated when Python Requests stop by using the Docker Compose condition: `service_completed_successfully`. This condition ensures that mitmproxy2swagger starts after Python Requests has completed.

mitmproxy2swagger accesses the shared volume to process the flow file generated by mitmproxy and uses this file as the input. The process of generating the OpenApi Specification

is explained in detail in the theory section about mitmproxy2swagger. We implemented a script to automate the process of generation. In addition to setting the backup variables, the mitmproxy2swagger script includes configuration for the OpenApi Specification generation.

```
mitmproxy2swagger -i ${SHARED_FOLDER_PATH}flow -o ${SHARED_FOLDER_PATH}mitm2swaggeroutput.
    yaml -p ${BASE_URL} --format flow && \

sed -i -e 's/ignore://g' ${SHARED_FOLDER_PATH}mitm2swaggeroutput.yaml && \

mitmproxy2swagger -i ${SHARED_FOLDER_PATH}flow -o ${SHARED_FOLDER_PATH}mitm2swaggeroutput.
    yaml -p ${BASE_URL} --format flow
```

Code Sample 4.7: mitmproxy2swagger OpenApi Specification generation script.

Our objective was to automate the program from start to finish. This objective was accomplished using the script portrayed in Code Sample 4.7. The script executes the mitmproxy2swagger tool, providing designated input and output files to create an OpenAPI specification from the input flow file. Further, the base URL for the requests saved is specified to know what endpoints to use for generation. All the `ignore:` prefixes are removed before the command is run again, generating the OpenApi Specification.

Data Formatting and Manipulation To Generate the OpenApi Specification for our HTTP client, initiating numerous requests and gathering a diverse collection of CVs are important as properties between CVs vary. We enriched the flow file with data by collecting diverse responses to ensure a complete and accurate data representation.

A downside of saving these responses was many duplicated properties within the OpenAPI Specification generated by mitmproxy2swagger. By identifying these common properties and systematically placing them in the global *components* section of the OpenApi Specification, we created a centralized repository for properties reused across multiple CVs.

To use a component section in an OpenAPI Specification, we use the `$ref` keyword to reference the properties and reduce property duplication. In our opinion, this approach lets us locate and update shared definitions as needed. Additionally, minimizing code duplication reduce the potential for errors and inconsistencies.

To automatically identify common elements and systematically place them in the global components section, we implemented a script in JavaScript. The script extracts properties from the schemas section and moves them to the components section.

The script uses the necessary packages for its functionality: ”node-jq” for JSON filtering, ”fs”

for file system operations, ”js-yaml” for YAML parsing, and ”console” for logging. A flow chart of the script is presented in Figure 4.10:

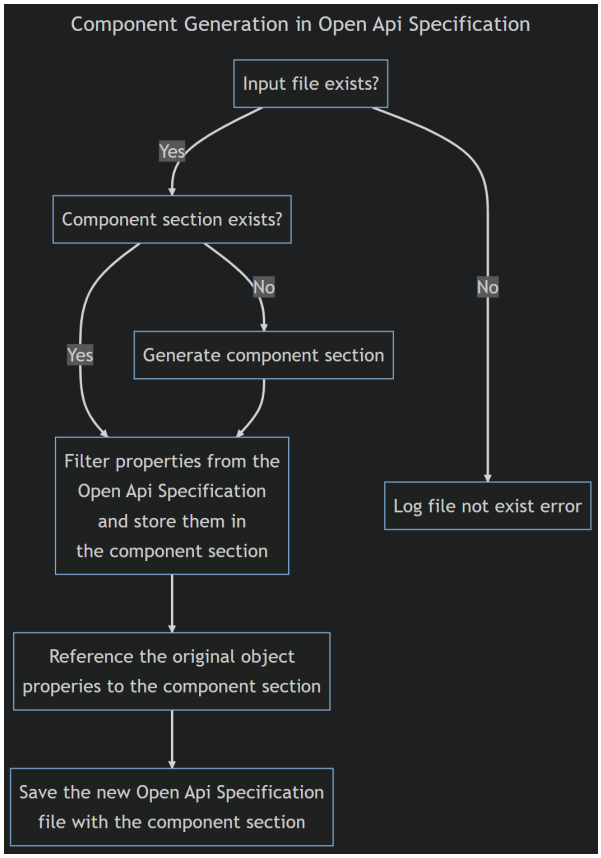

Figure 4.10: Components generation.

Http client generation

In the last step, we employ NSwag, a tool for generating c# HTTP clients.

NSwag is, as mentioned in the theory section primarily designed with a UI called NSwag Studio. However, we were required to run it within a Docker container and automate this process, which required additional steps. We had to configure NSwag for a non-interactive, command-line mode to make it compatible with our requirements, ensuring integration and operation within our Docker setup.

In our implementation, the container uses the *curl* command to download the NSwag tool zip file from the Nswags GitHub release page. After the installation is complete, the DockerFile

is instructed to run the *dotnet* command running the *dotnet-nswag.dll* file that is included in the NSwag directory. This file contains the code to run the NSwag.

The command also supports using a configuration file upon client creation. The start of the configuration file *nswagconfig.nswag* is presented in Code Sample 4.8.

```
{
  "runtime": "Net70",
  "defaultVariables": "InputSpec=Spec",
  "documentGenerator": {
    "fromDocument": {
      "json": "$(InputSpec)",
      "url": "",
      "output": null,
      "newLineBehavior": "Auto"
    }
  },
  "codeGenerators": {
    "openApiToCSharpClient": {
      "clientBaseClass": null,
      "configurationClass": null,
      "generateClientClasses": true,
      "generateClientInterfaces": true,
      ...
...
```

Code Sample 4.8: Start of NSwag configuration file.

Upon creation of the HTTP client, we extract the generated client files saved in the shared volume and integrate them into our project. In the event of API changes or contract updates, we re-run the process to generate an updated HTTP client and replace the existing project files with the new version. In our opinion, this approach ensures that projects remain up-to-date and compatible with the latest API modifications.

File backups

If the previous files generated by the program exist, we can save them to a separate folder specified in the environment variables named *.env*. This ensures that previously generated files are not overridden by created files. Additionally, the backup files are renamed to include the date and time of their creation for convenient access later. This naming convention, we believe, ensures that multiple versions can be stored and backups can be located and used if necessary.

A backup implementation may protect against data loss, ensuring that files are not lost due to hardware failure, software issues, accidental deletion, or other unforeseen events. In our case, backups provide a record of changes over time, which is useful for tracking progress, reviewing past work, or verifying changes made to a file.

4.3.2 Custom Fields Type Library

Fields is a generic and reusable custom library we developed in our software development project. It provides a collection of custom types, such as string, integer, DateTime, etc., that are designed to be easy to use and offer data validation functionality. By creating this library, we believe we provide a consistent and convenient way to work with different data types and enable extended functionality to validate our types. Fields library is designed to be generic and user-friendly to let other employees at Bouvet utilize it.

Challenge and Rationale

As part of our discussions with Bouvet regarding our microservice architecture approach, we identified a need for a custom library that provides reusable custom types with built-in validation capabilities. We found that existing libraries did not fully meet our requirements regarding customization, so we proposed the creation of the Fields library.

In a microservice architecture, data is often spread across multiple services. In our opinion, maintaining data consistency and integrity can be challenging, leading to data duplication and inconsistency. To mitigate this, we aimed to centralize the data type definitions and validation rules with Fields.

In addition, we designed the Fields library to be deployable to Bouvet's GitHub using NuGet. This ensures integration with our existing development workflow and makes it available for other teams and projects within the organization.

Design

The Fields library employs a configurable approach to data validation. A configuration file is defined by users to customize validation rules and settings. This configuration object, following the Options design pattern, is open for modification of validation rules or the addition of new ones without directly altering the source code during runtime.

The configuration is injected into the validator, supplying the necessary parameters for validation. The flow of our design is illustrated in Figure 4.11. This approach ensures integration of the configured validators while separating concerns. By decoupling the validation logic

from the application code, the resulting architecture is easier to maintain and extend, in our opinion.

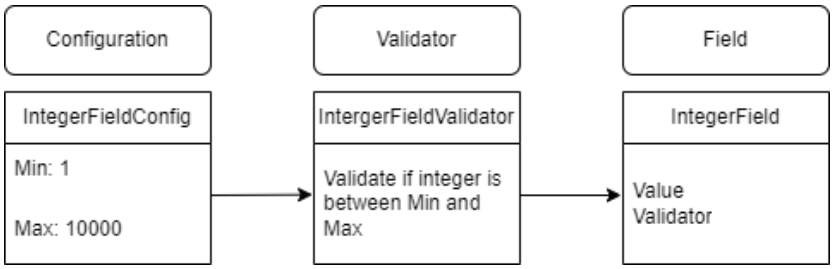

Figure 4.11: The flow of creating a field type.

In addition to the configurable approach using a configuration object, the Fields library utilizes the Factory design pattern. The Factory Design Pattern, a creational design pattern, enables flexible and modular object creation without revealing the underlying object creation logic to the client. It invokes a method or function from a factory class that generates objects based on input parameters or other conditions. Consequently, client code only needs to call the factory method or function, remaining unconcerned about the object creation process's implementation details.

Fields library utilizes the Factory Design Pattern. In our view, this pattern offers several benefits, such as enhanced modularity and flexibility, contributing to application maintenance, updates, and extensions. It enables adding new objects or modifying existing object creation processes without affecting client code, centralizing changes to the object creation process.

The Factory Design Pattern also facilitates features that simplify working with the Fields library. For instance, we implemented default values for validators and their configurations in the factory for the library to be ready to use without first needing to configure the validators.

Implementation

In implementing the Fields library, we utilized a standard interface called `Validate` from which all our custom types inherit. This interface served as a contract that enforced the presence of two validation methods: `DoValidate` and `DoValidateAndThrow`.

`DoValidate` provided a way to perform validation and return a `ValidationResult` indicating whether the validation passed or failed for a user to handle validation results.

On the other hand, the `DoValidateAndThrow` method performed the validation but addi-

tionally threw an exception with an error message when the validation failed. This provided a more robust approach for handling validation errors, which we believe allows users to handle exceptions in a centralized error-handling mechanism rather than manually checking the boolean result of the validation.

Fields library encompasses configuration classes, validators, and custom fields. The library's structural overview and its relationships are illustrated in Figure 4.12. The configuration classes form the foundation, supplying customizable settings for fields and validators.

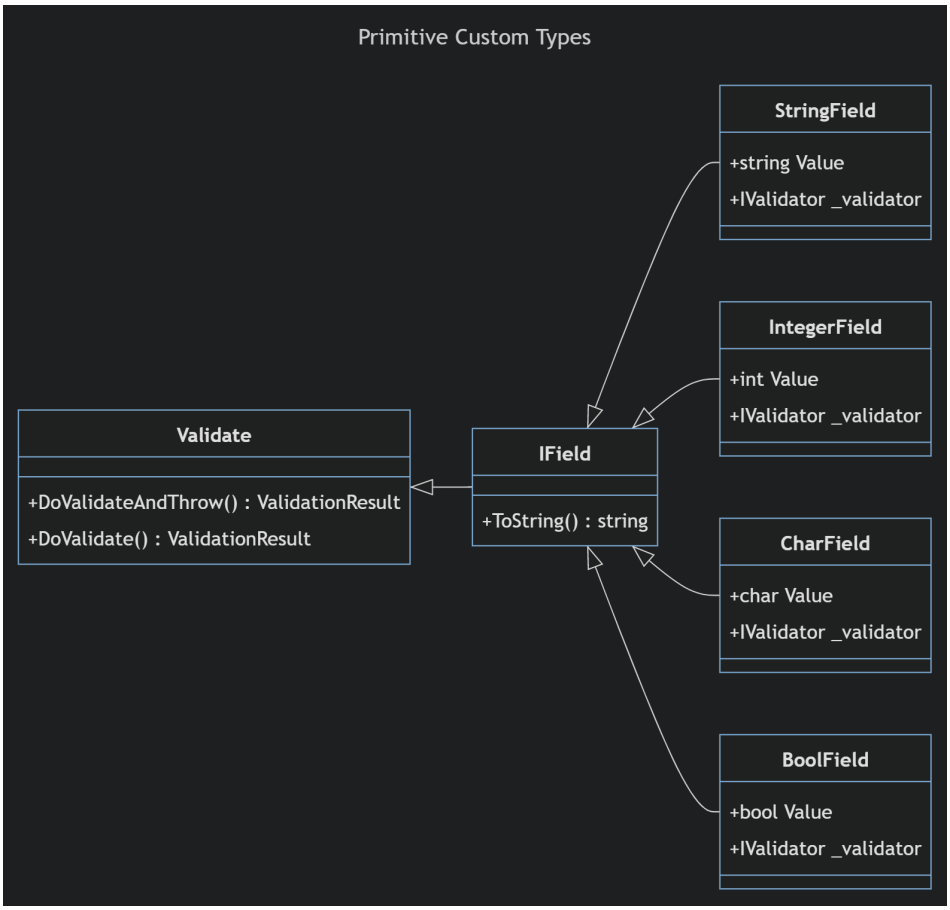

Figure 4.12: Primitive Fields types inheritance relationship.

Our Fields library implementation established custom types inherited from unique interfaces and the validate interface. For example, we incorporated the `IStringField` interface within the `StringField` class. Our implementation is shown in Code Sample 4.9, and all the custom field types adhere to this implementation. In our opinion, this fosters extensibility and modular development, enabling other developers to craft custom field types that align

with their specific needs.

```
public partial class StringField : IStringField, Validate {
    private readonly IValidator<IStringField> _validator = FieldsFactory.StringValidator();
    public StringField(string value, IValidator<IStringField> validator) {
      Value = value;
      _validator = validator;
    }
    public StringField() { }
    public string Value { get; set; } = string.Empty;
    public ValidationResult DoValidate() {
      return _validator.Validate(this, options => options.IncludeAllRuleSets());
    }
    public ValidationResult DoValidateAndThrow() {
      return _validator.Validate(this, options => options.IncludeAllRuleSets().
    ThrowOnFailures());
    }
  }
```

Code Sample 4.9: Custom field implementation example.

To validate our custom types, we employed the *.NET* package *FluentValidation* to create validators. FluentValidation enabled us to establish validation rules for each custom type, ensuring that our data meets the desired criteria and constraints. The implementation of a `StringFieldValidator`, used for validating the `StringField` type, is demonstrated in Code Sample 4.10.

```
public class StringFieldValidator : AbstractValidator<IStringField> {
    public StringFieldValidator(IOptions<StringFieldConfigOption> configuraiton) {
        RuleFor(x => x.Value)
        .NotNull()
        .NotEmpty() .WithMessage(SystemMessages.Validation.NullOrEmptyMessage)
        .MaximumLength(configuraiton.Value.Max)
        .MinimumLength(configuraiton.Value.Min)
        .WithMessage(SystemMessages.Validation.OutOfBoundary)
        .Must(x => Must.ContainsOnlylegalCharacters(x))
        .WithMessage(SystemMessages.Validation.IllegalCharacters);
    }
}
```

Code Sample 4.10: Custom validator for the `StringField` type.

By utilizing the standardized interface of the `AbstractValidator`, we facilitate the creation of validators for each corresponding interface for users to develop their custom fields while inheriting from our foundational interface.

In implementing configuration for the validators, we utilized a Configuration Option class that employs the Options Pattern to match the JSON data of the configuration file to the corresponding class. When creating the `StringFieldValidator` object in Code Sample 4.10, dependency injection passes the configuration parameter in the constructor. Additionally, we developed validators for the configuration to ensure that the provided configuration adheres to the required constraints and conditions.

Deployment

We deployed our Fields library as a *NuGet* package hosted on GitHub, enabling employees at Bouvet to install it using the NuGet package manager.

By choosing *NuGet* as our package manager, we addressed Bouvet's need to integrate the Fields library into other external existing projects. The command `dotnet add package fields` allows users to add the Fields library to their codebase, provided they have a NuGet configuration file containing the package location and the necessary API key.

We believe the built-in versioning support in NuGet is an advantage in deploying our library. We implemented versioning support by assigning a version identifier to 1.0.0 as this was our first iteration. This enables us to update the library without affecting users that are using earlier versions of the package, ensuring compatibility and stability in software projects.

Moreover, versioning assists in managing dependencies and resolving conflicts in software projects with multiple dependencies, ensuring that different libraries or packages used in the project can work together without conflicts.

