Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Implementing Variational Quantum Algorithms as Compositions of Reusable Microservice-based Plugins

Matthias Weilinger

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	M.Sc. Philipp Wundrack, M.Sc. Fabian Bühler

Commenced: April 19, 2023

Completed: October 19, 2023

Abstract

With its transformative processing capabilities, Quantum computing has ushered in a new computational era, presenting unparalleled opportunities and intricate challenges. One potential beneficiary of this quantum revolution is the Digital Humanities. With quantum computing, the field has the potential to enhance its quantitative analysis dramatically. QHAna, the Quantum Humanities Analysis tool explicitly designed for Quantum Digital Humanities, emerges as a pivotal system. This thesis focuses on enhancing QHAna by integrating variational algorithms and paving the way for Variational Quantum Algorithms in a modular manner. The objective is to encapsulate components of variational algorithms as distinct, interchangeable plugins, ensuring adaptability and enabling end users to adapt the algorithms. Addressing challenges like robust plugin communication and intuitive user experience, the research delves into this modular framework's design, implementation, and evaluation. Beyond the immediate application to Variational Quantum Algorithms, the insights and methodologies derived here lay the foundational groundwork for future modular system designs in the quantum computing domain.

Contents

List of Figures

List of Tables

Listings

Acronyms

DH	Digital Humanities. 15
OF	Objective Function. 20
QDH	Quantum Digital Humanities. 15
QHana	Quantum Humanities Analysis Tool. 7
RAMP	Reusable Microservice-based Plugin. 17
REST	Representational State Transfer. 17
UI	User Interface. 17
VQA	Variational Quantum Algorithm. 15

1 Introduction

The dawn of quantum computing has revolutionized the landscape of computational research. With its unparalleled processing capabilities and the potential to solve problems deemed impractical for classical computers, quantum computing heralds a new era of possibilities [Sho]. However, with these possibilities come challenges that demand innovative solutions, particularly in harnessing the power of quantum algorithms for diverse applications.

One domain that stands to benefit immensely from quantum computing is the Digital Humanities (DH). Traditionally, the humanities have been viewed through a lens of qualitative analysis. However, with the advent of digital tools and methodologies, a quantitative dimension has emerged, enabling researchers to analyze vast datasets, uncover patterns, and derive insights that were previously unavailable [BL19]. With its inherent strengths, Quantum computing offers the potential to elevate the DHs to new heights, enabling analyses of unprecedented complexity and depth [Bar22].

Enter QHana [BBH+22]. Initially conceived and meticulously crafted for Quantum Digital Humanities (QDH), its applicability has evolved, proving valuable not just for DHs but for broader quantum computing applications. The foundational architecture of the system is both robust and adaptable. A logical extension of this framework is the integration of variational algorithms, particularly Variational Quantum Algorithms (VQAs), in a modular fashion. The objective is to encapsulate components of variational algorithms as modular and interchangeable plugins within QHana. Such a modular approach promises enhanced adaptability towards a user-centric platform where researchers can precisely tailor their quantum optimization strategies to their needs.

However, the path to realizing this vision is full of challenges. As plugins are essentially microservices, QHana is a distributed system that brings its own challenges. Integrating optimization as multiple plugins demands a robust communication mechanism, ensuring seamless interaction and data sharing. Furthermore, the user experience must remain intuitive, allowing both quantum novices and experts to harness the full potential of the modular quantum optimization framework.

This work embarks on a journey to navigate those challenges. It aims to design and implement a modular framework within QHana, delving into the intricacies of the unique plugin-based architecture of QHana, the nuances of quantum optimization algorithms, and the challenges of ensuring seamless interaction between distinct components. Moreover, the implications of this research extend beyond the limits of VQA in QHana. By pioneering a methodology for VQA plugin interactions within QHana, this work lays the foundation for a broader paradigm of modular system design in the QDH.

The thesis undertakes a comprehensive journey through modular VQAs within QHana. Chapter 2 lays the groundwork by delving into the foundational principles and concepts that underpin the entire thesis. Chapter 3 sharpens the focus, presenting a clear and concise problem statement and outlining the objectives aimed to be achieved. With a clear understanding of the problem at hand, Chapter 4 dives deep into the methodology. The chapter explores the strategies and approaches employed to develop and evaluate the modular framework within QHana. Chapter 5 unveils the architectural blueprint of the modular framework. Moving from design to implementation, Chapter 6 delves into the details of the framework's implementation. In Chapter 7, the design and implementation are put to the test. The architectural design's efficacy and real-world implementation are measured through rigorous evaluations, drawing insights from the results. Chapter 8 delves into a thorough analysis of the findings and checks if set problems are adequately solved. To place the work in a broader context, Chapter 9 explores the landscape of related research. Finally, Chapter 10 brings the journey to an end, summarizing the resulting framework and looking ahead to future possibilities and roads for further extension.

2 Background

The study of QHana and its advanced plugin interactions is rooted in foundational principles underpinning its functionality. This chapter aims to offer a comprehensive introduction to these principles, encompassing key areas such as optimization algorithms, quantum computing, VQAs, Representational State Transfer (REST), and the core architecture of QHana. By delving into these foundational topics, readers will gain the necessary context to understand the innovative approaches adopted in this thesis.

2.1 QHana

QHana was conceived as a specialized application in the domain of DH and has since evolved into a versatile platform for quantum computing applications. Its primary design allows users to explore various machine-learning algorithms on designated datasets. While the primary vision of QHana is to assess the potential advantages of quantum algorithms within the DH community, the rise and cloud availability of quantum computers [Cas17] further underscore its relevance and timeliness.

QHana is designed to be extensible, allowing the integration of new data sources and quantum algorithms as plugins. Usually, plugins are built for specific applications, limiting their reusability in other contexts. Moreover, an application's plugins must be developed in the same programming phrasing as the application. Even if another application can reuse a plugin, a developer has to adapt the plugin's User Interface (UI) to the new application. Otherwise, users may need help understanding the plugin's functionality. To address this limitation, QHana is built on a novel concept of Reusable Microservice-based Plugins (RAMPs) [BBH+22]. This concept allows microservices with an integrated UI to be used as plugins by multiple applications, enhancing the reusability of the plugins across different applications.

2.1.1 QHAna's Architecture

QHAna's architecture, as depicted in Figure 2.1, is primarily microservice-oriented, emphasizing modularity and extensibility, which is evident from its use of RAMPs and the concept of *microfrontends*. Such an architectural choice is geared towards ensuring scalability and adaptability in the rapidly evolving field of quantum computing. Key components of QHana's architecture are:

QHAna UI Developed using Angular and TypeScript, the UI serves as the primary interface for users, facilitating interaction with the system's functionalities.

QHAna Backend The backend provides essential services, including a REST API for the UI, data management, and functions as a *RAMPs Registry* to discover and load RAMPs.

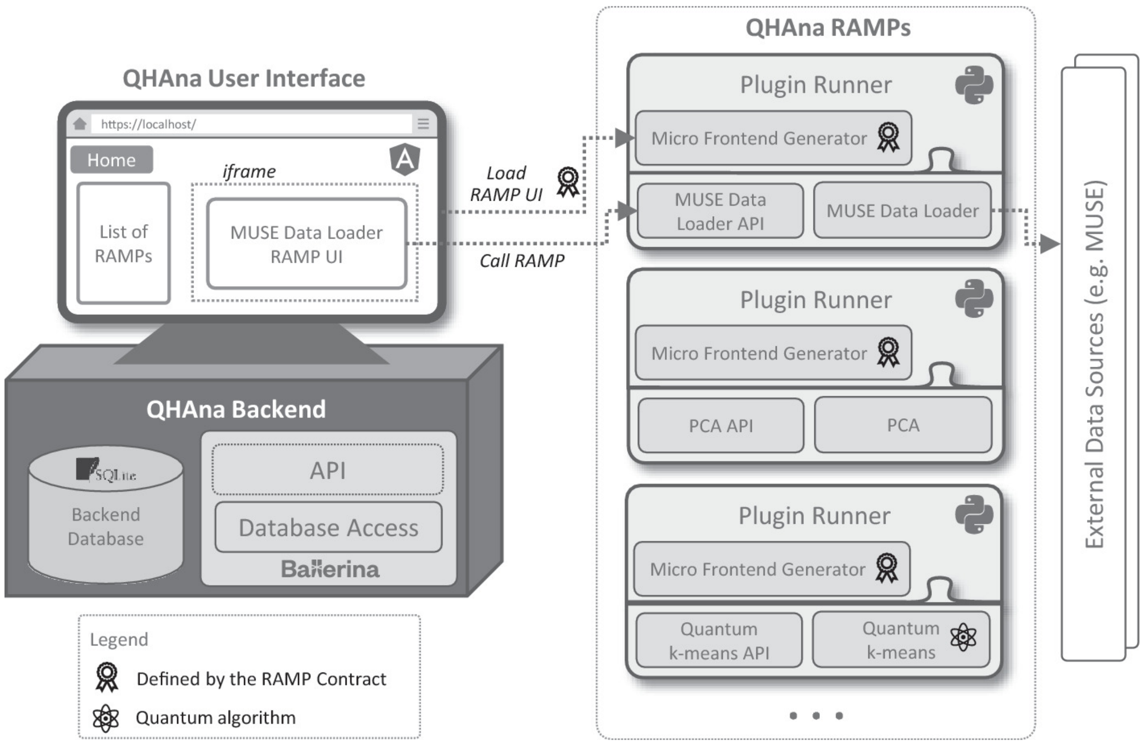

Figure 2.1: QHana's Architecture as depicted in [BBH+22]

RAMPs These plugins encapsulate specific classical and quantum algorithms. Their development is streamlined by the Plugin Runner, a Python framework designed to handle generic RAMP tasks.

Database This component is responsible for the persistent storage of data related to the experiments conducted within QHana.

The *QHana UI* offers users a selection of RAMPs. When a RAMP is selected, its microfrontend is embedded within the UI through an iframe. The *QHana Backend*, central to the system, manages data storage, acts as a RAMP registry, and provides necessary services to the UI. RAMPs, designed for modularity, can be developed independently and integrated into QHana. The *Plugin Runner* demands the adherence of RAMPs to system standards. Data from external sources is processed by RAMPs and stored in the database.

2.1.2 RAMPs: Bridging UI and Data Processing

RAMPs are designed to bridge the gap between traditional plugins and microservices. They offer a comprehensive approach to user interaction and data processing, distinguishing themselves from conventional plugins. One of the primary advantages of RAMPs is their ability to integrate seamlessly with UIs. Unlike traditional plugins, which might require modifications to fit an application's UI, RAMPs come with context-sensitive microfrontends. This ensures that the plugin's UI aligns effortlessly with the primary application's interface, enhancing user experience.

Furthermore, RAMPs introduce the possibility of multi-step UI interactions, where users can be presented with sequential interfaces based on prior inputs or processing results, enhancing the depth and interactivity of the user experience.

In summary, RAMPs blend the best of microservices and plugins. They combine the reusability and accessibility of microservices with the UI adaptability of plugins, making them a valuable asset in diverse application settings.

2.2 Quantum Computing

Quantum computing is a cutting-edge field that exploits the principles of quantum mechanics to process information. Unlike classical computers that use bits (0s and 1s) to store and process information, quantum computers use quantum bits, or *qubits*. Through superposition and entanglement, qubits can exist in multiple states simultaneously and correlate in ways that classical bits cannot [NC12].

Superposition allows a qubit to be in a state of a combination of 0 and 1, with a certain probability for each. This property enables quantum computers to perform many calculations simultaneously, vastly increasing their potential computational power. Entanglement allows qubits to be intimately linked regardless of the distance separating them. A change in the state of one will instantaneously affect the state of the other, a phenomenon that Einstein famously called spooky action at a distance [EPR35]. This property is essential for many quantum algorithms, quantum error correction, and quantum teleportation, making it a fundamental resource in quantum information processing [NC12; Pre98].

2.3 Variational Quantum Algorithms

VQAs combine the principles of quantum computing and optimization uniquely and powerfully. They are a class of hybrid quantum-classical algorithms that leverage the strengths of both quantum and classical computing to solve complex problems [MRBA16].

The central concept of VQAs is to use a sequence of quantum operations (a *quantum circuit*) controlled by specific parameters. These parameters are adjusted using classical optimization techniques to solve a specific problem. This problem, in many cases, involves finding the lowest energy state, or *ground state*, of a quantum system. This problem maps to finding the minimum of a particular function [PMS+13].

By leveraging classical optimization algorithms, VQAs become more resistant to quantum errors, as classical computers perform most of the computation. This combination of quantum and classical resources makes VQAs a promising algorithm for near-term quantum devices [MBB+17].

2.4 Optimization Algorithms

Optimization is a powerful tool ubiquitous in various scientific and technological domains. At its core, optimization is about finding the best solution from a set of possible choices. This section provides a snapshot of optimization's fundamental principles, paving the way for deeper exploration in the context of VQAs and plugin-based VQAs.

2.4.1 Objective Functions

Objective Functions (OFs) are fundamental to optimization problems, underpinning many computational algorithms and models. Depending on specific requirements, the aim might be to minimize or maximize these functions. Notably, within the realm of VQAs, the focus is primarily on function minimization [E17].

The core inputs to a OF typically encompass data points (denoted as x), corresponding labels or outcomes (represented by y), and a set of parameters or weights (often symbolized by θ or w). These parameters dictate how the model responds to the input data and makes predictions. Additionally, certain OFs may also include hyperparameters as input, which control the behavior and complexity of the model. In the context of optimization problems, the role of an OF is to capture both the problem we are attempting to solve and the strategy by which we are trying to solve it. It provides a measure of the 'goodness' or 'fitness' of our current solution or parameters, and the aim is to adjust these parameters to improve this measure.

One example of an OF is the Lasso (Least Absolute Shrinkage and Selection Operator) Loss function. The Lasso loss function has the form:

$$L(Y, X, W, \lambda) = ||Y - XW||_2^2 + \lambda ||W||_1$$

In this equation:

- Y is the vector of observed values.
- X is the matrix of input data points.
- W is the vector of weights, the model's parameters.
- λ is the regularization parameter, a non-negative hyperparameter.

This function consists of two terms:

1. The first term $||Y - XW||_2^2$ is the mean squared error between the predicted and actual outcomes. It measures the discrepancy between the model's predictions and the true values.
2. The second term $\lambda ||W||_1$ is a regularization term, where $||W||_1$ represents the L1 norm (sum of absolute values) of the weight vector. This term penalizes the absolute size of the coefficients, encouraging them to be small.

The hyperparameter λ governs the trade-off between these two terms. When $\lambda = 0$, the OF reduces to ordinary least squares regression, and the weights are chosen to minimize the mean squared error alone. As λ increases, more weight is given to the regularization term, and the solution becomes more sparse (i.e., more weights are driven to zero). Increasing the regularization term can help to prevent overfitting by effectively reducing the complexity of the model [SB14]

2.4.2 Minimization Functions

Minimization functions, generally called optimization algorithms, are pivotal in many computational models and algorithms. In essence, they serve to iteratively enhance the parameters of a model to reduce the value of the OF. These minimization functions aim to find the optimal set of parameters that yield the lowest possible value of the OF within the constraints of the problem [NW06].

