Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Plugin-based Workflow Integration for QHAna

Vincenzo Pisano

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Fabian Bühler, M.Sc., Benjamin Weder, M.Sc.
Commenced:	January 14, 2022
Completed:	July 14, 2022

Kurzfassung

Mit der Einführung von Computern in den Geisteswissenschaften, welches die digitalen Geisteswissenschaften darstellt, werden computergestützte Werkzeuge und Methoden mit vielversprechenden Resultaten eingesetzt. Die Integration der Quanteninformatik in den Geisteswissenschaften wurde von Barzen and Leymann [BL20] vorgeschlagen unter dem neu eingeführten Begriff der *Quantum-Humanities*. Quantencomputer stellen eine Technologie dar, die dazu verwendet werden kann, um komplexere Probleme zu lösen aufgrund von Eigenschaften solcher Computer, die zum Beispiel große Datenmengen in einem Schritt verarbeiten können. Um die Vorteile der Quanteninformatik in den Geisteswissenschaften zu nutzen, bietet das QHAna Projekt eine Reihe von Algorithmen für maschinelles Lernen, welche den Nutzern in Form von Plugins zur Verfügung stehen. Derzeit werden Workflows die mehrere QHAna Plugins verwenden manuell von Nutzern ausgeführt. In dieser Arbeit wird ein QHAna Workflow Plugin vorgestellt, das den Prozess der Workflow Verwaltung automatisiert, dazu werden Design-Entscheidungen und Implementierungsdetails diskutiert.

Abstract

With the introduction of computers in the humanities that constitutes the digital humanities, computational tools and methods are employed to assist researchers with promising results. Recently, the integration of quantum computing into the humanities was proposed leading to the term *quantum humanities* introduced by Barzen and Leymann [BL20] where quantum computers represent a new technology that promises to solve more complex problems due to properties of such computers, e.g., large amounts of data can be processed in a single step. To take advantage of quantum computing in the humanities the QHAna project provides a toolset of machine learning algorithms that are available to the users in form of plugins. Currently, workflows requiring the invocation of multiple QHAna plugins are carried out by users manually. In this work a QHAna workflow plugin is presented that uses a business workflow engine for the automation of such workflows, for this design choices and implementation details are discussed.

Contents

1 Introduction **11**

2 Background **13**
2.1 Workflow Technology . . . 13
2.2 Quantum Computing . . . 13
2.3 Digital and Quantum Humanities . . . 15
2.4 MUSE . . . 15
2.5 Quantum Humanities Analysis Tool (QHAna) . . . 16

3 Related Work **21**
3.1 Quantum Software Workflows . . . 21
3.2 Sharing Quantum Software . . . 23
3.3 Workflow Orchestration Platforms . . . 23

4 Integration Design **29**
4.1 Requirement Analysis . . . 29
4.2 Design Decisions . . . 30

5 Implementation **33**
5.1 Plugin Usage and Endpoints . . . 34
5.2 Clients . . . 35
5.3 Watchers . . . 35
5.4 Routes and Schemas . . . 42
5.5 Exceptions . . . 44

6 Conclusion and Outlook **47**

Bibliography **49**

List of Figures

2.1 Quantum Humanities Analysis Tool (QHAna) UI plugin list overview 20

3.1 Relevant phases from the simplified quantum software lifecycle 22

4.1 QHAna workflow meta-plugin simplified overview 30

5.1 Extended overview for the QHAna workflow meta-plugin 33
5.2 Sequence diagram for external components 39
5.3 Sequence diagram for the human task watcher 40
5.4 QHAna workflows meta-plugin instance status 41
5.5 Human task micro frontend of a demo workflow for the QHAna workflows meta-plugin 41

List of Tables

5.1 Structure of the workflow meta-plugin REST API 43

List of Listings

3.1 Example configuration file for a workflow using Orquestra, adapted from [Zap] . 24
3.2 Example workflow containing a sub-workflow using Covalent, adapted from [Agn] 26

Acronyms

API Application Programming Interface. 16

BPEL Business Process Execution Language. 22

BPMN Business Process Model and Notation. 11

HPC High Performance Computing. 23

MUSE MUster Suchen und Erkennen. 11

NISQ Noisy Intermediate-scale Quantum Era. 11

QHAna Quantum Humanities Analysis Tool. 8

QuantME Quantum Modeling Extension. 21

REST REpresentational State Transfer. 16

UI User Interface. 19

1 Introduction

Various areas such as medicine or chemistry can benefit from the use of quantum computing, where quantum computers provide properties such as being able to process vast amounts of data that allows for new problems to be approached [NC02]. An example represents the humanities where Barzen et al. [BFL18] outlines with MUster Suchen und Erkennen (MUSE) a use-case for the humanities where classical and quantum resources may be employed to facilitate research on vestimentary communication in films. For this, the identification of costumes using clustering and classification is required, a process that may be suited for the use of Noisy Intermediate-scale Quantum Era (NISQ) computers where quantum machine leaning algorithms could provide a substantial speedup compared to classical variants [RWJ+14]. To assist researchers with aforementioned addition of quantum resources within the field of humanities, QHAna represents a project that provides a toolset of machine learning techniques that can be run on classical and quantum computers and allows for users without a background in quantum computing to benefit from the use of quantum algorithms [Bar22].

Currently, QHAna incorporates three main components, namely the plugin runner that hosts a set of plugins implementing various algorithms, a web interface that can be used to manage plugins, i.e., create new instances, view instance progress and outcomes, and a backend. Use-cases within the quantum humanities often involve workflows comprising multiple algorithms, i.e., to produce results a set of QHAna plugins is necessary where users need to invoke multiple plugins in a specific order. As of now such workflows need to be carried out manually, thus users are required to keep track of intermediary results that are then potentially used as input for subsequent plugin invocations. This however leads to a time consuming and error prone process since not all plugin inputs can be specified at the beginning of a workflow and it is possible that intermediary results are mixed up when specifying inputs.

To tackle these challenges, the workflow meta-plugin introduced in this work implements the orchestration of QHAna plugins with the use of Camunda, an open-source workflow automation platform, and Business Process Model and Notation (BPMN) as the imperative workflow language for creating workflow models integrating hybrid workflows into QHAna, i.e., classical and quantum computers are integrated. The term meta-plugin is used to indicate that this plugin is able to invoke other QHAna plugins. Workflow models designed for the use with the newly presented plugin may utilize all available BPMN constructs, for this mappings are defined, e.g., between QHAna plugin invocations and native constructs alongside with workflow instance variable mapping for plugin input and output. Furthermore, the plugin provides components for workflow management such as workflow model deployment, instance creation and status update polling from selected Camunda queues and invoked QHAna plugins. Workflow instance variables can be updated through gathering input from the user by displaying human tasks on the QHAna web interface. Additionally, to leverage the benefits of using an imperative workflow language such as BPMN the plugin defines several exception types that can be thrown during workflow execution to allow for exception handling whilst modelling workflows.

Outline

The work is structured in the following way:

Chapter 2 - Background provides necessary basic information for the implementation of this work.

Chapter 3 - Related Work identifies and discusses relevant work.

Chapter 4 - Integration Design examines requirements and choices made for the implementation of a QHAna workflow meta-plugin.

Chapter 5 - Implementation explains functionality and implementation details for main components of the QHAna workflow meta-plugin.

Chapter 6 - Conclusion and Outlook summarizes the results of this work and illustrates possible approaches for future contributions.

2 Background

Fundamentals of this work are covered in this chapter, this includes specifying the term workflow used throughout the following chapters. The benefits of quantum computing and potential for integrating such a technology within the humanities is examined with a concrete use-case that represents the MUSE project. Furthermore, QHAna is introduced.

2.1 Workflow Technology

Within the context of this work the term workflow correlates directly to the specification of business processes using imperative workflow languages, such as BPMN. Thus, an imperative workflow model consists of activities as units of work where the partial order and data flow is represented by gateways. Activities, e.g., link to a specific script stored alongside the workflow model or invoke an external implementation, alternatively human task activities can be used to update workflow variables by gathering input from the user [Ant20]. Imperative workflow languages such as BPMN support nested structures through the use of subprocesses where, e.g., activities and gateways can be grouped into a subprocess and viewed as a singular activity. Activities and gateways are connected by directed edges that define the order of execution in a workflow. Using imperative workflow languages can be advantageous since many languages provide exception handling, i.e., events are constructs that can be used to activate a flow edge depending on the condition that the event type implements. For example, the error boundary event is pinned to an activity and activated when that activity encounters an exception. Furthermore workflow languages such as BPMN offer transactional processes where activities can be automatically rolled back in case of an exception during runtime. Transactions can also be utilized for activities that cannot be rolled back, therefore a set of compensation methods are used in this case. The use of workflow technologies may be beneficial since scalability and robustness are two key properties offered [Ell99].