NuGet's integration approach, versioning support, and compatibility management make it, in our opinion, a reliable choice for distributing our Fields library, addressing Bouvet's requirements for sharing the library with other employees and enhancing its accessibility and applicability in various applications.

4.3.3 Generic Rules Engine

In this section, we will examine and analyze the design and implementation of our generic Rules Engine.

Challenge and Rationale

The motivation for developing a generic Rules Engine was driven by the need for a reusable and extensible solution that caters to Bouvet's needs, such as the quality control of a CV

database. The Rules Engine should be as generic as possible, with Bouvet being able to run any workflow on any input and not just CVs, hence catering to Bouvet's requirements. As we see it, this approach ensures that the solution can adapt to future changes or expansions in the project's scope.

We analyzed various approaches for implementing a Rules Engine into our project. Build a Rules Engine from scratch or leverage existing open-source projects. We decided to build upon an existing open-source Rules Engine owned by Microsoft. In our opinion, this approach allowed us to develop rules and other application aspects, which give Bouvet more value. Our methodology of creating the most value in the least amount of time motivated this approach. The Microsoft Rules Engine provided the foundation, which we used to expand its functionality further.

We created custom entities and methods to expand the Microsoft Rules Engine capabilities. In our opinion, this enabled us to address the limitations of the Microsoft Rule Engine and provide additional functionality, such as incorporating complex logic, custom types as we created in our Fields library, and rule formats beyond lambda expressions. Our custom entities consist of *BWorkflow*, *BRules* and *BInputs* and their respective fields are portrayed with their properties in figure 4.13.

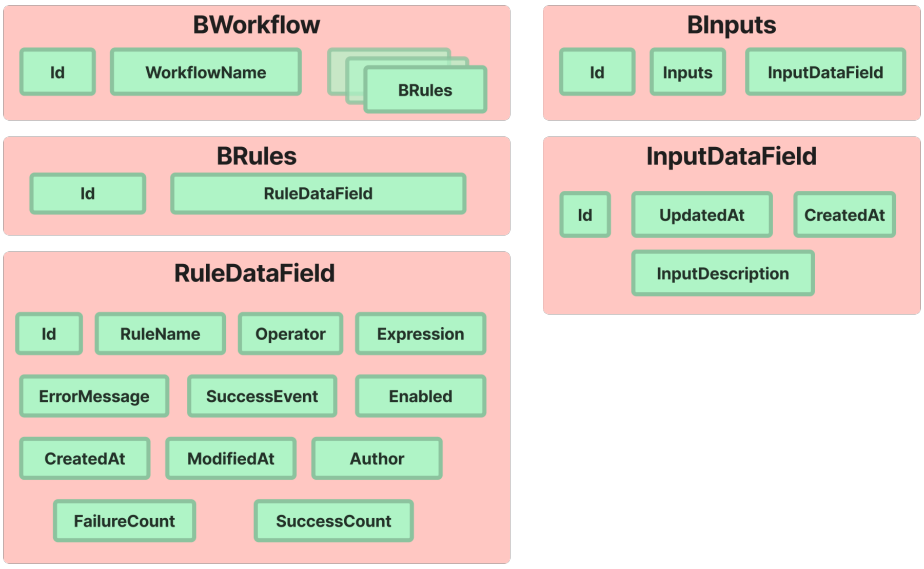

Figure 4.13: Rules Engine Entities: BWorkflow, BRules, RuleDataField, BInputs and InputDataField

Additionally, these entities allowed us to utilize EF for database integration. The flow of our Rules Engine is displayed in figure 4.14. The entities depicted in the figure will be examined following the presentation of the illustration.

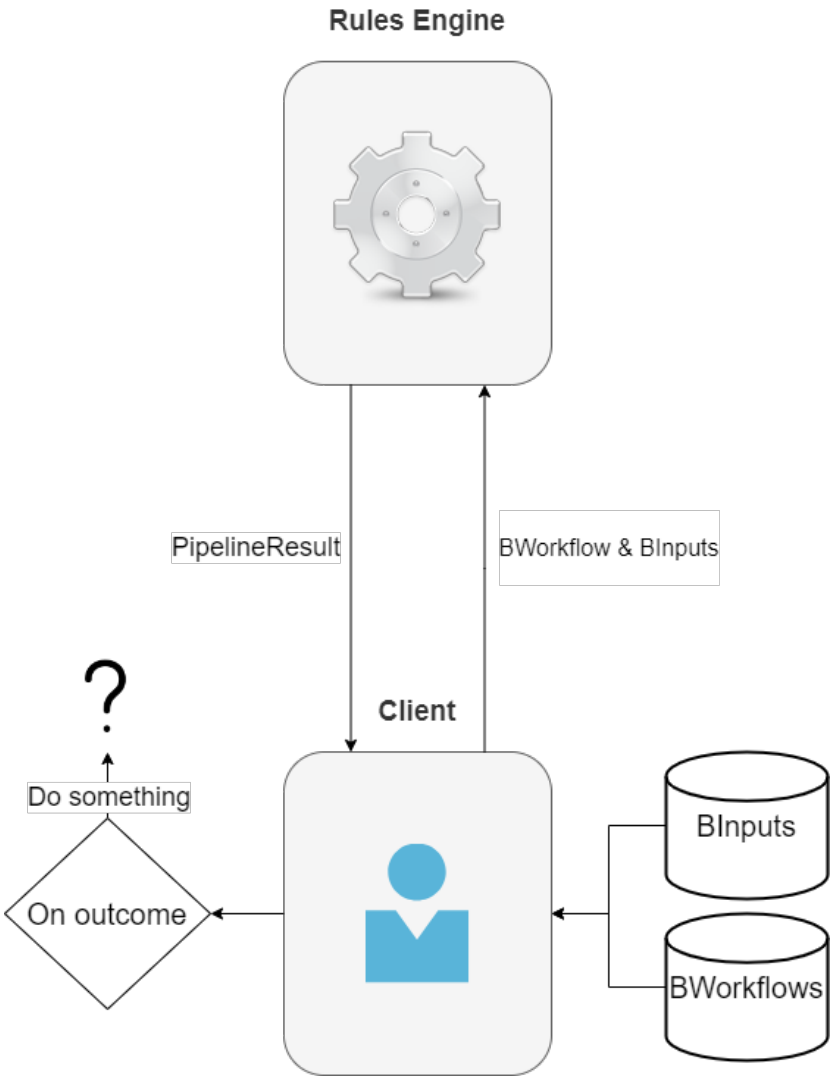

Figure 4.14: The flow of our Rules Engine depicting the entities in the form of BInputs and BWorkflows.

Our design considered users of our application to not have a direct dependency on the Microsoft Rules Engine, as they must adhere to our custom entities, BWorkflow, BRule, and BInputs, to utilize our generic Rules Engine.

Implementation

Firstly, we developed the class BRule to act as our rule class. It includes the rule ID, validator, and a Ruledata field. The Ruledata field contains all our necessary rule information, and we chose to separate the rule-related information into a separate field as it, in our view, offered

the following benefits:

Firstly, modularity is promoted by encapsulating the rule data into a separate field or object to manage and manipulate rule-related information. This modular design approach ensures that each component or field has a single responsibility or purpose, adhering to the SRP of SOLID, making the codebase more maintainable and extensible.

Secondly, separation of concerns is enabled, where the rule data is kept separate from other logic or functionality in the class. This promotes a more organized and structured approach to managing rule-related information, making the codebase more understandable and maintainable.

Thirdly, separating the rule data into a field allows for flexibility in handling different types of rule data, as it can be replaced or extended with objects or data structures as needed without impacting the overall structure or behaviour of the class. In other words, it adheres to the SOLID Dependency Inversion principle.

Enabling the use of our custom types from the Fields library, we created custom validators for our objects. Using our custom types, we created a `RuleDataFieldValidator` class that validates every property and conveniently reuses the multiple validators created in our Fields library. This design of creating custom validators using our custom types in our Fields library is followed throughout our implementation of the Rules Engine and its related classes.

The `BWorkflow` class serves as a container for multiple rules, being organized, managed, and executed as a cohesive unit, as previously shown in figure 4.14. This defines business workflows involving multiple rules with different dependencies, conditions, and actions, all encapsulated within a single `BWorkflow` instance.

To get input from the user, we created a `BInputs` class that can dynamically contain any object or type, respecting the input provided by the user. This is made possible by utilizing the *dynamic* type in C#.

The dynamic type, while technically a static type, enables an object to avoid static type checking. It typically behaves as if it were of the object type. The compiler presumes that a dynamic element can perform any operation. As a result, it is not necessary to figure out if the object receives its value from a COM API, a dynamic language like IronPython, the HTML Document Object Model, reflection, or any other part of the program. However, if the code is incorrect, errors will emerge during run time. (Microsoft, 2023e)

Using the dynamic type also comes with trade-offs. The trade-offs include reduced compile-time type checking, and potential runtime errors and should be used judiciously and with error handling in place. A challenge we faced creating the `BInputs` class was validation of the dynamic input as the specific type of the dynamic variable is checked at runtime. Discussing the issue made us settle for a simple null check validator. A check for *not null* would provide validation before executing the rule.

An exception would be thrown if the dynamic variable was not empty but contained other errors from the Rules Engine, which would then provide users of the Rules Engine feedback.

In our opinion, these are valid trade-offs as Bouvet's required a generic Rules Engine.

The Rules Engine features a single public method, `ExecuteWorkflowAsync` which accepts objects of type `BWorkflow` and `BInputs` as parameters. These objects encompass information needed to map our custom classes to the classes provided by the Microsoft Rule Engine to be able to run. The Rules Engine sequentially processes each rule and executes them based on specified conditions.

Upon completing the rule execution, the result is obtained, encompassing the rule execution outcome, any errors or failures encountered during the process, and details of the failed rule, if applicable. The result is then mapped to our custom result class, `PipelineResult`, which contains the execution outcome and relevant error information. The information provided in the `PipelineResult` ensures that the user receives a message containing information about the rule execution.

In our opinion, an advantage of using the Microsoft Rule Engine is its capability to support a range of expressions for rule evaluation through its `ReSettings` object. This functionality allows for the integration of complex types within business rules, such as RegEx, date-time objects, and custom objects. Additionally, this feature facilitates the development of additional rules that surpass the initial limitations of lambda expressions. Such additional rules will be presented in the next section.

4.3.4 Business Rules

In this section, we will explore the creation of business rules tailored to satisfy our requirements, specifically for quality control of a CV database using our generic Rules Engine.

By implementing our generic Rules Engine, as mentioned in section 4.3.3, it is possible to supply input objects and apply expressions to evaluate conditions as true or false. This

functionality facilitates the development of rules that address diverse aspects of CV database quality control. A custom rules static class has been chosen to expand the Rules Engine's capabilities, enabling the use of expressions and checks beyond the constraints of lambda expressions. This strategy harnesses the potential of programming to define logical algorithms and execute complex operations, thus supporting more advanced and adaptable rules that can effectively manage complex scenarios and edge cases.

Ensuring proper object retrieval was important for rules involving dynamic types was an issue. As mentioned earlier, these objects' data structures can change during runtime. To address this issue, we designed our rules to account for the dynamic nature of the data types. We had to ensure that each object was accessed one layer at a time, with checks and validations to prevent object reference errors. This approach required attention to detail, as any small errors can lead to serious issues in the CV quality control process. By designing our rules with this in mind, we ensured that our custom rules methods became reliable.

As the rules implemented in our system were primarily designed to demonstrate the proof of concept to Bouvet, showcasing the potential and versatility of our Rules Engine, these rules were not particularly complex. Serving as a foundation to exhibit the engine's capabilities and flexibility, this approach allowed Bouvet to gain insight into our system's potential applications and benefits. The following section will present a detailed overview of the rules implemented.

CV must be updated regularly A business rule implemented checks the time since the CV was updated. The CV must be updated regularly to ensure its relevance and accuracy.

To implement the *CvMustBeUpdatedRegularly* rule, we create a custom class that inherits from our base interface `IBouvetRule`, which states that every rule must have an id along with a *BRule* property type.

`CvMustBeUpdatedRegularly` define a lambda expression that checks whether the last update of a given CV falls within an acceptable time frame (e.g., within the past three months). The rule is considered unsatisfactory if the last update is beyond three months and a relevant error message is generated. The implementation of `CvMustBeUpdatedRegularly` is shown in Code Sample 4.11.

```
1 public class CvMustBeUpdatedRegularly : IBouvetRule {
2     public IdField<Guid> Id { get; set; }
3     public BRule Rule { get; set; }
4     public CvMustBeUpdatedRegularly(int days) {
5       Id = FieldsFactory.IdField(Guid.NewGuid());
6       Rule = new BRule(FieldsFactory.StringField("CV must be updated regularly"),
```

```
        FieldsFactory.StringField("SuccessEvent-None"),
        FieldsFactory.StringField("The CV must be updated regularly. Please update your CV to
    ensure it contains the most recent information."),
        FieldsFactory.StringField(@"CustomRules.IsUpdatedWithinDays(input, " + days + ")"),
        FieldsFactory.StringField("AND"),
        FieldsFactory.BoolField(true),
        FieldsFactory.StringField("Rules Engine Team"));
    }
  }
```

Code Sample 4.11: Implementation of the CvMustBeUpdatedRegularly class.

A `CustomRules` class to validate the condition a "CV must be updated" condition was implemented as shown below in Code Sample 4.12:

```
public static class CustomRules {
    public static bool IsUpdatedWithinDays(dynamic input, int days) {
      var ownerUpdatedAt = input.owner_updated_at;
      var createdAt = input.created_at;
      if (ownerUpdatedAt == null && createdAt == null) {
        return true;
      }
      if (ownerUpdatedAt == null && createdAt < DateTime.Now.AddDays(-days)) {
        return false;
      }
}
```

Code Sample 4.12: Implementation of the IsUpdatedWithinDays method.

CV Must Contain Only Categorized Technologies

This custom method checks the CV for any uncategorized technologies an employee has added to their resume. It flags them as invalid, ensuring that only categorized technologies are displayed to potential customers. This rule was introduced as CV Partner only exposed the categorized technology, not the uncategorized one.

The custom method to validate the rule checks whether any uncategorized technologies in a CV have associated skills. It takes a dynamic input, iterates through the technologies enumerable, and checks if the *technology_skills* and *technology_uncategorized* properties are set. If a technology is marked as uncategorized and contains any skills, the method returns *false*, indicating that the rule is not satisfied. If no such cases are found, the method returns *true*, signifying that the rule is satisfied.

By displaying only categorized technologies, we provide Bouvet's customers with a reliable

and comprehensive overview of a candidate's technical expertise.

CV Must Have A Profile Picture

In our implementation, we established a requirement for CVs to feature a profile picture to enrich their presentation and personalization. By incorporating a check for a profile image, we sought to create a more visually appealing and memorable representation of candidates, enabling potential employers to form a better connection with them.

Our implementation of the rule involved an if statement to examine the existence of an *image* property within our CV object. This approach made us determine whether the necessary image property was present and whether the CV satisfied the profile picture requirement.

CV Must Have Key Qualifications

We also mandated that CVs must include key qualifications to ensure a presentation of employees' skills and expertise. This requirement aimed to provide potential clients with a clear and concise overview of the candidates' most relevant and valuable qualifications. By highlighting key qualifications, employees have the ability effectively demonstrate their suitability for specific roles and distinguish themselves from employees at other firms.

To achieve this objective, we implemented an if statement to verify the presence of the *key_qualifications* property within the CV data. By checking for the existence of the *key_qualifications* property, the rule can differentiate between CVs that meet the key qualifications requirement and those that do not, prompting users to update their CVs accordingly.

CV Must Have Key Qualifications Over a Given Length

In our implementation, we emphasized the importance of including key qualifications in CVs that comply with a specified length. This requirement offered potential employers an overview of a candidate's relevant skills and expertise, enabling them to make hiring decisions. By mandating that key qualifications surpass a certain length, we encouraged candidates to detail their qualifications, demonstrating their suitability for specific roles and setting themselves apart from the competition.

To achieve this objective, we incorporated an if statement to confirm the presence and length of the *key_qualifications* property within the CV data. This step ensured that each CV included a list of pertinent qualifications that met the specified length requirement, promoting

consistency across all submissions. By verifying the existence and length of the *key_qualifications* property, our system can distinguish between CVs that fulfilled the key qualifications length requirement and those that did not, prompting users to update their CVs as needed.