The process of optimization involves a search through the parameter space. This search can be visualized as navigating a landscape of hills and valleys, with each point in the landscape corresponding to a different set of parameters and the height at each point representing the value of the OF. The goal of the minimization function is to find the lowest point in this landscape, corresponding to the minimum value of the OF [GBCB17].

The core inputs to a minimization function are the initial parameters of the model or weights (denoted as θ or w), the OF that needs to be minimized, and optionally, the gradient of the OF. Additionally, certain minimization functions may include hyperparameters as input, which control the behavior and complexity of the optimization process [VGO+20]. For instance, the learning rate is a typical hyperparameter that determines the step size in each iteration of the optimization process.

One of the most fundamental and widely used minimization functions is the Gradient Descent. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the function's gradient (or approximate gradient) at the current point.

The update rule of gradient descent has the form:

$$\theta_{t+1} = \theta_t - \alpha \nabla F(\theta_t)$$

In this formula:

- θ_{t+1} represents the parameters at the next time step.
- θ_t represents the current parameters.
- α is the learning rate, a positive scalar determining the step size.
- $\nabla F(\theta_t)$ is the gradient of the OF.

Here, the OF F is assumed to be a differentiable function. The gradient $\nabla F(\theta_t)$ provides the direction of the steepest ascent at the point θ_t, and $-\nabla F(\theta_t)$ provides the direction of the steepest descent. We move towards the minimum of the function by taking a step in this direction.

The choice of minimization function can significantly influence the efficiency and success of the optimization process. While some minimization functions may perform well on certain problems, they may yield different results on others. Therefore, it is crucial to understand the underlying mechanisms of these functions and their suitability to the specific problem at hand.

3 Problem Statement and Objectives

Optimization algorithms, with their ability to find the best possible solution from a set of feasible solutions, play a pivotal role in numerous computational domains. VQAs are a subset of these algorithms that leverage quantum computing principles, particularly in the realm of OFs and minimization techniques. However, the true potential of optimization is often hindered by rigid platforms where the components of these algorithms are tightly integrated, limiting adaptability and innovation.

QHana, with its unique environment tailored for experimenting with plenty of machine learning and quantum algorithms, presents an opportunity to redefine this paradigm. However, its current architecture does not fully exploit the modular benefits that can be achieved by decoupling the components of optimization algorithms. Furthermore, while developing this modular framework, allowing plugins to interact with each other is essential. This interaction-centric concept, once established, can be universally applied across QHana, not just for optimization but for any scenario where plugin interaction is required.

Problem Statement: How can we design and implement a modular framework within QHana that allows components of optimization algorithms, specifically OFs and minimization functions, to be implemented as distinct, interchangeable plugins? Furthermore, how can these plugins be structured to communicate and collaborate seamlessly, especially in the context of VQAs?

This problem encompasses several challenges:

- **Communication:** Establishing a robust communication mechanism that enables interaction, data sharing, and collaboration among these plugins.
- **Interchangeability:** Designing a system where different OF and minimization plugins can be effortlessly swapped, ensuring adaptability in optimization and VQAs. Implementing a consistent interface for these plugins, ensuring uniformity and compatibility across various OF and minimization plugins. The design should ensure that interchangeability accommodates future use cases beyond the optimization context.
- **User Experience:** Providing an intuitive UI for users to easily choose and experiment with different optimization components tailored to their needs.
- **Developer Experience:** Providing developers with an extensible framework to build new minimization and OF plugins.

Addressing this problem is essential to enhance the capabilities of QHana, transforming it into a dynamic, adaptable, and user-centric platform for experimenting with optimization and VQAs. When solving this problem, technical requirements and constraints must be considered. QHana, with its design centered on the Digital Humanities, provides an extensible platform for experimenting with various algorithms. QHana's architecture predominantly revolves around the concept of RAMPs

[BBH+22]. The objective is to leverage QHana's inherent modularity by enabling components of optimization algorithms to function as distinct and interchangeable plugins. This brings us to the significance of modularity in optimization. When VQAs are designed with modular components, it allows for increased flexibility. Specifically, having distinct OFs and minimization functions means parts of the algorithm can be adjusted or replaced seamlessly. This reflects the goals of modularity and flexibility intrinsic to QHana's design.

Interactivity in QHana should encompass various facets. No plugin should be aware of specific user input requirements of another plugin to enable interchangeability and loose coupling between plugins. This means that an invoking plugin should not have input fields that are specific to the invoked plugin. Therefore, a plugin should be able to invoke the microfrontend of another plugin where the user can provide the required input. Control should revert to the invoking plugin once the invoked plugin completes its tasks. Interactions between plugins can have varying degrees of complexity, so the interaction mechanism should be flexible enough to accommodate different scenarios. Both short-running tasks and long-running tasks should be facilitated. For the latter, a callback mechanism is proposed, wherein the invoking plugin is notified upon completion of a long-running task by the invoked plugin.

The need for such interactivity stems from the inherent nature of optimization. The OFs and minimization must closely interact to produce meaningful optimization results. Moreover, since an invoking plugin is unaware of the parameters or requirements of the invoked plugin, direct interaction becomes imperative for dynamic data exchange and collaborative processing.

Drawing inspiration from existing systems, the interaction between the minimization and OFs in implementations like SciPy's *optimizer.minimize* [VGO+20] function serves as a precedent.

This thesis introduces the concept of *interaction endpoints* to QHana to elevate the interactivity between plugins. While QHana already employs a metadata field for plugins known as *entry points* – endpoints invoked by the QHana UI to render the UI of a plugin – interaction endpoints extend this paradigm by marking specific endpoints as callable by other plugins.

A defining characteristic of an interaction endpoint is its *type*. Interaction endpoints sharing the same type uphold uniformity in their signature and return type. This typing ensures that they are invoked consistently by other plugins, irrespective of their specific implementation. Inspiration for this typing mechanism is drawn from the OpenAPI specification [MWG+21], which defines a standard, language-agnostic interface for REST APIs.

The proposed approach complements QHana's existing principles, resonating with its emphasis on extensibility, adaptability, and user-centric design. By enhancing the interactivity and modularity of plugins within QHana, we strive to elevate its capabilities, making it a more dynamic platform for optimization and VQAs.

The subsequent sections of this thesis will delve into the methods, implementations, and evaluations related to this problem.

4 Methods

This section outlines the method developed in this work for enabling modular VQAs consisting of multiple plugins. The chapter begins by defining the architectural design strategy and explaining the process of decomposing the optimization process into distinct plugins that interact. Next, the implementation strategy is defined, outlining the tools and technologies employed in the implementation process. Finally, the evaluation strategy is delineated, explaining the approach to evaluate the efficacy of the proposed design and implementation.

4.1 Architectural Design Strategy

The architectural design process is a critical phase that dictates how the solution's components will function and interact with each other. The final architectural design will be chosen by iteratively refining the existing architecture over multiple steps, considering different aspects like clarity, modularity, and extensibility. Before diving into the details of the design process, it was essential to understand the existing architecture of QHana.

Decomposition into Plugins: The first step involves decomposing the optimization process into distinct plugins. Decomposition is achieved by reviewing relevant literature to understand standard practices and methods in optimization [E17; NW06; SB14; VGO+20]. The main challenge is to find a decomposition so that, on the one hand, any functionality that should be interchangeable is encapsulated in an extra plugin, on the other hand, splitting the process into too many plugins would be inefficient due to the overhead of the required communication between plugins. The main question is how to encapsulate the OF and the minimization function and their coordination. In order to represent the decomposition, a UML component diagram is created.

Defining Plugin Responsibilities: Once the plugins are identified, the next step is to define the responsibilities of each plugin precisely. This ensures that every plugin has a well-defined purpose, preventing overlaps in functionality and ensuring clarity in their roles. Several crucial decisions delineate these responsibilities:

- Which one prompts the user for the OF hyperparameters?
- Which one gathers the minimization function hyperparameters from the user?
- Which one inquires about the user's preference for the OF and minimization algorithm?
- Which one solicits the input data from the user?
- Which one requests the target variable?

Universal Plugin Interface Design: Recognizing the vast possibilities and variations that interchangeable plugins might encompass, it is crucial to formulate a universally adhered-to interface. This interface acts as a *contract*, ensuring that a consistent mode of interaction exists irrespective of the specific implementation details of a plugin. The notion of interface contracts in microservice architectures is not new and is inspired by the OpenAPI specification [MWG+21]. Core attributes and functionalities, like querying the number of initial weights for the minimization process required, are defined as part of this universal interface. This approach fosters interchangeability and adaptability, as the defined interface could accommodate a multitude of interchangeable plugins, each with its unique implementations. In terms of optimization, this could mean that in the case of an OF plugin responsible for calculating the loss function, interfaces must be generalized to allow for any thinkable type of loss function. A UML component diagram visualizes the interfaces of each plugin and how they are connected.

Plugin Interaction Design: The next step is to design the interaction between plugins. This design process involved defining the various ways in which plugins could interact with each other. It is crucial to build a robust and flexible interaction design to accommodate many scenarios since the interaction between plugins in QHana should apply not only to optimization. This interaction design should lay the foundation for all future multi-plugin interactions within QHana.

Scenarios include, for instance, a plugin invoking the microfrontend of another plugin, or it could call a specific endpoint of another plugin. Additionally, the interaction design also encompasses the flow of control between plugins. A plugin should be able to invoke another plugin and then stop execution for as long as the invoked plugin is running. Alternatively, a plugin should be able to invoke another plugin and then continue with its tasks without waiting for the invoked plugin to complete its tasks. Intrinsic to this step is the design of a coordination mechanism that facilitates the interaction between plugins. A UML sequence diagram shows the complete process of a generalized optimization run to represent the interactions between plugins.

Feedback and Refinement: Throughout the design process, continuous feedback loops are integrated. This involves:

- Revisiting each stage, evaluating its alignment with overarching goals.
- Making necessary refinements to ensure the architecture is robust and flexible.

With the architectural design strategy in place, the next step is to define the implementation strategy.

4.2 Implementation Strategy

Before diving into the implementation, an in-depth study of the QHana documentation [Fab] and a related paper on QHana's architecture [BBH+22] was undertaken to gain a comprehensive understanding of the system. With component and sequence diagrams already in place, the next step is to define the implementation strategy.

4.2.1 Development Environment and Toolset

The choice of development tools and environment plays a pivotal role in the successful execution and maintenance of a project. The right toolset facilitates smooth development and enhances the system's efficiency, productivity, and scalability. The following list delineates the tools and technologies employed in the implementation process, each chosen for its specific capabilities tailored to the requirements of the project:

- **Visual Studio Code**: A versatile integrated development environment employed for its adaptability and support for Python development using extensions, facilitating comprehensive tools for coding, debugging, and testing.
- **Docker**: Utilized to run all components of QHana, ensuring consistent behavior across different computing environments and simplifying the deployment process.
- **Postman**: An API testing tool employed to validate and debug various endpoints, ensuring consistent and expected behavior of the plugin interactions.
- **Python**: As QHana is implemented in Python, which offers versatility and a vast library ecosystem.
- **Flask**: A lightweight Python web framework employed to develop the web services and RESTful APIs for the plugins.
- **Marshmallow**: A Python library pivotal for object serialization/deserialization, ensuring structured data transfer between the plugins.
- **Flask-Smorest**: An extension of Flask, offering tools for building RESTful APIs with Flask and Marshmallow, ensuring structured and accurate data transfer between plugins.
- **Celery**: A task queue to manage long-running tasks, particularly for minimizer plugins, allowing for asynchronous task execution.
- **Requests**: A Python library for interacting with HTTP endpoints. It invokes endpoints of plugins.

4.2.2 Key Principles of QHana Plugins

In the QHana ecosystem, plugins play a pivotal role in extending functionality. A set of principles governs the creation and integration of these plugins to ensure their seamless operation and interaction [Fab]. All plugins must adhere to these principles to ensure compatibility and consistency across the system. To understand the decisions made in this work, consider at least the following principles:

- **Plugin Definition:** A plugin is a Python module or package. Conventionally, it inherits from the `QHAnaPluginBase` class and resides in a directory specified by the `PLUGIN_FOLDERS` configuration variable. The plugin imports its implementation class and all associated Celery tasks.
- **Plugin Metadata and Endpoints:** A plugin should contain metadata and links to all its endpoints. It is reachable via the "./"path. The metadata includes crucial information like entry points.

- **UI Interaction:** Plugins define both `href` and `hrefUi` for each plugin with an UI. The ‘hrefUi‘ serves the microfrontend where users input data, and ‘href‘ is the underlying endpoint providing the functionality.
- **Data Handling in Multi-step Plugins:** Data is stored in a key-value store for plugins requiring multiple user input and processing steps. A plugin task is associated with a unique database ID, and subsequent endpoint URLs of a plugin typically include the ID in the URL path `http(s)://.../<ID>/<endpointName>`. That way, the plugin can retrieve the data from the database using the ID.
- **Handling Long Running Tasks:** For a plugin to be able to handle long-running tasks, QHana offers the Celery framework. Celery is a task queue that allows for asynchronous task execution.
- **File Loading from URLs:** Plugins have to be able to load files from URLs.
- **Data Format Specification:** Data formats, especially those shared across plugins, should be defined per the guidelines of QHana. For instance, for the `text/csv` format pertaining to entities:
  - The first column must be the ID column (named ID). Subsequent columns represent entity attributes.
  - The CSV file should contain a header row specifying all attribute names.

It is imperative to note that while the outlined principles are crucial to this thesis, QHana’s overarching documentation provides a more exhaustive list of best practices and guidelines for plugin creation [Fab].

4.2.3 Data Handling and Transfer

Flask-Smorest is instrumental in ensuring structured and accurate data transfer between plugins. It provides functions for validating the correctness of passed data, returning appropriate error codes, and managing errors efficiently based on schemas. This ensures that data exchanged between plugins is smooth and error-free.

4.2.4 Testing and Debugging

A multifaceted testing strategy addresses the paramount importance of the quality and reliability of plugins and their interactions:

1. **Static Code Analysis**:
   - **Purpose**: To ensure code quality and maintainability and to identify potential vulnerabilities or deviations from coding standards.
   - **Tools & Implementation**: The tool *flake8*, in combination with type hints, is utilized to conduct static code analysis on the Python codebase. *Flake8* generates a report detailing any code inconsistencies, potential errors, or areas for improvement and provides valuable feedback for refinement.

2. **Logging and Monitoring**:
   - **Purpose**: To capture, store, and analyze real-time information about the system's operations, aiding in troubleshooting and understanding system behavior.
   - **Tools & Implementation**: Python's in-built `logging` package is leveraged to track and record various events during the execution of plugins. By strategically placing logging statements within the code, it is possible to gain insights into the flow of operations, detect anomalies, and pinpoint areas that might require attention.
3. **Interactive Debugging**:
   - **Purpose**: To step through the code in real-time, inspect variables, and analyze the program's flow to identify and rectify issues.
   - **Tools & Implementation**: The integrated Python debugger in Visual Studio Code is employed. This debugger allowed for setting breakpoints, stepping through code, inspecting variable states, and examining the call stack, providing a granular view of the system's operations and aiding in issue identification and resolution.
4. **Manual Testing**:
   - **Purpose**: To capture nuances and potential issues that might be overlooked.
   - **Procedure**: A hands-on approach is adopted where the plugins are interactively used. This involved navigating through the UI, experimenting with different inputs, and observing the system's reactions to ensure it behaved as expected and met user requirements.