2.2 Quantum Computing

In the past the addition of quantum resources has benefited areas such as medicine or chemistry, where performing the largest chemistry simulations to date [IEE22] included quantum computers, such simulations can involve molecular reactions where the computational complexity increases exponentially with increasing molecules. Results may yield insights that are, e.g., applicable to the automotive industry where the development of new batteries with better properties contribute to the usage growth of electric vehicles. The integration of quantum technology into research and industry may be of interest due to following properties of such computers:

1. Certain problems can be solved faster [RWJ+14]
2. Large amounts of data can be processed in a single step [NC02]
3. Results produced are potentially much more precise compared to classical computers [HCT+19]
4. Results can be computed for some problems that were considered practically unsolvable due to time complexity [NC02]
5. There exist problems that are only solvable on quantum computers [RT19]
6. Usage of quantum computers is expected to be cheaper compared to classical supercomputers [C D19]

Although aforementioned properties assume an error-corrected quantum computer that does not yet exist, it is already possible for current quantum computers to provide a substantial speedup in solving certain problems that are of relevance to both research and industry. NISQ computers represent a near term solution for quantum computing albeit with constraints such as a limited number of available qubits in the range of 50 to a few hundred qubits and are not fault tolerant. NISQ computers available today are faster than classical computers for certain problems [RWJ+14] but the hardware used to solve such problems is susceptible to noise errors [Pre18] and thus performing computations on quantum computers requires the mitigation of such errors. During runtime errors due to environmental interactions lead to incorrect states, i.e., the result observed can deviate from the actual end state if no external influence would occur. The process of state changes over time due to external factors affecting qubits is also called decoherence [NC02]. An additional source of influence represents internal operations such as the action of performing measurements during runtime [EBL18], e.g., in order to retrieve the final state a measurement is required where it is possible that faulty measurements alter the end result.

Executing circuits on quantum hardware is a probabilistic operation, i.e., running an algorithm with the same input multiple times may produce distinct results each time due to, e.g., external and internal influences. Therefore multiple executions are needed to retrieve a probability distribution where the most frequent result is used [LBF+20; RP11] since the distribution can be affected by different factors such as faulty measurements. The probability distribution that is retrieved by running an algorithm with the same input multiple times needs further post-processing where an error mitigation task is used to change the distribution by dampening the effect of noise errors on the outcome. Error mitigation tasks are used to lessen the influence of errors during runtime and are dependent on the particular quantum hardware that is used, i.e., implementations for post processing tasks need to be adapted when different quantum computers are used since different error models apply [MZO20; WBL+20], thus the implementation demands deep mathematical and technical knowledge [WBLW20]. Furthermore, executing a quantum circuit requires not only post-processing but also pre-processing tasks. Running quantum circuits on available NISQ computers allows only the initialization of all zero states [Ley19], i.e., the state preparation pattern is used where a subcircuit is added to an existing quantum circuit to generate the required input for the algorithm during runtime [WBL+20].

2.3 Digital and Quantum Humanities

With the introduction of computers, the digital humanities employs computational tools and methods to assist researchers applying hermeneutical approaches, facilitating the process of, e.g., extracting and organizing information to allow for the interpretation of text [BL20]. Digital humanities incorporates the use of computers along with methods from computer science with promising results [Ber12; BL20]. Further research in the field of humanities may profit from the use of quantum computers that represent a new potential given their characteristics such as being able to solve certain complex problems due to a higher computing capacity, therefore the use of classical and now also quantum resources may further increase the productivity of researchers as seen before in the digital humanities.

The integration of quantum technology was proposed in social sciences in the past [KH13] and is now considered for the use in the humanities where the properties outlined in Section 2.2 are of interest. Despite the shortcomings of near term solutions, quantum computing allows for existing problems from the field of humanities such as literature, media science or history studies to be approached in new ways. To justify the usefulness of quantum computing in the humanities use-cases need to be outlined. More specifically, problems that benefit from quantum computing in the humanities are those that include clustering and classification steps [BL20]. A use-case that involves currently available quantum computers is outlined by Barzen et al. [BFL18] where the identification of costumes from films is based on quantum machine learning algorithms, thus necessary steps involve clustering and classification of clothes. These steps are executed on, e.g., NISQ computers that are part of a process representing a real world example where such technologies provide a speedup compared to classical computers and therefore underline the potential for research in the humanities.

2.4 MUSE

An application for the quantum humanities is illustrated by the MUSE project that aims to facilitate research on vestimentary communication in films. Through vestimentary communication information is delivered to the observer in a nonverbal manner, i.e., the goal should be to familiarize certain characters, the context of a scene or the overall setting of a film.

Barzen et al. [BFL18] propose a formal method that defines syntax and semantics to capture costumes enabling the creation of a collection usable for further studies. Based on a data set that can be used, MUSE, an acronym which stands for “search and identifying patterns” (dt. “MUster Suchen und Erkennen”), introduces methods to identify costume patterns and describes the implementation of a toolchain for such a task. A pattern hereby refers to a concept for capturing an abstract solution to a frequently occurring problem regarding a specific context, e.g., within the context of costumes a common question might be how a Queen is represented where a pattern for the role Queen provides the answer by specifying relevant vestimentary features. Answering complex questions can be achieved with the use of a pattern language that comprises a collection of interconnected patterns for a specific domain, e.g., software design patterns or architectural patterns. As an example, certain character roles in films are often associated with the same type of costumes, reoccurring features include specific colors or the type of clothing materials used. These features are helpful to classify such characters and especially important for supporting roles in films where screen time is limited

and thus creating a sense of familiarity with the character despite a short duration in front of the camera is necessary. An observer should feel familiar with that character without the need of much talking or moving through vestimentary communication, e.g., due to reoccurring features such as colors, materials used for partial or entire costumes and the way someone carries the costume [BFL18]. Patterns can be used to identify such features.

The concept behind MUSE involves the observation that films contain knowledge about costumes and their usage for vestimentary communication, the MUSE toolchain aims to detect and process this information for research and practical applications. Therefore, the goal is to capture costumes and create costume patterns through analysis that can be used to study the influence and methods of costumes for communication. An example for a costume pattern might be the pattern for a sheriff, thus the MUSE-Toolchain converts many sheriff costumes to one pattern named "sheriff" containing all significant features that classify this stereotype, e.g., colors, materials or how costumes are worn [BFL18]. Components contained within MUSE are a MUSE-Repository that holds concrete film costumes, the MUSE-Analysis-Tool that includes a multi step analysis to abstract costumes into costume patterns and a pattern repository named PatternAtlas, formerly known as PatternPedia [FBFL15]. MUSE includes the first data set analyzed as a use-case for the quantum humanities.

2.5 Quantum Humanities Analysis Tool (QHAna)

The Quantum Humanities Analysis Tool (QHAna) provides a toolset of machine learning techniques that allows users to run plugins on classical and quantum hardware. Plugins are software additions extending the functionality of QHAna. Such plugins implement, e.g., key steps in the MUSE-Workflow such as clustering and classification algorithms. Therefore, the implementation of a quantum algorithm like quantum k-means and the visualization of resulting data can be accomplished through the use of plugins in QHAna. Although the data set included with MUSE represents the first use-case, QHAna can be used for any data set. Plugins are utilized by QHAna users, external programs or other plugins and are managed by a plugin runner where a set of plugins can be hosted and executed. There are currently three supported types of plugins, namely processing, visualization and conversion plugins. Processing plugins receive input data that is then used by the underlying implementation to produce new data, visualization plugins consume data and return a human-readable representation, and conversion plugins receive input data in a given format and output the data in a different format. The plugin runner hosts a set of plugins and it has a REpresentational State Transfer (REST)-Application Programming Interface (API) with a database. Flask is a web application framework used by the plugin runner, i.e., it is a python library that contains a collection of libraries and other modules for writing web applications without having to work on low level aspects such as thread management or protocols. As a microframework, Flask offers a simple core where other functionality can be added through Flask extensions. Users interact with QHAna through the QHAna UI outlined at the end of this section. Information in this section is based on [QHA].

2.5.1 Plugin input and output

QHAna plugins can specify plugin dependencies that are required for a successful invocation, i.e., to terminate successfully the plugin may need to invoke the dependency. Specifying a plugin dependency can be done by explicitly stating the name of the plugin but it is also possible to define filtering methods that are less strict. This can be done by only allowing processing type dependency plugins, or by stating that the dependency plugin should, e.g., contain the tag “my-helper” or using “!bad-tag” to indicate that the dependency plugin should not have the tag "bad-tag". Additionally, a version range can be listed for dependencies, e.g., to exclude older versions. When invoking a plugin that expects one or more plugin dependencies the dependency is passed by reference, i.e., the link to the root endpoint of that plugin dependency is used.