CV Must Have Work Experience

Bouvet emphasized the importance of including work experience in CVs, even for newly graduated candidates—this requirement aimed to offer potential employers an understanding of a candidate's practical skills and experience. By mandating work experience in CVs, we highlighted the significance of hands-on experience and professional growth, recognizing that newly graduated candidates also have opportunities to learn and develop within their current roles, such as those at Bouvet.

To achieve this objective, we incorporated an if statement to confirm the presence of the *work_experience* property within the CV data. By verifying the existence of the *work_experience* property, our system can distinguish between CVs that fulfilled the work experience requirement and those that did not.

Email Must Be Valid

To implement the *EmailMustBeValid* rule, we utilized Regular Expression (RegEx), a pattern-matching tool used in programming for text processing and validation tasks. The choice to use RegEx was driven by its ability to concisely and effectively define patterns, making it a tool for validating email addresses. Using RegEx can be difficult to write and maintain, especially for complex patterns, and can lead to hard-to-debug errors if not used correctly. Thus, exercising caution and ensuring proper testing when employing RegEx in validation tasks is essential, which was solved using unit testing on all implemented rules. This will be further examined in section 4.7.

In the context of our Rules Engine, we needed to incorporate the RegEx pattern into the custom rules function. The RegEx used to evaluate the validity of the email address is `^[^@\s]+@[^@\s]+.[^@\s]+$`.

The email validation RegEx pattern checks whether these criteria are satisfied:

1. Starts with one or more non-whitespace characters that are not "@" symbols.

2. Followed by an "@" symbol.

3. Followed by one or more non-whitespace characters that are not ”@” symbols.

4. Followed by a period ”.”.

5. Ends with one or more non-whitespace characters that are not ”@” symbols.

A company owned by Bouvet, called Olavstoppen, also uses CV Partner as their place to keep CVs. But Olavstoppen uses a different domain name for their emails. We needed to exclude the domain name *@bouvet.no* from the RegEx pattern. To ensure that the email addresses of employees from Olavstoppen and any potential future acquisitions are not mistakenly flagged as invalid by our validation process.

By incorporating this RegEx pattern into our Rules Engine, we have ensured that the email addresses in the CV are valid and adhere to the commonly accepted email address format.

Employee Name Must Be Valid To ensure the quality and consistency of employee data in our system, we have implemented a rule using RegEx to validate that an employee's name is in a valid format. The rule is implemented as a class called `EmployeeNameMustBeValid` which is part of our Rules Engine and used to evaluate employee names entered into the system.

The RegEx pattern used in the implementation checks whether the name meets the following criteria:

1. Starts with an uppercase letter.

2. Followed by one or more letters, including letters with diacritical marks (such as åøæÀ-ÒØ-òø-ýĀ-ž), hyphens, and apostrophes.

3. Followed by one or more spaces, hyphens, apostrophes, and letters (including diacritical marks).

4. Ends with one or more letters (including diacritical marks) and an optional period.

By incorporating this RegEx pattern into our Rules Engine, we ensure the employee name in the CV database is valid and adheres to the commonly accepted format.

Implementation Approach We have utilized various tools and techniques to implement the quality control rules for the CV database. These tools have been incorporated into a standardized template for the rule implementation, ensuring consistency and ease of maintenance for the rules.

In our opinion, the implementation approach for the quality control rules in the CV database promotes consistency and scalability by adding rules that adhere to the same design principles and utilize similar patterns. This ensures that the workflow responsible for CV database quality control can be expanded to accommodate new rules while maintaining uniformity in the rule implementation. The approach serves as a demonstration of the workflow of our generic Rules Engine. It highlights the potential for further enhancements by creating specific rules to address the quality control needs of various CVs.

4.4 Microservices

This section will explore and analyze our three microservices: Bouvet.Rule.Engine.Service, Bouvet.CV.Service, and Bouvet.Notification.Service. We will discuss their specific use cases, delve into their design principles, and examine their implementation strategies.

4.4.1 Bouvet.Rule.Engine.Service

Bouvet.Rule.Engine.Service is a microservice designed to host our Generic Rules Engine for users to run any workflow on any input. This service is built upon our architecture utilizing microservices, pub/sub, and CQRS. In addition to our architectural choices, the Bouvet.Rule.Engine.Service follows several design patterns and strategies, including the mediator design pattern, the Chain of Responsibility pattern, and a Hosted Service.

Challenge and Rationale In the development of the Bouvet.Rule.Engine.Service several challenges and considerations were addressed to meet the project's requirements. A primary challenge was designing a generic, flexible service supporting various workflows, rules, and input types to accommodate Bouvet's requirements and enable other projects to utilize the service if needed.

We understand that users today appreciate immediate, real-time feedback, particularly in data processing and analysis. Delays or latency in system responses can negatively impact user experience. Consequently, we believe it was imperative to design the rule engine service to handle multiple simultaneous requests without performance degradation. Moreover, overlook the performance metrics, such as latency in our application, when other users leverage it. A poorly performing application can affect user engagement and satisfaction, negatively

impacting overall system usage and potentially leading to undesirable outcomes.

Design In the design of the Bouvet.Rule.Engine.Service we adopted design patterns to ensure the service was using proven solutions to problems. We structured the application flow to receive requests from our endpoint and route them to the respective handler using the mediator pattern. Furthermore, we employed a Chain of Responsibility pattern for the handlers, culminating in adding the workflow to the queue of the hosted Rules Engine, which would be executed in a background service.

When designing the Bouvet.Rule.Engine.Service, we created a sequence diagram to visualize and better understand the interactions between our components involved in the process.

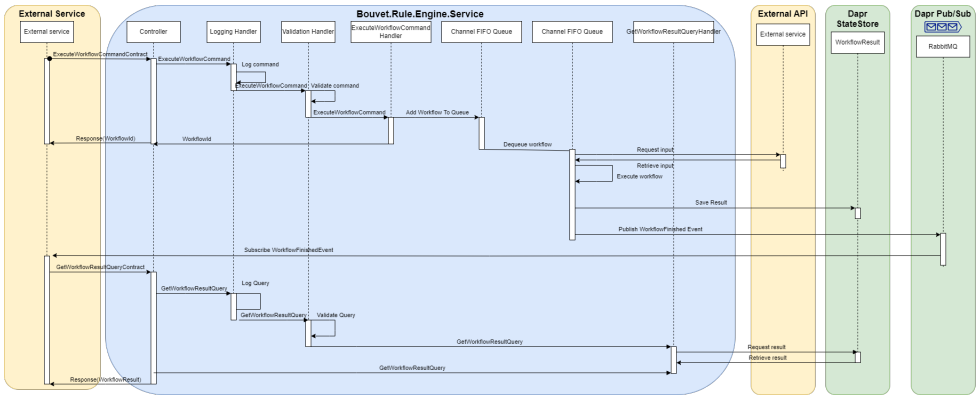

Figure 4.15: Our initial Sequence diagram for Bouvet.Rule.Engine.Service.

Figure 4.15 illustrates our initial effort to develop an approach where the input message identifies the data source, such as external APIs like *https://www.recipes.no/GetRecipes*. Our service would further need to fetch the input from an API or a data source based on the input message. This approach aimed to establish a flexible service that accommodates various inputs and addresses Bouvet's requirements.

However, the first approach had negatives leading us to reconsider the design. A drawback was the increased complexity associated with coordinating data retrieval from multiple external APIs. This added complexity would, in our opinion, lead to more challenging maintenance and debugging efforts. Additionally, potential points of failure were introduced as the service became more reliant on the reliability and availability of external APIs, which, in our view, would compromise the system's stability.

Another concern with the initial approach was the security implications, especially when han-

dling sensitive information. Since the service depends on multiple external sources, securing data access and transfer would be more challenging.

Upon evaluating these limitations and potential drawbacks, we decided to reevaluate our design approach and accept input directly from the request. By receiving input directly from the request, we addressed the aforementioned concerns.

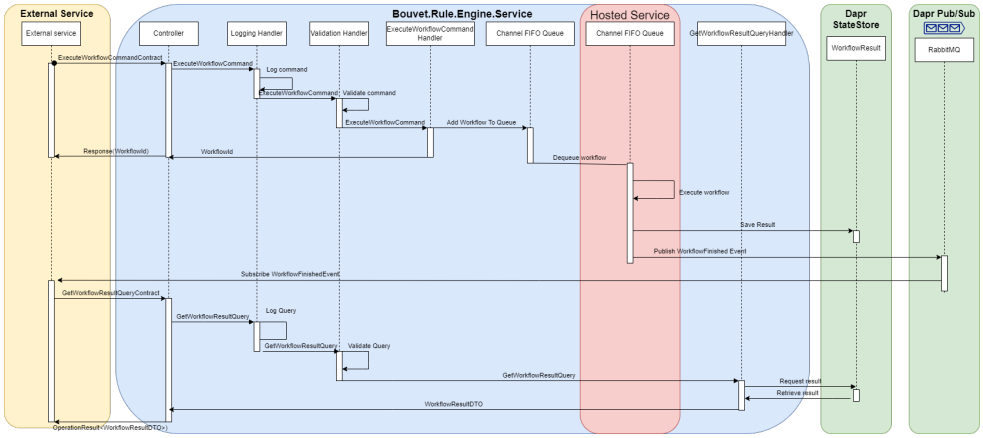

Figure 4.16: Sequence diagram for our final design choice in Bouvet.Rule.Engine.Service

The revised approach is shown in Figure 4.16 offers, in our opinion, a balance between flexibility and simplicity for the service to accommodate diverse inputs without the added complexity of coordinating with external APIs. Furthermore, we believe that Bouvet now integrates the service into its existing systems more easily, as the direct input method will optimize communication and data processing.

Within this context, the mediator pattern facilitated the implementation of CQRS architecture. Each request to the Rules Engine was encapsulated as a command or query and processed using the mediator Pattern. This reduced the direct dependencies among components, which we consider a cleaner and more modular codebase.

The Chain of Responsibility pattern was also employed in our system, dividing a request's processing into a series of handlers, each responsible for a specific aspect of the command or query, as illustrated in Figure 4.17. In our judgment, this design pattern promotes the separation of concerns within the codebase and simplifies adding or removing handlers as needed.

While the Chain of Responsibility pattern can potentially increase complexity with an ex-

cessively long chain or surplus handlers, we suggest that a concise and manageable chain, as illustrated in Figure 4.17, is advantageous. Our generic pipeline behaviours handled most of the pre and post-processing tasks, thereby avoiding undue elongation of our chains.

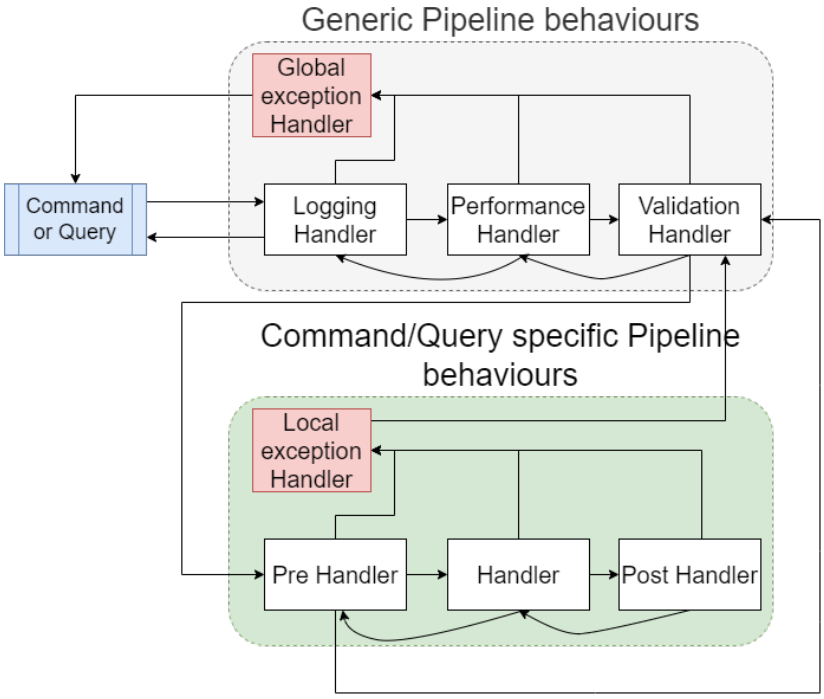

Figure 4.17: Separating the command or query into generic and specific handlers using the Chain of Responsibility approach.

The handler for the execution of the workflow was designed to send the response immediately when the workflow was added to our queue, not blocking for the workflow to complete execution. As the service does not know how long the execution for each workflow will take, delaying the response would impact the service's performance. Instead, the response is a workflow ID that can be used later to retrieve the result of the previously executed workflow.

To retrieve the result, the service subscribes to a *workflow finished* event triggered upon completing the workflow execution. We believe this approach enables non-blocking communication and enhances the service's scalability, eliminating the need to maintain an open connection while awaiting the workflow's completion. The service handles multiple requests simultaneously, and the response can be retrieved later when the workflow is complete.

To run our Rules Engine, we implemented a background-hosted service strategy to execute workflows in a separate thread. This enabled the service to process multiple requests simul-

taneously without impacting the Rules Engine's performance. Communication between the hosted service and the rest of the application was facilitated through a channel queue, enabling other threads to communicate with the hosted service and add workflows to the queue.

The background-hosted service provided several benefits. We believe asynchronous request processing improved scalability and performance by executing the Rules Engine in a separate thread.

We believe our approach, as depicted in figure 4.16, was an optimized solution, balancing flexibility and simplicity. Its modular and decoupled design, employing a mediator and Chain of Responsibility pattern, ensured separation of concerns, and the non-blocking communication and background-hosted service strategy reduced user latency.

Implementation

The implementation starts as the controller receives an *ExecuteWorkflow* request and uses the MediatR library to send a command of type *ExecuteWorkflowCommand*. The controller receives the request, and the body contains the *ExecuteWorkflowCommandContract*. This contract contains all the necessary information about the workflow that needs to be executed and the input that the workflow is supposed to execute. Once the controller receives the request, it creates a command using the `ExecuteWorkflowCommand`. This class contains a *correlationId*, our *BWorkflow*, and *BInputs* class. Lastly, the MediatR library sends this command to the appropriate handlers.

When a command is sent to the MediatR pipeline, it passes through a series of handlers, each responsible for a specific task. The pipeline behaviour adds functionality to the pipeline, such as logging, validation, exception, and performance handlers, without modifying the individual handlers.

Initially, the command is passed through the logging handler, logging the details of the command, such as the correlation ID and the time it was received. Once the logging handler completes its task, it passes the command to the validation handler. The validation handler checks the validity of the command, ensuring that all necessary fields are present and contain valid data.

Custom validators for each command were implemented, which used the previously mentioned validators created in the Rules Engine service to ensure that the command contained valid data. If the command is invalid, an exception handler picks it up and returns an error response

to the client without executing the command. The command passes it to the pipeline's next handler if valid. This prevents us from using unnecessary processing power if the command is invalid. To validate any command/query with any validators, we implemented a generic validation handler as shown in Code Sample 4.13.

```
1  public class ValidationBehaviour<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse
       > where TRequest : IRequest<TResponse> where TResponse : IOperationResult, new() {
2      public async Task<TResponse> Handle(TRequest request, RequestHandlerDelegate<TResponse>
       next, CancellationToken cancellationToken) {
3        if (_validators.Any()) {
4          var validationResults = await Task.WhenAll(_validators.Select(v => v.Validate
       Async(context, cancellationToken)));
5          var errors = validationResults.SelectMany(result => result.Errors).Where(x => x !=
       null).ToList();
6          if (errors.Count != 0) {
7            _logger.LogError("Error while validating - {RequestType} - Command: {@Command} -
       Errors: {@ValidationErrors}", typeof(TRequest).Name, request, errors);
8            throw new ValidationException(errors);
9          };
10       }
11       return await next();
12     }
13   }
```

Code Sample 4.13: MediatR pipeline behaviour.

Line 11 in Code Sample 4.13 showcase the code segment responsible for forwarding the request to the subsequent handler. This design is followed across all pipeline behaviours. Through this implementation, we believe it aligns with the principles of the Chain of Responsibility pattern.

We needed a way to communicate with a background service responsible for processing data in the background. We decided to use a channel queue to accomplish this. The queue was implemented as a simple First In First Out (FIFO) structure. Given our requirements, the workflows need to be executed in their received order without any prioritization. The FIFO structure queue catered to this requirement.

A class must hold the necessary data to pass information to the queue and execute workflows. In our system, the *BackgroundModel* class has been implemented for this purpose. This class contains relevant information about the workflow, including the workflow itself and the input on which it will be executed. We pass the necessary information to the queue using the *Background* model class.