Employing a combination of static code analysis, detailed logging, interactive debugging, and manual testing ensures functional correctness and that plugins adhere to coding standards and best practices.

4.2.5 Performance Optimization

The system employs several strategies to optimize performance in the microservice-based plugin architecture:

1. Reducing the number of interactions between plugins.
2. For each interaction, the amount of data transmitted across the network should be kept to a minimum to ensure efficient communication and faster response times.
3. For tasks that require extended processing, the Celery framework should be utilized, allowing these tasks to operate asynchronously and optimizing resource usage.

These strategies are critical in ensuring swift and seamless interactions between plugins.

4.2.6 Documentation and Extensibility

The code includes comprehensive documentation that details how to add new OF and minimization plugins, facilitating the development of new plugins. This documentation serves as a guideline for developers aiming to extend the capabilities of QHana with plugin interaction.

4.3 Evaluation Strategy

To validate the effectiveness of the proposed solutions and to assess whether the goals set out in the problem statement have been achieved, the following evaluation methods are adopted:

Performance Benchmarking: Performance is paramount, especially in a plugin-based architecture. A direct comparison shows the differences between the two plugin-based and non-service-based approaches. Critical metrics for this comparison include:

- **Objective Function Calculation Time**: This measures the time to retrieve the loss, directly impacting the overall optimization time. For the plugin-based approach, this is the time, as observed by the client calling the calculation endpoint, meaning it includes the network latency. As all system components run on the same machine, network, in this case, means the *localhost*.
- **Minimization Time**: This refers to the time needed to minimize the OF. The service-based approach includes the time taken for the minimization endpoint to return a result via the network.
- **Network Latency**: This exclusively quantifies the time required for a request to reach the server and for the corresponding response to be received. This metric does not include any computation time that occurs on the server. This metric is exclusive to the service-based approach.
- **Database Access Times**: It is essential to gauge the time required to retrieve data from the database since endpoints access context data during each invocation.

This evaluation hinges on quantitative metrics, with results graphically depicted for enhanced clarity. The *time.perf_count* function from Python measures the time taken for a function to execute, providing a reliable and accurate measure of performance.

Interchangeability: The accurate measure of interchangeability is the ability to swap components without causing disruptions. Accordingly, all allowed combinations of plugins were tested to validate seamless interchangeability.

Standardization Adherence: Standardization is checked to ensure compatibility and uniformity across diverse plugins. An evaluation determines whether all implemented plugins conform to the prescribed standards. This guarantees that all possible optimization plugins can be implemented uniformly, facilitating consistent and compatible integrations in the future.

Developer Experience: The developer experience has to be maintained. To measure the experience, the steps required to implement a new plugin are counted, and each step is evaluated for its complexity. The evaluation acknowledges any documentation that guides developers through the process. An assessment determines the ease and efficiency with which a developer can introduce a new plugin in plugin-based optimization in QHana. The goal is to ensure that the plugin-based approach is as easy to implement as the non-service-based approach.

This thesis thoroughly appraises the solutions concerning the challenges outlined in the problem statement by meticulously evaluating these parameters.

4.4 Test Data Generation for Evaluation

To robustly evaluate the solutions proposed in this work, testing them on diverse datasets with different sizes and complexity is essential. The objective is to mimic real-world scenarios where optimization problems can range from simple tasks with a few data points to complex challenges with many features and data points. The code used to generate the test data can be found in the Appendix. The following criteria are employed to generate the test data:

Datasets of different *sizes* are generated, spanning from a modest 200 data points to a substantial 1400 data points. This variation ensures that the optimization performance is assessed across different scales, from quick-to-process small datasets to computationally demanding large datasets. The *number of features* in each dataset is dynamically determined based on the dataset's size, calculated explicitly as $\lfloor \sqrt{\text{size}} \times 1.5 \rfloor$. This approach ensures that with the growth of the dataset, its complexity also increases, mirroring real-world scenarios where larger datasets often present more features or dimensions.

The `make_regression` function from the Scikit-learn library generates data. An added *noise* parameter introduces an element of randomness, making the optimization task more intricate and resembling real-world challenges.

To maintain the consistency and reliability of the generated datasets across multiple runs or evaluations, a fixed random seed (`numpy.random.seed(42)`) has been set. This ensures that the data, though synthetic and noisy, remains consistent across evaluations, enabling accurate comparisons, assessments, and *reproducibility*.

Two criteria are employed to adhere to QHana's data standards: Each data point in the dataset is allocated a unique ID in the format *entityX*, where X represents the entity number. This aligns with QHana's data standard that mandates every data point to have an identifiable ID. All datasets are stored in the CSV (Comma Separated Values) format, adhering to QHana's accepted data formats.

By employing such datasets, the methodology objectively evaluates the optimization solutions, assessing their performance, interchangeability, and user experience across various scenarios.

5 Resulting System Architecture

This chapter delves into the intricate architectural blueprint of the proposed plugin-based optimization framework. It explores the decomposition into distinct plugin types, each with clearly defined roles and responsibilities. The chapter also elucidates the universal plugin interfaces designed to ensure seamless interaction between different plugins, fostering modularity and extensibility. By the end of this chapter, readers will gain a comprehensive understanding of the system's interaction flow, its various components, and their interdependencies, all illustrated through detailed component and sequence diagrams.

5.1 Resulting Decomposition into Plugins and their Responsibilities

Following the decomposition strategy, the plugin-based optimization framework is divided into three primary plugin types: OF plugin, the minimizer plugin, and the coordinator plugin. These plugins have specific roles and responsibilities, ensuring a modular and efficient optimization process. This split is visualized in the component diagram in figure 5.1. It is important to note that this decomposition and the associated responsibilities remain consistent for both proposed plugin-based approaches.

Objective Function Plugin: The OF plugin is central to the optimization framework, encapsulating the mathematical function that defines the problem at hand. Its primary roles include:

- **Metadata Provision**: The plugin offers metadata about itself.
- **Hyperparameters Acquisition**: It prompts the user to provide the hyperparameters necessary for the loss function calculation.
- **Loss Calculation**: The plugin computes the loss based on the provided input data, representing the discrepancy between predicted and target values.
- **Gradient Calculation (Optional)**: For optimization algorithms that leverage gradient-based methods, the plugin can optionally compute the gradient of the loss function. The gradient aids in guiding the optimization process toward the desired minimum.

Minimizer Plugin: The minimizer plugin is responsible for iteratively adjusting parameters to minimize the loss provided by the OF plugin. Its functions include:

- **Metadata Provision**: Similar to the OF plugin, it provides essential metadata.
- **Hyperparameters Acquisition**: The plugin acquires the hyperparameters crucial for the employed minimization algorithm from the user.
- **Minimization Process**: Using the loss (and optionally the gradient) from the OF plugin, the minimizer plugin endeavors to find the parameter values that minimize this loss.

Coordinator Plugin: The Coordinator plugin acts as the orchestrator, ensuring seamless interaction between the OF and minimizer plugins and the user. Its primary responsibilities are:

- **Plugin Selection**: It prompts the user to select the desired OF and minimizer plugins for optimization.
- **Data Acquisition**: The plugin gathers the necessary input data and the target variable from the user, which the optimization process will use.
- **Endpoint Acquisition**: It obtains the necessary endpoints from the selected plugins.
- **Coordination Role**: It manages the interaction between the OF and minimizer plugins, ensuring that the loss (and optionally gradient) calculation function is provided to the minimizer plugin for the optimization process. Additionally, it coordinates the interaction between the user and the selected plugins, ensuring the user sees the necessary microfrontends.
- **Results Presentation**: Post-optimization, the coordinator plugin presents the optimization results to the user.

5.2 Universal Plugin Interface Design

The process of optimization is inherently complex, with a multitude of variations and nuances. It is imperative to establish universal plugin interfaces for each type of plugin to ensure a streamlined interaction between different plugins. This interface acts as a standard that every plugin of a specific type has to adhere to. The interfaces for each plugin are detailed below and are visualized in the component diagram in figure 5.1.

Objective Function Plugin Interfaces: The OF plugin interface accommodates a wide range of loss functions, including those that offer gradient computation. Its interfaces are:

- **Metadata**: The plugin provides metadata about itself as specified in the QHana documentation [Fab].
- **UIRef**: This endpoint returns the microfrontend for the OF plugin, where the user inputs the hyperparameters for the calculation. This interface also follows the QHana documentation [Fab].

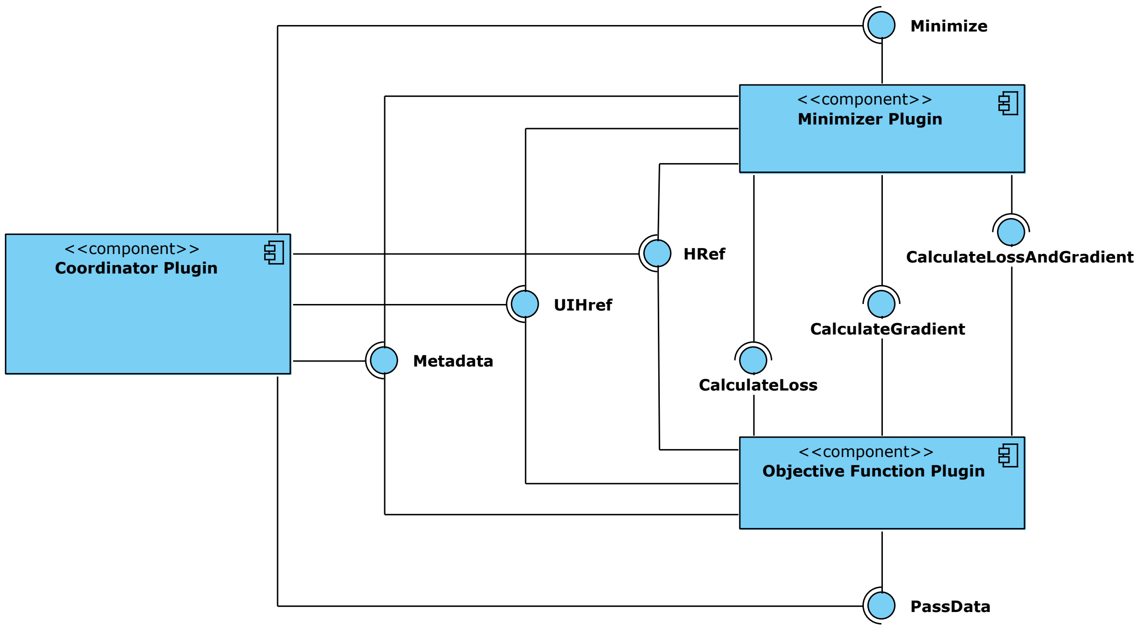

Figure 5.1: Component Diagram of decomposed optimization process

- **HRef**: This endpoint is used to process the input from the OF microfrontend and is therefore usually called the **processing** endpoint. It is usually triggered by the user clicking the *submit* button on the microfrontend. This interface also follows the QHana documentation [Fab].
- **PassData**: Via this endpoint, the coordinator passes the input and target data to the OF plugin. It returns the number of weights required for the optimization process. It returns the number of weights required by the objective function.
- **CalculateLoss**: This endpoint calculates the loss.
- **CalculateGradient (Optional)**: This endpoint calculates the gradient of the loss. It is optional since the gradient cannot be computed efficiently for all OFs.
- **CalculateLossAndGradient (Optional)**: This endpoint calculates the loss and the gradient of the loss function. Similar to the previous endpoint, it is optional.

Minimizer Plugin Interfaces: The minimizer plugin, responsible for optimizing the loss function provided by the OF plugin, offers these interfaces:

- **Metadata**: same as for the OF plugin
- **UIRef**: same as for the OF plugin
- **HRef**: same as for the OF plugin
- **Minimize**: This endpoint minimizes the loss function.

Coordinator Plugin Interfaces: The coordinator plugin, orchestrating the interaction between the OF and minimizer plugins, is equipped with the standard QHana interfaces **Metadata**, **UIRef**, and **HRef**.

A systematic, consistent, and efficient optimization process is ensured by establishing these interfaces and ensuring that each plugin conforms to them. This structured approach facilitates seamless interactions and fosters interchangeability, modularity, and extensibility, making it easy to add new OF and minimizer plugins in the future.

5.3 Plugin Interaction Design

The design of plugin interactions is pivotal to ensuring efficient and seamless coordination between different system components. Given the extensive possibilities of interactions within the QHana environment, the design phase is meticulous, considering various scenarios and ensuring adaptability. The two primary modes of interaction are short-running and long-running, each catering to specific requirements.

Short-Running Interaction: In short-running interactions, a plugin invokes another plugin's endpoint, typically via a 'GET' or a 'POST' request, and immediately receives a response. This mode of interaction is synchronous, wherein the invoking plugin waits for the response before proceeding. Instances of such interactions in the optimization context are:

- The coordinator plugin retrieves metadata from the OF and minimizer plugins.
- The coordinator plugin passes data to the OF plugin.
- The minimizer plugin calls the `CalculateLoss` endpoint of the OF plugin.

Long-Running Interaction: Long-running interactions come into play for processes requiring extensive computation time or involving multiple steps. In such a case, the calling plugin invokes the endpoint of the invocable plugin. The invoked endpoint then schedules a long-running task and returns immediately. The calling plugin can then continue with its execution asynchronously without waiting for the long-running task to finish. The calling plugin gets an additional endpoint that is called by the long-running task once it finishes. This endpoint is called a callback endpoint. Instances of such interactions are:

- The coordinator schedules the microfrontend of the OF plugin. The coordinator plugin provides a callback endpoint to the OF plugin that is called once the user enters the input and the input is processed.
- The coordinator schedules the microfrontend of the minimizer plugin. This mechanism works identically to the previous item.
- The coordinator plugin calls the minimization endpoint of the minimizer plugin. The minimizer plugin schedules the minimization process and returns immediately. Once the minimization process finishes, the minimizer plugin calls a callback endpoint of the coordinator plugin.

The design of these interactions ensures that plugins can interact seamlessly and efficiently regardless of the complexity or duration of tasks.

5.4 Introduction of Interaction Endpoints

Building upon the idea of plugins interacting with each other, as detailed in the previous section, a crucial question arises: How does one plugin discover the available endpoints of another? This thesis introduces a novel concept called *interaction endpoints* to QHana to address this very challenge.

While QHana already has a metadata field named *entry points*, which are endpoints invoked by the QHana UI to render a plugin's UI, interaction endpoints extend this idea further. They specifically define endpoints in the metadata that other plugins can invoke, facilitating seamless integration and interaction.