Apart from dependencies used as plugin inputs it is possible to provide files as input. File inputs are provided by reference, i.e., using the URL to the file, this includes multiple protocol schemes such as ’http(s)://’, ’file://’ or ’data://’. Passing files as reference is required since it allows large files to be opened as streams where the data is read incrementally, reducing memory consumption. Sometimes this is not only helpful but also necessary since input files can be too large to fit into memory on the machine that runs the plugin.

QHAna plugin inputs are specified in the root endpoint where a list of inputs is included. Each input contains information such as the parameter name, whether the input is required, the content type of the input, i.e., the encoding used for the data (for example application/json for JSON or application/xml for XML), and the data type used for data semantics. The content type specified does not need to match a specific type since it is also possible to use wildcards to allow for a set of content types given an input, e.g., ’application/*’ can be used to allow content types representing any kind of binary data. Thus, ’application/json’ or ’application/xml’ are valid content types if the input content type is set to ’application/*’.

To produce results QHAna plugins are able to compute intermediate files, such files are used to share data between plugin steps, i.e., between plugin (step) entry tasks. Intermediate files are shared between tasks using the associated id of a file, i.e., files should not be passed by value. Plugins produce intermediate and result files that are stored in a FileStore, by default this FileStore is configured to use the local file system although additional FileStores can be added by plugins. Result data is associated by specifying a file name, content type and data type.

2.5.2 Micro Frontend

In QHAna users interact with or provide input to plugins through the use of a micro frontend. Micro frontends are a similar concept to microservices where a monolithic structure is replaced by reusable components of a manageable size. QHAna loads each micro frontend in an iframe with each micro frontend generally containing HTML, CSS and javascript. To gather input for a plugin instance, a micro frontend is served to the user that contains an HTML form where each plugin input parameter corresponds to a form field. The micro frontend is sent when a user selects a plugin or when providing input for a new plugin instance step. Developers can create plugin micro frontends easily through the use of template macros that provide useful features such as generating native form elements from marshmallow schemas. When creating micro frontends developers can choose to mark specific form fields that are defined within the javascript part of micro frontends.

As a result such form fields are treated differently by the QHAna UI. For example, a form field can be marked with the attribute 'data-token', thus a password input can be treated as a token for an API that the developer specifies alongside the attribute. A form field with aforementioned attribute and context set to IBMQ will use the contents of the field as a token for any calls made to the IBMQ API. Another example represents the attribute 'data-private' that can be used to tell QHAna that such inputs should never be stored in permanent storage, a use-case for this are passwords. The attribute 'data-content-type' will allow users to only use inputs of the specified data content types. QHAna micro frontend forms include a validate and submit button, for which the attribute 'data-submit' can be used with values 'validate' or 'submit'. If any button is marked as validate then the user will see a validated micro frontend upon pressing the button. The contents of the user form are sent to the plugin runner that then returns a validated frontend, i.e., essentially identical to the non-validated frontend except for highlighted form fields where the user selected an invalid input. A button marked as submit will simply initiate the request to the entry task endpoint with inputs from the form fields.

2.5.3 Plugin data formats and data loaders

QHAna plugins may utilize other QHAna plugins by specifying dependencies or by including invocations to other plugins built in. This enables a higher degree of re-usability where existing plugins can be used in different scenarios. However re-usability can be limited when plugins are written by a multitude of developers, each using different data types describing data semantics for input and output of plugins. Thus, it can be difficult to follow the different formats used, resulting in the need for new data loaders each time existing loaders cannot be used with a newly encountered format.

To facilitate the use of data produced by other plugins QHAna defines a set of data (serialization) formats that can be used by QHAna plugins to exchange data. Thus plugins should use the following formats whenever possible to ensure compatibility across plugins, i.e., output files should follow formats defined for QHAna plugins, retaining the possibility for other plugins to use generated output files.

The format, i.e., the data model, is derived from the MUSE data model since QHAna is used for analyzing data produced by the MUSE project. With the specified format the data consists of attributes each including a name and values where names are of type string and values can be of any type. It should be noted that the name may not be identical to the reserved attributes 'ID', 'source', 'GRAPH_ID', 'entities' or 'relations'. The meaning of these attributes will be elaborated in the following. Attributes can be used in the input data to specify entities, i.e., a collection of attributes where the entity contains a name, an id that can be used to distinguish multiple instances of that entity, and a list of attributes. The id of an entity must be unique for all entities of the same type. When working with entity instances, all attributes can be modified but it is considered best practice to not alter the id of an entity instance, this is useful for tracking changes made to the entity. An entity contains a URL that points to the original location of an entity and may additionally contain an arbitrary number of additional attributes where the values can be of any supported type, i.e., boolean, numbers, strings, dates and times, enumerations, locations or the id of another entity allowing for complex entity types with nested structures.

The data format also specifies relations between entities, such relations are directed and point from one entity to another entity, i.e., a 'source' and 'target' entity is specified. Relations may also contain additional attributes that describe the relation in more detail. Entities and relations can be used together in constructs such as a graph that contains a set of entities and relations. A relation in a graph is only valid if both entities that are part of the relation are also contained within the set of entities of that graph, i.e., it is not possible to include an entity that is not part of the graph in a relation. Graphs can be of different types that can be used to not only describe directed graphs but also undirected, acyclic graphs, trees or lists. Graphs, like entities and relations, may also contain additional attributes.

Attributes for entities, relations or graphs can also be associated with attribute metadata. Such metadata contains useful information about an attribute, namely a title, description and the type of the attribute, e.g., boolean, integer, number, string, url, ref or a user defined type. Furthermore, it is possible to specify whether the attribute contains more than one value and if the order of the values is important as well as the separator used.

Using data formats specified by QHAna whenever possible is beneficial since QHAna also provides data loaders that can be used to load data that follows the introduced data format. Loaders for entities accept data in CSV or JSON content types, if the data is provided as CSV file(s) then the file must contain the 'ID' of all entities included as the first column and the link to the original source for each entity as second column such that the helper methods provided by QHAna can be used to load the entities, i.e., if the input data does not follow this requirement then a new data loader would be necessary. Using the JSON content type allows for an arbitrary order of attributes where each entity is specified in a separate line. This allows users to stream input data by processing one object at the time, this is especially helpful when the input data does not fit in memory.

2.5.4 QHAna UI

The QHAna User Interface (UI) was created using `Angular`[1]. Angular is a platform that can be used to build mobile and desktop web applications, it is based on the programming language Typescript and is an open source web application framework. The interface for QHAna is built with novice users in mind. The home page shows a list of experiments that the user has created by providing a name and optionally a description. An experiment contains a page with multiple tabs for general information, plugin listing, available result data and plugin instance status. The information tab can be used by the user to write notes about the experiments where notes can be written with markdown support. A new tab is used to list all available plugins as illustrated in Figure 2.1. Upon selecting a plugin the QHAna UI will try to fetch the microfrontend for the entry plugin task from the root endpoint of a plugin using a GET request. The user can then view the plugin form containing input fields that the user should fill out before submission. A POST request to the entry plugin task (entry processing task in case of a processing type plugin) is sent when the user presses the submit button of the form which results in a new plugin instance. Users can view the current status of the plugin instance after submitting the form in a separate tab where all instances are listed. Once the plugin has terminated the result files can be downloaded, generated logs and information about runtime