The *QueuedHostedService* is an always-running background service that hosts the generic Rules Engine and manages the execution of workflows within a Rules Engine system based on the queue. Implementing the QueuedHostedService ensures continuous operation by listening to a task queue for incoming items. When an item is added to the queue, the service dequeues a *Background model* item. When the hosted service dequeues the item, the workflow is executed. The result of the workflow is then saved to the services state store using Dapr, and an event is published to the event subscribers that the workflow is finished.

The Bouvet.Rule.Engine.Service incorporates an API endpoint named */GetWorkflowResult* to fetch the result of a completed workflow. This endpoint utilizes the same Chain of Responsibility design pattern as the workflow execution process. The service accesses the state store to retrieve the result in the corresponding handler and returns a DTO containing the workflow result.

Infrastructure

To accommodate the service's stateful design, the system employs a Dapr state store. In the context of a microservices architecture, each service maintains its state store. We believe that this approach ensures modularity by keeping the services independent from each other, allowing them to evolve without affecting the overall system. Separation of state stores also contributes to enhanced scalability, as each service can scale independently based on specific needs and demands. The state store for Bouvet.Rule.Engine.Service is set up with the document database MongoDB.

The Bouvet.Rule.Engine.Service needs to store the workflow result as a key-value pair with the workflow execution ID as the key and the workflow result *PipelineResult* as the value. Considering the role of the database, we believe that a relational database is not necessary for storing the workflows, given the lack of relations in our models, and that a document database is a better fit for this purpose.

There are some drawbacks to document-based databases we considered. One of the concerns with MongoDB is its lack of transactions, which can lead to data inconsistencies if multiple users are modifying the same document simultaneously. To solve this issue, Dapr has built-in support for distributed locks(Dapr, n.d.-b). However, this feature is currently in the alpha version. Using alpha versions in software development for security-critical systems is not recommended, as we believe these versions are unstable and can be prone to errors and vulnerabilities.

The decision to use the built-in Dapr state store rather than a direct Mongo database for Bouvet.Rule.Engine.Service was driven by scalability and performance considerations. Dapr state store provides an abstraction layer for storing and managing the state of the service(Microsoft, 2022b). This enables us to implement the business logic rather than the details. Dapr provides a uniform API for accessing the state store, regardless of the underlying storage mechanism. This implementation is displayed in Code Sample 4.14.

```
#"To save a key-value pair in the Dapr state store"
await _daprClient.SaveStateAsync("StatestoreName", "key", "value");

#"To retrieve the value from the Dapr state store"
var value = await _daprClient.GetStateAsync<TypeOfObject>("StatestoreName", "key");
```

Code Sample 4.14: Dapr state store implementation

This entails that Bouvet.Rule.Engine.Service can switch to a different storage provider, such as Redis or Azure Cosmos DB, without changing any of its code, making it, in our view, easier to adapt to changing requirements or integrate with different environments.

We updated the Docker Compose file to include a MongoDB container image to facilitate this. The Dapr configuration file was also created to specify the necessary metadata for connecting to the MongoDB database, including the database name, collection name, and authentication information.

Considering Bouvet's requirements, we consider MongoDB an optimal choice for managing the state store for Bouvet.Rule.Engine.Service. This choice fulfils the project's current requirements and offers support for future requirements in switching to a different database can be accomplished without impacting the existing codebase.

Incorporating design principles and infrastructure planning, we implemented Bouvet.Rule.Engine.Service to meet Bouvet's requirements. We believe our design's flexibility enabled us to create a generic Rules Engine that is adaptable to various external applications, precisely caters to our specific needs, and supports high demand without compromising on our service's performance or user experience.

4.4.2 Bouvet.CV.Service

Bouvet.CV.Service is the service in our microservice architecture responsible for persisting CVs fetched from CV Partner and rules. The service also supports CRUD operations. Bouvet.CV.Service follows the same architectural and design principles as Bouvet.Rule.Engine.Service includes microservices, Pub/Sub, CQRS, mediator design pattern, Chain of Re-

sponsibility pattern, and a background-hosted service.

Challenge and Rationale

The implementation of Bouvet.CV.Service is driven by the need for a service to integrate with CV Partner to retrieve their stored CVs. The service needed to fetch CVs without overwhelming their API and simultaneously meet Bouvet's requirements. The service was also the driver for the Rules Engine workflow, as it was responsible for sending requests to the Bouvet.Rule.Engine.Service when CVs were updated, or new CVs were stored on CV Partner.

There were several challenges in creating a solution that met our requirements for a scalable solution to handle the various scenarios. One challenge was ensuring that the Rules Engine was only executed when necessary. Bouvet wanted the Rules Engine to run when a new employee was hired or a CV was updated. In addressing this challenge, a system that can detect CV changes and trigger the Rules Engine accordingly was needed.

To address the challenge of retrieving the necessary CVs without stressing the CV Partner API, we would need a scheduled retrieval system to ensure we stayed within a reasonable amount of requests per second. In addition, Bouvet wanted a timeline of the CVs; hence we needed to save the new updated CVs as add-ons rather than replacing the old ones.

Further, we must be able to persist the workflow containing the business rules and enable CRUD operations on the rules, ensuring that the workflow is updated and maintained as the business rules evolve.

Bouvet.CV.Service is the initiator in our microservices architecture, responsible for retrieving and persisting the multiple CVs and driving the execution of Bouvet.Rule.Engine.Service. As a result, a tool that allows Bouvet to automate its processes of running business rules on its CV database is needed.

Design

Handling the daily retrieval of thousands of CVs presented challenges, particularly avoiding the overloading of the CV Partner API, which in our opinion, could lead to rate-limiting and other API-related issues. To mitigate this, the scheduler was designed to stagger requests with a built-in delay of 100ms, ensuring the API was not inundated with simultaneous requests. Moreover, the scheduler was set up to fetch CVs on a configured frequency. The default setting was once daily, given that most CVs are not frequently updated in practice.

In our system, detecting changes in a CV ensures that the service has up-to-date information and triggers updates as needed. To accomplish this, we introduced a delta variable to compare the current state of a CV with its previous state. However, to ensure accuracy and avoid bugs, we designed the system to detect changes in a string rather than an object.

In C#, the equality operator "==" compares reference equality for objects. If two objects reside in different memory locations, using the equality operator to compare them would yield a false result. (Vlad, 2023)

To accomplish our design goal of a scalability solution, the cv scheduler was integrated within a hosted service, enabling it to operate with minimal latency while preserving the efficiency of the service.

In the present work, the CRUD operations adhere to the same CQRS design principles and use the same design patterns as those employed in Bouvet.Rule.Engine.Service, maintaining a consistent flow across the system. Given the similarities in the implementation, a detailed discussion of the CRUD operations within this section is unnecessary. The focus will remain on other aspects of the system. Understanding the CQRS design and flow of commands and queries can be extrapolated from the descriptions provided for Bouvet.Rule.Engine.Service.

Implementation

We implemented the CV scheduler in a background service. We utilized the HTTP client obtained from our mitmproxy2client utility to fetch CVs from the CV Partner API explained in section 4.3.1.

The scheduler was implemented to start immediately upon service start and await a given period before execution started again. This was done by implementing a function called `TimeToWait`. This method calculates the time difference between the current time to the time set in the configuration. The configuration uses the Options Pattern.

Given our assessment that Bouvet employees are less likely to use the CV Partner license at night, we scheduled the service to operate at off-work hours, thereby minimizing potential disruption. This combination of the Options pattern and scheduling ensures, in our opinion, that the service remains flexible and efficient while accommodating the requirements of both Bouvet and CV Partner.

To ensure that Bouvet.CV.Service does not send unnecessary requests to run the Generic

Rules Engine every time we fetch CVs from CV Partner, a method to check if a CV has changed before processing it was implemented. If the CV exists in the state store, methods compared the current state of the CV with its previous state to see if it had been updated. This implementation minimized the processing power needed for Bouvet.Rule.Engine.Service and, in our view, led to a more efficient service that schedules CVs without overburdening Bouvet.Rule.Engine.Service.

In the implementation of Bouvet.CV.Service, we use Dapr service invocation to communicate with Bouvet.Rule.Engine.Service, as opposed to Dapr Pub/Sub. This decision was driven by the efficiency of *service invocation*. Service invocation enables a direct request to Bouvet.Rule.Engine.Service while transmitting all necessary information to execute the Rules Engine, resulting in only one request to execute a workflow on an input. Conversely, we believe Dapr Pub/Sub would have entailed a more complex communication process. This process could lead to delays or message delivery issues, as Bouvet.Rule.Engine.Service would then need to collect all required information and return a request to Bouvet.CV.Service, which would lead to twice the amount of communication.

Another factor in not opting for Dapr Pub/Sub for this specific request is the need for the requester to receive the workflow ID in the response, as events only send a notification without receiving a response. The returned workflow ID allows the requester to track the CV processing progress and retrieve the results upon completion.

Infrastructure

An aspect of *Bouvet.CV.Service* is the use of a combination of relational and document databases. While the document database is used to store CVs, the three relational database tables are used to store workflows, rules, and their respective rule data fields.

The database to store CVs uses MongoDB with Dapr state store with the same design as Bouvet.Rule.Engine.Service. The email would be the key, and the value is a list of the CVs corresponding to this email.

To integrate the Dapr state store and MongoDB, a new Dapr state store object was created that connected to another MongoDB database, using the same MongoDB host as the other components in the system. We used the same MongoDB host for both services in our pilot, which benefitted our application due to the improved resource utilization and reduced overhead.

In our view, a benefit of using the same approach for Dapr State Store for this service and Bouvet.Rule.Engine.Service is the ability to reuse some configuration logic. This approach saves time and effort in the development process, as we do not need to learn a new way of setting up the database or configure the system from scratch.

We used a relational database for workflows, rules, and rule data fields due to the relations between the data models. Specifically, workflows can have multiple rules, each with its rule data field. By using a relational database, Bouvet.CV.Service can ensure the integrity and create relationships between entities.

Regarding the specific design of the relational database, the workflows and rules entities follow a one-to-many relationship, where one workflow can have multiple rules. In contrast, the rules have a one-to-one relationship with their respective rule data fields. The entity relational diagram is depicted in Figure 4.18:

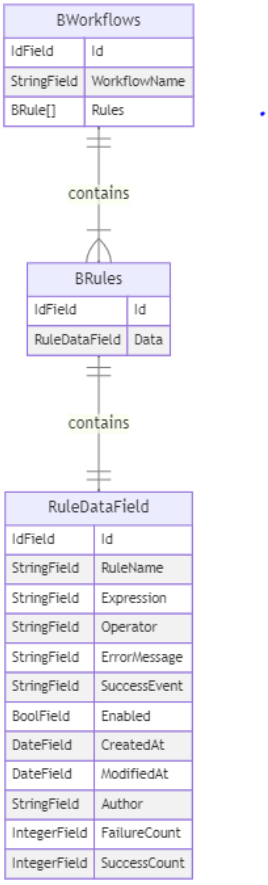

Figure 4.18: Bouvet.CV.Service entity relation diagram.

Our relational database was designed and implemented using a normalisation process to minimize data redundancy, improve data integrity, and optimize database performance.

Normalization is achieved through a series of normal forms, each representing a set of rules or conditions that a database schema must meet to ensure a specific level of data consistency and organization(GeeksforGeeks, n.d.). We will look at First Normal Form, Second Normal Form, Third Normal Form, and Boyce-Codd Normal Form and explain how our database meets each requirement.

1. First Normal Form:

    (a) **Requirement**: Each entity attribute holds atomic values, with no repeating groups(GeeksforGeeks, n.d.).

    (b) Our database meets this requirement as there are no repeating groups or arrays in the attributes. BWorkflows, BRules, and RuleDataField entities ensure each attribute stores a single value.

2. Second Normal Form:

    (a) **Requirement**: The primary key determines all the other attributes in the table, and therefore all non-prime attributes (attributes not part of the primary key) are fully functionally dependent on the primary key.(GeeksforGeeks, n.d.).

    (b) Our database meets this requirement since the entity has a primary key, and the property can only be accessed using this key. This ensures that all non-prime attributes in the entities are fully functionally dependent on the primary keys.

3. Third Normal Form:

    (a) **Requirement**: There should be no transitive dependencies between non-prime attributes(GeeksforGeeks, n.d.).

    (b) Our database meets this requirement since the Rule entity has a one-to-one relationship with the Data Field entity, with a foreign key in the Rule entity. This indicates no transitive dependencies between non-prime attributes, as each Rule directly relates to its corresponding data field.

4. Boyce-Codd Normal Form:

   (a) **Requirement**: For every non-trivial functional dependency X $\rightarrow$ Y, X must be a superkey(GeeksforGeeks, n.d.).

   (b) Our database meets this requirement as the primary keys in each table (Id in BWorkflows, Id in BRules, and Id in RuleDataField) are the determinants for all other attributes in their respective tables, making them superkeys. As a result, all functional dependencies have a superkey as their determinant.

EF is our choice to handle the data processing in our .NET microservice. We chose EF implementation for our project for several reasons.

First, our project primarily deals with relational databases and demands flexibility in handling data. Using EF, we can leverage object-oriented concepts instead of SQL queries to work with data(Microsoft, 2022c).

Second, we chose to use EF as, in our opinion, it improved our productivity by reducing the boilerplate code required to perform common data processing tasks.

Third, EF seemed fit for our project because it provides a flexible and extensible data processing framework. With EF, we can use Language-Integrated Query (LINQ) to write queries(Microsoft, 2021b) to express data processing logic clearly and concisely, as shown below in code sample 4.15:

```
var rule = await _db.Rules.Where(x => x.Id == query.ruleId.Value).
    FirstAsync(cancellationToken);
```

Code Sample 4.15: Implementation of how LINQ is used to manipulate data in the relational database

Fourth, we chose EF because it offers a flexible choice for our project if Bouvet's requirements change over time, without needing to change any code within our application, but rather a connection string. In addition to the aforementioned benefits, EF also serves as a security design choice by mitigating the risk of SQL injection attacks(Microsoft, 2021c). This aspect will be explored in a section 4.6.2.

Using a connection string to communicate with our database decouples the service from the database implementation. This means the service can be switched to a different database implementation, such as MySQL or PostgreSQL, without affecting the service code.

Secondly, a connection string can be configured using environment variables to deploy the service in different environments, such as development, staging, or production. In our opinion, this helps improve the maintainability and portability of the system, as it enables us to configure and deploy the service across different environments.

The consideration of both a relational and document database in Bouvet.CV.Service's design highlights that a one-size-fits-all approach to data management is not always effective, as different data types may necessitate distinct database solutions.

Implementation Entity Frameworks

The service utilized a Microsoft SQL Server. Following our microservice architecture and being deployed using Docker, we created a new container to host the SQL Server database.

We implemented our service to use an SQLite database in development which is a lightweight database. In Staging and production, our connection string switches to our SQL server. Our staging connection strings were implemented as displayed in code sample 4.16 by defining the server, database, user id, and password.

```
Server=sqldata;Database=WorkflowDb;User Id=NotOurActualUserId;Password=NotOurActualPassword;
    TrustServerCertificate=true
```

Code Sample 4.16: Database connection string.

A feature of EF is the ability to define complex types. In our system, we defined owned entity types to be able to save our custom types created in *Fields* library. This enables us to avoid our database's regular string and int types. This was implemented in the context file as shown in code sample 4.17.