The core of interaction endpoints is their *type*. All interaction endpoints with the same type must adhere to the same signature and return type. This uniformity ensures that other plugins can invoke them interchangeably. The OF plugin provides interaction endpoints of types *calc_loss*, *calc_grad*, *calc_loss_and_grad*, and *of_pass_data*. The minimizer plugin offers the *minimization* type. These interaction endpoints correspond to the endpoints defined in the previous section. The introduction of interaction endpoints significantly enhances the modularity and interchangeability within QHana, paving the way for a more dynamic and adaptable plugin ecosystem. More on how these interaction endpoints are implemented can be found in the implementation chapter 6.

5.5 Final Interaction Flow

The architecture's final interaction flow is split into three main parts, each ending in a new UI displayed to the user. The sequence diagrams in figures 5.2, 5.3, and 5.4 visualize the interaction flow.

The first part begins after selecting the optimization plugin and concludes when the OF plugin's UI is set as the next step. Here, the coordinator plugin retrieves the user-selected OF and minimization plugin metadata, including their interaction endpoints. This flow is represented in Figure 5.2.

The second part starts with retrieving the OF plugin's microfrontend and ends when the minimizer plugin UI is set as the next step. After the user inputs hyperparameters and submits, the OF processes the input and sends a callback to the coordinator plugin, The callback endpoint then passes the input and target data to the PassData endpoint. This sequence is depicted in Figure 5.3.

The last segment starts with the minimizer plugin's microfrontend retrieval and ends when the optimization process concludes. After users input the minimization hyperparameters, the minimizer processes the data and sends a callback to the coordinator. The coordinator then triggers the minimization endpoint, initiating a long-running minimization task that continuously calls the OF calculation endpoint. Once this task is completed, the minimizer makes a final call to the coordinator with the minimization results. Figure 5.4 shows this flow.

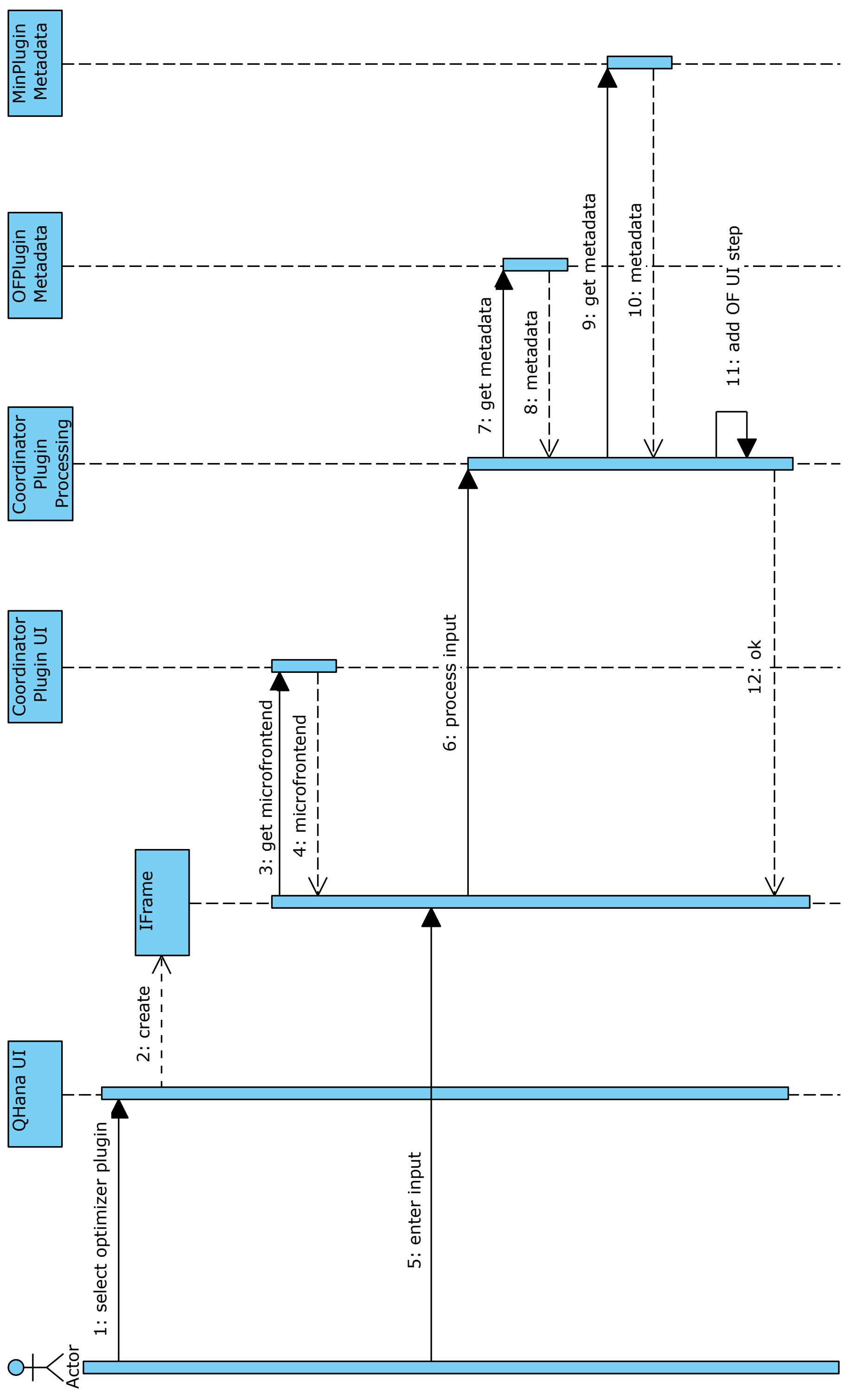

Figure 5.2: Final interaction flow of the optimization process (part 1/3).

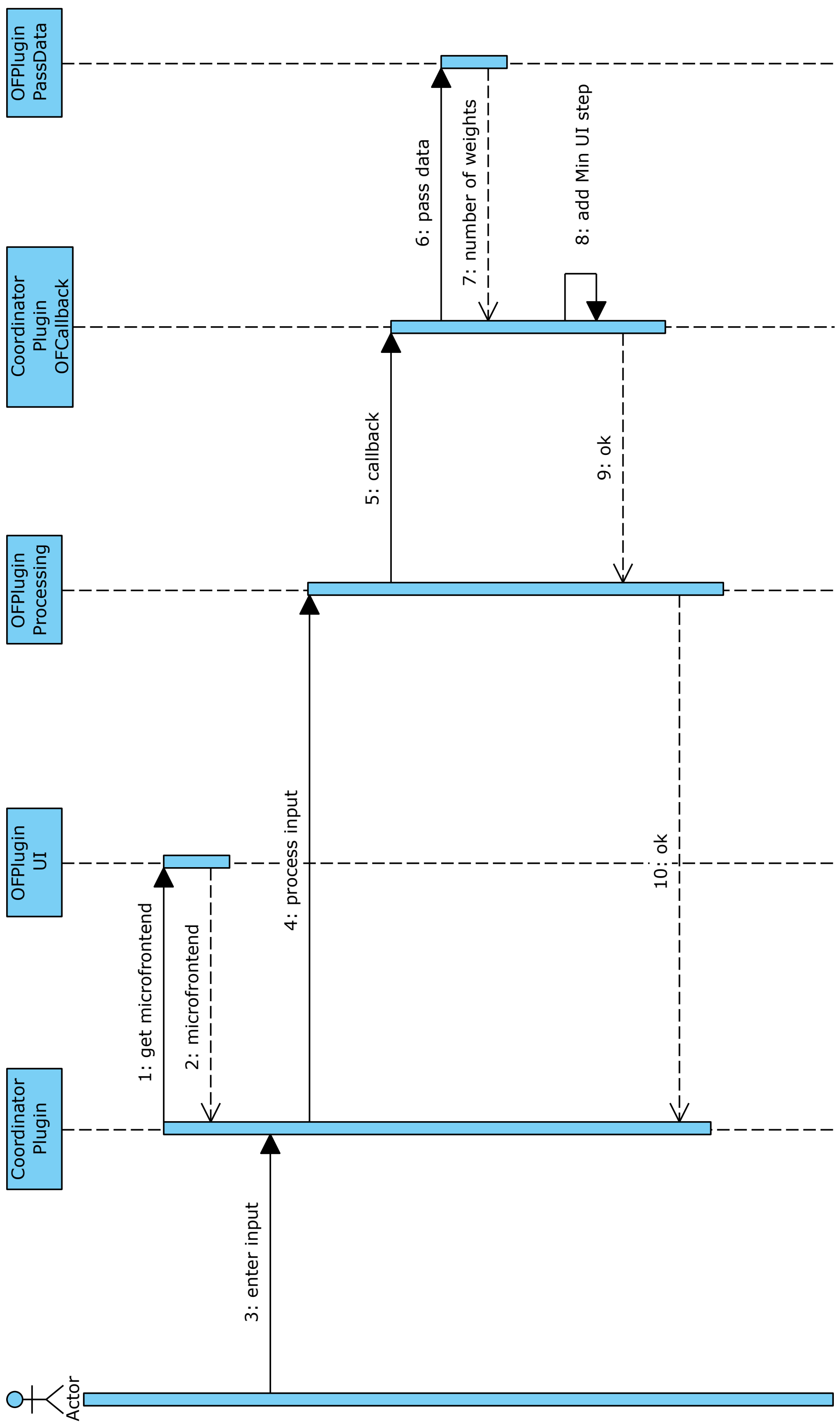

Figure 5.3: Final interaction flow of the optimization process (part 2/3).

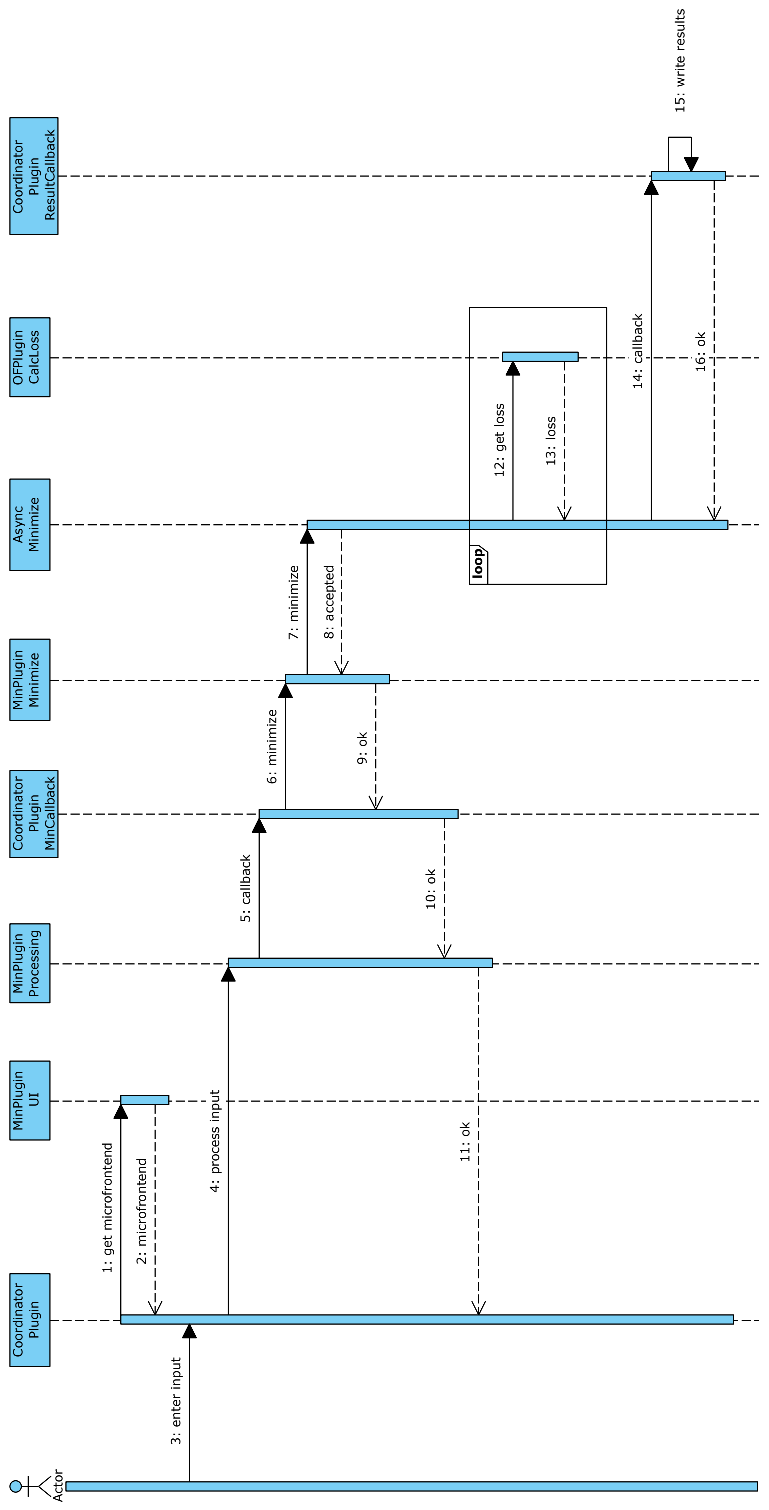

Figure 5.4: Final interaction flow of the optimization process (part 3/3).

6 Implementation

The reader can find the resulting code that realized two different approaches to a plugin-based optimization framework in QHana 's GitHub repository[1]. As examples, the code implements two types of minimizer plugins, the *scipy-minimizer* and *scipy-minimizer-grad* plugins, and three types of OF plugins, the *ridge-loss*, *hinge-loss*, and *neural-network* plugins. More on their implementation can be found in the following sections. This chapter does not aim to provide a comprehensive overview of the code but to highlight the critical implementation details and differences between the two approaches.

6.1 Directory Structure and Plugin Loading

QHana maintains two primary directories for plugin management: the *plugins* and the *stable plugins* directories. The former contains plugins undergoing development, while the latter contains deployment-ready plugins. Within these directories, individual plugins are neatly organized into dedicated subfolders. Upon initiation, QHana scans and loads plugins from these designated subfolders.

The focus of this thesis, the optimization plugin, resides in the *plugins/optimization* directory. Further divisions are made to promote clarity and a structured layout.

- *plugins/optimization/coordinator* – For the coordinator plugin.
- *plugins/optimization/objective_functions* – Where each OF plugin occupies its respective subfolder.
- *plugins/optimization/minimizer* – Where each minimizer plugin occupies its respective subfolder.

Given that QHana's native architecture does not support the direct loading of plugins from nested subdirectories, a recursive plugin loader is developed for this purpose. This loader traverses through the *plugins* directory and its subdirectories. The presence of an *__init__.py* file within a folder confirms the plugin's legitimacy. To exclude a plugin from the loading process, one places a *.ignore* file in its respective folder. The maximum recursion depth is limited to four to maintain system performance and a clear, structured layout. This threshold sufficiently accommodates the current plugin structure but can be adjusted upwards if future needs arise.

