HAW HAMBURG

BACHELOR THESIS
Jan Hedwig

Redesigning an AI-as-a-Service Platform: A maintainable microservice architecture for small development teams

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jan Hedwig

Neugestaltung einer AI-as-a-Service-Plattform: Eine wartbare Microservice-Architektur für kleine Entwicklungsteams

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Wirtschaftsinformatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Ulrike Steffens
Supervisor: Prof. Dr. Stefan Sarstedt

Submitted on: 15. Februar 2025

Jan Hedwig

Thema der Arbeit

Neugestaltung einer AI-as-a-Service-Plattform: Eine wartbare Microservice-Architektur für kleine Entwicklungsteams

Stichworte

Softwarearchitektur, Microservices, Machinelles Lernen, Datensynthese, Kubernetes

Kurzzusammenfassung

Diese Arbeit konzentriert sich auf die Neugestaltung einer AI-as-a-Service-Plattform mit dem Ziel, eine wartbare Microservice-Architektur zu entwickeln, die für kleine Entwicklungsteams geeignet ist. Die DaFne-Plattform, die im Mittelpunkt dieser Arbeit steht, bietet verschiedene Methoden zur Generierung und Evaluation von Daten. Die Herausforderung besteht darin, eine Architektur zu entwerfen, die Skalierbarkeit, Erweiterbarkeit und Wartbarkeit gewährleistet, und gleichzeitig die begrenzten Ressourcen eines kleineren Entwicklungsteams berücksichtigt. Um dies zu erreichen, wird die bestehende Architektur analysiert, um Schwachstellen zu identifizieren. Basierend auf dieser Analyse untersucht die Arbeit Best Practices und Architekturmuster, die eine Balance zwischen Komplexität, Leistung und Ressourceneffizienz bieten. Ein zentraler Bestandteil dieser Untersuchung ist die Bewertung verschiedener Microservice-Designansätze, einschließlich der Nutzung von reaktiven Frameworks und Kubernetes für die Container-Orchestrierung. Die vorgeschlagene Architektur wird beispielhaft durch die Implementierung eines Datengenerierungdienstes der Plattform umgesetzt. Die vorgeschlagenen Ansätze reduzieren die Komplexität und verbessern die Wartbarkeit innerhalb einer Microservice-Architektur. Die Arbeit schließt mit weiteren Themengebieten für zukünftige Arbeiten und gibt die nächsten Schritte in der Entwicklung der Plattform vor.

Jan Hedwig

Title of Thesis

Redesigning an AI-as-a-Service Platform: A maintainable microservice architecture for small development teams

Keywords

Software Architecture, Microservices, Machine Learning, Data Synthesis, Kubernetes

Abstract

This thesis focuses on the redesign of an AI-as-a-Service platform with the goal of creating a maintainable microservice architecture suitable for small development teams. The DaFne platform, which is the focus of this work, provides various data generation and evaluation methods. The challenge lies in designing an architecture that ensures scalability, extensibility, and maintainability while considering the limited resources of a small team. To address this, the existing architecture is analyzed to identify pain points and areas for improvement. Based on this analysis, the thesis explores best practices and architectural patterns that balance complexity, performance, and resource efficiency. A key part of this work is the evaluation of different microservice design approaches, including the use of reactive frameworks and Kubernetes container orchestration solutions. The proposed architecture is partially implemented by reworking a core service of the platform to serve as an example. The resulting approach offers a solution to reduce complexity within a microservice architecture and improve maintainability. The thesis concludes with recommendations for future research topics and outlines the next steps in the development of the platform.

Contents

List of Figures

List of Tables

1 Introduction

As machine learning is becoming more widespread, the need for data is greater than ever. Data quality strongly impacts the performance of machine learning models. Poor data quality can lead to unwanted biases or inaccuracies [JPN$^+$20]. However, not only the quality is important, but also the quantity of data. Keeping a large set of diverse data can help combat over-fitting and increase the robustness of a model as it is presented with a wider variety of information. This is another way biases can be removed, which is important in high-stakes areas such as healthcare and finances [RHW21]. Prioritizing data quality and variety is essential in developing accurate, fair, and adaptable machine learning systems. A large issue is that finding large high-quality data sets can be troublesome and sometimes even non-existent; this is where the need for DaFne was seen. DaFne is a platform that provides various mechanisms to facilitate data synthesization, which researchers use across multiple universities and institutions to generate more data for Smart City research [PKS22]. The DaFne platform qualifies as an AI-as-a-service platform containing multiple microservices that provide different strategies for data generation. Users can share data sets and data evaluation methods for research purposes. Such a platform needs a resilient architecture in order to function well. Previous works surrounding this platform contain well-thought-out architecture propositions, with most of them not fully implemented.

1.1 Objectives

The objective of this work is to collect and critically analyze previous work surrounding the DaFne platform and to provide suggestions to improve the current state of the platform and to shape a clear direction moving forward. It is currently being developed and maintained by a team of five developers. This work will contain suggestions that can be applied to large-scale applications, but all decisions will be made with a strong focus on maintainability and extensibility to reduce overall complexity. The work is divided into three parts. The first part provides an overview of the current state of the platform and describes existing pain points and issues. The second part discusses previous work on improving the architecture and gives suggestions defining the future path to follow during the development of the DaFne platform. In the final part, these propositions will be demonstrated by reimplementing parts of an existing microservice in the DaFne system.

1.2 Structure

This thesis begins with the **Introduction** 1, which presents the motivation and idea behind this work. The **Foundation** 2 chapter lays the foundation by providing important domain knowledge required in the following chapters, specifically for requirements engineering. In addition, it introduces the different data generation methods provided by the DaFne platform. In the **Requirements** 3 chapter, nonfunctional requirements are determined, serving as the basis for evaluating potential architecture improvements made in this work. The **Architecture Analysis** 4 captures and visualizes the existing state of the DaFne platform, offering brief descriptions and highlighting important nuances of each component within the architecture. In addition, it documents the technologies used, the current deployment strategies, and the communication patterns applied. It builds on this analysis by introducing improvements based on the importance of several nonfunctional requirements. Lastly, a decision is made to serve as the basis for future work. They are intended to define a standard for the entire platform with the goal of increasing the maintainability of the DaFne platform. The **Implementation** 5 exemplifies some of these suggestions by re-implementing the reproduction service. This section also presents additional optimization strategies while discussing potential weaknesses and made compromises. Finally, the **Summary & Outlook** 6 consolidates the proposals made in this work and provides an outlook on future work and the future of the platform itself.

2 Foundation

The following chapter introduces the DaFne platform. It contains a short overview of its services, users, and potential use cases. These will be used to further demonstrate the value proposition and allow analyzing functional and nonfunctional requirements. This will lay the foundation for the decisions made in later parts of this work.

2.1 DaFne

The core idea behind DaFne is to provide users with the ability to synthesize tabular data. Only authorized users are allowed to access this platform, which is mainly used by researchers across multiple universities within Germany. The platform will not be available for public use. Users have the ability to select existing data sets and upload their own data sets to use as input data for the selected synthesizing methods. After users have decided on their input data, they can select multiple methods depending on their needs. These methods will generate additional data to increase the quality and quantity of information within a data set. The generated data can then be run through a set of evaluation methods to filter out added information that does not meet a certain quality threshold. Training and saving of machine learning models including their weights and parameters is also possible, this allows users to reuse and customize existing generation models to their requirements.

2.2 Data Generation Methods

The following section briefly describes the data generation methods used within the DaFne system.

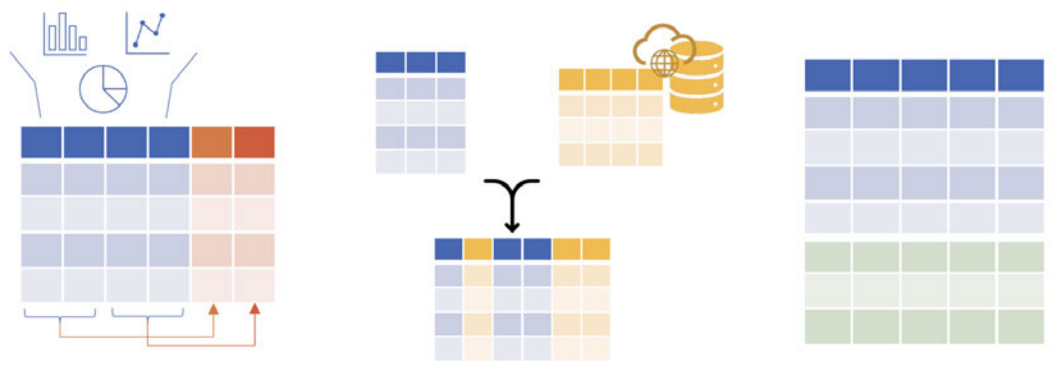

(a) Rule Based (b) Fusion (c) Reproduction

Figure 2.1: DaFne: Data Generation Methods [PKS22].

2.2.1 Data Reproduction

The reproduction of data is done by feeding existing input data into the so-called generation models, which will analyze the correlations between input data sets and try to generate new rows of data. These newly generated rows can then be run through multiple evaluation methods to ensure that the output meets certain quality standards. This method can be used to populate existing data with additional high-quality rows and therefore increase the quantity and diversity of the data set.

2.2.2 Data Fusion

Data Fusion describes the process of combining multiple existing data sets into one larger data set with the purpose of increasing the quantity and diversity of the data. These data sets can be taken from available public sources and can also be enriched with privately collected data. To exemplify this, one could take a public data set of Manhattan's weather history and combine it with a personally collected set of weather data from Buffalo to gain a more diverse and realistic set of weather data spanning the entire state of New York.

2.2.3 Rulebased Generation

This strategy revolves about picking certain rules as the basis to generate a new set of data without the need for any existing data. Such rules can be composed by combining common logical operators such as and, or, equals and less/greater than for each column. Such rules can also be used to generate custom data types, but are mainly used for the generation of numerical values, dates, or character sequences. Rule-based data generation can not only be used to create entire data sets from scratch but also to enhance already existing data by adding additional columns.

2.3 Potential Use Cases

2.3.1 Neighborhood Generation

The generation of synthetic neighborhoods enables the modeling of urban data using artificial intelligence. Users can create artificial neighborhoods based on an empty layout and a trained machine learning model. This approach can be applied to modeling cities or districts for various use cases. For example, it can help identify optimal locations for green spaces or public transportation hubs to improve accessibility and sustainability. The method generates recommendations for specific infrastructure components based on a given topology.

3 Requirements

Before making any judgment about the current architecture, it is important to understand the requirements of the platform and re-evaluate them. As functional requirements have been addressed in previous work independently for each service, this work will focus solely on nonfunctional requirements that have a strong impact on the proposed architecture.

Maintainability: The architecture should be designed in a way that is easy to understand. This requires maintaining clean and concise code to allow potential new developers to have a easier time understanding the architecture and adding new features. The time spent fixing errors, changing and adding features and for new developers to gain familiarity with the code base should be reduced as much as possible. Especially when critical issues arise, having a simplified architecture can drastically decrease the time to understand and address the issues.

Scalability: The architecture should allow horizontal scaling of individual services depending on the load they are experiencing. As this platform is not planned to be released to the public, concurrent user numbers remain much more limited, causing the load on the entire platform to be more predictable. With this in mind, the architecture should be able to respond to high loads by automatically scaling the respective service. For example: Efficient resource allocation for computationally expensive jobs is more important than having a highly scalable messaging architecture. The platform should be able to handle 1000 daily users on average.

Extensibility: The architecture should allow easy extensibility, as the need to add new data generation services could come up in the future. Through standardization and a decrease in complexity, developer efforts will be reduced and allow for more seamless integration of newer services.

Security: Only authenticated users should be allowed to access the platform. As the platform provides access to hardware clusters and potentially personal data, it is important to protect it from bad actors and reduce attack vectors. Since the platform itself is not open to the public and access is strictly monitored, potential abuse is easier to identify.

Availability: The system does not need to be available at all times. Jobs can not exceed a limit of running for a set amount of time to reduce blocking resources.

Reliability: The system performs computationally expensive tasks for prolonged periods of time. It should ensure that each job does not crash unexpectedly to be resource efficient. It can also be a cause of frustration if a job does not finish properly or if generated data are lost.

Integration: Machine learning models and the infrastructure needed to run them need to be made available through endpoints across the system.