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<BRule>(entity => {
      entity.HasKey(x => x.Id);
      entity.HasOne(x => x.Data);
    });
    modelBuilder.Entity<BWorkflow>(entity => {
      entity.HasKey(x => x.Id);
      entity.OwnsOne(x => x.WorkflowName);
    });
    modelBuilder.Entity<RuleDataField>(entity => {
      entity.HasKey(x => x.Id);
      entity.OwnsOne(x => x.RuleName);
      entity.OwnsOne(x => x.Expression);
      entity.OwnsOne(x => x.Operator);
```

```
15      entity.OwnsOne(x => x.ErrorMessage);
16      entity.OwnsOne(x => x.SuccessEvent);
17      entity.OwnsOne(x => x.Enabled);
18      entity.OwnsOne(p => p.Author);
19      entity.OwnsOne(p => p.SuccessCount);
20      entity.OwnsOne(p => p.CreatedAt);
21      entity.OwnsOne(p => p.ModifiedAt);
22      entity.OwnsOne(p => p.FailureCount);
23    });
24    base.OnModelCreating(modelBuilder);
25  }
```

Code Sample 4.17: Owned entities implementation.

By defining owned entity types in EF, we avoid needing separate tables for each type that would reference the object value with a foreign key. This approach enabled us to utilize and gain value from our custom types.

EF integrates with the database provider in our system by configuring the *ConfigureServices* method in the *program.cs* file. This method registers the *DbContext*, which embodies the database session and serves as the primary interface for interacting with the data store. The connection string, sourced from a secure secret store, is utilized by EF to establish a connection to the database. Moreover, in our opinion, registering the *DbContext* with the dependency injection system facilitates the integration of data access components across the application. In our opinion, a benefit of this approach is the availability of the *DbContext* within various application components, simplifying its usage. This methodology optimizes the development process by injecting the DbContext into the code, enabling interaction with the database.

We find using EF to be a beneficial decision for our thesis. It provided a solution capable of managing complex data processing workflows and facilitated CRUD operations on business rules. Its integration into our application architecture made our development process more effective.

4.4.3 Bouvet.Notification.Service

Bouvet.Notification.Service facilitates the Rules Engine outcomes for violations and alerts employees if the CVs do not meet specified quality control rules.

The service is not directly responsible for the notification process; instead, it dispatches an event to an external event bus, which manages the actual notification procedure to the business medium. Bouvet uses different communication mediums; the notification could use Slack,

Email or others. Another team in Bouvet is working on an event bus which our notification service will use as an external service.

Challenge and Rationale

When an employee's CV violates specific business rules, such as improper key qualifications, the Bouvet.Notification.Service is designed to issue a notification to the employee. This notification is intended to pinpoint the violation and provide guidance on rectifying the issue.

The principal challenge in this context was to establish a connection that would implement the notification logic and concurrently ensure that CV execution adheres to predetermined criteria. A further complication was the requirement for an external event bus integration.

Another challenge was designing the notification service to deliver timely and relevant CV violation notices to employees. The aim was to avoid inundating employees with excessive or irrelevant notifications while ensuring they receive necessary feedback.

Design

The design of the Bouvet.Notification.Service revolves around its ability to listen to events from the RabbitMQ event bus and be notified when a workflow has been run on a CV. The service is designed to manually check whether or not the CV is satisfied based on the workflow results. While the Rules Engine sends out an event when a workflow is finished, it does not provide the actual result. If one starts sending complex data through an event, the data no longer represents a description of change or notification. Hence, Bouvet.Notification.Service would be notified that a workflow has finished running and further fetch the workflow result from the Bouvet.Rule.Engine.

When the service has the result, the Bouvet.Notification.Service check to determine whether or not to notify the employee of a violation or if a rule is satisfied. We choose to subscribe to all workflow execution events. This decision enabled future enhancements, such as gathering metrics and statistics or performing additional logical functions based on failures and successes. By incorporating this feature, we believe it fosters greater adaptability and expandability, ensuring that our system can grow and evolve in response to changing requirements.

The design of the Bouvet.Notification.Service also follows the already mentioned structural approaches as the other services, such as CQRS architecture and mediator pattern. In this service, we use MediatR notification *event* to facilitate publish-subscribe Pattern, rather than

MediatR *requests*. This adheres to the publish-subscribe Pattern. This design decision was made based on the specific requirements of the service and the nature of the events being handled, as events are received when a workflow has finished running.

In the case of Bouvet.Notification.Service, we are handling events that do not require a response. Instead, notifications are broadcast to multiple handlers, making using MediatR notification events more appropriate.

The use of MediatR notification events provides greater flexibility in the implementation of the service, as it allows for multiple handlers to receive and process the same event. We believe it is particularly useful where multiple actions may need to be taken based on the same event as more processing and logic to notify an event might change or be added.

Furthermore, the design of the Bouvet.Notification.Service includes a configuration that enables the service to communicate with an external event bus, which would result in minimizing the code Bouvet would need to do when this external service is in production.

Implementation

The implementation of subscribing to a Dapr event in the Bouvet.Notification.Service involved using a .NET controller with the specified topic. This controller was responsible for receiving notifications from the Dapr event bus and handling them, shown in code sample 4.18.

```
[HttpPost(DAPR_PUBSUB_WORKFLOWFINISHEDTOPIC)]
[Topic(DAPR_PUBSUB_NAME, DAPR_PUBSUB_WORKFLOWFINISHEDTOPIC)]
public async Task WorkflowFinishedEvent(WorkflowFinishedEvent @event) {
  _logger.LogCritical("Event: {EventName} received", typeof(WorkflowFinishedEvent).Name);
  var validationResult = _validator.Validate(@event);
  if (!validationResult.IsValid) {
    _logger.LogError("Event: {EventName} is not valid with contents: {Contents}", typeof
    (WorkflowFinishedEvent).Name, @event);
    return;
  }
  await _mediator.Publish(@event);
}
```

Code Sample 4.18: Dapr events mplementation

This code snippet defines two constants in the topic of our controller, *DAPR_PUBSUB_NAME* and *DAPR_PUBSUB_WORKFLOWFINISHEDTOPIC*, which is used to specify the event bus name and event the controller subscribes to. The controller was designed with an *HttpPost* attribute. This enabled the controller to accept notifications transmitted to that specific

topic via the Dapr event bus.

When an event was received, the controller logged the name for debugging purposes. It then instantiated a validator to ensure the event was valid and met the required criteria. If the event was invalid, the controller logged an error message and returned without further processing. If the event was valid, the controller used MediatR to publish the event to any registered event handlers. This allowed Bouvet.Notification.Service to receive notifications from the Dapr event bus and handle them.

Once the events are published via the MediatR, our event handler intercepts them. The handler sends a request to the Bouvet.Rule.Engine.Service for the workflow result. Subsequently, the handler assesses the execution outcome and publishes an event in case of a false result. Conversely, if the outcome proved true, no action was taken.

We publish events through our event bus, as the external event bus project remains in the early stages of development. In the future, we will need to discuss the event contract with the other projects team and acquire information on connecting to their event bus. Once completed, the transition will involve a simple configuration change in our Dapr setup to switch the connection to the external event bus without any code modifications. Our current implementation is shown in code sample 4.19. This approach ensures minimal code changes by Bouvet when the external service becomes available.

```
1 await _daprClient.PublishEventAsync(DAPR_PUBSUB_NAME, topic, data, cancellationToken);
```

Code Sample 4.19: Publishing a notification event to Dapr Event Bus

4.5 Logging and Metrics

In this section, we will explore the implementation of logging and metrics within our system. A logging and metrics system is important for monitoring system performance, pinpointing bottlenecks, ensuring cohesive operation, and receiving feedback on the necessity of the application. Furthermore, using metrics provide feedback to Bouvet regarding the overall quality of the CVs and generates statistics on application usage. By incorporating logging and metrics components into our system, we aim to enhance maintainability, consolidate debugging, and offer essential insights for future improvements. This section will outline the specific logging and metrics techniques employed and the rationale behind our choices to demonstrate how they value our system.

Challenge and rationale

The driving force behind logging was our intent to address one of our major challenges - the complexity of debugging in a microservice architecture. While providing numerous benefits, our microservice application was difficult to debug due to its distributed nature. We aimed to achieve an equilibrium between capturing adequate data for meaningful analysis and debugging and preventing excessive logging, which can negatively affect the system's performance.

We implemented both distributed tracing and service logging, which we believe will provide monitoring and troubleshooting. Distributed tracing facilitates tracking requests across multiple services, while application logging captures specific events and data points within each service. By amalgamating these approaches, we can efficiently navigate our microservices system and resolve bugs.

4.5.1 Distributed Tracing

Our distributed system employed *Zipkin* for distributed tracing, leveraging its capabilities to enhance the monitoring and troubleshooting processes. As we utilized Dapr with Docker Compose, we integrated Zipkin as a container and configured Dapr to transmit tracing logs to the Zipkin endpoint. This was implemented by adding the following configuration into our Dapr configuration file shown in Code Sample 4.20:

```
tracing:
    samplingRate: "1"
    zipkin:
      endpointAddress: "http://zipkin:9411/api/v2/spans"
```

Code Sample 4.20: Dapr configuration

This setup enabled Zipkin to process the logs and present an interface for visualizing individual requests and the inter-dependencies among various components. We query the traces to get a full overview of all the requests between each service. The overview of our Zipkin interface is displayed in figure 4.19.

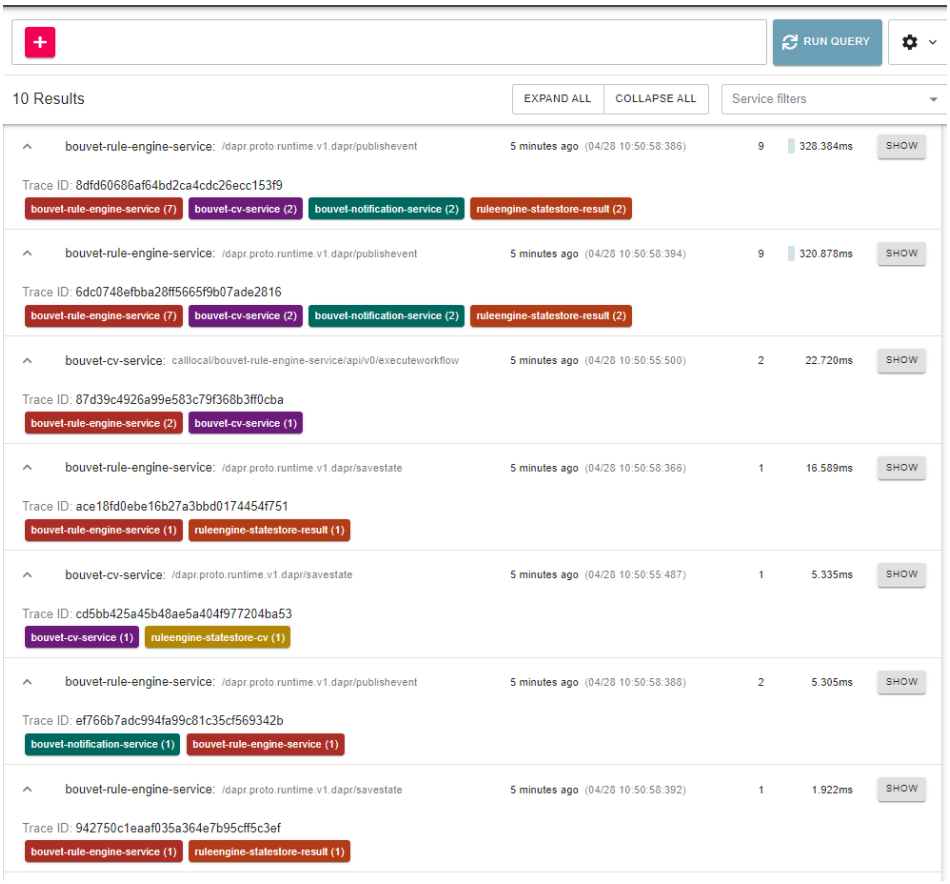

Figure 4.19: Zipkin trace overview.

Using *Zipkin*, we obtain a view of the dependency graph across various services. Depicted in Figure 4.20, we observe how the services interact with each other and Dapr-related components. On the left side of the graph are the Rules.Engine.Service, while its state store, the *WorkflowResult*, is located in the top right corner. Below the WorkflowResult node, the Bouvet.Notification.Service is displayed, illustrating the communication between these services. Further down, the Bouvet.CV.Service is shown, which depends on both the Rules.Engine.Service and its state store, where CVs are stored.

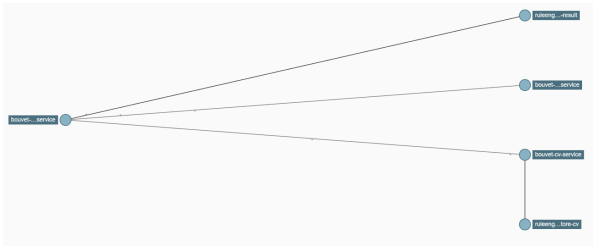

Figure 4.20: Overview of Zipkin dependency graph.

4.5.2 Application Logging

In our system, *Seq* was utilized for application logging, setting up a new container, and configuring it to receive logging messages from our services, similar to our approach with Zipkin. Using Seq, we believe we gained better visibility and control over our logging, enhancing our ability to monitor and troubleshoot the system.

Our services are configured to send logs to the Seq endpoint. To achieve this, we created an extension method *AddCustomSerilog* for the *WebApplicationBuilder* class, which takes *applicationName* as a parameter. We first retrieved the Seq server URL from the application configuration within this method. We then instantiated a new Serilog logger, configuring it to read settings from the application configuration, write logs to the console and the Seq server, and enrich logs with the application name as a custom property. Finally, we set this logger as the default logger for our application, ensuring that all logs generated by our application were sent to the Seq endpoint.

The Seq dashboard, as shown in Figure 4.21, portrays part of the logs, offering insights into the system's workings.

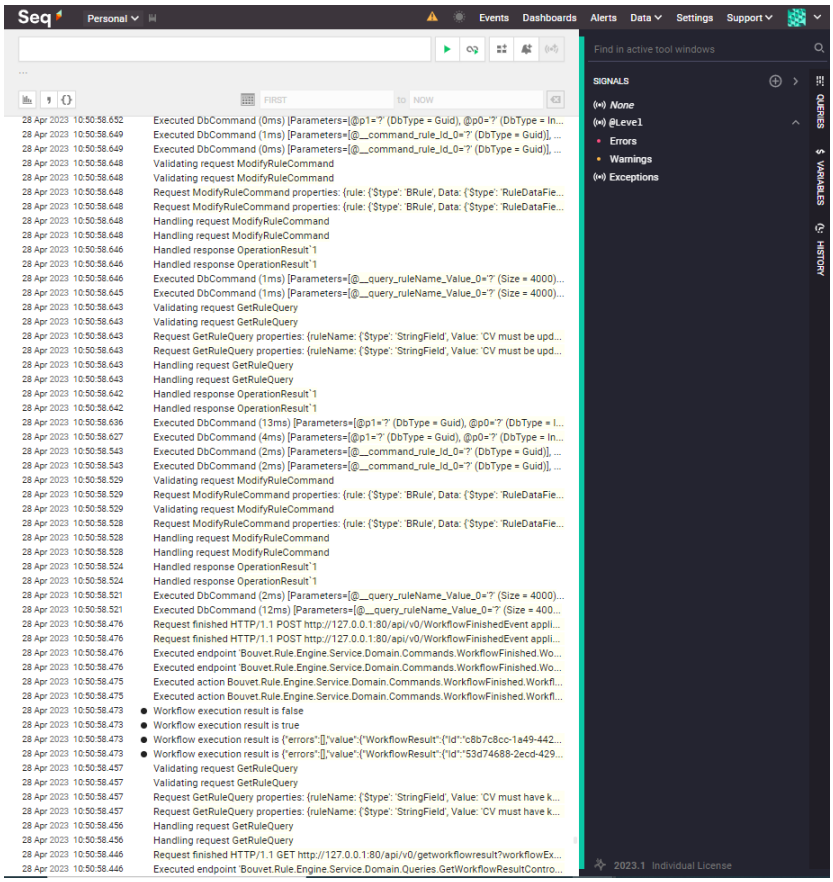

Figure 4.21: SEQ Dashboard.

4.5.3 Metrics-based Analytics

Beyond just logging messages, we recognized the need for a metrics system to obtain relevant feedback from our system. Implementing custom metrics was needed to assess the quality of Bouvet's CVs, and understand whether the pilot required further improvements. To address this, we combined Prometheus and Grafana, visualizing various business metrics.

By integrating Dapr with the monitoring system Prometheus, we capitalize on the relationship between these tools, allowing Prometheus to scrape and save metrics exposed by our service at a specific endpoint.

We then connected Grafana to Prometheus, processed the collected data and created a user-friendly dashboard that displays business information through various queries. This information encompasses performance indicators such as the number of satisfied versus unsatisfied CVs, the most frequently violated rules, general statistics about our solution, and other met-

rics indicative of our application's health.

We implemented custom metrics operations unique to our application to tailor our monitoring and reporting system to our business requirements. We believe these custom metrics enhance our understanding of our solution's performance based on the insights we obtain from the dashboard. The implementation for gathering custom metrics is demonstrated in Code Sample 4.21.