[1] https://angular.io/

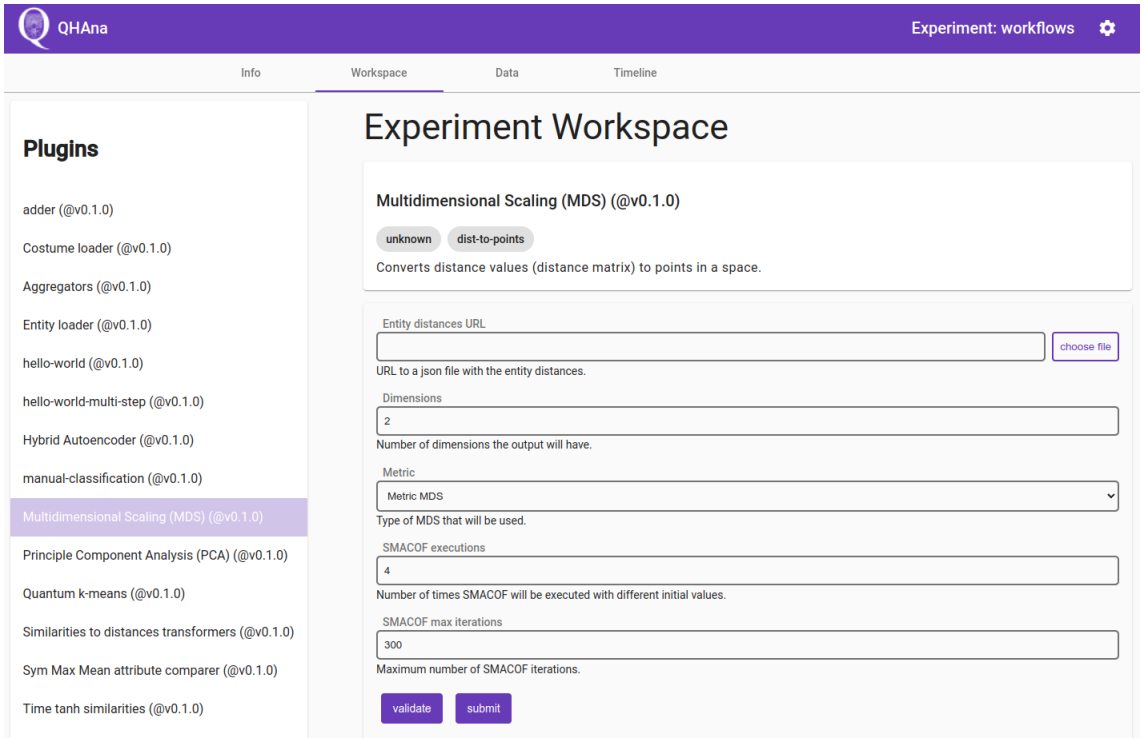

Figure 2.1: QHAna UI plugin list overview

length can be viewed. Resulting data is listed in the result data tab, users can select such data as input for other plugins. Furthermore, the QHAna UI includes a settings page where users can set the QHAna backend and provide URLs for plugin or plugin runner endpoints.

3 Related Work

There exists past work that focuses on workflows using quantum computers. In this chapter, two workflow orchestration platforms are introduced, compared and examined if such platforms can be used for the integration of workflow execution into QHAna. Furthermore, this chapter discusses the modeling extension for quantum circuits QuantME and the topic of sharing quantum software.

3.1 Quantum Software Workflows

The MUSE project provides an application for the use of quantum computers in the humanities, although for the execution of quantum algorithms further observations are required.

Due to the challenges outlined in Section 2.2, integrating quantum computing into classical workflow orchestration is therefore difficult as it requires the knowledge typically only acquired by quantum experts. To facilitate the task of creating hybrid workflows Weder et al. [WBLW20] introduce the modeling extension Quantum Modeling Extension (QuantME) for the imperative workflow languages such as BPMN that can be used to model quantum circuits, i.e., QuantME is a technology independent modeling extension which can be used to model quantum-classical workflows without exposing the details of each task used to execute a quantum circuit. A downside of introducing modeling extensions is reduced portability, for this the corresponding tool Quantum4BPMN is used to convert workflow models created using QuantME to native BPMN workflows, thus components of the extension can be mapped to constructs in BPMN. The conversion to native workflows is especially important since current workflow engines do not natively support the invocation of quantum circuits [WBL+20].

Weder et al. [WBLV21] explicitly introduce workflows as an entity and show how different stages of a simplified quantum software lifecycle as illustrated in Figure 3.1 can be represented with QuantME using extension components [WBLW20]. Different stages of the simplified quantum software lifecycle are examined in the following:

1. **Development phase**
   A quantum circuit is created in the development phase where a circuit incorporates an input state, gates to manipulate the quantum registry, e.g., to produce an input state from a zero state at the beginning of a run, and different type of measurements to retrieve information on a state. Therefore, quantum circuits can be seen as a blueprint to solve problem instances that can be used on NISQ computers. In this stage quantum circuits may include an oracle [Mos08] that allows to invoke a black-box function specific to the problem instance. The oracle then needs to be replaced by a concrete implementation and this step is known as an Oracle Expansion [LBF+20; Mos08].

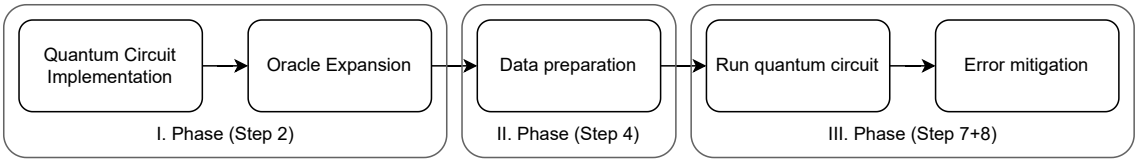

Figure 3.1: Relevant phases from the simplified quantum software lifecycle [WBLW20] mapped to corresponding steps in the quantum circuit lifecycle [WBLV21]

2. **Data preparation phase**
   The second stage of the software development lifecycle represents the data preparation phase where the goal is to run a given circuit on a selected quantum computer. At this point a quantum circuit is already available as a result from the development phase but it cannot be run on NISQ computers. The reason for this issue is that quantum computers available today can only run quantum circuits if initialized in an all zero state, this however is not feasible since in most cases the underlying algorithm is expected to be run with different inputs other than zero. The consequence is that quantum circuits from the first phase need to be adapted by prepending a subcircuit that is able to generate the required input state during runtime. Modifying a circuit for data preparation happens before deployment because inputs are generally not known beforehand, thus this subcircuit is integrated before deployment and execution which requires technical and mathematical knowledge [WBLW20].
3. **Execution and error mitigation phase**
   In the last stage the quantum circuit is deployed and run on a quantum computer or simulator followed by an optional error mitigation step where results influenced by readout errors are mitigated based on the error model of the used quantum computer [WBLW20].

To implement the steps in each of the three stages the use of workflow technologies seems promising [WBLW20], e.g., using a workflow language such as BPMN or Business Process Execution Language (BPEL). The use of workflow languages is beneficial since it allows to create workflow models describing, e.g., the execution order of a set of activities or the data flow during runtime. These workflow models can be used to achieve a certain goal through the orchestration of components with the use of a workflow engine [Ell99; LR99] where the use of a standardized workflow language such as BPMN means that different workflow engines can be used depending on the preference of the user and thus result in higher portability.

Workflows are not only used in business processes but also for running complex scientific simulations [Ell99; GSK+11; LR99], and are suitable for the presented stages of a quantum software lifecycle where tasks within stages can be represented as activities in a workflow model. Currently no modeling support for quantum applications using imperative workflow languages exists, therefore the modeling process without explicit support can be difficult and error prone, i.e., often only achievable by quantum experts. This issue can be mitigated by the introduction of abstraction layers, i.e., steps of the quantum software lifecycle are represented as single activities in a workflow model, thus users without deep technological knowledge can reuse these activities in various workflows without needing to know about the underlying implementation [WBLW20].

3.2 Sharing Quantum Software

Beyond running quantum algorithms and software, the distribution is another important aspect that should be inspected. Creating and managing algorithms or software for quantum computing is a different process compared to performing such tasks for traditional computers, e.g., different implementations may be required for available quantum hardware [TQ19]. A substantial amount of users are not sure if existing algorithms are suitable for a given situation, thus a lot of algorithms already exist but the task of selecting an algorithm and converting to specific quantum hardware is one that requires deep knowledge [LBF19]. As such Leymann et al. [LBF19] propose a quantum software platform that integrates a quantum algorithm catalog containing algorithms from various sources. This platform includes a public community that is able to move verified algorithms, i.e., algorithms fulfilling certain criteria, to the quantum algorithm repository.

The verification process involves members or crawlers searching for potential algorithms, community members discuss the maturity of found algorithms, where changes may be discussed until the algorithm is either discarded or moved to the quantum algorithm repository. As part of the review process the algorithm is analyzed, this results in metadata including information on source location, problems solved and properties of the algorithm, e.g., required qubits, being added [LBF19]. Developers can implement verified algorithms from the quantum algorithm repository that can be run on a set of quantum hardware and are stored in a quantum program repository.

3.3 Workflow Orchestration Platforms

The following section does not follow the introduced description for workflows in Section 2.1 but rather mentions the term in the context of data processing workflows adhering to the map-reduce programming model. Data processing workflows process large amounts of data using the map-reduce model in three phases, namely map, shuffle and reduce. For example, if the data set contains the grades of students then a possible task is to count for each grade the number of occurrences, i.e., in the map phase the data set is split into chunks and distributed across a set of map processors to use parallelism for a speedup in workflow execution where each processor creates intermediate results containing, e.g., a list of '1's for each grade observed in the respective chunk. For each grade all intermediate results from the map processors are merged, this is also known as the shuffle phase. Lastly, in the reduce phase all '1's of a list are added resulting in the count for a grade [CCA+10]. Thus, data processing workflows following the map-reduce programming model only define these three phases and do not include any of the components or characteristics introduced in Section 2.1.

Researchers can decrease the time required to run experiments by automating, e.g., the manual tasks of configuring systems used in the experiment, deploying code, executing algorithms and collecting data from results that is then moved across virtual machines, High Performance Computing (HPC) or clusters for subsequent tasks. Workflow orchestration platforms provide tools to run experiments as workflows where time intensive tasks that were carried out manually are now automated. Platforms enabling the execution of hybrid workflows often include easy to use deployment methods to a variety of quantum or classical backends. In the following two such platforms, namely Zapata Orquestra [Zap] and Agnostiq Covalent [Agn], are examined.