Handling of Bottlenecks: Jobs that rely on machine learning models or other computationally expensive methods for data generation can introduce bottlenecks depending on resource availability. Also, crucial components within the architecture, such as the API gateway, should not become unavailable during times of increased load.

4 Architecture Analysis

4.1 Frontend

The frontend is composed of multiple micro frontends with the idea to encapsulate and separate different business logic. Micro frontends follow the same idea as microservices, splitting a monolithic frontend into smaller frontends that teams can work independently on [Mez22]. They can be implemented using a variety of different technologies each, depending on requirements. In this case, multiple frontend frameworks such as Vue.js [You], React.js [MP] and Next.js [Vera] were used. As for the deployment, the most recent version of each repository is built and the resulting build-files are served through a AWS Simple Storage Solution (S3) [Sera] bucket, secured with AWS Identity and Access Management (IAM) [Serb] and using AWS CloudFront Content Delivery Network (CDN) [Serc] to deliver the files from the nearest edge locations.

Micro frontends

The frontend is split up into four micro frontends all within the same repository, which are being served using AWS Simple Storage Solution (S3) Buckets. Every micro frontend uses different frameworks, folder structures and routing. This makes it harder to maintain due to the varying frameworks and non-standardized structure within the repository. The suggested approach would be to agree on a standardized structure for each repository as well as agreeing on using the same frameworks where applicable. As two out of the four frontends are using Next.js as their frontend framework, the other two should be adapting the same technology. Next.js offers poly-repository micro frontend support which could help with developing and maintaining them.

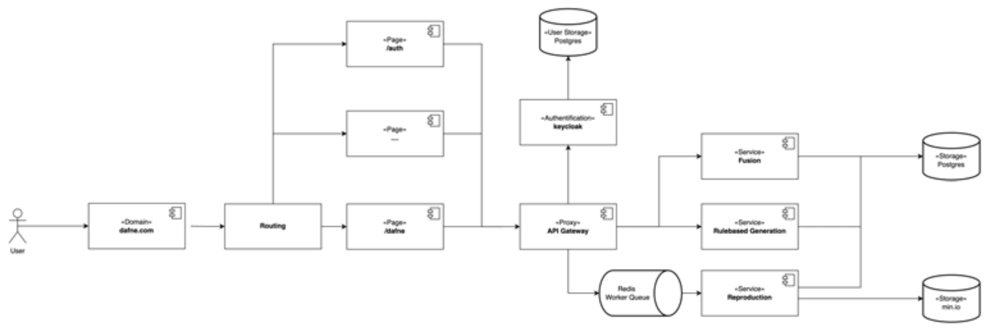


Figure 4.1: A broad overview of the DaFne system.

Poly-repository pattern

Having a poly-repository structure can help with creating a clear separation of concerns, decrease the repository size and most importantly allow for multiple teams to work independently, as each frontend has a respective repository. In addition, the advantage of allowing teams to work independently is definitely important in large-scale companies or large teams in general. As DaFne is an on-going research project, the team members are likely to change over time. Without having a separate environment to develop in, especially while experimenting during research projects, interfering with each other is much more likely. Having multiple repositories would give developers more freedom and reduce the likelihood of interfering each other. Whilst these advantages seem pretty convincing at first, it is important to consider potential downsides when going with the poly-repository structure. Having multiple repositories comes with an increase in project complexity, not only due to spreading the frontend across multiple repositories but also increasing efforts for managing them. For example, each repository has to be set up, which usually contains configuring pipelines, such as CI/CD, potentially containerizing the application and installing dependencies. As all of the frontends are supposed to be using Next.js, it will be common for dependencies to overlap. Another issue with micro frontends is maintaining standardization, especially since a Next.js application usually is made up of hundreds of components. Being able to share already created components helps with creating a consistent user experience across all different domains within the DaFne platform. A standardized tech-stack across domains also allows developers to support each other in different domains, as the project structure, technologies and components are already familiar.

Decision

Maintaining a micro frontend architecture is not exactly needed, as the use-cases for each frontend are similar and do not benefit from varying technology stacks. Also, micro frontends come with the incurred cost of increased bundle size, potentially increasing loading times [Mez22]. As for DaFne, the advantages of realizing a poly-repository approach for each frontend do not outweigh the disadvantages drastically because of three main factors: the small team size, the small code base being small enough to fit into a singular repository without driving up the complexity and the currently non-existent need for being able to deal with exponential growth as a platform.
While interference during research is possible, sticking to certain guidelines, such as local development, working in separate branches and not deploying breaking changes into the productive environment can avoid most issues.
Whether the increased management effort that comes with applying the poly-repository outweighs possible interferences needs to be further discussed within the team. As of now, sticking with the mono-repository approach should suffice.

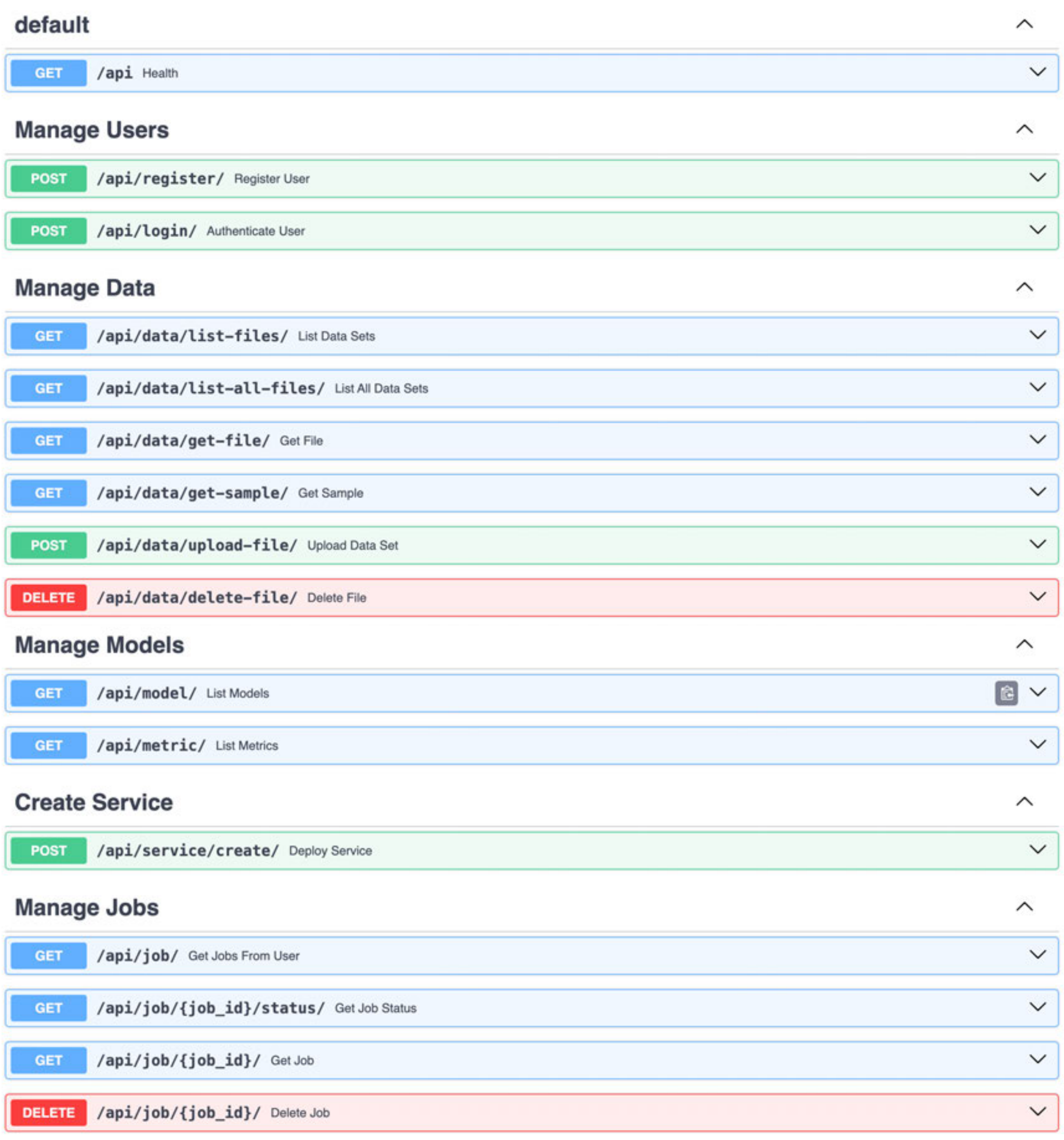
4.2 Backend

4.2.1 Gateway

The API Gateway serves as the entry-point to the system and acts as a secure reverse proxy and load balancer, hiding the running microservices behind a standardized API [TLP18]. The API Gateway pattern allows for easy extensibility as new services can be on-boarded by simply adding new endpoints and connecting them to the endpoints of the new service. It also serves as a guarantee, that service implementations can be exchanged easily without having to modify the outgoing endpoints. Having a centralized point of entry also allows for easier organization and maintainability as everything is in one place. The use of this pattern can also come with downsides such as having a single point of failure and introducing potential bottlenecks. However, this can be addressed by the use of a load balancer and horizontally scaling the API Gateway service, allowing better load distribution and automatic scaling. If the number of microservices behind an API Gateway are steadily increasing, different routing alternatives could come in handy as the API Gateway may become increasingly complex and harder to maintain [TLP18]. For authentication the whole system relies on Keycloak, an open source identity and access management solution, which is responsible for maintaining users and their roles across multiple areas within the system and offers integration into various services [Key24]. For storing credentials, Keycloak additionally uses a separate Postgres [Pos24] instance. Through the gateway several endpoints are exposed which can be seen in Figure 4.2.

4.2.2 Storage

DaFne enables users to access a wide range of machine learning models and data sets. This requires a robust and scalable storage infrastructure to efficiently manage, organize, and retrieve these resources. For data set storage and retrieval, the reproduction service relies on using min.io, an efficient and simple object storage solution that specializes in AI data storage. For storing and providing machine learning models, the solution differs. Machine learning models are provided as containers that are stored in a (private) Docker Registry. A list of available models, together with additional information, such as model type and parameters, is being stored with reference to the container inside a Postgres database. As soon as a model is registered within this database, it is made available to all users. In addition, a Redis [Red] instance is there to serve as a basic worker queue, which currently is only implemented for the reproduction service. Lastly, as mentioned in the previous section, Keycloak uses an additional Postgres instance to store user credentials. Postgres and min.io are currently storing data sets, parameters, and weights together with the metadata of each machine learning model. Each model is in a separate Docker container, exposing functionality through REST endpoints. A model manager and data manager exist as an abstraction away from the storage solutions; however, there are still direct calls made to each storage solution without making use of the existing abstraction. This can be observed within the sidecar container [Kub24c] inside of a UserPod. Here, the sidecar accesses the data through the respective storage instance, without making use of the existing abstraction by calling its endpoints instead. This is considered a bad practice as it tightly couples the UserPod with the used storage technology. Any changes made to the storage schema or the technology used could lead to breaking changes in the reproduction service. In addition, the use of shared data storage for multiple microservices is considered an antipattern that is commonly found within monolithic applications [Pac18]. However, for microservices, a common approach is to apply the database per service pattern 4.3. Although data storage sharing is an antipattern, the premise of DaFne is to allow users to combine different methods of data generation to modify existing data sets. Therefore, it would be rather impractical



default

GET /api Health

Manage Users

POST /api/register/ Register User

POST /api/login/ Authenticate User

Manage Data

GET /api/data/list-files/ List Data Sets

GET /api/data/list-all-files/ List All Data Sets

GET /api/data/get-file/ Get File

GET /api/data/get-sample/ Get Sample

POST /api/data/upload-file/ Upload Data Set

DELETE /api/data/delete-file/ Delete File

Manage Models

GET /api/model/ List Models

GET /api/metric/ List Metrics

Create Service

POST /api/service/create/ Deploy Service

Manage Jobs

GET /api/job/ Get Jobs From User

GET /api/job/{job_id}/status/ Get Job Status

GET /api/job/{job_id}/ Get Job

DELETE /api/job/{job_id}/ Delete Job

Figure 4.2: Endpoints of the API Gateway

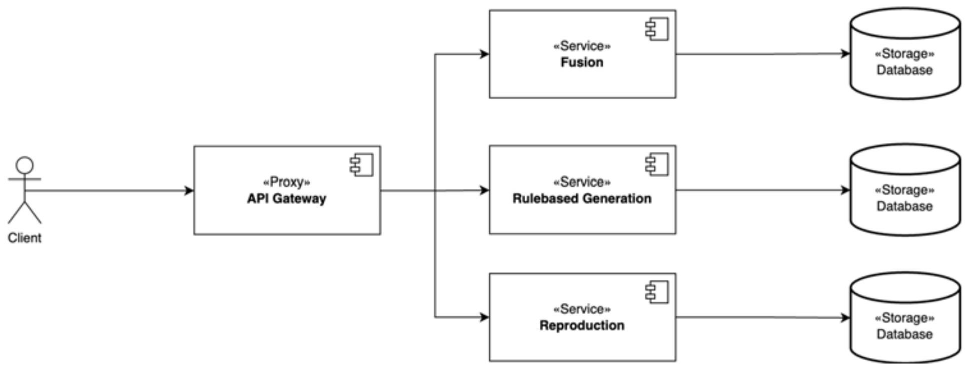


Figure 4.3: Visualization of database per service pattern inside of DaFne.

to manage replication of newly created data sets across all individual storage services. This would not only introduce additional network load, but would additionally be very impractical. Having separate data storage for each microservice would also result in large chunks of redundant data, which takes up to three times as much storage space.