```
public static class CVMetrics {
    public static readonly Counter CVUpdatedCounter = Metrics.CreateCounter("cv_updated_total
    ", "Total number of times CVs has been updated");
    public static readonly Counter CVFetchCounter = Metrics.CreateCounter("cv_fetch_total", "
    Total number of times CVs has been fetched from CV partner");
    public static readonly Counter RulesExecution = Metrics.CreateCounter(
    "rule_runs",
    "Counts the number of successful runs of the rules",
    new CounterConfiguration {
      LabelNames = new[] { "rule_name", "status" }
    });
}
```

Code Sample 4.21: Custom metrics implementation.

To generate a generic metric based on the rules executed by our Generic Rules Engine, we utilized *CVMetrics.RulesExecution.WithLabels* method as demonstrated in Code Sample 4.22. This metric provides a breakdown of each rule and its success or failure count and is displayed in our Graphana dashboard.

```
CVMetrics.RulesExecution.WithLabels(rule.Data.RuleName.Value.Replace(" ", ""), "success").
    IncTo(rule.Data.SuccessCount.Value);
```

Code Sample 4.22: Counter for rule success or failure.

Using Graphana, we present a dashboard with a visual representation of our metrics and a view of our monitoring and reporting processes. We organized the Graphana dashboard into three categories: One category for displaying CV metrics with data and the other two categories for service status and system metrics. Service status included various performance metrics, latency, execution count, and more, while system metrics displayed CPU usage, memory allocation, and so on. In total, the dashboard comprised 22 panels for presenting metrics. Part of the dashboard is shown in figure 4.22

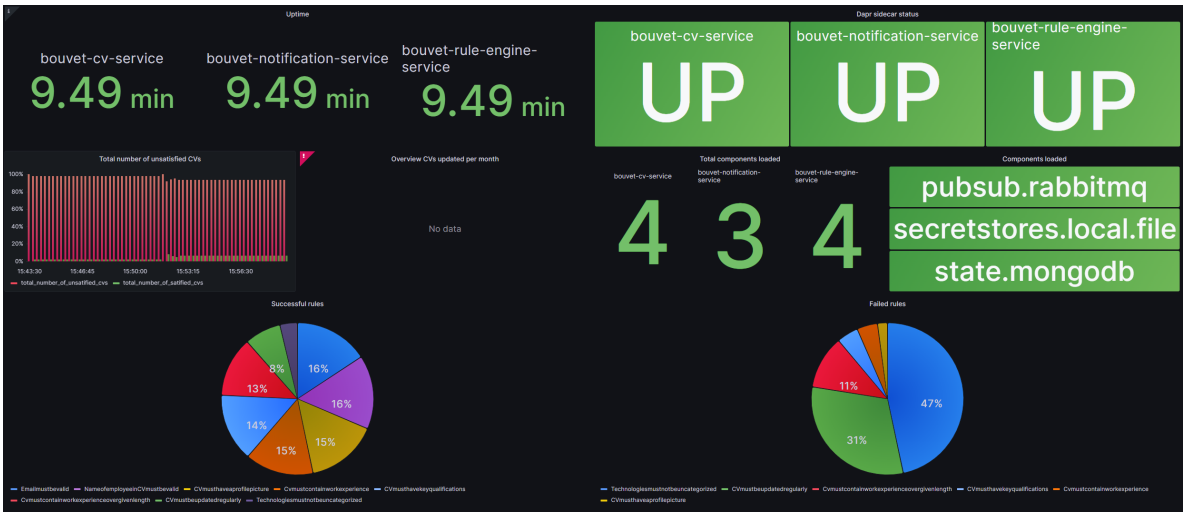

Figure 4.22: Graphana Dashboard.

To create these panels and display the metrics, we utilized *PromQL* queries to query the Prometheus data and present it according to our preferences. PromQL supports various mathematical operations, and with Grafana, we can choose from multiple display methods such as time series graphs, bar charts, pie charts, etc.(Prometheus, n.d.-b). We opted for various display methods to provide the end-user with the most information possible. The Code Sample 4.23 demonstrates our query for calculating the frequency of Rules Engine executions per second.

```
1 rate(rules_engine_executed_total[$__rate_interval])
```

Code Sample 4.23: PromQL query.

The below list of custom metrics was developed for the CV quality control solution. As these metrics are stored in the Prometheus time-series database, Bouvet can monitor their behaviour over a specified period.

1. Overview of Satisfied/Unsatisfied CVs:

   (a) The number of CVs that meet or fail predefined criteria.

2. Total Number of Rule Outcomes:

   (a) The total count of each rule's success and failure instances.

3. Rule Success vs. Failure Rate:

    (a) The comparison of how often specific rules succeed versus the rule fail provides insights into the most challenging rules.

4. Number of Employees Notified:

    (a) The count of employees who has been notified about their CV status showcasing the reach and effectiveness of the notification system.

5. Number of Employees Notified:

    (a) The variation in the number of CVs within Bouvet over a month illustrates trends in employee engagement and CV updates.

4.6 Security

In this section, we present the results of our security analysis. The analysis was guided by the principles and risk categories outlined in the OWASP Top 10. This approach provides a view of potential security vulnerabilities and helps identify the risks in our project. By aligning our analysis with the OWASP Top 10, we aim to ensure that our application adheres to good practices in web security.

4.6.1 Cryptographic Key Vault

One of the security features implemented in our application is integrating a *secret key store* as a Dapr component, as shown in figure 4.23. By employing a secret key store, we address some vulnerabilities outlined in the OWASP Top 10, such as Sensitive Data Exposure and Broken Access Control, by ensuring that sensitive data is securely managed and protected from unauthorized access.

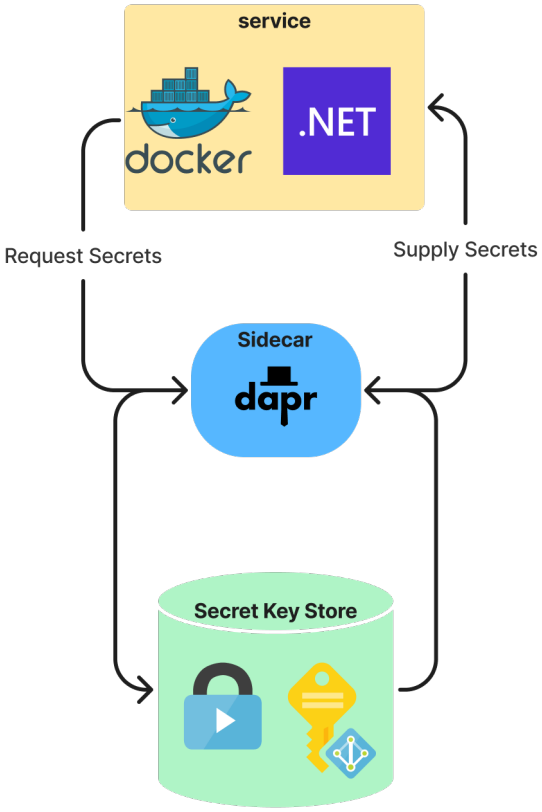

Figure 4.23: Overview of how a Cryptographic Key Vault works.

The project utilizes a JSON file as a source for storing secrets. While this implementation serves as a proof of concept and demonstrates the functionality of the Dapr secret store component, it is not a secure option for a production environment. We would use a more robust and secure solution in production, such as Azure Key Vault. Transitioning to Azure Key Vault or a similar service would require a configuration change in the Dapr secret store component, allowing for integration into the existing application when the application is production ready.

To access the secret store in our startup file, we load the configuration using the code in Code Sample 4.24.

```
builder.Configuration.AddDaprSecretStore("ruleengine-secretstore", new DaprClientBuilder().
    Build());
```

Code Sample 4.24: How we load secrets to our application.

4.6.2 SQL Injection Prevention

EF is used to mitigate SQL injection vulnerabilities, highlighted in the OWASP Top 10 as ”Injection”. EF’s built-in methods can eliminate the risk of standard SQL injection attacks arising from raw SQL queries and insecure string concatenation. (Microsoft, 2021c)

The built-in methods provided by EF automatically generate parameterized SQL queries, which separate data from the query itself. This separation ensures that user input is never treated as executable SQL code, making it impossible for a traditional SQL injection attack. (Microsoft, 2021c)

By avoiding the use of custom SQL queries and relying on the built-in functionality provided by EF, we believe that we adhere to secure coding practices that protect our application from SQL injection vulnerabilities.

4.6.3 Security Logging and Monitoring

The pilot application applies logging practices to ensure visibility into the system’s activities and potential security incidents. Doing so directly addresses the OWASP Top 10 vulnerability ”Security Logging and Monitoring Failures”.

Maintaining detailed and accurate logs, we think that it enhances our ability to detect vulnerabilities during the application’s development and operation phases.

4.6.4 Software and Data Integrity

In developing our application, we have been cautious in selecting and utilizing third-party packages, emphasizing using well-maintained and up-to-date libraries. This approach helps us stick to the OWASP Top 10 vulnerability category ”Software and Data Integrity Failures” and ”Vulnerable and Outdated Components”. By choosing reputable packages that are actively maintained, we minimize the risk of introducing vulnerabilities, outdated dependencies, or unsupported software into our application.

4.7 Software Testing

Software testing plays a role in software development, ensuring that applications are, in our opinion, reliable, stable, and meet the desired functional requirements. It involves a series of methodologies and practices designed to validate an application's performance, security, and usability, contributing to its quality. This section will discuss our project's various testing strategies and techniques.

Challenge and rationale

Incorporating testing into our project presented challenges, necessitating a balance between QA and efficient use of development resources. The rationale for implementing testing was to ensure the delivery of a reliable application.

A challenge in implementing tests was determining the appropriate level of testing coverage while considering factors such as code quality, adherence to Agile processes and early bug detection. We implemented a moderate test coverage because it helped identify and address issues early in the development process.

Adhering to Agile processes by incorporating testing enabled us to add features, knowing that our test suites would help validate the functionalities and ensure that existing features remained intact. This approach facilitated a more efficient development process to quickly adapt to changing requirements, as aligned with our methodology.

Implementation

We implemented testing by utilizing the *xUnit* framework, which facilitated the organization and execution of our test suites. We structured our code to accommodate both integration and unit tests.

Unit tests were employed to validate most of our classes, methods, and tasks, evaluating the application's components. These tests allowed us to identify and address any issues, thereby, in our view, improving the quality and reliability of our pilot.

Integration tests ensured integration and functionality across our system and enabled us to identify potential bottlenecks or inconsistencies.

Integration tests are slower than unit tests, involving multiple components and often requiring external resources such as databases or APIs. Consequently, an extensive suite of integration tests may lead to longer test execution times, slowing the development process and increasing

the project's cost. However, we believed that the positives of testing the integration of our application outweigh the time it took to validate our codebase. We measured our test sweep to take approximately 1 second to execute our combined 127 unit and integration tests.

Our tests adhered to a structure known as the "Three A's of unit testing" – Arrange, Act, and Assert. This approach entails arranging the necessary preconditions for the test, acting on being tested, and asserting the expected outcome(Cruz, 2021).

We used *FluentAssertions*, an assertion library, to improve the readability and fluency of our test assertions, moving towards a more functional programming style. FluentAssertions enabled us to write more expressive and readable tests, reducing ambiguity and enhancing the codebase's maintainability. A general example of FluentAssertions is demonstrated below in Code sample 4.25.