Listing 3.1 Example configuration file for a workflow using Orquestra, adapted from [Zap]

```yaml
apiVersion: io.orquestra.workflow/1.0.0
name: example-workflow

imports:
- name: some-algorithm
type: git
parameters:
  repository: "git@github.com:user/repo.git"
  branch: "main"

steps:
- name: first-step
config:
  runtime:
    language: python3
    imports: [some-algorithm]
    parameters:
      file: some-algorithm/file.py
      function: foo
outputs:
- name: foo
  type: data

- name: second-step
passed: [first-step]
config:
  ...
inputs:
  - data: ((foo.attribute))
    type: data
outputs:
  ...

types:
- data
```

3.3.1 Orquestra

With Orquestra, Zapata Computing introduces a platform for scientists and enterprises to run quantum-classical workflows on a suite of quantum hardware and classical resources. Users can create, deploy and execute workflows on local machines, cloud infrastructures or HPCs as well as quantum computers or simulators. During workflow creation, YAML configuration files are used to describe workflows, tasks declared within a workflow lead to the invocation of, by default, specified Python scripts during runtime. The addition of user defined languages allows for task implementations that are not written using Python [Zap].

Listing 3.1 shows a configuration file representing a workflow named `example-workflow` containing two tasks `first-step` and `second-step` (Orquestra uses the term step). Tasks must specify a runtime configuration including imports and input parameters, imports may point to external implementations through the specification of git repositories. Orquestra additionally provides a set of built-in implementations. Included algorithms are either publicly available or proprietary and

can be used in workflows without linking to a repository. By default tasks will run in parallel, a chain of sequential tasks may be constructed through the use of dependencies. The example uses the attribute `passed` for task `second-step` to indicate that the execution may start only after completion of the task `first-step`. Workflows can be built using Jinja templates, e.g., if the output of a preceding task is a list, it is now possible to use loops where a new task for each element in the list is created and all tasks can be run in parallel. However this does not allow for complex workflows since such templates are of limited use as they need to be replaced manually before the workflow is executed. For example, if in a template loop a task is defined then the loop is replaced by pasting the task multiple times and therefore the number of iterations must be known before running the workflow. Within a workflow configuration file the user can utilize different deployment locations for each task, allowing for hybrid workflows where both quantum hardware and classical resources are used.

To run a workflow it must be submitted to the Quantum Engine, a collection of components for the execution and result store of workflows. The Quantum Engine contains a workflow engine that integrates the Argo workflow engine, thus configuration files representing Orquestra workflows cannot run natively and must be converted to Argo workflows as an intermediary step by a transpiler. Furthermore, the execution order of tasks is handled by the transpiler. Due to current limitations it is not possible to write workflows that embed sub-workflows, a feature that would allow for the creation of meta-workflows composed of simpler building blocks and result in less code duplication since tasks do not need to be copied over to other workflows. Code duplication can lead to issues, e.g., following the observation that currently assigned resources are insufficient for a given task, a configuration change in resource requirements would have to be made for every usage location, a manual operation that might result in missed changes for one or more locations. The Quantum Engine fulfills a typical requirement for orchestration platforms that allow for the execution of quantum algorithms namely being able to run algorithms on quantum or classical backends, i.e., ensuring that a compatible version with the selected backend is deployed.

Data management is handled within Orquestra by a correlation service where multiple databases are used to store workflows and instance results including user logs produced during workflow execution. A task can produce artifacts, a collection of data representing an object, e.g, a circuit or molecule, that is then stored for later usage as input to a different task or as part of the workflow output. An output corresponds to a single result file that contains user logs, artifacts and other information such as execution start and ending times. Additionally, users can generate access links to share data such as configuration files.

The Quantum Engine exposes endpoints through a REST API making it possible to embed the orchestration platform into applications, performing actions such as submitting files or retrieving data requires the use of the authentication service. It should be noted that Orquestra workflows can only be used with the Orquestra Engine provided by Zapata, i.e., users need to pay in order to utilize the platform. Orquestra offers a command line interface and a graphical interface as an extension for Microsoft Visual Studio Code for users to interact with the platform.

3.3.2 Covalent

A different option for the use of workflow orchestration platforms represents Covalent by Agnostiq where users are offered similar functionalities to Orquestra, i.e., create, deploy locally or to cloud infrastructures, HPCs as well as quantum computers or simulators. Covalent offers additional

Listing 3.2 Example workflow containing a sub-workflow using Covalent, adapted from [Agn]

```python
import covalent as ct

@ct.electron
def return_excitement(s1, s2):
    return f"{", ".join([s1, s2])}!"

@ct.lattice
def workflow_a(s1, s2):
    phrase = return_excitement(s1, s2)
    return phrase

sub_workflow = ct.electron(workflow_a)

@ct.lattice
def workflow(s1, s2):
    phrase1 = sub_workflow(s1, s2)
    phrase2 = sub_workflow(s2, s1)
    return phrase1, phrase2

dispatch_id = ct.dispatch(workflow)("Hello", "World")
```

features compared to Orquestra such as increased re-usability by allowing the inclusion of sub-workflows and the option to create more complex workflows. Creating workflows for experiments as a researcher using Covalent can be accomplished by writing Python scripts. An example is provided in Listing 3.2 where it is demonstrated that existing implementations for algorithms can be turned into tasks, called electrons in Covalent, with one line of code using Python decorators. Tasks may contain any valid Python code and the invocation of sub-tasks from within a task is allowed. An advantage for the use of Python scripts for workflows over configuration files is that input and output parameters along with types are inferred from existing implementations and do not need to be listed in additional files [Agn].

Workflows can be defined as Python functions that are able to call tasks and include conditional statements or loops. However, unlike tasks only a restricted subset of valid Python is allowed within such functions as certain restrictions imposed by Covalent apply. For example, if a workflow function invokes one or more task functions and the result is, e.g., a list then reading items or slices from that list works, trying to iterate over a result list, assigning a different value to items or getting the length of that list will lead to an invalid workflow that cannot be run. In general object manipulation in such methods is not allowed, thus any modification to a task result has to be carried out in a separate task. When a workflow is submitted, as a first step Covalent analyzes the code: all functions marked as workflows are invoked while tasks are excluded. Thus, if a workflow function tries to, e.g., modify the value of a result object the operation will lead to an exception since the object is undefined. The consequence is that such functions cannot be used to manipulate the results of tasks, instead they orchestrate tasks to create a workflow.

Covalent is able to execute tasks in parallel by identifying independent parts of a workflow. Conditional statements and loops are supported, loops that invoke tasks are also executed simultaneously. This is beneficial since tasks may require a lot of time to compute resulting in a substantial runtime reduction.

The conversion of experiments into workflows can help researchers save time by automating previously manually carried out tasks, after finishing the development of a workflow it is conceivable that researchers may want to run the experiment using different inputs. When using Orquestra multiple configuration files, one for each input, are required. Therefore tasks are duplicated across configuration files, this decreases the overall maintainability and clutters the workspace. Listing 3.2 shows how `workflow_a` is converted to a sub-workflow and then used in a different context where multiple instances are created with different input parameters. This enables the creation of meta-workflows where nested experiments can be performed, more complex sequences can be built upon simpler existing constructs. A workflow that includes a single task represents the simplest construct, in contrast it is possible to create more advanced constructs such as workflows with multiple layers where each layer from the bottom up results in further abstraction.

Migrating to Covalent is made easier by providing helper functions that allow the user to run tasks implemented in different programming languages, i.e., by default other languages besides Python are supported. In case a language is not supported, many tools exists to facilitate the `integration`[1].

Covalent is built upon a microservice architecture, in the following existing services and interactions are discussed. When a workflow is submitted one of the first steps that the Covalent dispatcher server, a component that handles incoming requests, performs is create a directed acyclic dependency graph. A dependency graph can be used to illustrate how tasks with inputs and outputs relate to each other or the current execution status by highlighting completed and ongoing tasks. The dispatch service receives requests from a queue consumer that is connected to a queue service via a message service, thus the user submits a new or modified workflow through the Covalent SDK that is then forwarded to the queue service. Once ready, the dependency graph is sent to the workflow runner and the user interface. Covalent offers a user interface where information such as task status, metadata or the dependency graph as well as results can be viewed. It is possible to track ongoing workflows with the user interface, showing finished and ongoing tasks, task results, start and ending times where results and metadata can also be retrieved by code through the Covalent SDK.