Decision

As for the data storage within DaFne, the approach of abstracting data access away through a separate service is the right choice. Having shared data storage across multiple microservices is considered an antipattern. Not applying this pattern within DaFne allows for reduced complexity, easier maintainability, and less resource usage through sharing storage space. However, it comes with the downside of having cross-domain coupling within the architecture itself. Horizontally scaling data storage services and introducing replication can prevent bottlenecks and increase reliability within the system. In the future, each service should access the data through the intended service and should never access the storage solutions directly 4.4.

4.2.3 Reactive Frameworks

So far, all services responsible for data generation are implemented using different frameworks and programming languages. While some of them are still prototypes or serve as a proof of concept, settling on a standard and sticking to the same programming language

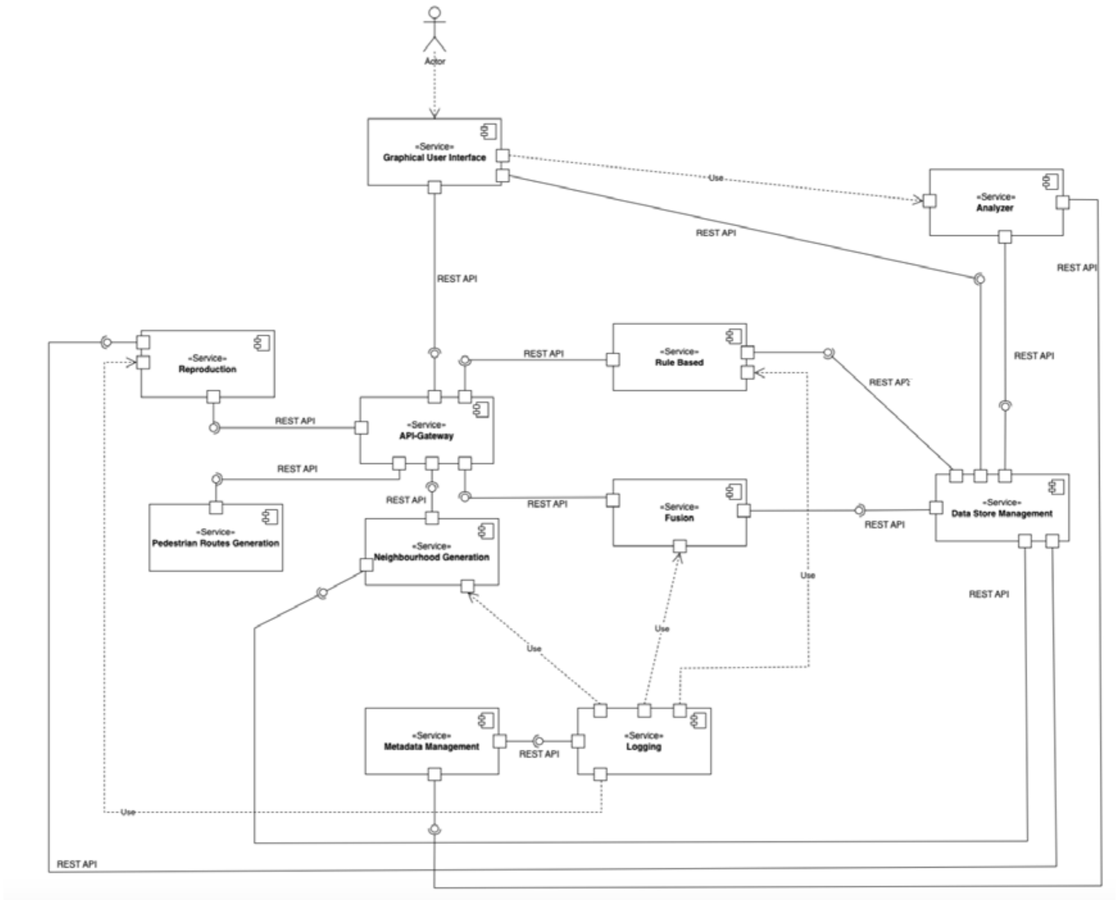


Figure 4.4: Visualization of the shared data storage approach [Kra24]

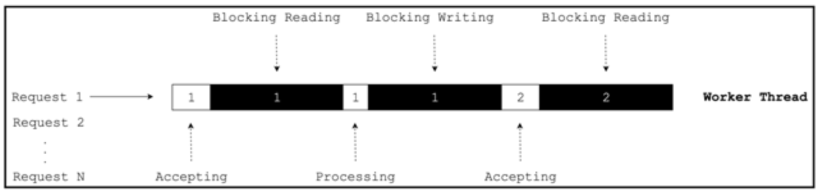


Figure 4.5: Blocking request and response processing [DL18].

and frameworks is important. The only reason to use different technologies is if they provide clear advantages that outweigh the additional complexity, for example using Python for a data-science heavy service.

Reactive Frameworks

The use of reactive frameworks has become more common in systems handling thousands of requests per second (RPS) requiring low-latency and higher throughput. Reactive frameworks allow for much higher throughput per instance due to worker threads being able to handle multiple concurrent requests in contrast to the common one request per thread model [Sha18]. Usually, the requests are queued up and processed sequentially. During processing, blocking operations, such as waiting for I/O tasks, take up the majority of processing time which could be used for other requests to start processing. Resuming with the initial request once the blocking action completed [DL18]. This comparison becomes more evident when comparing the figure 4.5 and 4.6. In 4.5 the black chunks represent blocking operations within a thread. Here requests are handled sequentially, whereas in the non-blocking approach the worker thread can continue processing new requests, while the blocking operation is ongoing. This leads to more work being done while maintaining lower resource usage on average. Popular reactive frameworks include Spring WebFlux [Fra] and Vert.x [Verb].

While this sounds promising, there are a few challenges and downsides with using this approach. Due to the shift to the reactive paradigm, the data flow within a code base becomes much harder to follow and may cause confusion for developers unfamiliar with the system or paradigm. In addition, debugging also becomes much more difficult as reactive frameworks are callback based. Stack traces do not provide exact locations of exceptions that happened inside the code. The ability of being able to sequentially debug code will be lost and make it more difficult to understand the context. It is worth noting that there are solutions to make debugging reactive applications easier [DL18].

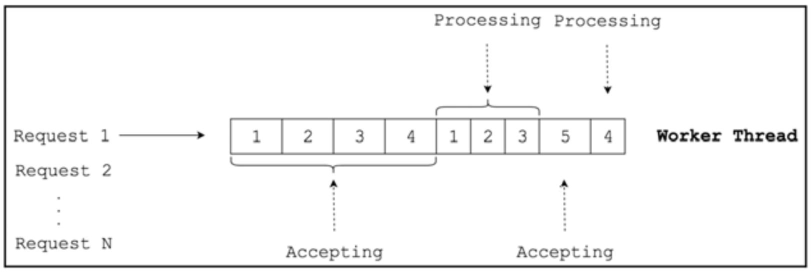


Figure 4.6: Asynchronous non-blocking request processing [DL18].

Estimating requests per second (RPS)

As mentioned earlier, the platform is not planned to be available to the public. With this in mind, we can make some estimates for the needed RPS. Assuming 2000 users are given access to the platform, out of which 10% (200 users) will be using DaFne per day. Most of these users will be accessing the platform during normal business hours and won't be scheduling more than 5 jobs per day. This makes up for a total of 1000 jobs needing to be scheduled and completed. Most requests are made to create jobs, whilst some are there for requesting status updates and fetching results, resulting in approximately 3000 requests in total, spread out over 12 hours or 43.200 seconds. This would leave us with roughly 0.07 RPS or around 4 requests per minute (RPM). Even if all 2000 users were to use this platform every day the usage might peak at around 0.7 RPS and 42 RPM.

Little's Law

The previously estimated numbers raise no concerns when it comes to not being able to handle the expected load. Non-reactive systems using Tomcat [Tom24] and Spring Boot together with Virtual Threads from Java 21 are able to handle thousands of requests per seconds depending on the requested action and hardware used [Oh25], [Cun25]. However, another important aspect is the average time it takes to complete a request. If each requests takes 1 second on average to complete, having an incoming 2 requests per second will cause back-pressure to build up, assuming only 1 thread is available. Before making a decision about where to use reactive frameworks, following data points need to be collected for each service:

- Average request completion time

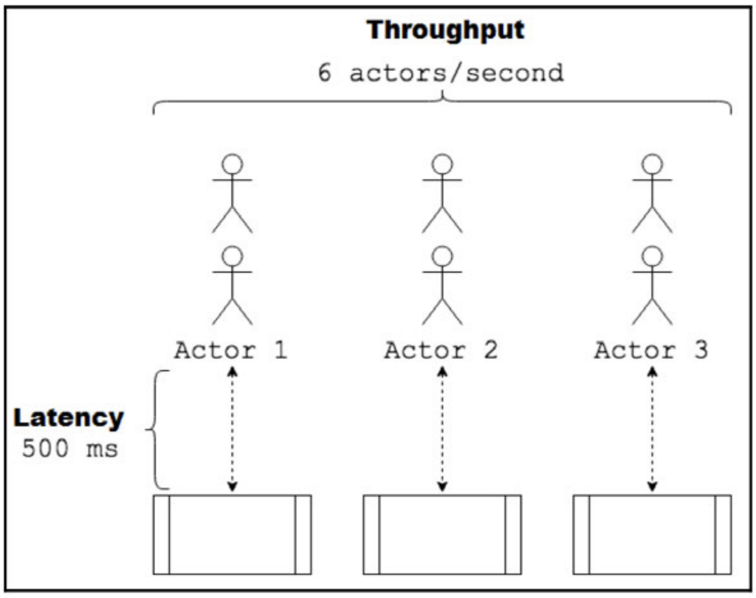


Figure 4.7: Ideal simultaneous process [DL18].

- Verify the estimated load (requests per second)
- Maximum number of available threads per server instance
- Determining if requests are computationally expensive

After gathering these data points, we can apply Little's Law to calculate the number of required threads per instance:

$$N = X * R \tag{4.1}$$

In which the number of requests processed simultaneously (N) equals the throughput or the requests per second (X) times the average latency or response time (R). As can be seen in figure 4.7.

However, the result of this calculation is not always applicable to real-world applications, as there is usually some kind of concurrent access to shared resources within each thread (for example, shared database access), which requires additional resources and potentially causes a bottleneck. These are taken into consideration inside *Ahmdahl's Law* and the *Universal Scalability Law* which are explained in detail in [DL18].

Decision

Optimally, running stress tests simulating high-load situations and measuring request completion time would suffice to make a judgment about the performance implications of using a non-reactive approach. As for this work, the orchestrator component will be implemented non-reactively to allow for further research into the limits of a non-reactive approach. The other services, such as the data fusion service, which run computationally expensive jobs, might make more use of the current reactive implementation until the above data points have been collected. Lastly, all backend services should be implemented using Java or Python to reduce the number of different frameworks and dependencies used.