```
string result = "THESIS";
result.Should().StartWith("TH").And.EndWith("IS").And.Contain("ES").And.HaveLength(6);
```

Code Sample 4.25: The variable *result* from a test is asserted and returns *true* or *false*.

In addition to the testing methodologies mentioned, we employed TDD. By using TDD, we better understood the desired functionality and requirements before implementation. This approach allowed us to focus on the expected outcomes and design the solution accordingly, ensuring that our code met the criteria from the outset. The iterative nature of TDD also facilitated the continuous improvement of code, as we refined and optimized the implementation during the refactoring phase without compromising the passing test status.

In addition to our testing strategy, CI/CD was implemented by automating our tests to run every time new code was added to the codebase, as shown in figure 4.24.

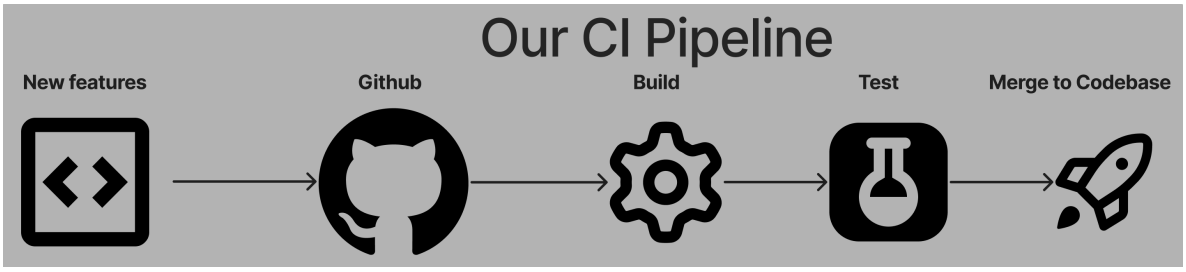

Figure 4.24: An overview of how our CI/CD Pipeline was implemented

This automation was achieved using *GitHub Workflows*. GitHub Workflows ensure the tests

are executed every time we implement new features into our codebase. By integrating CI/CD into our project, we identified and addressed any issues during development, which, in our opinion, resulted in a more robust and reliable application. The automated tests provided immediate feedback on the impact of any changes made to the codebase.

Chapter 5

Results and Interpretation

This chapter presents the results and interpretations of our thesis research. We will evaluate how we planned and developed the solution using Agile Scrum and XP methodology and assess the pilot implementation and limitations. Ultimately, we will ascertain whether the initial objectives of the thesis have been accomplished.

5.1 Holistic Assessment

The goal of this thesis, outlined in section 1.2, was developing a software solution to quality control Bouvet Employees' CVs. This was accomplished by applying business rules through a generic Rules Engine and subsequently notifying employees of required updates or modifications. The pilot has some limitations and is not production-ready. It successfully demonstrates the execution of business rules on Bouvet's CV database using a generic Rules Engine; hence our goals were met. Although, the notification process has limitations as an external notification service has not been developed by external parties, and additional security implementations are needed to transition the project into a production environment.

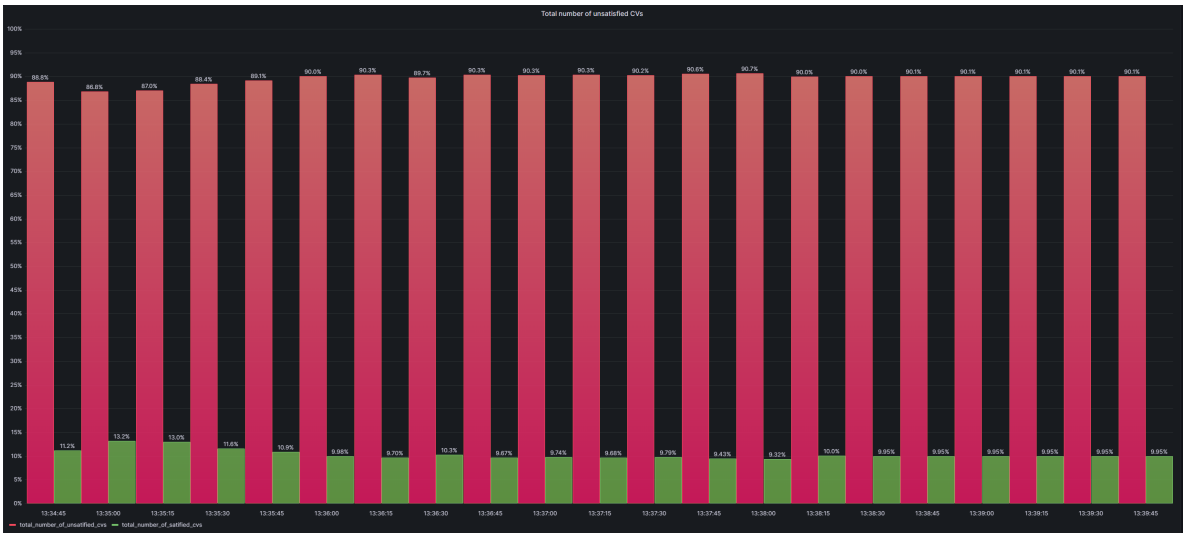

Figure 5.1: Graphana panel using our custom metrics to display that approximately 90% of CVs in Bouvet is not satisfied according to our present rules.

Our study reveals that 90% of the CVs submitted by Bouvet failed to comply with the rigid business rules, as demonstrated in figure 5.1. Specifically, there were two rules that the CVs often failed to meet. Firstly, 50.8% of the CVs had not been updated within the recommended three-month period. Secondly, 76.7% of the CVs included uncategorized technologies.

Figure 5.2: Overview of statistics on each incorporated business rule.

These results, in our opinion, underscore our pilot is providing Bouvet with valuable insights and areas for improvement. Our software can assist Bouvet and its employees in strategic decision-making by identifying areas where the current CVs can be improved. This will elevate the quality of CVs presented to potential clients and partners for Bouvet, enhancing their reputation and customer retention.

5.2 Assessing the use of Scrum and Extreme Programming

The Agile Scrum and Extreme Programming methodologies were pivotal to guiding our software development process, which we implemented throughout this thesis. These methodologies laid the groundwork for us to deliver a customer-centric software solution. By utilizing an iterative approach, we aligned our processes with our goals. To facilitate our methodology, we utilized GitHub Projects, which offered us essential tools to comply with Agile Scrum and Extreme Programming methodologies by providing a visualization and overview of the backlog.

One of the more difficult tasks in our implementation process involved breaking each feature into smaller, manageable components, particularly due to the intricate nature of the project and the ambiguity surrounding the best implementation method. To tackle this challenge, we could have initially established a well-defined and mutually agreed-upon implementation plan instead of sometimes modifying our code. This approach could have enabled us develop specific features more rapidly and effectively.

Through the course of our project, we recognized that the 4-week sprints we had planned were not suited to the nature of our project. Consequently, we concluded that future sprints should be time-boxed for shorter periods, increasing our responsiveness. This decision was prompted by the realization that certain requirements and features took longer than expected to develop. Shorter sprints would allow us to prioritize and potentially adjust these requirements in consultation with Bouvet.

XP helped us focus on delivering high-quality code that adhered to Bouvet's standards. Pair programming was particularly useful in solving complex problems, and our CI pipeline allowed us to test and develop the application quickly and confidently.

Reflecting on our experience, implementing TDD proved beneficial. It helped us ensure the functionality of the features using it while promoting clean, readable, and maintainable code. Despite the initial time investment required by TDD on the need to understand the feature implementation before actual development. We believe the long-term benefits outweighed the additional upfront time. The testing process enabled us to catch bugs early, reducing debugging time and leading to an efficient development process overall. In conclusion, TDD was a suitable fit for our project, despite the slight increase in initial development time.

In conclusion, our chosen methodologies, including Agile Scrum, XP, provided, in our view, value and helped guide us throughout this pilot. Despite a few initial challenges, these method-

ologies enabled us to deliver a customer-focused solution, manage complex tasks, adapt to changing requirements, and develop a high-quality software pilot.

5.3 Assessment of the Pilot implementation

In this section, we will concentrate on two aspects of our pilot: First, we will identify the shortcomings and limitations, and second, we will evaluate our implementation. We recognize that specific goals could not be fully achieved due to time constraints and dependencies on other internal systems within Bouvet, such as a notification service and the absence of an admin API key. An in-depth discussion will elucidate how these unimplemented features impact the pilot's effectiveness.

5.3.1 Limitations and Shortcomings

In this section, four limitations of the experiment will be highlighted. First, the limitation related to an external event bus notification process intended to notify employees through Bouvet's communication mediums. Second, the constraint posed by not receiving a perpetually valid admin API key from Bouvet to access CV Partner. Third is the security limitations of not having implemented Role-Based Access Control (RBAC) and the API gateway.

External Event Bus

The unavailability of the external event bus, another project at Bouvet still in development, limits the functionality of our application, as it prevents us from effectively integrating with Bouvet's existing communication channels. Without the event bus, our notification implementation remains incomplete, meaning employees cannot receive a notification if their CV is not within the quality control rules. Although this issue is beyond our control, it affects the application in fulfilling its intended purpose and end-functionality.

Admin API-key

An admin API key, for conveniently fetching CVs from CV Partner was not acquired. While Bouvet facilitated the acquisition of such a key, the application's non-production status led us to postpone this approach. Consequently, our application is limited in that it requires manually replacing the cookie and the CSRF token with valid ones inside our configuration. New cookies and CSRF tokens would be obtained from our browser using our account. This limitation hinders the application's ability to run for extended periods without changing the cookie and associated credentials, affecting its overall efficiency in fetching CVs. Although the application is designed to accommodate a configuration change for the admin API key,

the current constraint impacts the user experience and the application's capacity to perform optimally over longer durations.

Role Based Access Control

Although we planned to integrate an RBAC system called *Bicep* and connect it to Bouvet's Active Directory (AD), time constraints prevented us from accomplishing this goal. The proposed implementation aimed to enable certain roles to access specific endpoints, such that only business unit managers at Bouvet are authorized to operate the Rules Engine and add rules. We believe this approach would have ensured secure and controlled access to the application's functionalities, as access would have been restricted to authorized personnel only.

This security limitation is a concern, as it impacts the overall integrity of the application and user trust. Before the application can be considered production-ready, implementing RBAC is essential to guarantee access control and safeguard sensitive information.

API Gateway

Due to time constraints, we have not implemented an API gateway. An API gateway would have facilitated an entry point for all services and managed control access to these services. The absence of an API gateway limits the application's ability to manage and control requests. In addition, the fact that the RBAC system needed to work in conjunction with the API gateway further compounded the challenge of implementing both features within the limited time frame.

5.3.2 Interpretation of Microservice-, Event-driven- and CQRS architecture

Microservice architecture is a commendable approach to developing and deploying our microservice application. It broke down our pilot into smaller, independent services, which scaled them separately. As a result, we added new services while conveniently developing the pilot without disrupting other components.

Moving on to EDA in our application. While we do not strictly adhere to EDA, we found it a good fit for publishing updates to multiple services without expecting a response. However, it was not optimal when needing direct communication with a single service and also requiring a response.

Literature authored by Rocco Scaramuzzi(Scaramuzzi, 2022) reinforces our viewpoint, which proposes that it is essential to assess each scenario individually to determine whether a command or an event-based solution is more suitable. Scaramuzzi also highlights the significance of attaining equilibrium between events and commands to achieve an excellent EDA.

Our combination of EDA and regular service invocation is successful. We realize that not forcing our strict architecture but rather having a more flexible approach for each use case would probably have saved us time and resources.

Next, concerning the CQRS architecture, it is beneficial in our pilot as it separates the responsibility of handling read and write operations into separate components, simplifying our code and improving service performance by optimizing read and write operations. Despite requiring additional effort in coding work, we believe that the benefits we have seen from incorporating CQRS inside each microservice have justified the extra development time.

5.3.3 Interpretation of Design Principles, Strategies and Patterns

The result of using the mediator pattern, utilized using the *MediatR* NuGet package, was a centralized communication distributor, eliminating tight couplings within our services. This made the services more maintainable and scalable, making it easier for us to add functionality to microservices without affecting others. For the future, we believe this pattern will make it easier to build upon for us or other developers.

Another beneficial design pattern is the Options pattern. This pattern allows for changes to the configuration of our services at runtime. This means that we could change the configuration while the application was running, leading to less downtime and having to restart the app to make changes. If the pilot moves on to production, the Configuration pattern may come in handy as users can make changes to the configuration more easily.

The API contracts strategy in our communication between microservices establishes clear communication contracts and specifies each service's requests and responses. The outcome of this strategy may aid future software engineers in Bouvet's clear overview of what our services expect in terms of inputs.

5.3.4 Interpretation of mitmproxy2client

The automatic generation of the HTTP client was successful for multiple APIs, most importantly, our target for this pilot, CV Partner. It also reverse-engineered the API of Storm.no, a weather-forecast service. This indicates that our HTTP generation can be applied to different

APIs. It can reduce the manual work of reverse-engineering APIs when creating or updating an HTTP client.

However, reverse-engineering some APIs causes issues. For example, we were not able to reverse-engineer the API of Pent.no. This could be because servers recognise that the requests come from a script rather than a browser. Servers may use techniques to differentiate between script requests and web browsers. This could include checking for the presence of headers that are typically sent by browsers but not by scripts or looking for patterns in the timing or sequence of requests characteristic of scripts. Further, there might be checks for JavaScript execution, which Python Requests, an HTTP library, does not support. Further investigation is needed to understand why our approach did not work with these APIs.

After a search online, we found that this issue was also reported by others(Treadway, 2019) trying to gather data from websites. It seems a potential fix is to use a package named *requests_html* instead of Requests to be able to run JavaScript. We encourage this to be further investigated in future work.

These findings suggest that while our approach has potential, challenges remain. Future work could focus on improving compatibility with more secure or sophisticated APIs, possibly by incorporating more advanced techniques for mimicking a browser. This could expand the range of APIs our approach can handle, making it more broadly applicable.

5.3.5 Interpretation of Fields

The implementation and usage of our custom library, ”Fields”, are successful within our organization. This library, containing custom types with built-in validation leveraging the FluentValidation package, provides a foundation for our services by ensuring type safety and adherence to the defined configuration. The configuration limits the properties of the types, such as a maximum and minimum value to a number. This ensures the data adheres to the expected constraints, preventing invalid data.

Further, the Fields types contain default values that are never null, eliminating the potential for null reference errors.

However, a trade-off to the increased validation is that the custom types introduce a layer of abstraction that can make them cumbersome to use with standard C# objects that do not accept these custom types. We and other users of the package need to use the. Value property to extract the underlying value(int, string etc.), which can be a minor inconvenience for users. Despite this issue, the library is in use within the organization

Library users have given us their input, expressing that the ”introduced abstraction layer contributes to the complexity and requires an adjustment period”. Conversely, they ”value the prevention of null reference errors” and consider that the ”incorporated validation could potentially be beneficial in the future, though so far, beyond the basic validation setup, has not proven necessary”.

The issue with the layer of abstraction is something that could be improved in the future. One potential solution might be to implement implicit conversion operators that automatically convert the custom types to their underlying types when necessary, reducing the need for developers to extract the value manually.

5.3.6 Interpretation of Dapr

Using Dapr as a communication middleware is beneficial for our application. Its service-to-service invocation, publish and subscribe, observability, state management, and secret-store capabilities have enhanced our application. Following are the results of each feature.

The state store feature of Dapr simplified the connection and utilization of a database. However, the difficulty in creating tests and mocking the state store was a drawback, as it made it harder to ensure the correctness of our code in a non-Dapr environment.

The service-to-service invocation resulted in beneficial communication between services without specifying URLs. The communication is passed using service names instead. This may become useful in the future if the application enters production environments where URLs may frequently change. However, the generic error messages made debugging this implementation challenging.

The publish and subscribe feature simplified the connection to the RabbitMQ event bus and explicitly removed the need to name each event to each service subscriber. We found this feature convenient to set up, experiencing no drawbacks using this feature,

Dapr’s observability feature provided valuable insights into our services’ performance and flow, improving our ability to monitor our application. However, the lack of persistence in traces and logs can make debugging more difficult, especially in abrupt program exits. A future enhancement to the application may be to persist these traces and logs using our logging service.

The secret store allowed for the secure handling of secrets like API keys, passwords, and connection strings, improving our application’s security. However, the extended start-up time

for this service sometimes caused other parts of the application to fail. This drawback was rectified with the manipulation of the start-up sequence.

5.3.7 Interpretation of Generic Rules Engine

The result of our Generic Rules Engine implementation shows that it can execute any workflow on any input. By building upon the Microsoft Rules Engine's foundation, we made it generic and with added functionalities. For instance, we can create more complex rules since we are not limited by lambda expressions. As a result, our decision to utilize the Microsoft Rules Engine saved us considerable time and effort in laying the groundwork for our Generic Rules Engine.

5.3.8 Interpretation of our Microservices

Our microservice architecture comprises three main services and sixteen other services, comprising Dapr sidecars, logging, tracing etc. This composition results in a successful implementation of an autonomous quality control system for Bouvet's CV database. We will below discuss and interpret the outcomes of the individual services we developed.

Bouvet.CV.Service

The Bouvet.CV.Service successfully retrieves CVs from our CV Partner using the generated HTTP client. However, as part of future work, we need to engage with CV Partner to discuss their API limits, especially regarding requests per second.