[1] `https://github.com/UST-QuAntiL/qhana-plugin-runner`

The *interaction_utils* directory is dedicated to housing utility functions for generalized plugin interactions. Additionally, there is a *shared* directory, which stores data structures and schemas utilized across the plugins related to the optimization plugin. A visual representation of the final folder structure, with all plugins implemented for this thesis, is shown in Figure 6.1.

```
plugins
└── optimizer
    ├── coordinator
    ├── interaction_utils
    ├── minimizer
    │   ├── scipy_minimizer
    │   └── scipy_minimizer_grad
    ├── objective_functions
    │   ├── hinge_loss
    │   ├── neural_network
    │   └── ridge_loss
    └── shared
```

Figure 6.1: QHana Plugin Folder Structure.

6.2 Maintaining Context Across Plugin Endpoints

The implementation leverages the multi-step plugin concept inherent to QHana to preserve the continuity of information across different plugin endpoints. In the *processing* endpoint of a plugin, a database task is established, identifiable by a unique database ID. This task stores all relevant context data, such as the hyperparameters designated for the minimization process. Subsequent endpoints, like the *minimization* endpoint, can access this stored context data by referencing the database task. Notably, the endpoints' URL incorporates the database task's ID.

6.3 UI Creation for Plugin Interaction

The QHAna platform provides a streamlined mechanism to craft simple UIs with input fields generated from marshmallow schemas. The primary step is to inherit from QHAna's `FrontendFormBaseSchema` class. For the OF plugin, the input fields are the hyperparameters the loss function needs. Listing 6.1 shows the *ridgeloss* plugin's hyperparameter, which is the regularization parameter *alpha*.

```
class HyperparamterInputSchema(FrontendFormBaseSchema):
    alpha = marshmallow.fields.Float(
        required=True,
        allow_none=False,
        metadata={
            "label": "Alpha",
            "description": "Alpha variable for Ridge Loss function.",
            "input_type": "textarea",
        },
    )
```

Listing 6.1: Ridge Loss Plugin Hyperparameter Schema.

The minimizer and OF plugin selection field is a critical input field for the coordinator plugin. The `PluginUrl` class, a custom class that extends marshmallow's `fields.Url` class generates that field. It ensures that a user can only select plugins of the correct type by checking a plugin's metadata for the correct label. Imperative for this to work is that all minimizer plugins have the *minimization* and all OF plugins have the *objective-function* label. Listing 6.2 shows the minimizer plugin selection field in the coordinator plugin's UI.

```
class OptimizerSetupTaskInputSchema(FrontendFormBaseSchema):
    minimizer_plugin_selector = PluginUrl(
        required=True,
        allow_none=False,
        plugin_tags=["minimization"],
        metadata={
            "label": "Minimizer Plugin Selector",
            "description": "URL of minimizer-plugin.",
            "input_type": "text",
        },
    )

    # more fields...
```

Listing 6.2: Coordinator Plugin UI Schema.

As the UIs of the OF and minimizer plugins are designed to be invoked by the coordinator plugin, the *UIHref* which serves the UI needs to accept a callback URL. Therefore, all *UIHref* endpoints of invocable plugins must include the *CallbackUrlSchema* marshmallow schema as a query parameter.

6.4 Invocation of Plugin Microfrontends by the Coordinator Plugin

As highlighted in the previous chapter, the coordinator plugin invokes the microfrontends of both the OF and minimizer plugins. An adaptation of QHana's *add_step* function allows the invocation of a plugin's microfrontend by another plugin. The *add_step* function's original purpose is to allow a multi-step plugin to show another of its microfrontends to the user. The function is enhanced and renamed to *invoke_task*. With this modification, the coordinator plugin can invoke another plugin's microfrontend by supplying the appropriate *UIHref* and *Href*. Furthermore, the coordinator plugin can provide a callback URL to the invoked plugin. This callback URL is a query parameter within the *Href* and *UIHref* URLs.

6.5 Callback to the Coordinator Plugin

The coordinator plugin requires callbacks from invocable plugins in two distinct scenarios:

1. After the user submits the microfrontend of either the minimization or the OF plugin.
2. Once the asynchronous minimization process concludes.

For the first scenario, the process unfolds as follows:

- The coordinator invokes the microfrontend of the invocable plugin using the *invoke_task* function, as elaborated in Section 6.4.
- On user submission of the microfrontend, the callback URL is passed to the processing endpoint of the invocable plugin via a query parameter.
- Subsequently, the invoked plugin makes a post request to the callback URL.

In contrast, the second scenario operates through the following mechanism:

- The coordinator plugin initiates a post request to the minimizer plugin's minimization endpoint, embedding a callback URL within the body.
- The minimizer plugin schedules an asynchronous celery task for loss function minimization. It registers the callback URL in the database under the designation *status_changed_callback_urls*. This label is pivotal for subsequent retrieval of the callback URL.
- The endpoint returns, and the celery task commences the minimization process.
- Following the task's completion, its status updates through the already implemented *save_task_result* function. This action now triggers a signal when it finishes or runs into an error, courtesy of Python's *blinker* library, indicating the status change and passing the database ID of the task as an argument.
- A dedicated signal handler retrieves the callback and task view URL from the database. The handler then orchestrates a post request to the callback URL, embedding the task view URL within the body.

Listing 6.3 presents the implementation of the signal handler, while Listing 6.4 illustrates the juncture where the signal is emitted.

```python
TASK_STATUS_CHANGED.connect(task_status_changed_handler)


def task_status_changed_handler(sender, db_id: int):
    task_data: ProcessingTask = ProcessingTask.get_by_id(id_=db_id)
    callback_urls = task_data.data.get("status_changed_callback_urls", [])
    task_url = task_data.data.get("task_view", None)
    for callback_url in callback_urls:
        requests.post(callback_url, json={"url": task_url, "status": task_data.status})
```

Listing 6.3: Signal Handler for Callbacks.

```python
TASK_SIGNALS = blinker.Namespace()

TASK_STATUS_CHANGED = TASK_SIGNALS.signal("task-status-changed")


@CELERY.task(name=f"{_name}.save-result", bind=True, ignore_result=True)
def save_task_result(self, task_log: str, db_id: int):
    # ...
    task_data: ProcessingTask = ProcessingTask.get_by_id(id_=db_id)
    # ...
    task_data.task_status = "SUCCESS"
    # ...
    TASK_STATUS_CHANGED.send(self, db_id=db_id)


@CELERY.task(name=f"{_name}.save-error", bind=True, ignore_result=True)
def save_task_error(self, failing_task_id: str, db_id: int):
    # ...
    task_data: ProcessingTask = ProcessingTask.get_by_id(id_=db_id)
    # ...
    task_data.task_status = "FAILURE"
    # ...
    TASK_STATUS_CHANGED.send(self, db_id=db_id)
    # ...
```

Listing 6.4: Signal Emitter for Callbacks.

6.6 Implementation of Interaction Endpoints

The essence of the plugin-based optimization framework lies in its interaction endpoints. These endpoints standardize the interchangeability of inputs and outputs. Marshmallow schemas are the backbone of the implementation to manage this standardized data exchange.

Interaction Endpoint Schemas: The repository's shared folder contains the defined schemas. Each interaction endpoint type has an associated input and output schema. If a plugin wishes to invoke another plugin's interaction endpoint, it can populate the input schema and parse the response using the output schema. In the implementation, the OF plugin defines schemas for the following interaction endpoints:

- **pass_data**: Uses *ObjectiveFunctionPassDataSchema* for input and *ObjectiveFunctionPass-DataResponseSchema* for output.
- **calc_loss**: Uses *CalcLossOrGradInputSchema* for input and *LossResponseSchema* for output.

For the minimizers' *minimization* endpoint, the input schema is *MinimizerInputSchema*. The output, meanwhile, is a simple 200 status code, signaling the initiation of the minimization process. Another implementation challenge revolves around ensuring that plugins can discover the endpoints of other plugins, which is solved in section 6.6, but first section 6.6 introduces a custom marshmallow field for input and target data.

Custom Marshmallow Field for Data: To streamline the input and target data exchange between plugins, a custom marshmallow field, *NumpyArray*, is introduced (see 6.5). Specifically crafted to handle NumPy arrays, this versatile field can accommodate arrays of any dimensionality with JSON-serializable data types. It aids in data transfer between endpoints and ensures efficient data storage in the database.

```
class NumpyArray(marshmallow.fields.Field):
    def _serialize(self, value, attr, obj, **kwargs):
        if value is None:
            return None
        return {"data": value.tolist(), "dtype": str(value.dtype), "shape": value.shape}

    def _deserialize(self, value, attr, data, **kwargs):
        return numpy.array(value["data"], dtype=value["dtype"]).reshape(value["shape"])
```

Listing 6.5: Custom Marshmallow Field for NumPy Arrays.

Metadata Endpoint and Interaction Endpoint Discovery: The metadata of plugins houses the definitions for interaction endpoints. Traditionally, the *EntryPoint* field in the metadata guides the QHana UI to a plugin's microfrontend. However, this thesis expands the *EntryPoint* field to encompass interaction endpoints. The *interactionEndpoints* field is essentially a list, potentially containing multiple interaction endpoints. Each list entry is a dictionary containing a **type** and an **href** field. The **type** field, a string, denotes the endpoint type. It is made user-friendly by providing the *InteractionEndpointType* enum in the shared folder, which lists all possible endpoint types for the optimization process. The **href** field houses the endpoint's URL. Listing 6.6 illustrates this using the *ridge-loss* plugin's metadata.

```
interaction_endpoints = (
    [
        InteractionEndpoint(
            # define the type of interaction endpoint
            type=InteractionEndpointType.of_pass_data.value,
            href=url_for(
                f"{RIDGELOSS_BLP.name}.{PassDataEndpoint.__name__}",
                _external=True,
            ),
        ),
        InteractionEndpoint(
            type=InteractionEndpointType.calc_loss.value,
            href=url_for(
                f"{RIDGELOSS_BLP.name}.{CalcCallbackEndpoint.__name__}",
                _external=True,
            ),
        ),
    ],
)
```

Listing 6.6: Metadata of the Ridge Loss Plugin.

For multi-step plugins, the interaction endpoint's definition in metadata slightly deviates. As discussed in 6.2, multi-step plugins incorporate a runtime-specific database ID into the URL. The *href* field functions as a format string that allows the substitution of a placeholder with the actual database ID. Listing 6.7 offers a glimpse into this, showcasing the *ridge-loss* plugin implemented as a multi-step entity. The invoking plugin replaces the placeholder with the actual database ID when known. This distinction is foundational to the differences between the two approaches, explored further in the subsequent section.

```
interaction_endpoints = (
    [
        InteractionEndpoint(
            # define the type of interaction endpoint
            type=InteractionEndpointType.of_pass_data.value,
            # define the endpoint URL
            # use the url_for_ie function to generate the URL with a task_id placeholder
            href=url_for_ie(f"{RIDGELOSS_BLP.name}.{PassDataEndpoint.__name__}"),
        ),
        InteractionEndpoint(
            type=InteractionEndpointType.calc_loss.value,
            href=url_for_ie(f"{RIDGELOSS_BLP.name}.{CalcLossEndpoint.__name__}"),
        ),
    ],
)
```

Listing 6.7: Metadata of the Ridge Loss Plugin as a Multi-Step Plugin.

6.7 Exemplary Minimizers and Objective Functions

The implementation encompasses two distinct minimizer plugins and three OF plugins to illustrate the capabilities of the plugin-based optimization framework. Specifically, the minimizer plugins *scipy-minimizer* and *scipy-minimizer-grad* and the OF plugins *ridge-loss*, *hinge-loss*, and *neural-network*.

Minimizer Plugins: At the core, both plugins use *scipy.optimize.minimize* as their minimization algorithm. The *scipy-minimizer* plugin leverages this method to minimize the loss function without needing a gradient. Its sole input hyperparameter is the minimization method, restricted to methods that operate without a gradient.

On the other hand, the *scipy-minimizer-grad* plugin employs the same method but incorporates the gradient for loss minimization. It accepts input hyperparameters that exclusively pertain to methods necessitating the gradient. Regardless of their approach, both plugins yield the weights that optimize the loss function's minimization.

Objective Function Plugins: The *ridge-loss* plugin encapsulates the ridge loss function, as depicted in 6.8. It operates with the regularization parameter *alpha* as its input hyperparameter. In a similar vein, the *hinge-loss* plugin embodies the hinge loss function, showcased in 6.9, and uses the regularization parameter *C* as its input.

The *neural-network* plugin employs a neural network featuring a single hidden layer. Its hyperparameters include the count of neurons present in this hidden layer. Notably, this plugin facilitates the gradient calculation for the loss function, making it compatible with the *scipy-minimizer-grad* plugin. The user is encouraged to refer to the code repository for a more in-depth understanding of how a more sophisticated method is implemented for loss calculation.

```
def ridge_loss(X: numpy.ndarray, w: numpy.ndarray, y: numpy.ndarray, alpha: float) -> float:
    # Calculate the predicted values using the weight vector
    y_pred = X.dot(w)

    # Compute the mean squared error between actual and predicted values
    mse = mean((y - y_pred) ** 2)

    # Calculate the ridge penalty using the weight vector
    ridge_penalty = alpha * sum(w**2)

    # Combine the mean squared error and ridge penalty to get the total ridge loss
    total_loss = mse + ridge_penalty

    return total_loss
```

Listing 6.8: Ridge Loss Calculation

```
def hinge_loss(X: numpy.ndarray, w: numpy.ndarray, y: numpy.ndarray, C: float) -> float:
    # Get the number of samples from the input matrix
    n_samples, _ = X.shape
    # Initialize the loss to zero
    loss = 0.0

    # Iterate over each sample in the dataset
    for i in range(n_samples):
        # Calculate the score for the current sample using the weight vector
        score = np.dot(w, X[i])

        # Compute the hinge loss for the current sample and accumulate
        loss += max(0, 1 - y[i] * score)

    # Apply the regularization term to the accumulated loss
    loss = C * loss / n_samples

    # Add the l2 regularization term to the loss
    loss += 0.5 * np.dot(w, w)

    return loss
```

Listing 6.9: Hinge Loss Calculation

6.8 Comparative Analysis of Plugin-Based Implementation Strategies

The implementation resulted in two distinct methods for constructing a plugin-based optimization framework. This section outlines the intrinsic differences that characterize each strategy.

6.8.1 Decoupled Plugin Approach

The foundational idea behind the first approach is promoting maximal decoupling between plugins, which particularly applies to the OF plugin. This idea entails minimizing data transfer between plugins and emphasizing data retrieval directly from the database whenever feasible.

Specifically, the OF plugin archives its hyperparameters in the database. The coordinator invokes the *pass_data* endpoint, which stores input and target data in the database. Thus, for the *pass_data* endpoint, which determines the number of input weights, the coordinator plugin only passes the input and target data since the OF plugin fetches the hyperparameters from the database. The loss calculation endpoints *calc_loss*, *calc_grad*, and *calc_loss_and_grad* solely need the weights as input, leveraging already stored hyperparameters and data. Listing 6.10 illustrates an exemplary schema for the *calc_loss* endpoint.