4.2.4 Reproduction

The reproduction service, as described in 4.8, is responsible for providing the functionality necessary to allow users to upload public and custom data sets, select a generation model, and evaluate the quality of the generated data. The orchestrator is a component responsible for preparing jobs by creating and managing UserPods on a per-user basis. A UserPod is a Kubernetes pod, which is the smallest deployable unit in Kubernetes that contains multiple Linux containers [Kub24b]. A UserPod is made up of a generative container and multiple evaluation container, all of which can be communicated with through different ports. A UserPod can only contain a single generative container, which contains a generative, pre-trained machine learning model to serve as a generation strategy for the data reproduction. However, multiple evaluation containers can run within a single UserPod. The evaluation containers contain an implemented evaluation method, chosen by the user depending on the quality requirements and data set type. Judging the output data quality, it filters out rows that do not match the requirements. The data sets and models inside the containers will be shared through a mounted shared volume across running UserPod instances to reduce bandwidth usage [AM23]. In case a model is not available within the shared volume space, the sidecar container fetches the needed resource from external sources and loads the model container from the Docker Registry. The reference to the model container is stored inside the Postgres database. It also provides the interface for uploading high-quality data sets, updated model weights, and parameters to the respective data storage. Users can schedule multiple jobs to be run within the same UserPod through the API Gateway. This publishes a new message

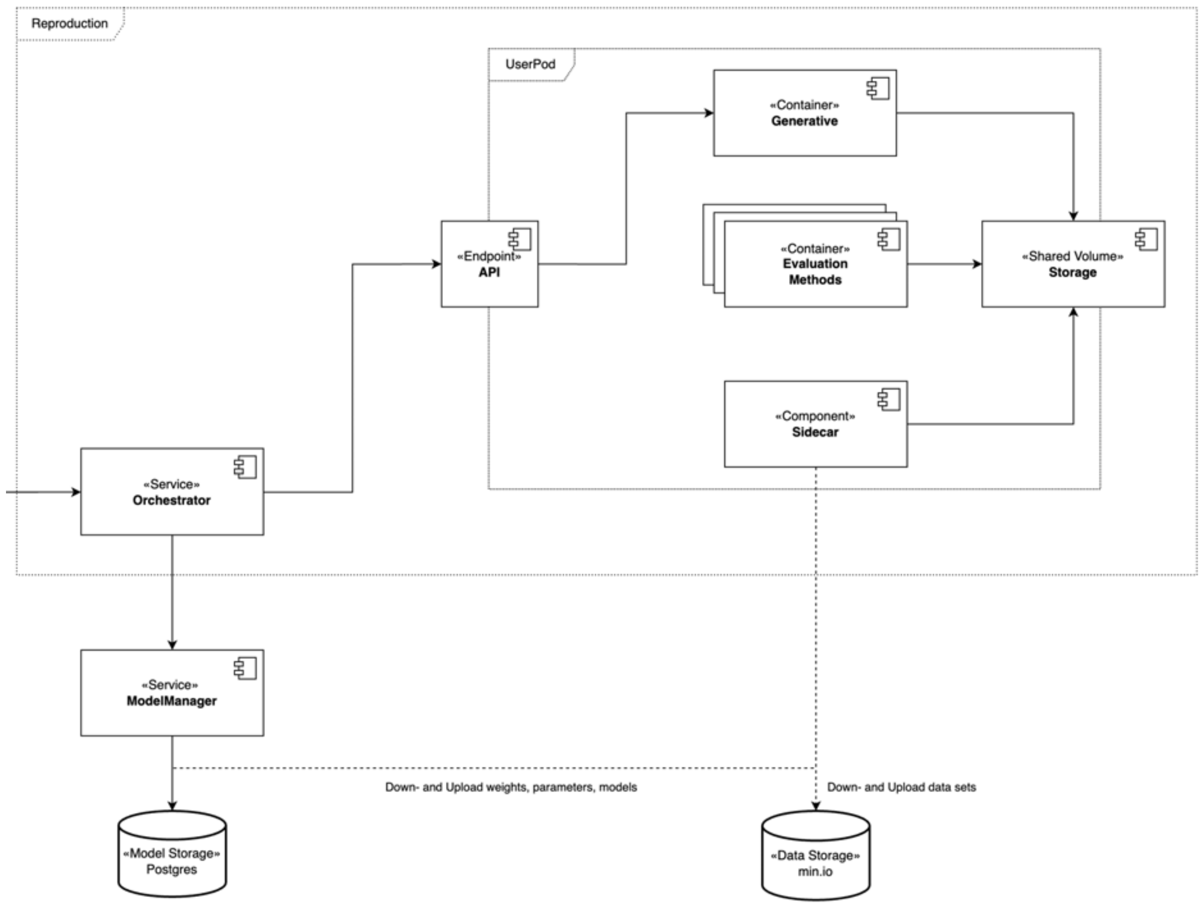


Figure 4.8: A visualization of the current reproduction service.

within the Redis worker queue, which is then passed down to the subscriber, the sidecar container, within the UserPod instance. During execution of the job, the user can query each job's status using separate requests. In terms of implementation, the orchestrator is implemented in Java using the reactive framework Reactor [Rea24]. The orchestrator is inside a docker image and is deployed using Kubernetes.

Decision

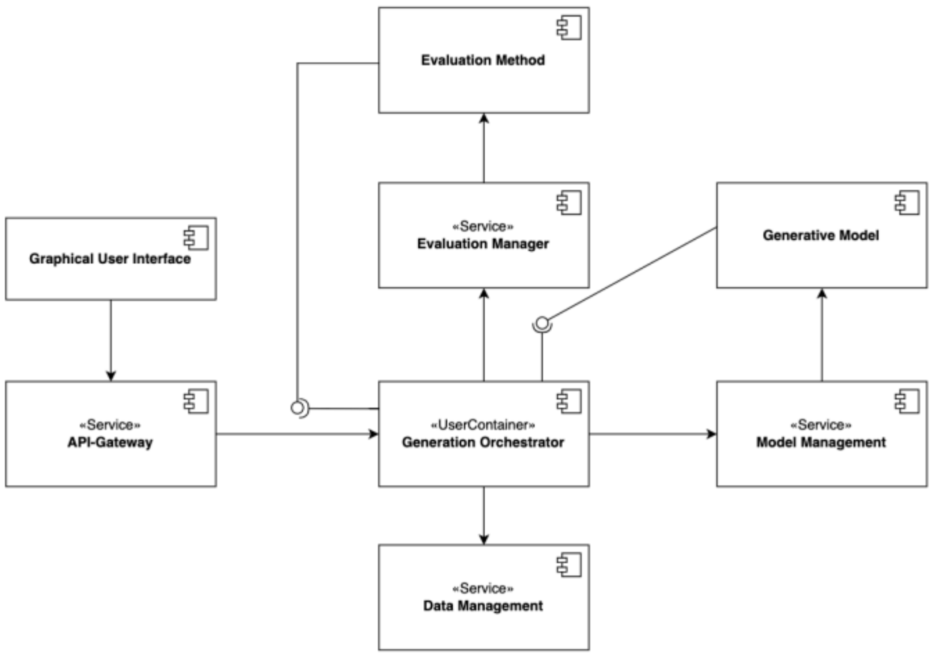


Figure 4.9: Proposed components within the reproduction service [Kra24].

The reproduction service is the most complex service in terms of business logic, it combines multiple domains, providing data generation and evaluation in a single service. Currently, evaluation access is limited to the reproduction service only. However, evaluation should also be made available for other generation methods. For this, another service responsible for the evaluation of the generated data sets should be implemented. This eliminates the need to manage a UserPod altogether, reducing the size and complexity of the code within the reproduction service. This suggestion is also proposed by [Kra24] in 4.9. It is also important to remove all direct access to storage within the Reproduction service to reduce the tight coupling of this service to the data storage solution, especially when an abstraction away from the data storage already exists. Additionally, the current implementation contains dead code, which further drives the need for refactoring. The following action items should serve as guidelines during refactoring:

- Separate data generation and evaluation and move the evaluation into a separate service.
- Remove the UserPod concept and manage simple pods, jobs, or other alternatives mentioned in 5.
- Make use of the data storage service within the Sidecar component 4.8, instead of accessing the data storage directly.
- Implement the service using a non-reactive framework to reduce code complexity and increase maintainability.
- Avoid over-abstraction, as it complicates the code.
- Remove dead code from the current implementation.

4.2.5 Fusion

The fusion service contains four components. Firstly, an external API connector is implemented to allow fetching additional data from outside, such an API could provide additional weather data for a certain location. The data sets obtained can then be joined within the fusion component, which provides several strategies for combining data sets. These fusion strategies are grouped under the term JOIN-algorithms [Moh23]. The Comma-Separated Values (CSV) File Handler assists in parsing CSV files into the internal format used within the fusion component and parses the output data back into the CSV format. While the fusion jobs are running, users can query the state of their jobs through an endpoint provided through the API Gateway. This state is handled by the respective component, as shown in 4.10. The entire service is written in Kotlin using Quarkus, a native Kubernetes framework for Java and Kotlin applications [Hat24]. It also offers an in-house reactive messaging framework which the fusion service also makes use of. The components defined within the fusion service are concise and do not need drastic refactoring. The implementation is easy to understand and has a well-thought-out structure. The documentation is clear and contains a detailed class diagram, making it easy to understand the interactions between each class [Moh23].