Before running experiments on external classical or quantum hardware it is recommended that researchers execute their workflows on local machines due to the potentially high computational costs of such external resources, thus during development issues can be fixed and tests can be run locally until a production ready version is achieved. Running tests locally is possible since the tool is open source. Once a workflow is executed a result object and a workflow instance identifier is created that can be used to pull status updates and results within code, the runner service implements multiple types of executors, an executor is responsible for taking a task and deploying to a resource. Depending on the type of executor tasks are either deployed locally or to cloud infrastructures, HPCs as well as quantum computers or simulators. Covalent provides a set of executors that can be used in workflows, users can also create custom executors to include additional resources that can be used.

The result service manages result objects, i.e., it retrieves, stores and updates workflow outcomes whereas result objects are used for reproducible workflows by including data such as task input parameters, output and metadata. Further information is added such as computation start and end times, computation status and a log of print statements produced by tasks. A use case for such a feature would be the option to run a workflow once for a subset of available backends. The Covalent

[1] https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages

user interface displays results once a run has terminated and provides tools for researchers to analyze data. Unlike Orquestra Covalent does not provide built-in task implementations that can be used, one advantage although is that the project is `open-source`[2].

3.3.3 Comparison and limitations

The documentation for Orquestra does not provide any information on the implementation of subworkflows, a feature that is present within the documentation for Covalent. Additionally, users may only define loops with Orquestra where the number of repetitions are known before runtime, a requirement that is not always fulfilled. Furthermore, with Covalent workflows can be more easily run with different input parameters by converting the workflow to a subworkflow that is then treated as a task, Orquestra requires the user to create a configuration file for each experiment workflow input combination. This results in duplicated workflow fragments scattered across multiple files where modifying parts of the workflow can result in missed changes in one or more configuration files. Covalent also supports other languages and is open-source. Using Orquestra can be less challenging for novice users as the configuration files are relatively simple and do not require the user to write any code.

The main reason for not utilizing workflow orchestration platforms such as Orquestra or Covalent for the integration of workflows into QHAna is that both options ultimately realize the automated execution of data processing workflows. Thus, many important features such as exception handling, transactions, events or the ability to visually model workflows are not supported. Furthermore, Orquestra and Covalent provide no audit trail and no extensive recoverability.

[2] https://github.com/AgnostiqHQ/covalent

4 Integration Design

This chapter focuses on the requirements and design decisions for the implementation of a QHAna workflow meta-plugin. This includes the workflow engine Camunda and workflow language BPMN where mappings between QHAna components and BPMN components are defined. Furthermore, this chapter introduces watchers that perform periodic status updates on a set of endpoints.

4.1 Requirement Analysis

QHAna users are able to create new plugin instances using the QHAna UI, oftentimes use-cases for QHAna involve the execution of multiple plugins in a workflow. As of now the user thus needs to manually proceed through the workflow, i.e., for this, plugins are manually invoked and the resulting data needs to be provided as input for consecutive tasks by the user. It is not unusual for an experiment to require several plugin invocations where managing workflows in a manual way is time consuming and error prone, e.g., oftentimes multiple workflow instances need to be run using the same workflow model implementing the experiment but with different workflow variable assignments. Therefore, the user needs to keep track for each workflow instance the progress and acquired intermediary results without confusing any of the data when completing plugin input forms. Manual management of workflows is a time consuming task where plugins sometimes run for many hours, as such plugins may terminate during the night and the workflow instance may not proceed further until the next day at earliest. Therefore, for each intermediate step in the workflow the user is required to provide input through the use of plugin forms. Executing workflows in an automatic way can help with utilizing computational resources during the night since the results of a QHAna plugin can be used as inputs for subsequent plugin invocations without requiring the user's attention which can help save time for users as there is no need to wait for results during the day.

In this work a meta-plugin for the integration of workflow execution into QHAna is presented that aims to solve the issues mentioned above with running workflows manually. The primary requirement for this new meta-plugin is the capability to deploy and run a workflow model on a workflow engine to automate the process of workflow management. Figure 4.1 offers a first simplified view over invocations between various components. Allowing for the use of as many constructs as possible provided by the workflow language when defining a workflow model is of main interest, i.e., defining a model for the use with the new meta-plugin should not result in a restriction to the existing possibilities when using the workflow language. An additional requirement for the meta-plugin is that it must be able to update workflow instance variables by gathering new input from the user during workflow execution, by using a micro frontend that displays a form the user should complete, and then use a set of workflow instance variables to invoke QHAna plugins with the necessary inputs. This plugin should be implemented as a QHAna processing plugin since both visualization and conversion type plugins are not suitable for the task at hand. When a workflow meta-plugin instance terminates the user should be able to view the results of

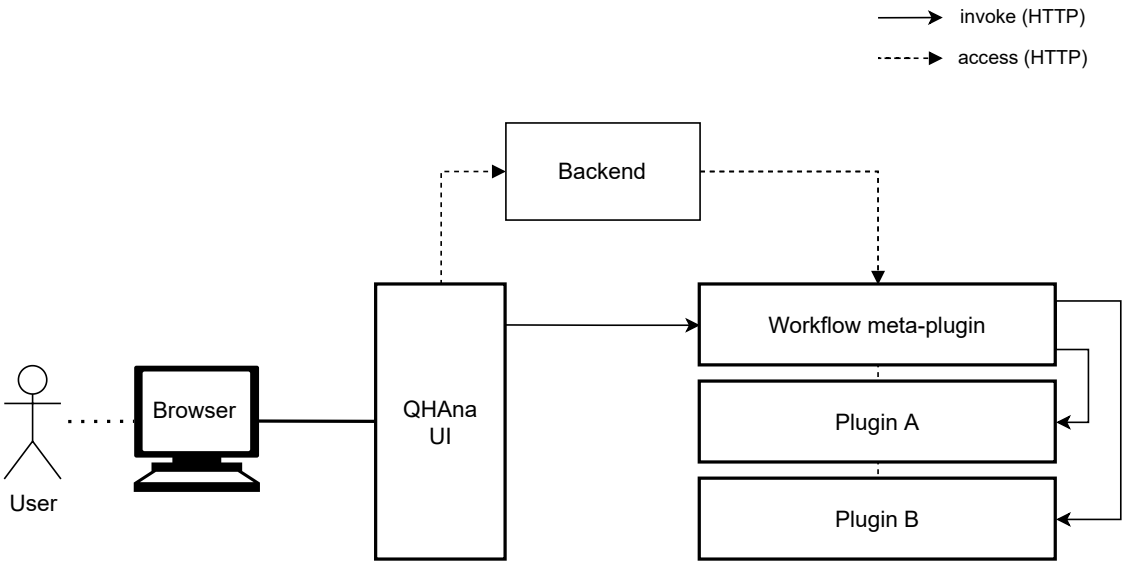

Figure 4.1: QHAna workflow meta-plugin simplified overview

the workflow process instance, therefore during the process of creating a workflow model for the workflow meta-plugin the user needs to be able to mark workflow instance variables that should be included in the results. Finally, using the meta-plugin should not require user interaction with external components, i.e., to allow for an easy to use environment, especially for novice users, all steps should be performed within the QHAna UI, e.g., it should not be necessary to use a web interface of the workflow engine to complete certain tasks.

4.2 Design Decisions

The Camunda platform with its Camunda Engine was chosen as the workflow automation platform and consequently BPMN is used. BPMN is an imperative workflow language that allows users to visually model constructs using, e.g., the Camunda modeler, such as subworkflows, gateways, activities, transactions and various events such as triggering an event when an exception occurred in an activity. BPMN allows for the definition of different exception types, a feature that is important for the workflow meta-plugin since it utilizes different exception types depending on the context. Furthermore, activities can be associated with different properties, this is useful to link an activity to a specific QHAna plugin. Using the Camunda Engine to run BPMN models is easily achieved in Java through the Camunda Java API, however it is also easy to integrate Camunda into other programming languages due to the provided REST API. Thus the decision was made to implement a Camunda Client in Python that implements a subset of the Java API functionality using the Camunda REST API. Camunda was chosen for this work since it provides important characteristics for the execution of workflows in the context of QHAna such as accountability, recoverability and an audit trail. Recoverability is important since it helps QHAna users save time from manually fixing issues and for this the audit trail that provides a record of performed actions is used. Furthermore, Camunda is scalable, thus many users can be served at the same time.

Functionality provided by the workflow meta-plugin is implemented as Celery tasks, as part of this work a standalone version of the meta-plugin was created in the first phase, i.e., a workflow runner that is not integrated into QHAna as a processing plugin, and for this Python threads were used to implement the runner tasks. Once a proof of concept was achieved the runner and the underlying tasks were ported to QHAna as a QHAna processing plugin using Celery tasks. Celery was selected as the distributed task queue system for the meta-plugin tasks since without a queuing system it is difficult to accommodate multiple plugins where many users can be served. Furthermore, existing QHAna plugins use Celery to implement plugin tasks, thus adapting the workflow runner is facilitated due to the existing integration of Celery into the QHAna plugin runner. Figure 4.1 shows how the user interacts with QHAna through the QHAna UI and may invoke the workflow meta-plugin that can then call any QHAna plugin. Both QHAna plugins and the QHAna UI utilize a backend.