The current approach of sending requests synchronously, with a built-in delay of 100ms per request, fetches 100 CVs in 56.6 seconds. However, executing the process asynchronously without a built-in delay only takes 4.5 seconds. The graph in Figure 5.3 shows the difference in time performance fetching 100 CVs synchronously and then switching to fetch 100 CVs asynchronously. This suggests that if we remove the custom delay, which adds up to $100ms * 100CV = 10$ seconds in this case and switches to fetching CVs asynchronously, we achieve a 10.36 times faster performance.

Figure 5.3: Time difference between synchronous and asynchronous fetching of CVs.

Our service achieved a faster processing time by utilizing multiple threads. The process utilized 18 additional threads, as demonstrated in Figure 5.4. We can observe the green line with a base of 51 threads for the first execution around 10:50. We execute the process again in asynchronous mode with the blue dotted line, and the green line spikes to a total of 69 threads. This test suggests that asynchronous fetching of CVs would significantly boost our system's performance, particularly if implemented in a larger organization than Bouvet.

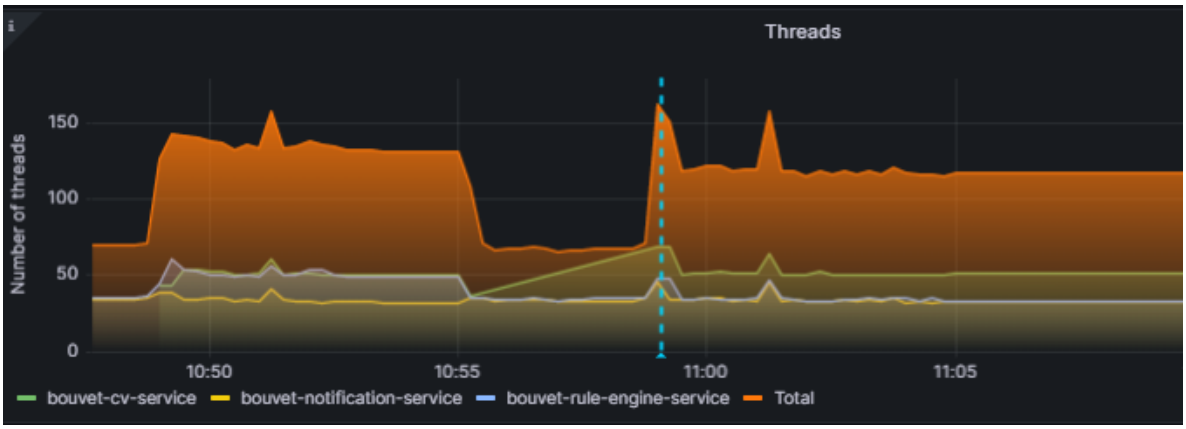

Figure 5.4: Thread usage comparison using synchronous and asynchronous CV fetching processes in our program.

The Bouvet.CV.Service is successful in persisting workflows and rules. Bouvet can send API calls to our service and perform CRUD operations on the business rules used to quality control the CV database. Currently, the sole means of interacting with our service is via a Swagger interface tool. However, we recognize the potential benefits of creating a front end in future work for non-technical business leaders to modify the workflow without technical knowledge. This could enhance the service's usability and provide greater flexibility to Bouvet.

Bouvet.Rule.Engine.Service

Our implementation of the Bouvet.Rule.Engine.Service has shown that the Generic Rules Engine is generic and capable of executing any workflow on any input. Using a background service that executes the Rules Engine in a background task and gets the task from the queue has resulted in us being able to handle large volumes of workflow requests without staggering our service.

Furthermore, by publishing an event when execution is finished, we have created a more decoupled service from other services. The application can be more adaptable in future work by relieving clients from subscribing to events. By providing a natural request-response for systems that cannot connect to our event bus and listen to events, the system will not force other services to conform to our architecture.

Bouvet.Notification.Service

The notification service is capable of retrieving results from the Bouvet.Rule.Engine.Service

and should be able to notify employees when connected to an external event bus. The external event bus we were supposed to use is still in development and was unavailable. Due to this circumstance, the service was tested using a mock event bus. The test shows that the service publishes events and is ready when the external event bus is completed.

Moving forward, a plan to add more logic to our notification process to provide additional employee satisfaction could be beneficial. For instance, a scheduling system to send out notifications at specific times and circumstances, minimizing the number of notifications sent to employees.

5.4 Feedback from Bouvet

Bouvet som selskap er fundamentert på «smarte hoder» dvs. ansatte med kompetanse og erfaring på etterspurt teknologi. Vi er samfunnsbyggere og oppgaven er en liten med viktig del i det vi mener er samfunnsbygging. Studenter som gjør en oppgave fra Bouvet skal få en relevant og engasjerende oppgave hvor målet er å gi studentene erfaringer innen forventingsstyring, estimering, programmering, design og arkitektur og testing. Bouvet eksistens er basert på kunnskap og erfaringer til våre ansatte og ingenting annet.

CV Alle ansattes i Bouvet har en CV som er registret i et fagsystem. Ansatte forplikter seg til å vedlikehold innholdet slik at dokumentet oppsummerer den ansattes kompetanse, erfaringer og et sammendrag av deg som ansatte og eller person. Bouvet ansatte med leveranse ansvar bruker fagsystemet når kundene etterspør Bouvet tjenester.

Oppdraget Kvaliteten på våre ansatte er dokumentert i CV fagsystemet. Systemet er en ekstern tjeneste og ofte brukt av våre konkurrenter. Systemet har ikke rapporteringsfunksjonalitet som gir ansatte og deres ledere informasjon om kvalitet på innhold, statistikk på hvem og hva som er oppdatert etc. Globale initiativer som f.eks. sikkerhet (Mål for 2023) kan ikke med dages system kvantifiseres. Oppdraget adresserer behovene for kontroll, oppfølging og måling av trender.

Gjennomføringen Gjennomføringen har fra Bouvet siden vært meget god. Oppgavene og utfordringer har blitt løst i forkant og underveis. I starten fokuserte vi på objekt orientert programmering og viktigheten av domene objekter. Viktigheten av å Innkapsulering av funksjonalitet ble kjørt som en parallell aktivitet. Studentene var imøtekommende og viste stor interesse for prinsippene som bør følges for å skape kildekode av høy kvalitet. Prinsippene og metodikken ble anvendt uoppfordret videre i prosjektet uten påtrykk fra undertegnede. Studentene valgt en løsningsarkitektur basert på mikrotjenester. Av egen erfaring vil en slik

løsning være kompleks og tidskrevende og implementerte. For å komme i mål ble studentene utfordret på å identifisere biblioteker av god kvalitet og med egenskaper som tidsakselerasjon og redusere kompleksitet. Valgene resulterte i hurtige leveranser av flere mikrotjenester! Underveis måtte studentene også ta bruke verktøy som virtualisering av tjenester og «reverse engineering» av APIer. Studentene har løst oppgaven på en eksemplarisk måte. I tillegg har de hjulpet ansatte og andre studenter med kompetanse og klapp på skulderen. Bouvet har sett ideen i praksis og at resultatet av oppgaven er begynnelse til et produksjonsklart system.

Chapter 6

Conclusion

In this bachelor's thesis, we have demonstrated the process of developing a software solution to improve the quality control of CVs for employees at Bouvet. The results of our pilot have shown that having a Scrum with Extreme Programming as a project management system in place provided us with a clear overview and helped us manage our project effectively.

Through our research and implementation of our pilot, we were able to develop a generic rule engine with extended functionalities to effectively control the quality of a CV database. Our results show that 90% of the CVs at Bouvet do not meet their business standards.

We developed a scalable and maintainable pilot using Microservices, Event-Driven, and Command and Query Responsibility Segregation architecture. Our pilot enables us to change the parameters of the quality control rules, supporting future modifications using create, read, update and delete operations. Due to time constraints, our pilot is not production-ready due to the absence of Role-based access control and an API-Gateway. Once the project of another group at Bouvet, the event bus, is completed, our implementation will be able to notify employees about the state of their CVs, further improving the overall quality control process.

Bibliography

AgileAlliance. (2023). *Extreme programming*. Retrieved March 13, 2023, from https://www.agilealliance.org/glossary/xp/

Arun, R. (2023a). *An introduction to the world of user stories*. Retrieved May 14, 2023, from https://www.simplilearn.com/tutorials/agile-scrum-tutorial/user-stories

Arun, R. (2023b). *What is agile: Understanding agile methodology and principles*. Retrieved February 27, 2023, from https://www.simplilearn.com/tutorials/agile-scrum-tutorial/what-is-agile

Baeldung. (2022). *The dto pattern (data transfer object)*. Retrieved May 10, 2023, from https://www.baeldung.com/java-dto-pattern

Beck, K., Beedle, M., Arie van Bennekum, A. C., Cunningham, W., Martin Fowler, J. G., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Principles behind the agile manifesto*. Retrieved February 27, 2023, from https://agilemanifesto.org/principles.html

Bogard, J. (n.d.). *Mediatr home*. Retrieved May 1, 2023, from https://github.com/jbogard/MediatR/wiki

BusyBox. (n.d.). *Busybox: The swiss army knife of embedded linux*. Retrieved April 15, 2023, from https://busybox.net/about.html

Chauhan, S. (2022). *Chain of responsibility design pattern - c#*. Retrieved May 14, 2023, from https://www.dotnettricks.com/learn/designpatterns/chain-of-responsibility-design-pattern-dotnet

CHERVENKOVA, M. (2022). *What are feedback loops and why you need to implement them?* Retrieved May 14, 2023, from https://kanbanize.com/blog/feedback-loops/

Cruz, J. (2021). *The three a's of unit testing*. Retrieved May 14, 2023, from https://dev.to/coderjay06/the-three-a-s-of-unit-testing-b22

Cummings, I. (2018). *The operation result pattern — a simple guide*. Retrieved May 3, 2023, from https://medium.com/@cummingsi1993/the-operation-result-pattern-a-simple-guide-fe10ff959080

Dapr. (n.d.-a). *Build connected distributed applications faster*. Retrieved March 27, 2023, from https://dapr.io/

Dapr. (n.d.-b). *Distributed lock overview*. Retrieved May 4, 2023, from https://docs.dapr.io/developing-applications/building-blocks/distributed-lock/distributed-lock-api-overview/

Dapr. (n.d.-c). *Try out secrets management*. Retrieved May 14, 2023, from https://docs.dapr.io/developing-applications/building-blocks/secrets/secrets-overview/

Docker. (n.d.). *Docker overview*. Retrieved March 27, 2023, from https://docs.docker.com/get-started/overview/

Drumond, C. (n.d.). *What is scrum?* Retrieved February 27, 2023, from https://www.atlassian.com/agile/scrum

Erinç, Y. K. (2020). *The solid principles of object-oriented programming explained in plain english*. Retrieved March 28, 2023, from https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/

EventStore. (n.d.). *Cqrs (command-query responsibility segregation)*. Retrieved April 19, 2023, from https://www.eventstore.com/cqrs-pattern

Fernando, R. (n.d.). *Evaluating performance of rest vs. grpc*. Retrieved May 15, 2023, from https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da#:~:text=gRPC%5C%20is%5C%20roughly%5C%207%5C%20times,of%5C%20HTTP%5C%2F2%5C%20by%5C%20gRPC.

FluentValidation. (n.d.). *Fluentvalidation*. Retrieved May 1, 2023, from https://docs.fluentvalidation.net/en/latest/

Foundation, P. S. (n.d.). *Requests: Http for humans™*. Retrieved April 10, 2023, from https://requests.readthedocs.io/en/latest/

Francino, Y., & Denman, J. (2023). *Agile retrospective*. Retrieved May 14, 2023, from https://www.techtarget.com/searchsoftwarequality/definition/Agile-retrospective

Frühauff, D. (2022). *Improve your pipeline maintainability with test categories in xunit*. Retrieved May 15, 2023, from https://dateo-software.de/blog/test-categories-in-xunit

GeeksforGeeks. (n.d.). *Normal forms in dbms*. Retrieved May 8, 2023, from https://www.geeksforgeeks.org/normal-forms-in-dbms/

Gillis, A. (2023). *Mongodb*. Retrieved May 14, 2023, from https://www.techtarget.com/searchdatamanagement/definition/MongoDB

GitHub. (n.d.-a). *About issues*. Retrieved April 19, 2023, from https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues

GitHub. (n.d.-b). *About milestones*. Retrieved April 19, 2023, from https://docs.github.com/en/issues/using-labels-and-milestones-to-track-work/about-milestones

GitHub. (n.d.-c). *About projects*. Retrieved April 19, 2023, from https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects

GitHub. (n.d.-d). *About pull requests*. Retrieved April 19, 2023, from https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests

GitHub. (n.d.-e). *About workflows*. Retrieved April 19, 2023, from https://docs.github.com/en/actions/using-workflows/about-workflows

GrafanaLabs. (n.d.). *What is grafana?* Retrieved April 19, 2023, from https://grafana.com/oss/grafana/

Harris, C. (n.d.). *Distributed scrum: How to manage scrum remote teams*. Retrieved May 13, 2023, from https://www.atlassian.com/agile/scrum/distributed-scrum

Inc., D. (n.d.-a). *Control startup and shutdown order in compose*. Retrieved April 10, 2023, from https://docs.docker.com/compose/startup-order/

Inc., D. (n.d.-b). *Use bridge networks*. Retrieved April 11, 2023, from https://docs.docker.com/network/bridge/

Istio. (n.d.). *The istio service mesh*. Retrieved May 8, 2023, from https://istio.io/latest/about/service-mesh/

JQ. (n.d.). *Jq*. Retrieved March 13, 2023, from https://stedolan.github.io/jq/

Kaseb, K. (2022). *The layered architecture pattern in software architecture*. Retrieved May 4, 2023, from https://medium.com/kayvan-kaseb/the-layered-architecture-pattern-in-software-architecture-324922d381ad

Khine, T. (2019). *Red, green, refactor!* Retrieved April 7, 2023, from https://medium.com/@tunkhine126/red-green-refactor-42b5b643b506

Kleier, T. (2020). *How to version a rest api*. Retrieved May 10, 2023, from https://www.freecodecamp.org/news/how-to-version-a-rest-api/

Kooijman, S. (n.d.). *Scrum vs lean*. Retrieved February 27, 2023, from https://leansixsigmagroup.co.uk/scrum-vs-lean/

Lane, K. (2021). *What is an api contract?* Retrieved May 14, 2023, from https://apievangelist.com/2019/07/15/what-is-an-api-contract/

Larkin, K., & Anderson, R. (2022). *Options pattern in asp.net core*. Retrieved May 12, 2023, from https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-7.0

Larkin, K., Smith, S., & Dahler, B. (2023). *Dependency injection in asp.net core*. Retrieved May 12, 2023, from https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-7.0

Mathew, S. (2021). *Software design principles every developers should know*. Retrieved March 28, 2023, from https://bootcamp.uxdesign.cc/software-design-principles-every-developers-should-know-23d24735518e

Microsoft. (2023a). *Background tasks with hosted services in asp.net core*. Retrieved May 10, 2023, from https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-7.0&tabs=visual-studio

Microsoft. (2022a). *Compare newtonsoft.json to system.text.json, and migrate to system.text.json*. Retrieved April 16, 2023, from https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/migrate-from-newtonsoft?pivots=dotnet-7-0

Microsoft. (2022b). *The dapr state management building block*. Retrieved May 3, 2023, from https://learn.microsoft.com/en-us/dotnet/architecture/dapr-for-net-developers/state-management

Microsoft. (2023b). *Database providers*. Retrieved May 14, 2023, from https://learn.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli

Microsoft. (2023c). *Databases*. Retrieved May 14, 2023, from https://learn.microsoft.com/en-us/sql/relational-databases/databases/databases?view=sql-server-ver16

Microsoft. (2022c). *Entity framework*. Retrieved May 14, 2023, from https://learn.microsoft.com/en-us/aspnet/entity-framework

Microsoft. (2021a). *Entity framework overview*. Retrieved May 14, 2023, from https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview

Microsoft. (2023d). *Grpc*. Retrieved May 9, 2023, from https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/grpc

Microsoft. (n.d.). *Publisher-subscriber pattern*. Retrieved March 28, 2023, from https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber

Microsoft. (2021b). *Querying data*. Retrieved May 14, 2023, from https://learn.microsoft.com/en-us/ef/core/querying/

Microsoft. (2022d). *Records (c# reference)*. Retrieved May 2, 2023, from https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record

Microsoft. (2021c). *Security considerations (entity framework)*. Retrieved May 14, 2023, from https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ef/security-considerations

Microsoft. (2023e). *Using type dynamic*. Retrieved May 2, 2023, from https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/interop/using-type-dynamic

mitmproxy. (n.d.-a). *Mitmproxy*. Retrieved April 10, 2023, from https://docs.mitmproxy.org/stable/concepts-howmitmproxyworks/

mitmproxy. (n.d.-b). *Mitmproxy is a free and open source interactive https proxy.* Retrieved March 13, 2023, from https://mitmproxy.org/

mitmproxy2swagger. (2023). *Mitmproxy2swagger*. Retrieved March 13, 2023, from https://github.com/alufers/mitmproxy2swagger

Nassi, T. (2022). *Benefits of migrating to event-driven architecture*. Retrieved April 28, 2023, from https://aws.amazon.com/blogs/compute/benefits-of-migrating-to-event-driven-architecture/

Nowak, M. (2023). *What is a rules engine and why do you need it?* Retrieved May 15, 2023, from https://www.hyperon.io/blog/what-is-a-rules-engine

Nswag. (n.d.). *Nswag*. Retrieved March 15, 2023, from https://github.com/RicoSuter/NSwag

nuget. (n.d.). *Create .net apps faster with nuget*. Retrieved March 27, 2023, from https://www.nuget.org/

OWASP. (2021). *Owasp top ten*. Retrieved May 8, 2023, from https://owasp.org/www-project-top-ten/

PAL, S. K. (2023). *Software engineering extreme programming*. Retrieved March 13, 2023, from https://www.geeksforgeeks.org/software-engineering-extreme-programming-xp/

Prometheus. (n.d.-a). *Overview*. Retrieved April 19, 2023, from https://prometheus.io/docs/introduction/overview/

Prometheus. (n.d.-b). *Querying prometheus*. Retrieved May 8, 2023, from https://prometheus.io/docs/prometheus/latest/querying/basics/

Rabbitmq. (n.d.). *Rabbitmq is the most widely deployed open source message broker*. Retrieved March 27, 2023, from https://www.rabbitmq.com/

RedHat. (2021). *What is containerization?* Retrieved March 28, 2023, from https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization

Refactoring.Guru. (n.d.-a). *Criticism of patterns*. Retrieved May 1, 2023, from https://refactoring.guru/design-patterns/criticism

Refactoring.Guru. (n.d.-b). *Factory method*. Retrieved April 19, 2023, from https://refactoring.guru/design-patterns/factory-method

Refactoring.Guru. (n.d.-c). *Mediator*. Retrieved April 19, 2023, from https://refactoring.guru/design-patterns/mediator

Richardson, C. (n.d.). *What are microservices?* Retrieved March 27, 2023, from https://microservices.io/

Scaramuzzi, R. (2022). *Event-driven microservice architecture, don't use only events but use commands too!* Retrieved May 12, 2023, from https://medium.com/rocco-scaramuzzi-tech/event-driven-microservice-architecture-dont-use-only-events-but-use-commands-too-b8694d370436

seq. (n.d.). *Overview*. Retrieved March 27, 2023, from https://docs.datalust.co/docs

Streng, T. (2022). *.net performance #2: Newtonsoft vs. system.text.json*. Retrieved April 17, 2023, from https://medium.com/@tobias.streng/net-performance-series-2-newtonsoft-vs-system-text-json-2bf43e037db0

T. Bray, E. (2017). *The javascript object notation (json) data interchange format*. Retrieved April 16, 2023, from https://datatracker.ietf.org/doc/html/rfc8259

Treadway, A. (2019). *Scraping data from a javascript webpage with python*. Retrieved May 2, 2023, from https://theautomatic.net/2019/01/19/scraping-data-from-javascript-webpage-python/

Vlad, P. (2023). *Understanding string equality in c#: Comparing objects, value vs. reference, and string interning*. Retrieved May 3, 2023, from https://blog.devgenius.io/in-c-a-string-is-an-object-7b5dc0cf3e30

West, D. (n.d.). *Agile scrum roles and responsibilities*. Retrieved May 13, 2023, from https://www.atlassian.com/agile/scrum/roles

Whyatt, T. (2023). *What is the waterfall model?* Retrieved March 31, 2023, from https://www.one-beyond.com/pros-cons-waterfall-software-development/

Zipkin. (n.d.). *Zipkin*. Retrieved March 27, 2023, from https://zipkin.io/