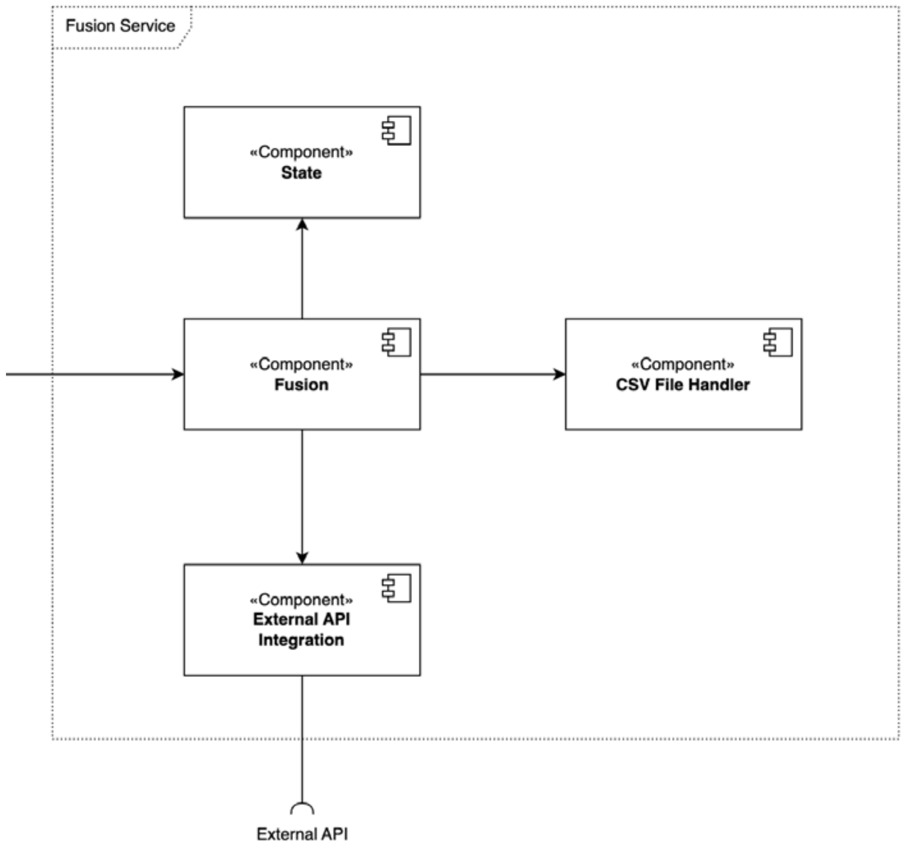


Figure 4.10: Overview of the components inside the Fusion service

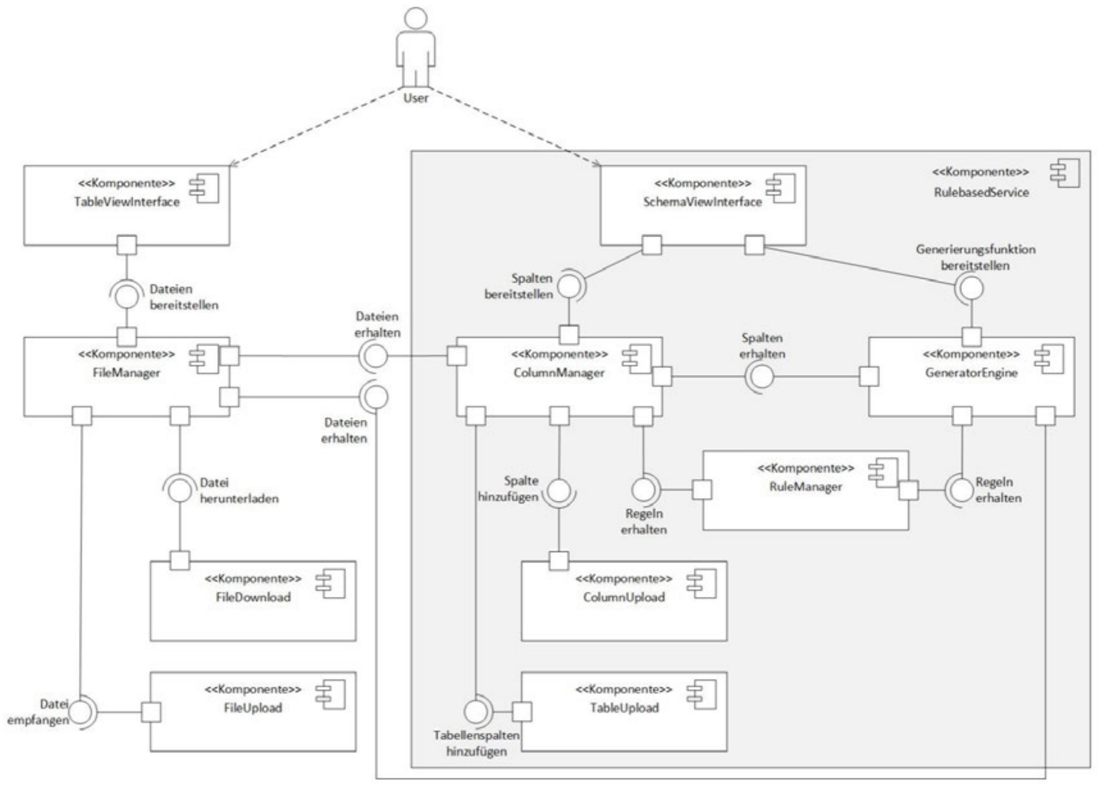

User
<<Komponente>> TableViewInterface
<<Komponente>> SchemaViewInterface
<<Komponente>> RulebasedService
Dateien bereitstellen
Spalten bereitstellen
Generierungsfunktion bereitstellen
Dateien erhalten
Spalten erhalten
<<Komponente>> FileManager
<<Komponente>> ColumnManager
<<Komponente>> GeneratorEngine
Dateien erhalten
Datei herunterladen
Spalte hinzufügen
Regeln erhalten
<<Komponente>> RuleManager
Regeln erhalten
<<Komponente>> FileDownload
<<Komponente>> ColumnUpload
<<Komponente>> FileUpload
<<Komponente>> TableUpload
Datei empfangen
Tabellenspalten hinzufügen

Figure 4.11: Overview of the components inside the rule-based generation service [Bee23].

Decision

This service has a concise implementation and does not need major refactors. Although some improvements can be made, they merely serve as suggestions and as such are not time sensitive.

- Standardize and document API endpoints by applying the OpenAPI Specification [Swa24], using the API Gateway as reference.
- Migrate the implementation to a full Java implementation.
- Migrate to a non-reactive framework such as Spring Boot.

4.2.6 Rule-based Generation

The components that are needed to implement a rule-based data generation service are shown in 4.11. The key component is SchemaViewInterface which allows users to generate new columns with the ColumnManager by providing the various generation methods through the GeneratorEngine. The user can upload single columns or all columns from a table through either the ColumnUpload or TableUpload component. Existing tables can already be downloaded and uploaded by users through the FileManager component. These can serve as basic configurations for common or more specific data sets and allow users to export the generated tables for further processing and use. The microservice is implemented in Python using the open source Django framework [Fou] and has been containerized in Docker for deployment in Kubernetes. The current implementation serves as a prototype.

Decision

Since the implementation is still a prototype, there are many tasks pending. Using Python for this service makes sense, as it facilitates implementation and is already used in other sections of the platform, through which no additional complexity is added.

- Create a finalized implementation of the rule-based generation service based on the component diagram 4.11.
- Create a proper frontend for the rule-based generation using Next.js and TypeScript.
- Write in-depth unit and integration tests, especially for handling edge cases.
- Containerize the new service and create deployment configurations, using existing services such as the reproduction service as a guideline.
- Standardize and document API endpoints by applying the OpenAPI Specification, using the API Gateway as a reference.

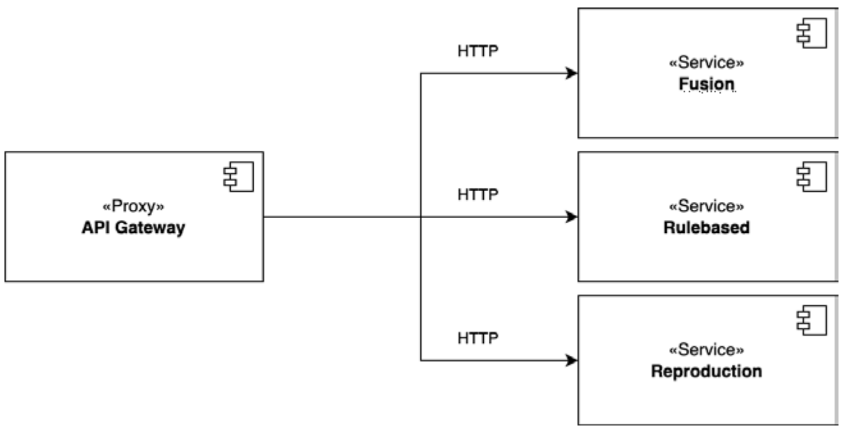


Figure 4.12: An overview of communication protocols used between services.

4.2.7 Communication

After a user sends an HTTP request to the API Gateway, depending on the endpoint, the request will be handled and additional calls to the respective services will be made. In microservice architectures, it is common to use asynchronous communication between systems or to schedule tasks [Bon16]. In this case, asynchronous communication has been implemented only within the reproduction service, where it serves as a work and result queue. For simplicity, Redis is used for the message queue. The orchestrator publishes jobs that are then consumed by the sidecar container. Once the jobs are completed, the result is written into a result queue which can then be consumed by the frontend. As suggested in the previous work [Fis24], the use of messaging for communication seems to be the preferable solution given the asynchronous nature of most communications that occur between the services. While this is indeed correct, it is important to also keep in mind the additional complexity it adds to the architecture and the less clear data flow resulting from implementing asynchronous communication. This could potentially impact efficiency in smaller development teams. A full REST communication between all services, providing standardized and well documented endpoints would reduce the complexity of the microservice architecture and make it easier to manage. It also results in a clearer data flow within the system [Kra24]. When it comes to HTTP, an API with JSON is the standard for synchronous communication. However, a common approach for internal synchronous communication, with regard to latency and data translation, is the use of binary traffic [Pac18]. Another important point is that using binary traffic is not tied to any specific technology, and changing the communication interface with this technology is extremely simple. [Pac18]. However, if using messaging is crucial for

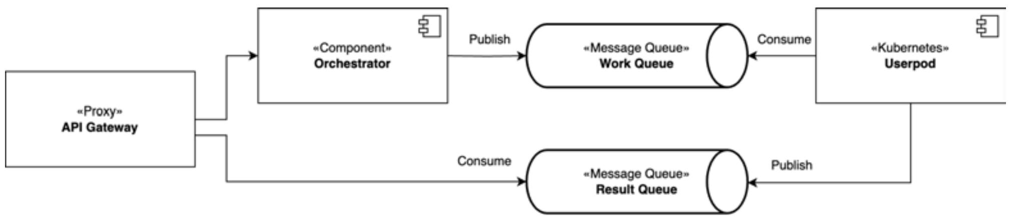


Figure 4.13: Message queue being used inside the reproduction service.

this architecture due to potential bottlenecks, then a good direction would be to use RabbitMQ. Due to its simplicity and high performance with up to 20.000 RPS per node, it gives a balance between simplicity and scalability. Whereas competitors such as Kafka can serve up to 100.000 RPS per node, which is five times more than RabbitMQ but comes at the cost of a much more complex system [Pac18]. Furthermore, such high throughput will not be needed in the near future.

Decision

The reproduction service is already making use of a message queue. However, it is only implemented within the orchestrator, resulting in a tightly coupled message queue being inaccessible from other services. The message queue should be extracted into a separate service with endpoints for consumers and producers. This means that each data generation service should maintain a separate topic within the message queue, responsible for holding scheduled jobs. Implementation within the reproduction service is done using Redis 4.13. Although this probably served as a proof of concept, it is important to find more robust alternatives. Spring Data Redis [Spr24a] offers straightforward integration for pub/sub messaging and streams [Spr24b] with very little configuration and supports imperative and reactive paradigms. However, Redis pub/sub lacks reliability as it stores messages in-memory. This means that messages could potentially be lost if the message queue becomes unavailable. Redis Streams offers support for message persistence by enabling the Redis Database (RDB) to take periodic snapshots of the memory and store it on disk. Potentially pairing this with replication should increase the reliability of messages being consumed. If this is still an issue, then it would make sense to migrate to RabbitMQ as it provides a persistence layer. Another advantage of using RabbitMQ is its in-built message confirmation, which allows detection of a processed message that has not resulted in a completed job. This could happen if the consumer becomes unavailable

	Redis Pub/Sub	**Redis Streams**	**Redis List**	**RabbitMQ**
Message Delivery	*At most once*	*At least once*	*At most once*	*At least / most / exactly once*
Persistence	*No*	*Possible*	*Possible*	*Yes*
Consumer Groups	*No*	*Yes*	*Yes*	*Yes*
Effort	*Low*	*Medium*	*Low*	*High*

Table 4.1: Comparing message queue implementations using Redis and RabbitMQ [Red], [Inc24]

during job execution. The message would then be re-add the message to the queue for further processing [Inc24].

4.3 Summary

The following is a list of action items regarding standardization and optimization for increased maintainability

- Migrate all existing micro frontends to Next.js with Typescript and configure the deployment in S3.
- (Optional) Split all existing micro frontends into separate repositories to reduce interference during independent research.
- Migrate all services to Java or Python with standardized API documentation using the OpenAPI specification [Swa24].
- Document each service's components in one location together with function and nonfunctional requirements, allowing developers to easily onboard to the project and make meaningful contributions.
- Refactor existing services to communicate with data storage services instead of accessing storage solutions directly.
- Run performance tests on both implementations of the reproduction service to determine the potential throughput of the non-reactive and reactive approach.
- Implement data storage service as an abstraction instead of managing data access separately in each service.
- Implement an evaluation service that allows for the calculation of metrics to judge the quality of data sets, as implemented within the reproduction service.

- Implement a simple message queue using integrated Spring Boot Redis List support for scheduling jobs and create one for each service.

This action plan should serve as a guide for shaping the future of the DaFne platform and will result in easier maintainability and a faster development process while reaching a more refined and standardized state within the platform. This will be exemplified in 5, where the suggestions made will be applied by re-implementing crucial components of the reproduction service and will provide guidance for the suggestions proposed in this chapter.

As discussed in the previous chapter 3, the suggestions focus on satisfying a specific set of requirements.

Maintainability: The architecture should be designed in a way that is easy to understand. This requires maintaining clean and concise code to allow potential new developers to have a easier time understanding the architecture and adding new features. The time spent fixing errors, changing and adding features and for new developers to gain familiarity with the code base should be reduced as much as possible. Especially when critical issues arise, having a simplified architecture can drastically decrease the time to understand and address the issues. The concerns regarding maintainability have been addressed in the frontend by reducing the amount of technologies used, implementing standardized user interface components and introducing TypeScript to ensure type safety, therefore also increasing code readability. As for the backend, writing a proper documentation for each service, especially for the reproduction service, will improve maintainability. Furthermore, separating the reproduction service into two individual services removes the need for UserPod management within the orchestrator. As shown in 5, this reduces the complexity and code size of the service as a whole. Further investigating the required communication patterns and the need for reactive frameworks reduces the complexity of the architecture.

Scalability: As throughput is also highly dependent on hardware, calculating exact metrics is difficult without proper stress testing. However, as shown in 4.2.3 and [Oh25] the expected user load should be easily handled by existing request handlers. Scalability is ensured through existing Kubernetes deployment configurations, allowing hardware-independent deployment within clusters and the ability to scale horizontally. If the above suggestions do not suffice, changes to the architecture are proposed that address potential scalability issues.

Extensibility: The architecture should allow easy extensibility, as the need to add new data generation services could come up in the future. Through standardization and a decrease in complexity, developer efforts will be reduced and allow for more seamless integration of newer services.