Consequently, for the minimizer plugin, there is no occasion where knowledge of the OF 's hyperparameters or its data is essential, leading to a lean input schema for the *minimization* endpoint, as depicted in Listing 6.11. However, it mandates that the task's database ID be communicated to the coordinator plugin through the microfrontend callback process, essential for managing interaction endpoint URLs, as discussed in 6.6.

```python
class CalcLossOrGradInputSchema(MaBaseSchema):
    x0: NumpyArray = NumpyArray(required=True, allow_none=False)
```

Listing 6.10: *CalcLossOrGradInputSchema* Schema for multistep-plugin approach.

```python
class MinimizerInputSchema(MaBaseSchema):
    x0 = NumpyArray(required=True, allow_none=False)
    calc_loss_endpoint_url = marshmallow.fields.Url(required=True, allow_none=False)
    calc_gradient_endpoint_url = marshmallow.fields.Url(required=False, allow_none=True)
    calc_loss_and_gradient_endpoint_url = marshmallow.fields.Url(required=False, allow_none=
True)
    callback_url = marshmallow.fields.Url(required=False, allow_none=True)
```

Listing 6.11: *MinimizerInputSchema* Schema for multistep-plugin approach.

6.8.2 Integrated Plugin Approach

The second strategy embodies a tighter integration. This approach does not implement the OF plugin as a multi-step plugin. Such an implementation implies that no shared context exists among the plugin's endpoints, necessitating the passing of all required data during each endpoint invocation.

This design requires the microfrontend callback process to relay the hyperparameters to the coordinator plugin. The coordinator then transmits the hyperparameters and data to the OF plugin's *pass_data* endpoint. In this setup, the minimizer plugin must also store the hyperparameters and data since it must relay this to the OF plugin's loss calculation endpoint. Listing 6.12 exhibits the input schema for the *calc_loss* endpoint, while Listing 6.13 demonstrates the *minimization* endpoint's schema. Passing the hyperparameters of a OF plugin ensures that neither the coordinator nor the minimizer plugin requires in-depth knowledge of the hyperparameters irrespective of the OF 's internal mechanics. As a trade-off, there is no need to relay a task's database ID to the coordinator.

```python
class CalcLossOrGradInputSchema(MaBaseSchema):
    x: NumpyArray = NumpyArray(required=True, allow_none=False)
    y: NumpyArray = NumpyArray(required=True, allow_none=False)
    x0: NumpyArray = NumpyArray(required=True, allow_none=False)
    hyperparameters: dict = marshmallow.fields.Dict(required=True, allow_none=False)
```

Listing 6.12: *CalcLossOrGradInputSchema* Schema for non-multistep-plugin approach.

```python
class MinimizerInputSchema(MaBaseSchema):
    x0 = NumpyArray(required=True, allow_none=False)
    x = NumpyArray(required=True, allow_none=False)
    y = NumpyArray(required=True, allow_none=False)
    hyperparameters = marshmallow.fields.Dict(required=True, allow_none=False)
    calc_loss_endpoint_url = marshmallow.fields.Url(required=True, allow_none=False)
    calc_gradient_endpoint_url = marshmallow.fields.Url(required=False, allow_none=True)
    calc_loss_and_gradient_endpoint_url = marshmallow.fields.Url(required=False, allow_none=
True)
    callback_url = marshmallow.fields.Url(required=False, allow_none=True)
```

Listing 6.13: *MinimizerInputSchema* Schema for non-multistep-plugin approach.

7 Result Validation

This chapter explains the outcomes achieved with the current implementation, emphasizing system performance, plugin interchangeability, and developer usability.

Performance benchmarks are conducted on a MacBook Pro. The system's detailed specifications include:

- **Model:** MacBookPro18,1
- **CPU:** Apple M1 Pro
- **RAM:** 32 GB
- **GPU:** Apple M1 Pro (16 Cores, Metal 3 support)
- **Operating System:** macOS Version 13.4.1

7.1 Benchmark Results

Given its recurrent invocation, the loss function call is the most resource-intensive phase of the minimization process. The benchmarks employ the *ridge-loss* plugin as the OF, configuring *alpha* at 0.5. The *scipy-minimizer* plugin acts as the minimizer, with the method set to *L-BFGS-B*.

Table 7.1 displays the frequency of loss evaluations during the minimization process for varying datasets, underscoring the significance of efficient loss function computation.

Number of Data Points	**Number of Evaluations**
200	264
400	341
600	370
800	430
1000	480
1200	520
1400	570

Table 7.1: Number of times the loss function gets evaluated during minimization across different dataset sizes.

Figure 7.1 shows the time it takes for a single call of the loss function for different dataset sizes for the decoupled plugin-based implementation approach as described in 6.8.1. The *Calculation Time* captures only the actual time it takes to calculate ridge loss. The *Database Time* measures the time

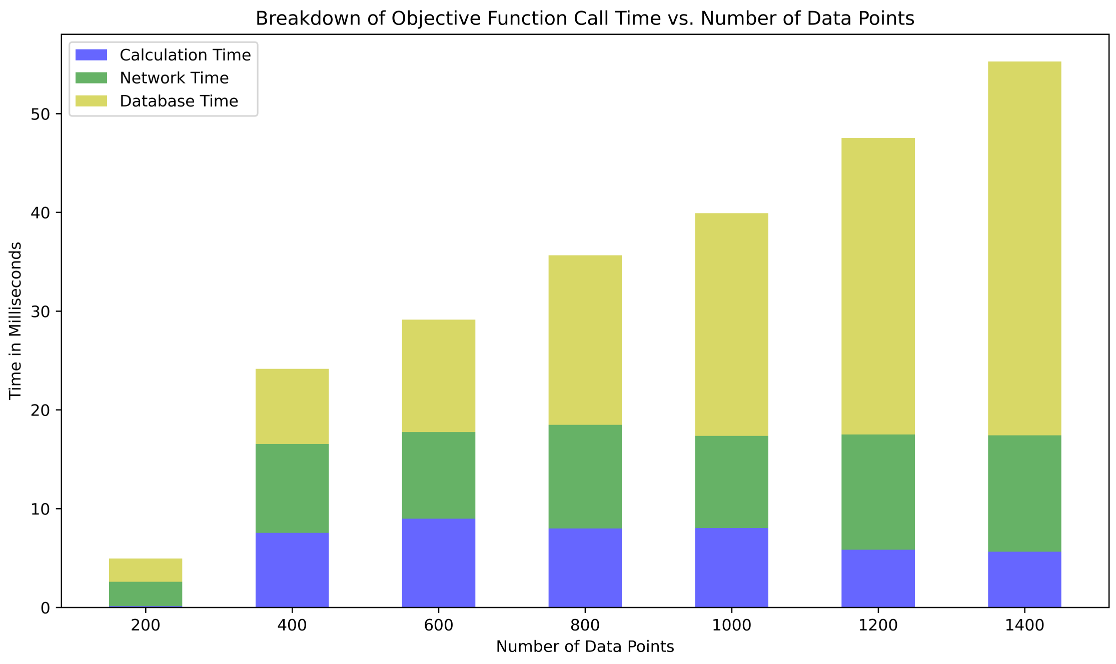

Figure 7.1: Time it takes for a single call to the *calc_loss* endpoint of the *ridge-loss* plugin for different dataset sizes for the decoupled plugin-based implementation approach.

it takes to retrieve the data from the database. The *Network Time* measures the time it takes to send the data to the OF plugin and receive the result. The total time it takes from the point at which the minimizer plugin calls the calculation endpoint until it receives the result is the sum of the *Calculation Time*, *Database Time*, and *Network Time*. The *Database Time* is the most significant contributor to the total time as the data is retrieved from the database for every call. With increasing dataset size, the time it takes to retrieve the data also increases. The *Network Time* is comparably shorter but still significant. It increases only slightly with an increasing dataset size since the data sent over the network does not change significantly.

Figure 7.2 captures the same measurements as 7.1 but for the integrated plugin-based implementation approach as described in 6.8.2. The diagram does not show the *Database Time* since this version does not make calls to the database. The *Network Time* significantly increases for this approach as the vast amounts of data previously retrieved from the database are now sent over the network. Additionally, the *Network Time* increases with increasing dataset size since the amount of data sent over the network also increases.

Figure 7.3 compares the total times it takes to get the loss value across different implementations, including the Jupyter Notebook implementation. The Jupyter Notebook implementation is the fastest as it does not have to deal with network or database latency.

Figure 7.4 measures the same metrics as Figure 7.3, but this time it ignores the *Database Time* and the *Network Time* only measuring the actual *Calculation Time*. The result shows that a plugin-based implementation approach does not significantly impact the time it takes to calculate the loss function.

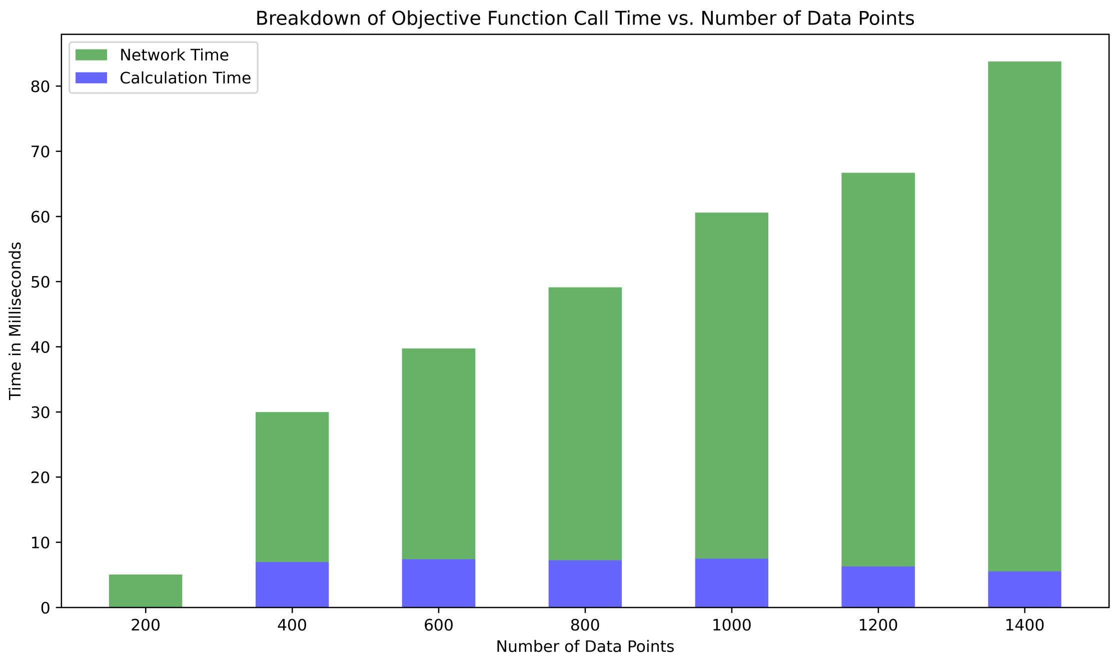

Figure 7.2: Time it takes for a single call to the *calc_loss* endpoint of the *ridge-loss* plugin for different dataset sizes for the integrated plugin-based implementation approach.

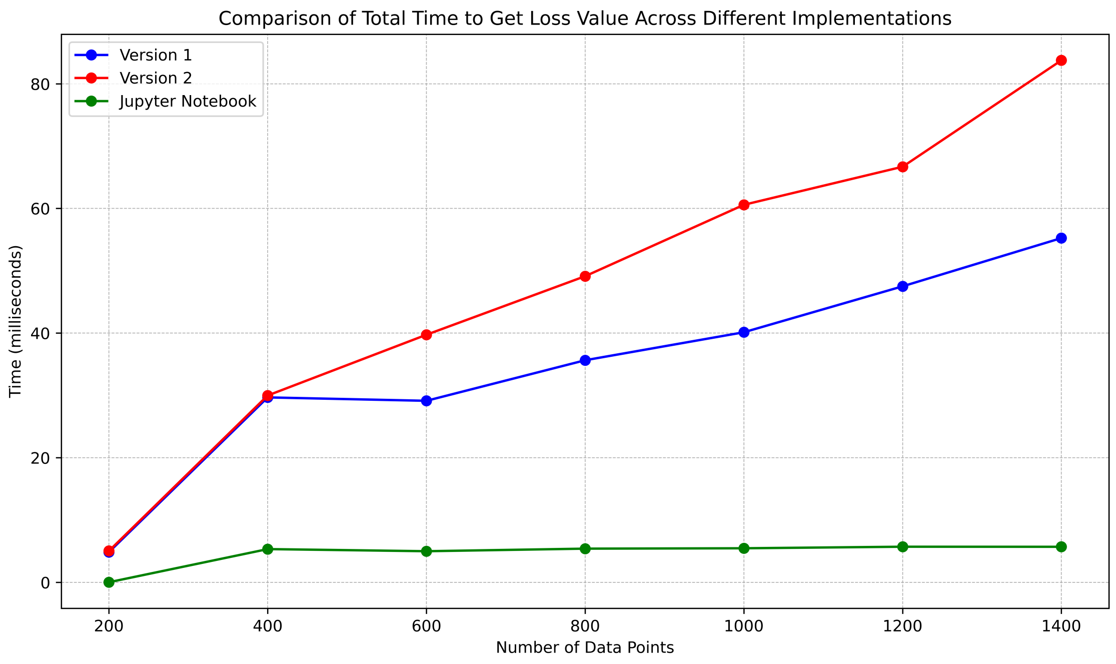

Figure 7.3: Comparison of the total time it takes to get the loss value across different implementations, including the Jupyter Notebook implementation.

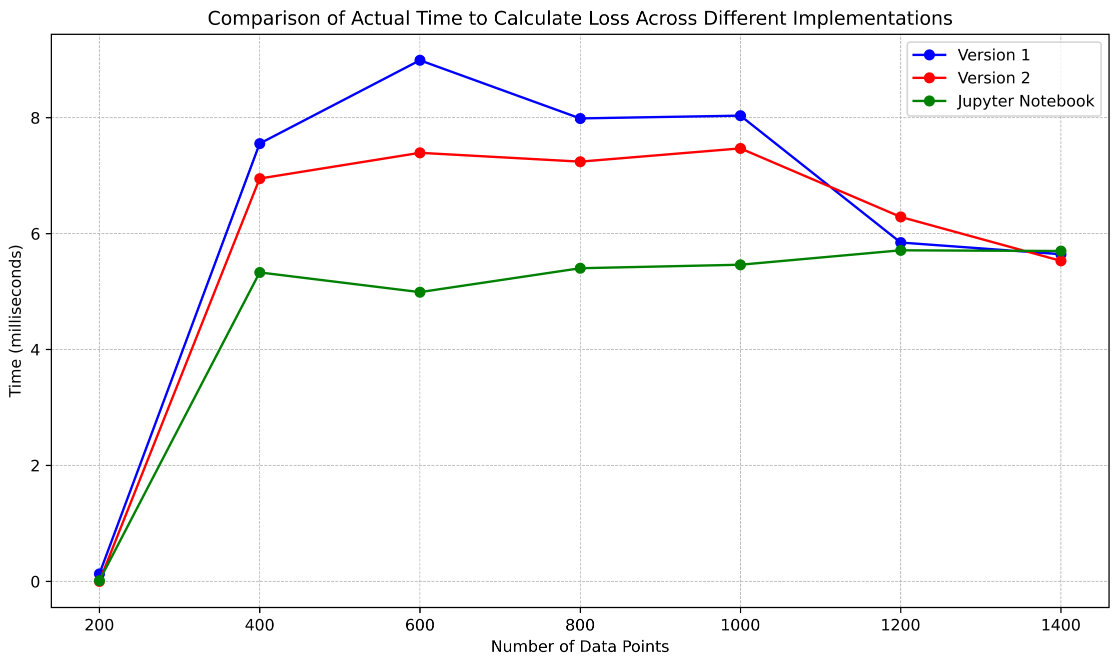

Figure 7.4: Comparison of the time the actual loss calculation takes across different implementations, including the Jupyter Notebook implementation.