Two main components that a user may want to utilize in a workflow model created for the use with the workflow meta-plugin are QHAna plugins and QHAna human tasks, i.e., the user should be able to specify the invocation of plugins and input gathering in a model. For this a mapping from the two QHAna components to corresponding BPMN constructs is required, thus the decision was made to represent QHAna plugins as service tasks with the property 'external implementation' set to contain the prefix 'plugin.', i.e., in Camunda service activities set to external implementation are also called external tasks, and QHAna human tasks are represented as human tasks in BPMN. Service activities were chosen since such an activity can specify external implementations. Scripting activities may also define the script type as 'external resource' but the resource needs to contain Java code and is thus not suited for the existing workflow meta-plugin implementation that is written using Python.

When the workflow model is executed by the Camunda Engine the meta-plugin requires periodical status updates. For this three different watchers are used in this work, polling for a specific Camunda Engine endpoint or a QHAna plugin instance endpoint until a certain condition is met that depends on the response received. The watchers are used to determine if new QHAna plugins need to be invoked, if a QHAna plugin instance is still running and if there are new human tasks that should be displayed to the user.

5 Implementation

Key components for the implementation of a workflow meta-plugin are discussed in this section, this includes clients to handle calls to QHAna REST-APIs and the Camunda REST-API whilst also mentioning the new endpoints added by the workflow meta-plugin. Furthermore, watchers are introduced that poll certain endpoints to gather status updates and exception types are discussed. An overview of components and corresponding interactions detailed in this chapter is provided in Figure 5.4.

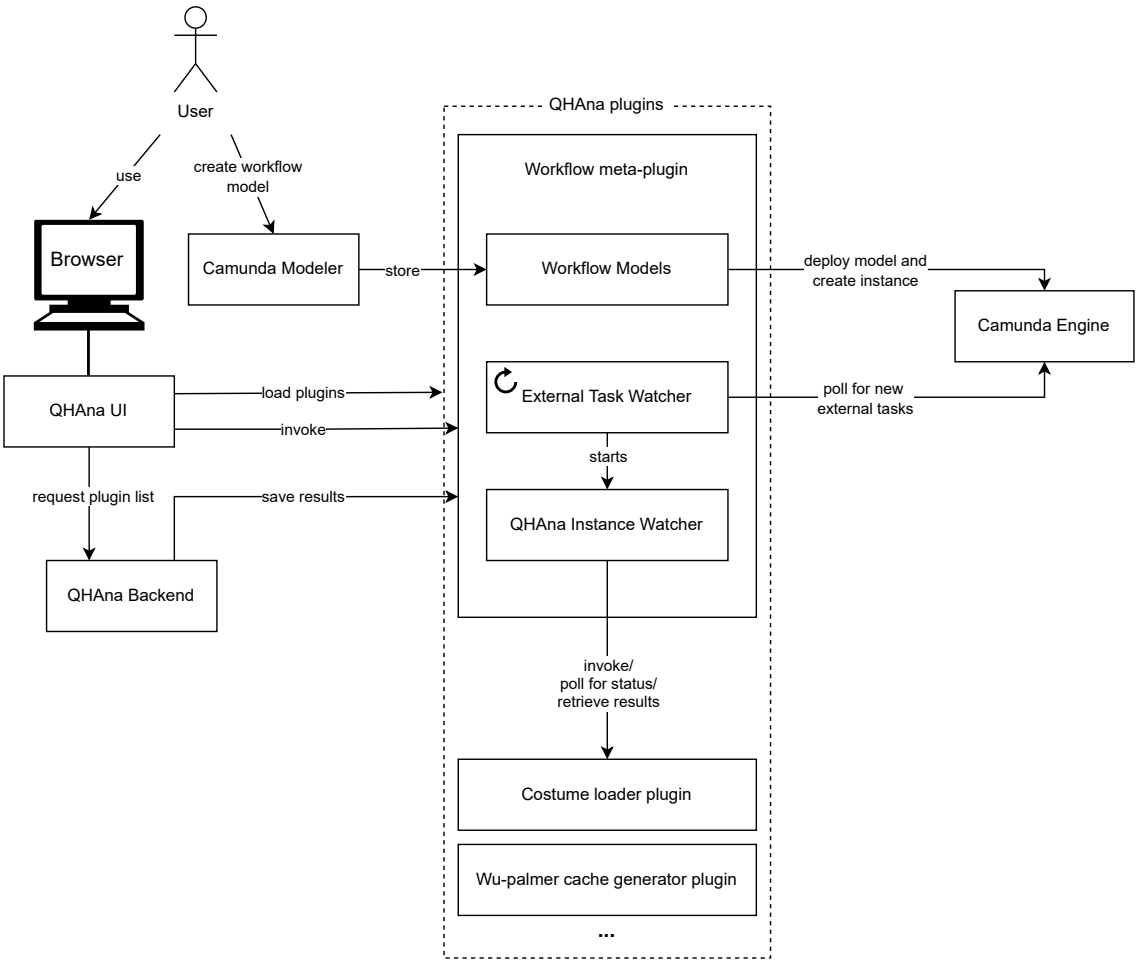

Figure 5.1: Extended overview for the QHAna workflow meta-plugin

5.1 Plugin Usage and Endpoints

Plugins, such as the workflow meta-plugin, and external programs are able to invoke QHAna plugins by using a REST API to call specific endpoints. QHAna plugins include the root endpoint, where links are specified for retrieving the user input form and for invoking the plugin given the input. Some plugins only require input once before the plugin entry task is started, it is however sometimes necessary to gather additional input during plugin execution. Therefore, QHAna can be used to create multi-step plugins such as the workflow meta-plugin that contain more than one entry tasks. Thus, QHAna plugin steps may contain a set of tasks that are run within that step, the first task of the initial plugin step that is executed is called the (plugin) entry task and subsequently first tasks of a new plugin step are called (plugin) step entry tasks. It may not always be possible to provide input that can be used for all plugin steps before the first task is run since it can happen that, e.g., the number of steps is not known before runtime leading to the requirement for multiple steps. Thus, the general procedure for using a multi-step plugin looks as follows: first the plugin is invoked with the required inputs by other plugins, external programs or through submitting a user form to the entry endpoint. The plugin can now compute a first batch of intermediary results until it determines that additional input is required to proceed with the computation, i.e., as a consequence, the plugin creates a new step containing information such as the link to retrieve the user input form that may differ from the initial user input form used for the plugin entry task and a new endpoint that should be called with valid input to start the next step. Within the context of this work, the workflow meta-plugin does not compute a batch of intermediary results but rather continues with the execution until new input is required from the user for the workflow instance. Each step contains an identifier, this information can be useful since inputs for plugin steps are logged hence allowing users to repeat plugin invocations, i.e., the same step id and input produces identical results for each repeated run. This newly created step is then appended to the steps list attribute available in the root endpoint where each item contains the aforementioned properties and a flag 'cleared' to indicate whether necessary input for the step entry task was submitted. Only the last step in the list can have the 'cleared' flag set to false and it is not possible for a plugin to extend that list unless the last step is cleared. Once the new step is added to the list the plugin is stopped to avoid consuming resources since it cannot proceed. Thus, when the entry task endpoint of the new step is called the plugin can continue computing either a next batch of intermediary results followed by one or more additional steps or alternatively the plugin result.

Entry task endpoints return the database id of the plugin instance when invoked. The database id is used to fetch the status of a task by calling the endpoint '/<db_id>/'. The REST endpoint of a plugin instance returns a response that contains the status of a plugin, currently supported status types are 'PENDING' for plugin instances that are still running but may be completed at a later time, 'SUCCESS' and 'ERROR' indicate whether the plugin instance terminated successfully or if an error was thrown during runtime. Since the workflow meta-plugin handles QHAna plugin invocations this information is used to determine when the results of a plugin instance can be passed to the Camunda workflow instance. If the status is 'SUCCESS' then an output list is included in the response data where each element in the output list contains a reference to the (raw) data, a file name that the plugin assigned to the data which should fit the contents of that file as well as the content and data type. Therefore, output data should only be populated if the status is 'SUCCESS' or 'ERROR', i.e., fetching the status of a plugin instance before completion will not include already computed result data.

5.2 Clients

The workflow meta-plugin utilizes a Camunda Client and a QHAna task client that implement required functionality for the orchestration of plugins.

5.2.1 Camunda Client

A Camunda client requires a Camunda config that contains information about Camunda, such as the base url used, the id for the workflow process instance, the id for the worker of that workflow process instance and the deployment. Initially, when the workflow meta-plugin is invoked, only the base url is set in the Camunda config. Once the Camunda config is provided one of the first tasks after the invocation that the client needs to complete is to deploy the BPMN model. After deploying the model the Camunda config is updated with the new deployment that contains the deployment id and definition id. The next step after deployment is to create a workflow instance using the uploaded BPMN model, also updating the Camunda config with the new process instance id. A Camunda client contains methods that perform calls to the various Camunda endpoints, e.g., locking an external task or getting the global variables of a workflow instance. For such calls to the Camunda REST API different information is needed depending on the endpoint, e.g., the deployment id or process id.