Security: Access control has already been implemented in existing services using Keycloak. This solution should be the standard for the future.

Availability: As reliability was chosen over availability within this architecture, availability drops can be expected. However, given the nature of DaFne, this should not create a large-scale impact.

Reliability: Reliability was defined as the guarantee of job completion. Job completion is addressed in 4.2.7 by implementing exactly-once semantics within the message queue. Enabling Kubernetes Jobs retry logic, as shown in 5, can further contribute to reliable job completion.

Integration: Machine learning models and data sets can be retrieved and uploaded via the data storage service. Integrating a variety of evaluation methods will be possible with the introduction of the suggested evaluation service. Further services can be integrated due to loosely coupled components and having clearer data flow within the architecture.

Handling of Bottlenecks: Components that can pose as potential bottlenecks are identified in the architecture analysis 4 together with suggestions to improve the mentioned components.

5 Implementation

5.1 Preparation

Before proceeding with the process of implementing the reproduction service and applying the refactoring mentioned, it is important to document the endpoints and understand each part of the code. This ensures that after refactoring, the service still contains the same features. For public endpoints, a look at the existing controllers is enough to determine the functionality and use cases that accompany it 5.1. Next, existing models need to be analyzed in conjunction with the functionality of each class, as there is a lack of clear documentation for either.

Endpoint	**Method**	**Function**	**Responsibility**
/health	*GET*	Returns a heartbeat to track service availability	Health
/work/create	*POST*	Creates an entry for the user in the work queue	Work Queue
/userpod/create	*POST*	Creates a UserPod with chosen models and writes a job to the work queue	User Pod & Jobs
/userpod/{jobId}	*DELETE*	Deletes the UserPod tied to an existing job	User Pod
/service/pods	*GET*	(Deprecated) Returns a list of all running pods from Kubernetes	User Pod
/service/yml	*GET*	Returns Kubernetes Deployment and Service YAML configuration files	Kubernetes

Table 5.1: Summary of endpoints present in the orchestrator component within the reproduction service.

Currently, the architecture of the reproduction service is fairly complex, as can be seen in the sequence diagram 5.1. The main reason for said complexity stems from trying to do the metric evaluation for generated data sets inside the reproduction service. This is outside the domain of a service responsible for data generation. In addition, evaluating the quality of the generated data can be beneficial for other generation methods. With the current implementation, this evaluation can only be used for the reproduction method. The evaluation will be migrated to a separate, loosely coupled service that allows the calculation of metrics on any type of generated data set, following the proposal made in [Kra24]. This will in turn reduce the complexity of the reproduction service and remove the need to manage specific UserPods and maintain a work queue for each. A simplified

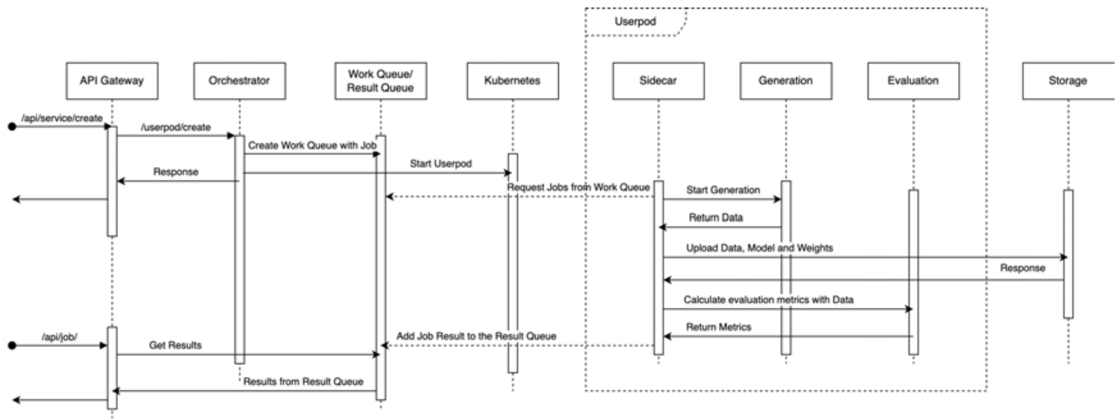


Figure 5.1: Sequence Diagram of generating data using the current reproduction service.

approach for the reproduction service is visualized in the following sequence diagram 5.2. The importance lies within the removal of the evaluation from the reproduction service and removing the concept of the UserPod entirely. The evaluation is going to be implemented as a separate service where users can select any available data set and perform metric analysis on it without having to perform data generation in the first place. Reducing management efforts further, as users will have to request the evaluation themselves. This chapter will address the limitation of this approach while guiding its implementation.

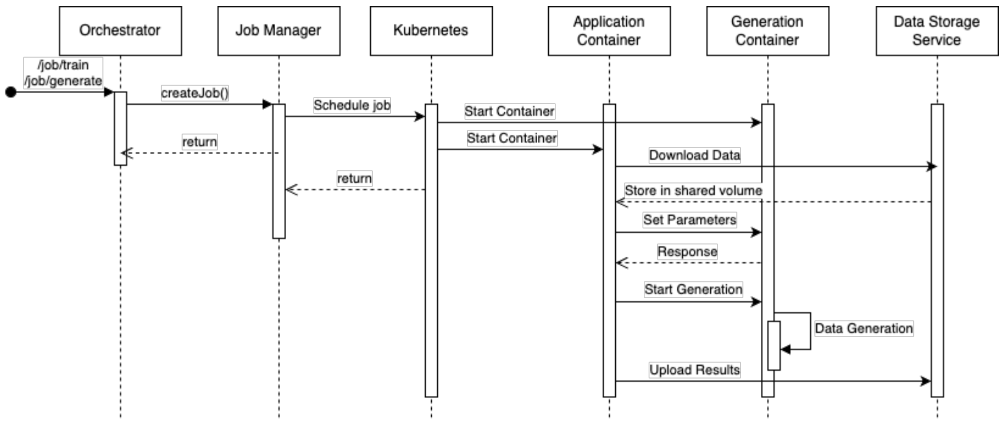


Figure 5.2: Sequence diagram of the simplified reproduction service without the evaluation component.

5.1.1 Functional Requirements

5.2 Orchestrator Implementation

The existing orchestrator implementation handles the creation of a UserPod, containing three different types of containers: a generation container, multiple evaluation containers, and a sidecar container. Upon the creation of a new UserPod, the orchestrator writes multiple job entries to its internally managed message queue, which will then be consumed by the sidecar container within the UserPod. This will cause it to start the generation and also handle the evaluation of the data after the generation process. The results will then be pushed to a queue of results 5.1. Since the refactored reproduction service should not include data evaluation, UserPods will no longer be necessary. However, the issue of running a job after creating a new pod containing the generation container persists. There are multiple options for interacting with the running container:

- Managing a job queue, where the orchestrator publishes a job and the container consumes the job from the front of the queue.
- The container runs a web server and exposes an endpoint for starting a job. This would include port forwarding and accessing the running pod by getting its IP address and the forwarded port.
- Create Kubernetes jobs to execute the data generation.
- Pass a command during Kubernetes pod creation to automatically run a job on startup.

The communication between the orchestrator and the running pod poses a problem: The pod needs to know which job it should process and then the user needs to be able to access the data generation results that lie inside the pod. So, in reality, we have three core functionalities that need to be supported:

- Pass the job including the request parameters to the Generation Container.
- Keep track of the job progress to allow user queries.
- Store the generated data and model weights.

Furthermore, there are multiple messages published into the work queue during UserPod creation. These messages are made up of multiple calls to different endpoints of the generation container, separated as start tasks and generation tasks:

```
      public List<TaskDTO> getStartTasks(Principal user,
          InstructionDTO instruction, UserPodDescription description)
          {
      return Stream.of(
                  List.of(storageTaskFactory.download(
                      instruction.getPaths().getDownload()
                  )),
                  List.of(generationTasksFactory.putParameters(
                      instruction, description)
                  ),
                  List.of(generationTasksFactory.fit(description)),
                  generationTasksFactory.loadModel(user,
                      instruction, description))
              .filter(Objects::nonNull)
              .flatMap(Collection::stream)
              .toList();
  }

  public List<TaskDTO> getGenerationTasks(InstructionDTO
      instruction, UserPodDescription description, int runNum) {
      return
          Stream.of(List.of(generationTasksFactory.train(description,
          runNum),
                      generationTasksFactory.saveModel(description,
                          runNum)),
                  evaluationTaskFactory.getMetricTasks(instruction,
                      description, runNum))
              .flatMap(Collection::stream)
              .toList();
  }
```

Listing 5.1: Existing implementation of the task scheduling within the reproduction service

5.3 Job Scheduling

5.3.1 Passing Job Parameters through Redis Store

Passing the job parameters to the container can be achieved by storing the parameters as key-value pairs in a Redis store. Using the key pattern `job:<job-id>` and storing the `pod-id`, `status` and `parameters` as values. The orchestrator will create the job entry and assign the `pod-id` to equal the created pod. Each pod is assigned a unique `pod-id` and will have to query the Redis store to find the job assigned to its `pod-id` with the status `CREATED`, change the status to `RUNNING` and start processing the job. Once the job is completed, the status should be changed to `DONE` or `ERROR`. This approach would allow the creation of an endpoint to check a job's status with minimal effort and introduce little overhead. The main downside to this approach is that the completion of a job will not cause the termination of a running pod. The orchestrator would need to have a separate worker thread to clean up unused pods after the assigned job has been finished. Also, retries for failed jobs have to be managed by the orchestrator.

The pod creation can be achieved by creating the specification of the pod and passing it to the `createNamespacedPod` function.

```
public V1Pod createPodSpec(String namespace, String podName,
    String containerName, String image, int port) {
   V1Container container = new V1ContainerBuilder()
      .withName(containerName)
      .withImage(image)
      .withImagePullPolicy("Never")
      .withPorts(new V1ContainerPortBuilder()
         .withContainerPort(port)
         .withProtocol("TCP")
         .build())
      .build();

   V1PodSpec podSpec = new V1PodSpec()
      .containers(Collections.singletonList(container))
      .overhead(null);

   V1ObjectMeta metadata = new V1ObjectMeta()
```

```
            .name(podName)
            .namespace(namespace);

        return new V1Pod()
            .apiVersion("v1")
            .kind("Pod")
            .metadata(metadata)
            .spec(podSpec);
    }

    public void schedulePod(String namespace, V1Pod pod) throws
        ApiException {
        try {
            V1Pod createdPod = coreV1Api.createNamespacedPod(namespace,
                pod).execute();
            System.out.println("Pod scheduled: " +
                createdPod.getMetadata().getName());
        } catch (ApiException e) {
            System.err.println("Failed to schedule pod: " +
                e.getResponseBody());
            throw e;
        }
    }
```

Listing 5.2: Implementation of a pod specification using the Kubernetes Client Library

5.3.2 Managing a Job Queue and Result Queue

This is a similar approach to passing job parameters using the Redis Store. Instead of having an entry in a long list of key-value pairs, a separate queue with jobs is being managed for each pod. Additionally, a result queue exists that serves as storage for completed jobs. On completion of the job, an entry into the result queue is made from within the pod, which can then be accessed from outside the pod to fetch the results for each completed job. Although this approach easily allows multiple jobs to be processed during the lifetime of one pod, it creates a large overhead, as two queues are required from outside and within the pod. More flexibility is provided with the cost of increasing complexity and additional failure points.