One can see that the Jupyter Notebook implementation is the fastest across the board as it does not have to deal with network or database latency. Especially for the decoupled plugin-based implementation, the database latency is a bottleneck that significantly decreases the performance, but that can be mitigated by caching the data, as described in the next section.

7.2 Caching of Data

A caching mechanism is implemented for the decoupled plugin-based approach to avoid database latency. Since the input and target data and the hyperparameters do not change during the minimization process, they can be cached. That means the data is retrieved from the database during the first call of the *calc_loss* endpoint and cached for subsequent calls. The implementation uses the *flask-caching* library and keeps the cached data in memory. Figure 7.5 shows the average time for a single call of the *calc_loss* endpoint for different dataset sizes when using caching.

Figure 7.6 compares the total times it takes to get the loss across different implementations, including a cached version of the decoupled plugin-based implementation approach.

Finally, Figure 7.7 shows the total time it takes to complete the minimization process for the different implementations. The measurements for the plugin-based implementations do not include the time it takes a user to input data but instead start when the coordinator plugin calls the minimization endpoint of the minimizer plugin.

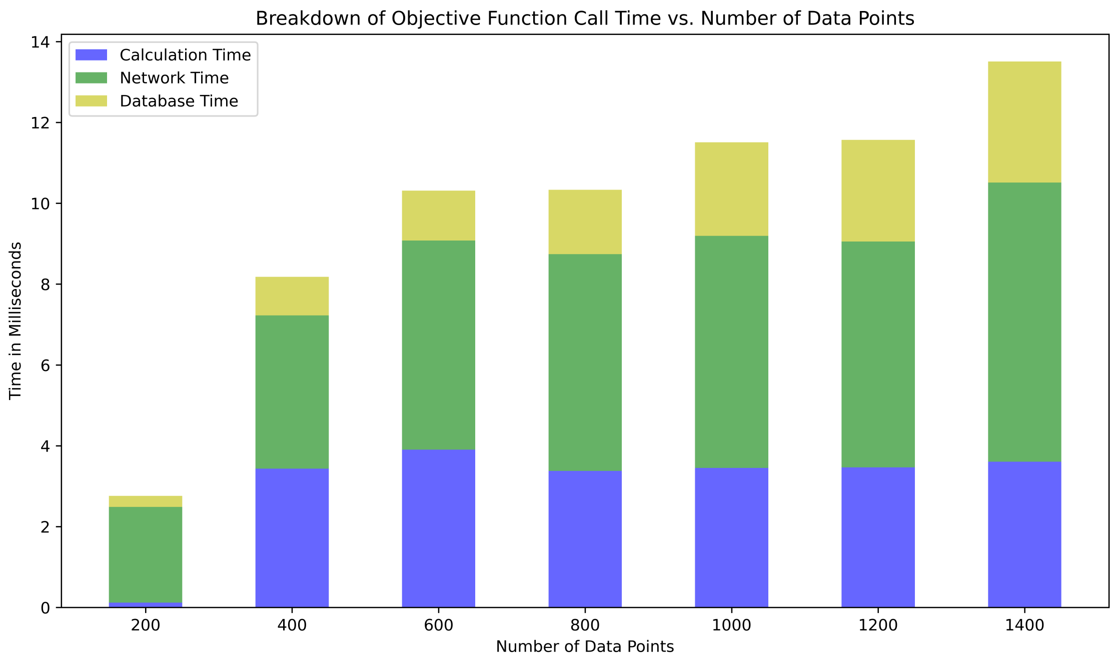

Figure 7.5: Time it takes for a single call to the *calc_loss* endpoint of the *ridge-loss* plugin for different dataset sizes with caching.

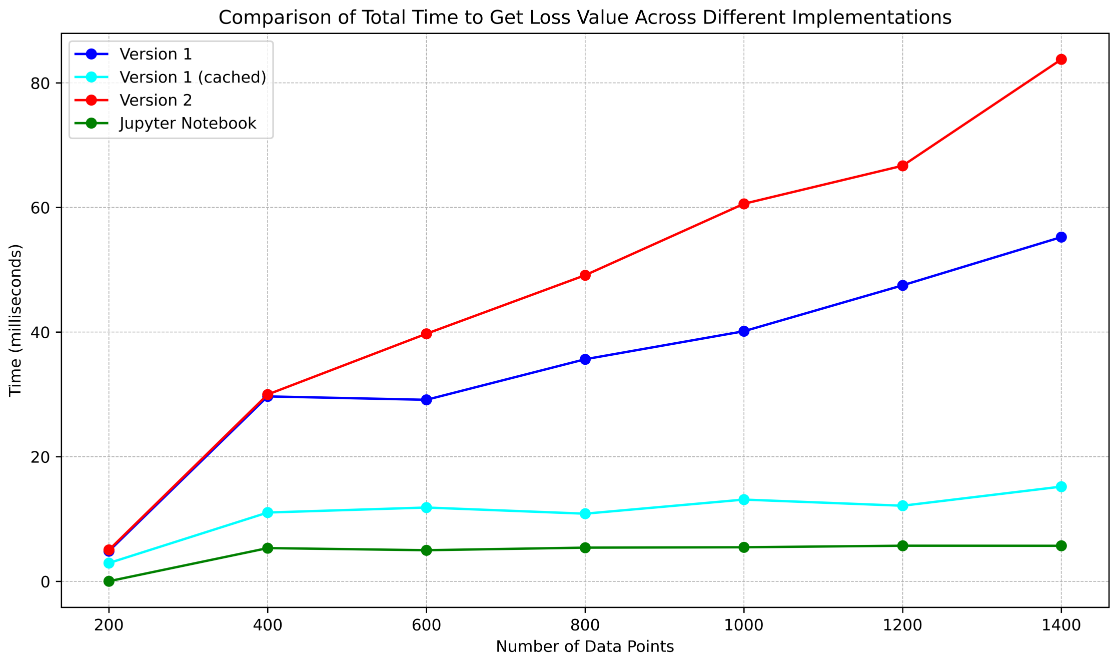

Figure 7.6: Comparison of the total time it takes to get the loss value across different implementations, including a cached version of the first plugin-based implementation approach.

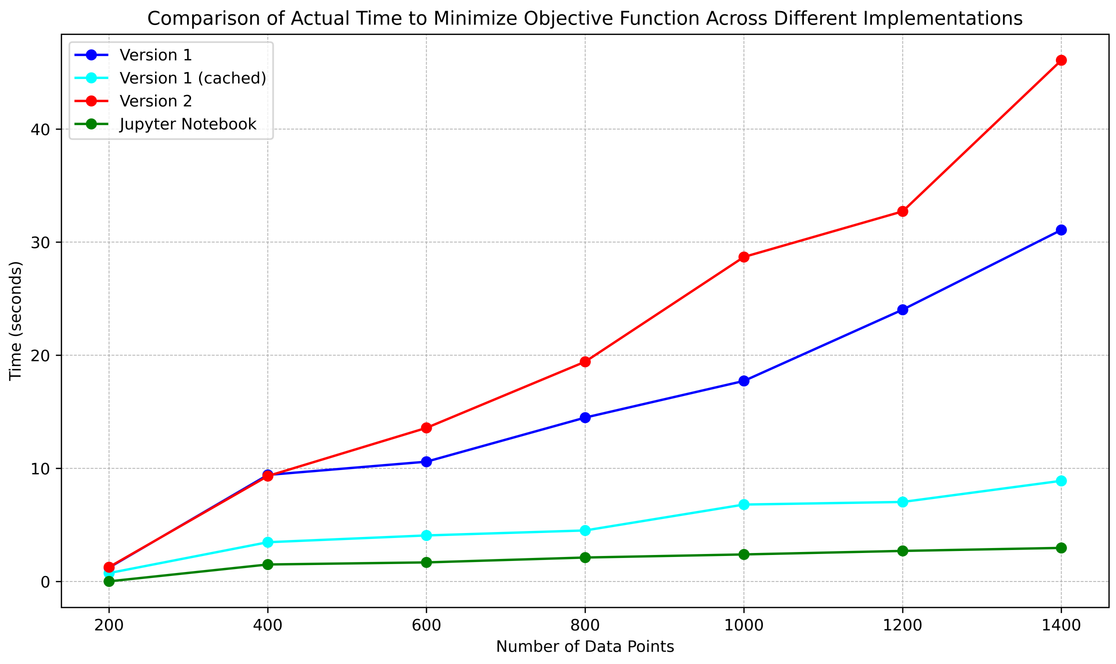

Figure 7.7: Comparison of the total time it takes to complete the minimization process across different implementations.

7.3 Interchangeability and Standardization

This segment evaluates the plugins' adherence to predefined standards and their interchangeability. The governing standards, as enumerated in Chapters 5 and 6, encompass:

- All invocable plugins should list interaction endpoints in their metadata.
- Such plugins must also offer interaction endpoints compliant with respective argument and response schemas.
- Both invoking and invoked plugins should uphold the callback mechanisms delineated in Section 6.5.
- Each plugin should retain its intended functionality as specified in Section 5.1.

Successful compliance ensures plugin interchangeability. All example plugins in the implementation segment conform to these standards, ensuring mutual compatibility. Users can combine any minimizer plugin with any OF without changing any code. It is important to note that to leverage the gradient calculation provided by the *neural-network* plugin, one has to use the *scipy-minimizer-grad* plugin.

The combination of all minimizer plugins with all OF plugins is extensively tested manually, and the results have been documented in Table 7.2 to ensure that the plugins are fully interchangeable.

	ridge-loss	hinge-loss	neural-network
scipy-minimizer	✓	✓	✓
scipy-minimizer-grad	✓	✓	✓

Table 7.2: Compatibility between example implementation of plugins.

7.4 Developer Usability Assessment

This section assesses the system's developer-friendliness. The steps required to implement new minimizer and OF plugins are documented to underpin the evaluation. For every new plugin, the first step is to copy an existing plugin into a new directory as a template.

Minimizer Plugin Development: For crafting a novel minimizer plugin:

1. Replicate the *scipy-minimizer* plugin into a distinct directory.
2. Update the plugin's name in the *__init__.py* file.
3. Integrate requisite hyperparameters into the *MinimizerSetupTaskInputSchema* schema.
4. Implement the minimization algorithm within the *minimize_task* function.

Objective Function Plugin Development: For devising a new OF plugin:

1. Duplicate the *ridge-loss* plugin into a fresh directory.
2. Modify the plugin's name in the *__init__.py* file.
3. Incorporate necessary hyperparameters into the *HyperparamterInputSchema* schema.
4. Implement the input weight calculation within the *of_pass_data* endpoint.
5. Construct the loss function's computation in the *calc_loss* endpoint.

The outlined steps demonstrate the approach to developing new plugins within the system. By starting with existing templates, developers are spared the repetitive tasks of setting up basic functionalities and can focus on the unique aspects of their specific plugins.

8 Discussion

This chapter delves into the critical discussions surrounding architectural choices, the model's practical execution, and the consequential outcomes. It starts by evaluating the plugin-based optimization framework's architectural decisions and how it is implemented. Subsequently, it analyzes the performance of the plugin-based optimization framework and compares it to the Jupyter Notebook implementation. Finally, this concept addresses how the results influence the developer experience when implementing new minimization and OF plugins.

8.1 Architecture and Implementation

Decomposing the minimization process into separate minimizer and OF plugins aligns perfectly with the plugin-based optimization framework's ethos. This delineation ensures clarity and specificity and establishes a transparent interface between the two plugins. The success of libraries like SciPy and how it handles the minimization process [VGO+20] reinforces the effectiveness of this decomposition. Entrusting the coordinator plugin with the responsibility of overseeing the optimization eliminates the need for the minimizer and OF plugins to manage interactions, enabling developers to concentrate solely on enhancing plugin functionalities.

Interaction endpoints serve as the core of the plugin-based optimization framework, offering a unified mode of interaction across plugins. This standardization supports developers by delineating available endpoints and their operational functionalities. By presenting these endpoints even before passing through the UI process, developers can now bypass the UI and directly trigger the requisite endpoint in a standardized and, thus, reliable way. Such a setup is invaluable for those seeking automation. By precisely defining input-output expectations, this approach also ensures seamless plugin interchangeability.

The callback mechanism is an essential facet of the plugin-based optimization framework. Its design ensures a distinct separation of responsibilities. The coordinator plugin, devoid of internal insights into the minimizer or OF plugin, solely relies on available endpoints. This mechanism prevents the need for database or API polling, offering updates upon minimization completion. The distinctive implementation of the callback mechanism for the minimization endpoint further underlines the modularity, as it fully decouples the procedure from the minimizer plugin. The event-driven model [HW11] drives adaptability for all kinds of asynchronous processes in QHana

The rationale behind the *pass_data* endpoint merits discussion. While it introduces an additional layer, the trade-offs justify its existence. Directly passing vast datasets as query parameters to the UI endpoint is not feasible. Transferring data as files to the UI endpoint burdens the OF plugin with the data extraction process, which should not be its responsibility. Receiving data as a NumPy array allows the OF plugin to focus on its primary role: loss value computation.

The two plugin-based implementation strategies show the trade-off between the complexity of handling interaction endpoints and the amount of data that has to be passed between plugins. The decoupled approach, where the OF plugin is implemented as a multi-step plugin, is more complex since the coordinator plugin has to format the interaction endpoint URL to reflect the database ID. Nevertheless, it also has the advantage that the minimizer plugin does not have to pass the hyperparameters and data to the OF plugin since the OF plugin retrieves them from the database. In general, that means by passing along a single parameter, the database ID, any volume of data can be loaded and used for context. The integrated approach, where the OF plugin is not implemented as a multi-step plugin, is less complex since the coordinator plugin does not have to handle formatting the interaction endpoint URL. However, it also has the disadvantage that the minimizer plugin has to pass the hyperparameters and data to the OF plugin since the OF plugin cannot retrieve them from the database. Both approaches have advantages and disadvantages and should show that the plugin-based optimization framework is flexible enough to support both approaches. As the benchmarking results show, by caching the data, the first approach is arguably the better one of the two.

Standardization is paramount in the plugin-based optimization framework. The metadata, interaction endpoints, callback protocols, and plugin functionalities are meticulously defined to foster uniformity. This organized structure creates seamless plugin interchangeability. The modular nature of this framework not only simplifies the optimization process but also encourages a modular development course, paving the path for universal plugin interactions within QHana.