5.3.3 Scheduling Jobs using Kubernetes Jobs

Using Kubernetes Jobs addresses the issue of having to clean up unused pods or pods that completed a job. A Kubernetes Job specification, in its simplest form, contains the container, in this case any generation container, and a console command to start the task. As all generation containers host a web server with the `/start` endpoint, the command could be a CURL or execute a Python script. The resulting data set would then be uploaded to the S3 bucket by communicating with the data storage service. The job status can be fetched by querying Kubernetes as each job will have a status such as `SuccessfulCreate` or `Completed` [Kub24a]. While this approach reduces management efforts by automatically deleting pods after job completion and supports error handling including restart policies, it requires significant restructuring of the existing containers and prevents multiple jobs being run on the same pod, therefore increasing bandwidth usage and execution time.
The following YAML snippet shows what a basic job configuration could look like for the reproduction service. The job name will match the assigned `job-id` to facilitate querying the job status in Kubernetes. The specified container contains a standardized wrapper that accepts and parses the command-line arguments provided to an application container and a generation container as a sidecar container. For this, the CTGAN model is being used for demonstration purposes [XSCIV19]. To allow usage of various generation containers, the application container will take a parameter `image` to allow users to choose a given image from a set of images of allowed generation. The backoffLimit specifies the number of retries before a job will appear as failed; in this case, we will allow one retry to prevent extensive resource usage from jobs that throw exceptions multiple minutes into the generation process. `ttlSecondsAfterFinished` describes the number of seconds to wait before automatic deletion of the pod after completion of the job. The environment variables will be used to pass the URL to the storage service to upload the generated data sets and updated weights.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: <job-id>
spec:
  backoffLimit: 3
  template:
```

```
  spec:
    restartPolicy: Never
    ttlSecondsAfterFinished: 100
    containers:
    - name: application-container
      image: <docker-registry>/application-container:latest
      command: <command-with-parameters>
      env:
        - name: STORAGE_SERVICE_URL
          value: <storage_service_url:port>
      volumeMounts:
          - name: shared-data
            mountPath: /opt/app/data
    initContainers:
      - name: generation-container
        image: <docker-registry>/ctgan-restapi-docker-image:latest
        restartPolicy: Always
        ports:
          - containerPort: 8080
            protocol: TCP
        volumeMounts:
          - name: shared-data
            mountPath: /opt/app/data
  imagePullSecrets:
    - name: <secret>
```

Listing 5.3: An example of a job specification inside Kubernetes

The above solution using Kubernetes Jobs provides a simple and maintainable solution by using native Kubernetes features, eliminating the need to implement retry logic and perform pod cleanup within the orchestrator. While other solutions provide more flexibility by, for example, allowing multiple jobs to be run on the same pod, they come with the cost of increasing code size and complexity. Early optimizations are not needed. Especially since the number of concurrent users is not expected to grow, as the platform has limited accessibility. As mentioned in the approach described above, some adjustments need to be made; this means creating a container to handle the business logic. This container must satisfy the following requirements:

5.4 Container Implementation

5.4.1 Requirements of the application container

Training the model on a custom dataset:

- Download the custom data set to train the model on
- Set model specific parameters, such as `epochs` and `generator_dim`
- Fit the model with the provided data set
- Save the model to the model storage using the storage service

Loading a pre-trained model to generate synthesized data:

- Download the pre-trained model weights from the model storage through the storage service
- Load the weights into the model
- Generate synthesized data
- Upload the generated data to the storage by communicating with the data storage microservice

It becomes evident that the requirements specified above define two separate areas of concern. While one focuses on preparing a machine learning model for data generation by training it on a specific data set, the other focuses on the data generation itself by loading such a pre-trained model from the model storage. This clear distinction opens up the possibility of further breaking down the service into two services. One is responsible for the training the models, whilst the other handles the generation. However, this separation can also be presented by providing two endpoints:

- `/job/train`: Responsible for training and storing the model, taking in the model, training parameters, and the data set identifier
- `/job/generate`: Responsible for data generation and storing the generated data, taking in a pre-trained model, and generation parameters such as the number of rows to generate.

Since the machine learning models are implemented in Python, the wrapper will also be implemented as a Python module to easily interact with the model module. The following implementation uses the existing sidecar container implementation for the basis, which can also be specified in the job specification.

Once each of the containers has been implemented, the last step is to implement the creation of jobs, using the functions implemented to create pods. In order to create a job, first, a pod specification needs to be defined. This is also the place where the arguments need to be passed to the application and sidecar container. In this case, the sidecar container is going to be the Python webserver that exposes interactions with the CTGAN model. The application container is responsible for handling business logic, which includes downloading data sets and weights, interacting with the machine learning model, and storing the results. This separation results in easier maintainability as changes to business logic will not require changes within each machine learning model image, only to the single application container image. Since the containers are separate instances, a Kubernetes volume can be set up as a shared volume to allow shared access to downloaded data. This will serve as the target location for the downloaded files and allow the machine learning model to access the required data sets and weights for further processing. The mount path will be defined as `/opt/app/data`, as this is the default input data path defined within the existing generation container.
The following implementation of the application container handles both model training and data generation by interacting with the generation container provided and fulfills the core requirements specified in 5.4.1.

```python
import argparse
import json
import requests
import urllib
import os

from data_storage import get_dataset, get_weights, upload_dataset,
    upload_model

generation_container_url =
    os.environ.get('GENERATION_CONTAINER_URL',
    'http://localhost:8080')
```

```
data_storage_service_url = os.environ.get('DATA_STORAGE_SERVICE_URL')
shared_directory_path = os.environ.get('SHARED_DIRECTORY_PATH',
    '/opt/app/data')

parser = argparse.ArgumentParser(
                prog='Generation Worker',
                description='Calls a running DaFne machine learning
                    model container',
                epilog='For more information visit dafne.info')

parser.add_argument('-j', '--job', help='Determines the job type to
    execute, valid values are "training" and "generation"',
    required=True)
parser.add_argument('-p', '--parameters', help="Training: JSON
    String of the configuration parameters passed to the machine
    learning model; Generation: The generation job configuration such
    as the number of rows to generate.", required=True)
parser.add_argument('-d', '--dataset', help="The path to the dataset
    resource, required for a training job")
parser.add_argument('-w', '--weights', help="The path to the weights
    of a pre-trained model, required for a generation job")
parser.add_argument('-o', '--output', help="The name of the
    resulting output artifact", required=True)

def main():
   args = parser.parse_args()

   parameters = json.loads(args.parameters)

   if args.job == 'training':
      training_job(args.dataset, parameters, args.output)
   elif args.job == 'generation':
      generation_job(args.weights, parameters, args.output)
   else:
      print('Invalid job type, please see help for valid job types')

def training_job(dataset: str, parameters: dict, output: str):
   # Download the dataset into the storage folder
   dataset_path = get_dataset(data_storage_service_url, dataset,
       shared_directory_path)
   if dataset_path == None:
      return

   # Train the machine learning model to the downloaded datasets
   model_path = start_training(dataset_path, parameters, output)

   # Upload the resulting weights to the storage
   upload_model(data_storage_service_url, model_path)


def generation_job(weights, parameters, output):
   # Download the dataset into the storage folder
   weights_path = get_weights(data_storage_service_url, weights,
      shared_directory_path)

   if weights_path == None:
      return

   # Execute the generation job on the pre-trained model
   dataset_path = start_generation(weights_path, parameters, output)

   # Upload the resulting weights to the storage
   upload_dataset(data_storage_service_url, dataset_path)
```

Listing 5.4: Implementation of the main python module inside the application container

The module `data_storage` is responsible for the up- and download of data sets and weights by interacting with the data storage service. Although this service has not been implemented yet, a proposal for this service has been made in [Kra24] and the implementation of this service should be a priority. Not having any form of abstraction and directly accessing different storage technologies, such as min.io or AWS S3, can lead to tight coupling and cause maintainability issues. This also allows the implementation of the `data_storage` module being concise. As this implementation is only a proof-of-concept, authentication using Keycloak has not yet been implemented. The existing sidecar container stored in the `data-sidecar` repository already handles authentication which can be used to extend the functionality of this container. The implementation of the `data_storage` module assumes that the data storage service has already been implemented and outlines the idea of how communication with this service should work.

```
import requests

def get_dataset(url, dataset, output_path):
   try:
      response = requests.get(f"{url}/dataset/{dataset}",
         stream=True)
      response.raise_for_status()

      file_path = f'{output_path}/{dataset}'

      with open(file_path, 'wb') as file:
         for chunk in response.iter_content(chunk_size=8192):
            if chunk:
               file.write(chunk)

      return file_path

   except requests.exceptions.RequestException as e:
      print(f"Failed to download dataset: {e}")

def get_weights(url, weights, output_path):
   try:
      response = requests.get(f"{url}/weight/{weights}", stream=True)
      response.raise_for_status()

      file_path = f'{output_path}/{weights}'

      with open(file_path, 'wb') as file:
         for chunk in response.iter_content(chunk_size=8192):
            if chunk:
               file.write(chunk)

      return file_path

   except requests.exceptions.RequestException as e:
      print(f"Failed to download weights: {e}")
```

```
def upload_model(url, model_path):
   try:
      with open(model_path, 'rb') as model_file:
         response = requests.post(f"{url}/weight", files=model_file)
         response.raise_for_status()
   except Exception as e:
      print(f"Failed to upload model: {e}")

def upload_dataset(url, dataset_path):
   try:
      with open(dataset_path, 'rb') as dataset_file:
         response = requests.post(f"{url}/dataset",
            files=dataset_file)
         response.raise_for_status()
   except Exception as e:
      print(f"Failed to upload dataset: {e}")
```

Listing 5.5: Implementation of the data storage module inside the application container

The above module communicates with the data storage service to fetch weights and data sets before starting and after completion of a job. It uploads the resulting artifacts to the respective storage solution. The downloaded files will always be stored on the shared volume, so the generation container can access the required files for further processing.

With these two modules complete, the foundation for the application container is implemented. The next step is to implement the generation of the Kubernetes job specification to connect the two separate containers and allow for dynamic processing of various generation and training jobs. The code below handles the creation of the training and generation job specifications. It supports dynamic configuration by passing arguments to the application container and choosing the image for the generation container.

```
   public void createTrainingJob(String namespace, String
       parameters, String dataset, String resultPath, String image,
       String sidecarImage, int port) {
      List<String> args = List.of("--job", "--training",
         "--parameters", parameters);
```

```
        List<String> sidecarArgs = List.of("--dataset", dataset,
            "--result", resultPath);

        V1PodTemplateSpec templateSpec =
            createPodTemplateSpec(namespace, image, sidecarImage, args,
            sidecarArgs, port);

        V1Job generationJob = createJob(namespace, "job",
            templateSpec);
        batchV1Api.createNamespacedJob(namespace, generationJob);
    }

    public void createGenerationJob(String namespace, String
        parameters, String weights, String resultPath, String image,
        String sidecarImage, int port) {
        List<String> args = List.of("--job", "generation",
            "--parameters", parameters);
        List<String> sidecarArgs = List.of("--weights", weights);

        V1PodTemplateSpec templateSpec =
            createPodTemplateSpec(namespace, image, sidecarImage, args,
            sidecarArgs, port);

        V1Job generationJob = createJob(namespace, "job",
            templateSpec);
        batchV1Api.createNamespacedJob(namespace, generationJob);
    }

    public V1PodTemplateSpec createPodTemplateSpec(String namespace,
        String image, String sidecarImage, List<String> args,
        List<String> sidecarArgs, int port) {
        V1Container applicationContainer = new V1ContainerBuilder()
            .withName("application-container")
            .withImage(image)
            .withImagePullPolicy("Never")
            .withArgs(args)
            .build();
        V1Container sidecarContainer =
            createContainerWithPort("sidecar-container", sidecarImage,
            port);

        V1PodSpec spec = createPodSpec(namespace,
            List.of(applicationContainer), List.of(sidecarContainer));
        V1ObjectMeta metadata = new V1ObjectMeta()
            .name("job-pod")
            .namespace(namespace);


        return new V1PodTemplateSpecBuilder()
                .withSpec(spec)
                .withMetadata(metadata)
                .build();
    }
```

Listing 5.6: Implementation of dynamic Kubernetes job generation using the Kubernetes Client Library

The above code snippet is responsible for dynamically scheduling Kubernetes jobs that execute training and data generation tasks based on user requests.

The `createTrainingJob` method is responsible for starting a training job within the Kubernetes namespace. It constructs a list of arguments that include the –job and –training flags, along with a set of user-defined parameters that guide the training process. Additionally, a secondary list of arguments is created for a sidecar container, specifying the data set to be used and the result path where the output should be stored. These arguments are then passed to the `createPodTemplateSpec` method, which generates a Kubernetes pod template containing both the main application container and a sidecar container. Using this template, a Kubernetes job is created using the `createJob` method, which is subsequently submitted to the cluster using the `batchV1Api.createNamespacedJob` method.

Similarly, the `createGenerationJob` method follows the same structure, but is tailored for generation tasks. Instead of training-related parameters, it sets up arguments for a generation job, including the –job and generation flags, as well as parameters that define the amount of data to generate. The sidecar container arguments specify the weights to be used in the generation process. As for the training job, the method constructs a pod template using `createPodTemplateSpec`, creates a Kubernetes job, and submits it to the cluster.

The `createPodTemplateSpec` method plays a crucial role in defining the execution environment for scheduled jobs. It constructs a primary container using the specified application image and sets the execution arguments. The method also configures a sidecar container using the `createContainerWithPort` function, ensuring that it is properly configured to communicate over the defined port. Then both containers are assembled into a pod specification using the `createPodSpec` method. To ensure proper identification within the cluster, a metadata object is created to assign the job a certain `job-id` for status queries and a namespace. Finally, the complete pod template is built and returned as a `V1PodTemplateSpec` object.

By structuring job creation in this way, the implementation ensures that training and generation tasks can be dynamically scheduled, parameterized, and executed within a Kubernetes cluster.

The orchestrator must accept and handle incoming requests by providing the required endpoints. The `/userpod` endpoints mentioned in 5.1 are now deprecated, as the concept of UserPods no longer exists and has instead been replaced with the implementation of Kubernetes jobs. The `/work/create` endpoint is replaced by the `/job/train` and `/job/generate` endpoints. In order to track the state of a job, the endpoint `/job/{job-id` will respond with the status of a given `job-id`.