8.2 Performance Insights

Performance metrics underscore the plugin-based optimization framework's viability when compared against the Jupyter Notebook model. Absent caching, both plugin methods suffer from network and database latencies. However, Figure 7.4 stands out, highlighting that the intrinsic time for loss function computation remains unaffected across methods.

Caching within the first plugin-based model increases performance significantly, as evidenced by Figure 7.5. However, network latencies persist and cannot be minimized further as data sent and received is minimal. Additionally, some database overhead is still present, as the data has to be retrieved from the database once and stored in the cache. As dataset sizes increase, these latencies diminish in relative significance. Additionally, the primary use case of this optimization framework is VQA, where the actual loss function calculation takes significantly longer than the network and database latencies. When fortified with caching, the decoupled plugin-based approach emerges as a compelling alternative to the Jupyter Notebook implementation.

8.3 Developer Experience

The plugin-based optimization framework manifests developer-centricity. Shielded from the difficulties of plugin interactions, developers can emphasize plugin functionalities. The regimented structure of the interaction endpoints liberates developers from concerns about other plugins. This contrasts with the Jupyter Notebook implementation, where developers must know the entire optimization process when implementing a new minimizer or OF. As the results in 7.4 show, implementing new minimizer and OF plugins is straightforward and can be done in a few steps. What simplifies the process even more is that all plugins of the same type have the same structure, meaning that the first step is only to copy an existing plugin. By copying an existing plugin, everything concerning the plugin's metadata, interaction endpoints, and callback protocols is already implemented. The standardized architecture means developers can focus on hyperparameter customization and core calculations during implementation. With several existing example plugins, developers can leverage these to understand how the core implementations are realized. Detailed documentation of the plugin-based optimization framework that explains the architecture and implementation further enhances the understanding.

9 Related Work

Beisel et al. [BBG+23] detail the implementation of VQAs within a workflow-based system. They delve into the intricacies of incorporating the quantum aspect of the algorithm into such a system while emphasizing the interaction between its various components less. In their approach, the conventional computing segment of the algorithm is split into two services: the *Objective Evaluation Service* and the *Optimization Service*. Similar to this thesis, the *Optimization Service* utilizes SciPy for optimization tasks.

In a subsequent paper [BBB+23], Beisel et al. elaborate on the architecture of the workflow-based system, shedding light on the interplay between its components. They explain that the system's components are realized as microservices, each offering a RESTful API for interaction. Central to this architecture is a *Gateway*, which handles the internal communication with the various microservices and presents a unified RESTful interface to the outside.

Mayer et al. [MMS03] advocate simplifying the intricacies of expansive software systems through a *plugin-based architecture*. They argue that the system gains flexibility and becomes more maintainable by minimizing the core system and encapsulating most functionalities as plugins. They introduce the concept of *Plugin-in Component Architecture* (PICA), where each plugin must provide a list of interfaces the core can call and a description of how to use them.

Wolfinger et al. [WDPM06] present a plugin architecture tailored for the .NET platform, drawing parallels with the Eclipse platform. The authors emphasize the structured interaction between plugins through the concepts of *slots* and *extensions*. A plugin host defines slots that indicate how to extend the plugin, while plugin contributors provide extensions to fill these slots. This structured interaction ensures seamless integration of plugins. The architecture also leverages .NET features to embed relevant plugin information directly within the application's source code, arguing for improved readability and maintainability.

Birsan et al. [Bir05] delve into the plugin architectures of the Eclipse platform. Birsan underscores the importance of well-defined *extension points*, allowing plugins to interact seamlessly. The work highlights the intricate interplay between plugins, emphasizing the need for structured interfaces for effective interaction.

Thullier et al. [THG21] introduce a *machine learning workbench*, emphasizing a modular approach to machine learning processes for smart homes. The workbench divides the machine learning workflow into distinct services: the *Windowing Module*, the *Feature Extraction Module*, and the *Machine Learning Module*. This modular design ensures efficient parallel processing and scalability. The paper does not aim to split the machine learning service into different components.

A similar optimization loop, wherein a OF is evaluated iteratively to minimize the loss value, is adopted by popular machine learning frameworks like TensorFlow [ABC+16] and PyTorch [PGM+19]. These frameworks process input data, compute the discrepancy or loss between predictions and actual values, and refine the model's parameters based on this computed loss. The iterative process continues until a specified stopping criterion, such as a predetermined number of epochs or a desired accuracy threshold, is met. While there are architectural and functional distinctions between TensorFlow and PyTorch, at a high level, their optimization loops embody this core principle. However, these frameworks do not offer a distributed optimization loop, a crucial feature of the plugin-based optimization framework.

In a comprehensive survey by Verbreaken et al. [VWK+20], the authors delve deep into the paradigms and challenges of *Distributed Machine Learning*. The authors categorize *Distributed Machine Learning* techniques based on their key characteristics, such as data distribution, model updates, and communication strategies. The study sheds light on various challenges, including communication overhead. They further highlight the significance of an efficient communication strategy, emphasizing the need for a robust communication protocol. As discussed in the result section of this work, the decoupled plugin-based implementation already minimizes communication overhead by caching data and reducing the amount of data sent over the network.

10 Conclusion and Outlook

This thesis embarked on the challenging journey of conceptualizing and realizing the plugin-based optimization framework within the QHana environment. What emerged is an innovative approach to optimization and a pioneering framework that establishes protocols for plugin interactions within QHana. This framework's versatility hints at its potential applicability beyond just optimization tasks.

The example implementations of both the minimizer and the OF plugins are a testament to the system's functionality and adaptability. Performance benchmarks highlight the framework's robustness and efficiency, especially with caching, trying to minimize network and database latencies from a distributed system. A particularly commendable aspect is the strict adherence to standardization, which ensures modularity and paves the way for seamless plugin interchangeability. This design philosophy also creates a developer-friendly system, simplifying plugin development while ensuring versatility.

The current ecosystem provides a substantial opportunity to diversify by introducing a broader array of minimizer and OF plugins. Expanding this repertoire will not only enrich the optimization solutions available but also enhance the robustness and adaptability of the system.

As the thesis lays out the complete foundation to implement VQAs, a new OF plugin that handles the quantum part would be a valuable and easily implementable addition. The energy state of a quantum system is then perceived as the loss value, while an existing minimizer plugin is used to minimize this loss.

Building upon the inherent interchangeability and modularity of the current design, a novel concept draws attention: viewing the OF or the minimizer as hyperparameters themselves. In a traditional setting, hyperparameters are parameters set before the learning process begins. However, given the modularity, one could envision running parallel optimization processes with different OF plugins but the same minimizer. This would be similar to a hyperparameter search, where each OF is a candidate and aims to find the best fitting OF for a given dataset or problem. Similarly, different minimizers can be benchmarked against a fixed OF.

In conclusion, with its modularity, standardization, and user-friendliness, the plugin-based optimization framework offers many opportunities. From diversifying its offerings to delving deep into quantum algorithms, the journey ahead is teeming with potential.

Bibliography

[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng. *TensorFlow: A system for large-scale machine learning*. 2016. DOI: 10.48550/ARXIV.1605.08695 (cit. on p. 63).

[Bar22] J. Barzen. "From Digital Humanities to Quantum Humanities: Potentials and Applications". In: *Quantum Computing in the Arts and Humanities*. Springer International Publishing, 2022, pp. 1–52. DOI: 10.1007/978-3-030-95538-0_1 (cit. on p. 15).

[BBB+23] M. Beisel, J. Barzen, M. Bechtold, F. Leymann, F. Truger, B. Weder. "QuantME4VQA: Modeling and Executing Variational Quantum Algorithms Using Workflows". In: *Proceedings of the 13th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2023. DOI: 10.5220/0011997500003488 (cit. on p. 63).

[BBG+23] M. Beisel, J. Barzen, S. Garhofer, F. Leymann, F. Truger, B. Weder, V. Yussupov. "Quokka: A Service Ecosystem for Workflow-Based Execution of Variational Quantum Algorithms". In: *Service-Oriented Computing – ICSOC 2022 Workshops*. Springer Nature Switzerland, 2023, pp. 369–373. DOI: 10.1007/978-3-031-26507-5_35 (cit. on p. 63).

[BBH+22] F. Bühler, J. Barzen, L. Harzenetter, F. Leymann, P. Wundrack. "Combining the Best of Two Worlds: Microservices and Micro Frontends as Basis for a New Plugin Architecture". In: *Service-Oriented Computing*. Springer International Publishing, 2022, pp. 3–23. DOI: 10.1007/978-3-031-18304-1_1 (cit. on pp. 15, 17, 18, 24, 26).

[Bir05] D. Birsan. "On Plug-ins and Extensible Architectures". In: *Queue* 3.2 (Mar. 2005), pp. 40–46. DOI: 10.1145/1053331.1053345 (cit. on p. 63).

[BL19] J. Barzen, F. Leymann. "Quantum humanities: a vision for quantum computing in digital humanities". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1-2 (Aug. 2019), pp. 153–158. DOI: 10.1007/s00450-019-00419-4 (cit. on p. 15).

[Cas17] D. Castelvecchi. "IBM's quantum cloud computer goes commercial". In: *Nature* 543.7644 (Mar. 2017), pp. 159–159. DOI: 10.1038/nature.2017.21585 (cit. on p. 17).

[E17] W. E. "A Proposal on Machine Learning via Dynamical Systems". In: *Communications in Mathematics and Statistics* 5.1 (Mar. 2017), pp. 1–11. DOI: 10.1007/s40304-017-0103-z (cit. on pp. 20, 25).

[EPR35] A. Einstein, B. Podolsky, N. Rosen. "Can Quantum-Mechanical Description of Physical Reality Be Considered Complete?" In: *Physical Review* 47.10 (May 1935), pp. 777–780. DOI: 10.1103/physrev.47.777 (cit. on p. 19).

[Fab] P. W. Fabian Bühler, ed. *QHAna-Plugin-Runner's documentation*. Comp. software. University of Stuttgart. URL: https://qhana-plugin-runner.readthedocs.io/ (visited on 09/30/2023) (cit. on pp. 26–28, 34, 35).

[GBCB17] I. Goodfellow, Y. Bengio, A. Courville, F. Bach. *Deep Learning*. Vol. 521. 7553. MIT Press, 2017, pp. 436–444. ISBN: 9780262035613. DOI: 10.1038/nature14539 (cit. on p. 21).

[HW11] G. Hohpe, B. Woolf. *Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*. Programming Simplicity, LLC, 2011, p. 683. ISBN: 9780321200686 (cit. on p. 59).

[MBB+17] N. Moll, P. Barkoutsos, L. S. Bishop, J. M. Chow, A. Cross, D. J. Egger, S. Filipp, A. Fuhrer, J. M. Gambetta, M. Ganzhorn, A. Kandala, A. Mezzacapo, P. Müller, W. Riess, G. Salis, J. Smolin, I. Tavernelli, K. Temme. "Quantum optimization using variational algorithms on near-term quantum devices". In: (2017). DOI: 10.48550/ARXIV.1710.01022 (cit. on p. 19).

[MMS03] J. Mayer, I. Melzer, F. Schweiggert. "Lightweight Plug-In-Based Application Development". In: *Objects, Components, Architectures, Services, and Applications for a Networked World*. Springer Berlin Heidelberg, 2003, pp. 87–102. DOI: 10.1007/3-540-36557-5_9 (cit. on p. 63).

[MRBA16] J. R. McClean, J. Romero, R. Babbush, A. Aspuru-Guzik. "The theory of variational hybrid quantum-classical algorithms". In: *New Journal of Physics* 18.2 (Feb. 2016), p. 023023. DOI: 10.1088/1367-2630/18/2/023023 (cit. on p. 19).

[MWG+21] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, U. Sarid, eds. *OpenAPI Specification v3.1.0*. Feb. 15, 2021. URL: https://spec.openapis.org/oas/v3.1.0 (visited on 10/15/2023) (cit. on pp. 24, 26).

[NC12] M. A. Nielsen, I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, June 2012. ISBN: 9781107002173. DOI: 10.1017/cbo9780511976667 (cit. on p. 19).

[NW06] J. Nocedal, S. Wright. *Numerical Optimization (Springer Series in Operations Research and Financial Engineering)*. Springer, 2006, p. 664. ISBN: 9780387303031 (cit. on pp. 21, 25).

[PGM+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. DOI: 10.48550/ARXIV.1912.01703 (cit. on p. 63).

[PMS+13] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, J. L. O'Brien. "A variational eigenvalue solver on a quantum processor". In: (2013). DOI: 10.48550/ARXIV.1304.3061 (cit. on p. 19).

[Pre98] J. Preskill. "Reliable quantum computers". In: *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 454.1969 (Jan. 1998), pp. 385–410. DOI: 10.1098/rspa.1998.0167https://doi.org/10.1088/1367-2630/18/2/023023 (cit. on p. 19).

[SB14] S. Shalev-Shwartz, S. Ben-David. *Understanding Machine Learning*. Cambridge University Press, May 2014, p. 410. ISBN: 9781107057135. DOI: 10.1017/cbo9781107298019 (cit. on pp. 20, 25).

[Sho] P. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE Comput. Soc. Press. DOI: 10.1109/sfcs.1994.365700 (cit. on p. 15).

[THG21] F. Thullier, S. Hallé, S. Gaboury. "LE2ML: a microservices-based machine learning workbench as part of an agnostic, reliable and scalable architecture for smart homes". In: *Journal of Ambient Intelligence and Humanized Computing* 14.6 (Oct. 2021), pp. 6563–6584. DOI: 10.1007/s12652-021-03528-8 (cit. on p. 63).

[VGO+20] P. Virtanen et al. "SciPy 1.0: fundamental algorithms for scientific computing in Python". In: *Nature Methods* 17.3 (Feb. 2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2 (cit. on pp. 21, 24, 25, 59).

[VWK+20] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, J. S. Rellermeyer. "A Survey on Distributed Machine Learning". In: *ACM Computing Surveys* 53.2 (Mar. 2020), pp. 1–33. DOI: 10.1145/3377454 (cit. on p. 64).

[WDPM06] R. Wolfinger, D. Dhungana, H. Prähofer, H. Mössenböck. "A Component Plug-In Architecture for the .NET Platform". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 287–305. DOI: 10.1007/11860990_18 (cit. on p. 63).

Appendix

```python
import numpy as np
import pandas as pd
from sklearn.datasets import make_regression

# Set random seed for reproducibility
np.random.seed(42)

# Example dataset size
size = 1000

# Number of features determined by size
n_features = int(np.sqrt(size) * 1.5)

# Create data
X, y = make_regression(n_samples=size, n_features=n_features, noise=10)

# Generate IDs for each row
ids = ["entity" + str(i) for i in range(1, len(X) + 1)]

# Create a dictionary to hold data
data_dict = {"ID": ids}
for i in range(n_features):
    data_dict[f"x{i+1}"] = X[:, i]
data_dict["y"] = y

# Create a DataFrame with the data
data = pd.DataFrame(data_dict)

# Save the DataFrame to a CSV file
data.to_csv("data.csv", index=False)
```

Listing 10.1: Source code for generating a sample test dataset for benchmarking with 1000 samples, 47 features, 10 noise and 1 target.

.

Declaration