```java
@RestController
@RequestMapping("/job")
@Log4j2
public class ModelController {
    @Autowired
    private KubernetesClient kubernetesClient;

    @PostMapping("/train")
    @ResponseBody
```

```
public ResponseEntity<String> train(@RequestBody TrainingJobDTO
    trainingJobRequest) {
   try {
      String parameterString = new
          ObjectMapper().writeValueAsString(
         trainingJobRequest.getParameters()
      );
      kubernetesClient.createTrainingJob(kubernetesClient.getNamespace(),
          parameterString, trainingJobRequest.getDataset(),
          "<path-to-result>", "application-container",
          trainingJobRequest.getImage(), 0);
      return ResponseEntity.accepted().body("");
   } catch (JsonProcessingException ex) {
      Logger.getLogger(ModelController.class.getName()).log(Level.SEVERE,
          null, ex);
      return ResponseEntity.badRequest().body(<exception>);

   }

}

@PostMapping("/generate")
@ResponseBody
public ResponseEntity<String> generate(@RequestBody
    GenerationJobDTO generationJobRequest) {
   try {
      String parameterString = new
          ObjectMapper().writeValueAsString(
         generationJobRequest.getParameters()
      );
      kubernetesClient.createGenerationJob(kubernetesClient.getNamespace(),
          parameterString, generationJobRequest.getModel(),
          "<path-to-result>", "application-container",
          generationJobRequest.getImage(), 0);
      return ResponseEntity.accepted().body("");
   } catch (JsonProcessingException ex) {
      Logger.getLogger(ModelController.class.getName()).log(Level.SEVERE,
          null, ex);
      return ResponseEntity.badRequest().body(<exception>);
```

```
        }
    }

    @GetMapping("/{jobId}")
    @ResponseBody
    public String getById(@PathVariable("jobId") String jobId) {
        return
            kubernetesClient.getJobStatusMessage(kubernetesClient.getNamespace(),
            jobId);
    }
}
```

Listing 5.7: Implementation of the controller responsible for handling job creation and status queries inside the orchestrator

5.5 Summary

The above implementation reduces the complexity within the reproduction service by eliminating the UserPod concept and separating the service into two individual services. Alternatives to an internal work and result queue have been identified and proven to be applicable in the given context. Choosing Kubernetes Jobs helped further reduce complexity by relying on Kubernetes to handle job completion and retry logic. Although this implementation is not complete, it gives an idea about implementing the core components of the service. One topic that has not been handled in this chapter is the result retrieval. The result retrieval can be implemented by querying the data storage service for the specified output data set name. Another approach could be to store the status together with a field that points to the resulting artifact in object storage and expose endpoints to update the job status in persistent storage through the data storage service. Furthermore, thorough logging should be introduced into the application container to allow precise status messages and assist during debugging and bug fixing.

6 Summary and Outlook

6.1 Summary

This work focuses on two research questions with the goal of improving maintainability of a data synthesis platform in a small development team. The first question is about identifying the components within the existing architecture and highlighting issues. It involves identifying tightly coupled components, complex code sections, and other areas that need improvement. Reading through previous works documenting the requirements and ideas served as a strong foundation throughout this work. The second question discusses best practices and determining compromises between decreasing complexity, code size, and resource usage at the cost of latency, throughput, and availability. Also removing optional features if they cause unnecessary complexity. This leads to a set of recommendations designed to guide the future development of this platform.

In order to answer these questions, the first step was to analyze the existing architecture. This required gaining an in-depth understanding of the architecture and the data flow across all microservices. However, due to non-existent or incomplete documentation, understanding the system became a challenging task. During this stage, the importance of improving the maintainability became much more evident. Capturing the architecture was the result of reading existing work regarding the development of the platform. This also involved reading through the code base itself and gaining crucial information about deployment and communication between the implemented microservices.

The knowledge gathered throughout the documentation process emphasized existing issues with the platform as a whole. In Chapter 5 4, each component within the architecture was described and inspected, followed by a list of action items that make up the final decision for each component. These decisions are based on best practices and the discussion of alternative solutions with the potential to reduce complexity within the system. At the same time, highlighting advantages and disadvantages of each alternative to allow further discussion in the future. In more ambiguous sections of the system, such as the communication, the functional and nonfunctional requirements had to be defined to properly match the users' needs. The summary of the architecture analysis provides a list of high-priority tasks regarding future changes to be made to the system.

In chapter 6 5, the previous suggestions are exemplified by guiding through an implementation of the most complex service within the DaFne platform, the reproduction service. This implementation follows ideas and suggestions from an earlier work [Kra24] and applies the suggested changes from the previous chapter to further demonstrate them. This implementation only serves as a foundation by addressing the most crucial components of the reproduction service and fulfilling core requirements, while still allowing for further discussions in future work.

6.2 Outlook

The issues mentioned during this work significantly impact the maintainability of the DaFne platform and thereby also reducing the developer's efficiency when making changes to an existing service. It further highlights the importance of reducing technical debt, creating and updating documentation as early as possible and making compromises between features and code complexity, especially for non-crucial features.

The next step is for the changes to be applied to the platform. As such, future work can finish implementing the updated reproduction service. This involves piecing together the components as described in 5, adding authentication using Keycloak and creating a deployment configuration for Kubernetes. A performance comparison between each implementation should be made, for example, by running extensive stress tests. This process involves determining system throughput, latency, and availability to allow making a data-based decision on the usage of reactive frameworks within the DaFne platform. For this, the rule-based generation and data fusion service should also be part of the stress test. Documenting these results and getting an estimate for the breaking point of the platform can serve as a foundation for future decision making in architecture discussions.

Another work could focus on creating the data storage service, as suggested in [Kra24]. An abstraction away from direct access to storage solutions, such as min.io and S3, which are being used as model and dataset storage. This also requires modification to all existing services and would reduce the tight coupling introduced by direct access to each storage method. By introducing this service, independence from the storage technology can be guaranteed and give more freedom for future changes due to technical limitations or changing requirements. Other advantages such as central error handling and easier implementation of replication contribute to this decision.

For work focusing on the frontend architecture, a full migration to Next.js with Typescript using a standardized component library should be made. The importance here lies in architectural and stylistic redesigns, meaning the project needs to be planned thoroughly by wireframing pages and creating stylistically fitting components, focusing on requirements of each existing micro frontend. Based on this, reusable components and layouts can be identified and be broken down. This will automatically result in a more finalized look of the DaFne interface, while still maintaining separation of concerns inside the repository. A component library such as shadcn [sha] could reduce the implementation effort; however, depending on the customization needed for the controls, it could make more sense to implement the components from scratch.

Doing more research on potential bottlenecks within the system, specifically focusing on the API gateway, can provide insights into the necessity for additional load balancing and even the introduction of a message queue. Here it would make sense to discuss the configuration and implementation of a message queue using RabbitMQ, focusing on fail-over, retry logic and ensuring each message is being delivered exactly once.

Bibliography

[AM23] Sebastian Gedigk Alexander Menk. Creating an architectural draft for a data-synthesization platform. Technical report, HAW Hamburg, 2023.

[Bee23] Mareile Beernink. Protoypical implementation of rule-based data generation. Technical report, HAW Hamburg, 2023.

[Bon16] Jonas Bon. *Reactive Microservices Architecture*. O'Reilly, 2016.

[Cun25] Gonçalo Trincao Cunha. Reactive vs. Synchronous Performance Test — dzone.com. https://dzone.com/articles/spring-boot-20-webflux-reactive-performance-test, 2025. [Accessed 09-02-2025].

[DL18] Dokuka, Oleh and Lozynskyi, Igor. *Hands-On Reactive Programming in Spring 5: Build cloud-ready, reactive systems with Spring 5 and Project Reactor*. Packt Publishing, 2018.

[Fis24] Marius Fischmann. Identification of a suitable communication pattern between microservices of a data generation platform for artificial intelligence, 2024.

[Fou] Django Software Foundation. Django - the web framework for perfectionists with deadlines.

[Fra] Spring Framework. Spring webflux.

[Hat24] Red Hat. Quarkus - supersonic subatomic java, 2024.

[Inc24] Broadcom Inc. Consumer acknowledgements and publisher confirms | RabbitMQ, 2024.

[JPN+20] Abhinav Jain, Hima Patel, Lokesh Nagalapatti, Nitin Gupta, Sameep Mehta, Shanmukha Guttula, Shashank Mujumdar, Shazia Afzal, Ruhi Sharma Mittal, and Vitobha Munigala. Overview and importance of data quality for machine learning tasks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3561–3562, New York, NY, USA, 2020. Association for Computing Machinery.

[Key24] Keycloak. Open source identity and access management, 2024.

[Kra24] Tom Krause. Design of a software architecture for the integration of use cases into a platform for data synthesis. Technical report, HAW Hamburg, 2024.

[Kub24a] Kubernetes. Kubernetes Jobs documentation, 2024.

[Kub24b] Kubernetes. Kubernetes Pods documentation, 2024.

[Kub24c] Kubernetes. Sidecar container concept, 2024.

[Mez22] Luca Mezzalira. *Building Micro-Frontends: Scaling Teams and Projects, Empowering Developers*. O'Reilly Media, 2022.

[Moh23] Abdullah Al Mohammad. Design and implementation of a generic data fusion service. Technical report, HAW Hamburg, 2023.

[MP] Inc Meta Platforms. React.

[Oh25] Daniel Oh. Demystifying Virtual Thread Performance: Unveiling the Truth Beyond the Buzz — dzone.com. `https://dzone.com/articles/demystifying-virtual-thread-performance-unveiling`, 2025. [Accessed 09-02-2025].

[Pac18] V.F. Pacheco. *Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices*. Packt Publishing, 2018.

[PKS22] Olaf Zukunft Pamela Kunert, Tom Krause and Ulrike Steffens. A platform providing machine learning algorithms for data generation and fusion - an architectural approach. Technical report, HAW Hamburg, 2022.

[Pos24] Postgres. PostgreSQL - The world's most advanced open source database, 11 2024.

[Rea24] Project Reactor. Project reactor, 2024.

[Red] Redis. Redis.

[RHW21] Yuji Roh, Geon Heo, and Steven Euijong Whang. A survey on data collection for machine learning: A big data - ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1328–1347, 2021.

[Sera] Amazon Web Services. Amazon s3 - cloud object storage.

[Serb] Amazon Web Services. Aws identity and access management (iam).

[Serc] Amazon Web Services. Cdn cloud service - amazon cloudfront.

[sha] shadcn. Shadcn component library.

[Sha18] Rahul Sharma. *Hands-On Reactive Programming with Reactor: Build reactive and scalable microservices using the Reactor framework.* Packt Publishing, 2018.

[Spr24a] Spring. Pub/sub messaging :: Spring data redis, 11 2024.

[Spr24b] Spring. Streams :: Spring data redis, 11 2024.

[Swa24] SmartBear Software Swagger. Openapi specification, 11 2024.

[TLP18] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. In *International Conference on Cloud Computing and Services Science*, 2018.

[Tom24] Apache Tomcat®. Apache tomcat®, 2024.

[Vera] Vercel. Next.js.

[Verb] Eclipse Vert.x. Eclipse vert.x.

[XSCIV19] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. *Modeling tabular data using conditional GAN.* Curran Associates Inc., Red Hook, NY, USA, 2019.

[You] Evan You. Vue.js - the progressive javascript framework | vue.js.

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

[redacted]

Ort	Datum	Unterschrift im Original