5.2.2 QHAna Task Client

This client contains methods for invoking QHAna plugins, i.e., given an external task the client checks whether the implementation of the external task was set to the format 'plugin.plugin_name'. It then checks whether the plugin with name 'plugin_name' is available and if the format matches then the client will fetch all variables for the external task that should be used as input for the QHAna plugin. In the workflow model a service task set to an external implementation with the aforementioned format may define inputs, i.e., an input specified as 'qinput.paramA: variableName' where the prefix 'qinput' indicates that the qhana plugin should receive for the parameter 'paramA' the contents of the workflow instance variable with name 'variableName'. Therefore, the contents of the workflow instance variable need to be fetched.

5.3 Watchers

Watchers are used to poll a specific endpoint until a break condition is met and then invoke a callback function. This break condition is dependent on the received response that is retrieved from a specified endpoint in a predefined interval. This work uses two types of watchers, i.e., those started by the workflow meta-plugin and periodic scheduled watchers are required. Celery is a distributed task queue system that can process messages and provides task scheduling. Celery task queues accept as an input a unit work that is called a task, in the context of QHAna this can be, e.g., the entry processing task for the workflow meta-plugin or one of the watchers detailed in the following sections. Workers in turn act as consumers, i.e., they remove a task from the queue by constantly monitoring task queues. There can be one worker or many others, by default QHAna requires a

single worker that will process plugin invocations although it is in this work that a second mandatory worker will be introduced which is responsible for processing periodic tasks. Non-periodic watchers are run by the Celery worker that is already used in the QHAna plugin runner to execute plugin instances whereas periodic watchers are run by the newly added worker. Celery tasks are started by sending a message to the queue, within the context of this work tasks can be initiated by users or other tasks. For example, after retrieving the micro frontend the user submits a form containing the input data, the plugin runner then receives this request and places the message into the Celery task queue where the message mentions the task to invoke and the parameters. Tasks may not only be initiated as soon as a worker is ready to process them but can also be scheduled to run after a certain amount of time by specifying the delay in seconds or setting a specific date and time. This is useful since periodic tasks introduced in this work used for polling specify a delay in seconds to ensure that the next task does not follow directly after, avoiding too frequent requests. Celery offers periodic tasks that represent often occurring events that are run in a specified interval or other custom start times such as setting the start time to solar events, e.g., at sunrise. Periodic tasks are handled by the Celery beat scheduler, the involved beat happens at a fixed configurable time interval, e.g., every 5 seconds and creates new messages for the task queue when new periodic tasks are available, however it should be noted that specifying tasks with a delay or interval means that the task will be executed at earliest when the delay or interval is over, but the task might also run at a later point in time if, e.g., no workers are available to process the task. Thus specifying a delay or interval is not a guarantee that tasks are run at the time the delay or interval is over, i.e., the task is sent to the task queue where other tasks may still need to be processed before the task can be run.

5.3.1 External Task Watcher

This is a periodic task in Celery, i.e., an entry exists in the beat schedule that will spawn a new external task watcher in a specified interval. Thus, it is possible that multiple external task watchers run at the same time. The main task of an external task watcher is to listen for the Camunda external task queue, i.e., a queue containing external tasks that are added by the Camunda Engine whenever the workflow instance process reaches a Service task that utilizes an external implementation. For this work external tasks, i.e. service tasks with the implementation method set to external, represent QHAna plugins. Thus, when the workflow instance execution process reaches the external task the external watcher hands over the task to a subsequent watcher of different type that then invokes the QHAna plugin with the specified name in the topic of the external task.

The Camunda external task queue is not instance specific and therefore external tasks within that queue need to be filtered depending on the instance the watcher belongs to. The topic name of a task in the queue is important since there may be many watchers but a task is typically consumed by only one. In this case external tasks that have a topic name set with the configurable prefix 'plugin' are processed by the external task watcher. When fetching a list of external tasks that are in the queue the data is returned in serialized JSON format and therefore this data is then deserialized to a list of ExternalTask objects. Working on deserialized data through the use of Python objects rather than directly on serialized data has the advantage of available autocompletion in an IDE, i.e., it is not necessary to look up the fields that are available in a response, a lookup of available response properties is only necessary when specifying the object representing the (partial) serialized response data.

If a matching external task is found then the external task watcher checks if the task is locked. If locked, then the task is skipped since it is already being processed by a different watcher. In case that it is not locked, the watcher locks the task to avoid a situation where a task is consumed more than once. Locking tasks requires specifying a lock duration to ensure that external tasks are not sitting in a queue indefinitely. After locking an external task the external task watcher will spawn a QHAna instance watcher and passes the newly found external task as argument. Note that the watcher may find multiple matching external tasks and thus spawns a QHAna instance watcher for each external task found. The external task however is not yet removed from the Camunda external task queue, this ensures that is any of the following steps fail, for example due to the QHAna instance watcher crashing, the external task can be picked up again after the lock duration time has expired. In case the processing of an external task fails multiple times it is possible to use the included property 'retries' of an external task to avoid situations where one task is processed indefinitely and thus throw a BPMN exception in case the number of available retries reaches zero.

5.3.2 QHAna Instance Watcher

Unlike the external task watcher a QHAna instance watcher is not periodically scheduled but rather called whenever the periodic external task watcher detects new entries in the Camunda external task queue. Therefore, an external task watcher spawns a QHAna instance watcher for each matching external task found.

The QHAna instance watcher is responsible for creating new QHAna plugin instances. For this is passes the external task to the QHAna task client. The task client fetches the input for the external tasks, thus also inputs for the QHAna plugin represented by the external task, that are stored as task local variables in Camunda. Camunda has different variable scopes, at the top most level is the root scope, all workflow variables within that scope are available throughout the entire workflow model. The next lower level represents the subworkflow scope where variables are available only within that subworkflow, if a variable with the same name is defined in the subworkflow and the parent scope then the parent scope for that variable is shadowed. If the workflow process instance execution exits a scope then existing variables from that scope are merged into the parent scope, thus either a new variable is created if no variable with the same name exists in the parent scope or the value is updated if it exists. The lowest level represents the task scope where variables are only available throughout task execution. Shadowing of variables from parent scopes applies also to the task scope, merging variables into parent scopes only happens in task scopes if the variables are marked as outputs, unlike subworkflows where all existing variables within that scope are merged.

Once the local task variables are fetched each variable is checked if it is an input variable, i.e., input variables contain as part of the variable name 'qinput.'. Therefore, the variables containing 'qinput.' as name are used as inputs for a QHAna plugin, it should be noted that the user creating the workflow model is responsible for ensuring that at this point the required inputs are available. If input is missing the user may add an additional activity to the workflow model to generate a form that can be filled out to gather inputs. This approach is described in the next subsection. The input variable with name 'qinput.foo' indicates that the workflow instance variable with name 'foo' should be used as input. Thus, the contents of the workflow instance variable 'foo' need to be fetched first. The QHAna instance watcher will poll for status updates once the QHAna plugin instance is

created Figure 5.4. If the plugin instance terminates successfully the instance watcher fetches the results that are deserialized into a QHAnaResult object that contains a list of QHAnaOutputs and the corresponding QHAna task.

Once the QHAnaResult object is available the external task that was locked in the Camunda external task queue is completed by an external task watcher. When completing the external task, a result JSON object is specified. The result object contains for each output in the QHAna result the name, content type, data type and reference link to the output data. This approach allows for subsequent workflow activities or gateways to utilize results produced by QHAna plugins. An overview on interactions between external components and Camunda is presented in Figure 5.2. The two watchers introduced are considered to be external since such watchers do not depend on QHAna workflow plugin instances but rather run independently.

5.3.3 Human Task Watcher

Like the QHAna instance watcher the human task watcher is not a periodic task. Instead the human task watcher is invoked by the workflow meta-plugin and there is only at most one human task watcher per workflow plugin instance at any given time. The human task watcher listens for the Camunda (human) task queue and will only process tasks that have the same workflow process instance id as the workflow plugin instance is assigned to and if the delegation state of the task is set to 'PENDING', i.e., the task is not yet completed.

When a new processable human task is found it includes form variables that are used by the workflow meta-plugin to create the QHAna form that QHAna users can utilize to provide additional input to workflow process instances. Thus, each form variable corresponds to one form field in the QHAna form. Camunda offers a '/form-variables' endpoint that takes the id of the Camunda task as a parameter although unfortunately, despite the name suggesting that such an endpoint should be used to fetch form variables, the endpoint returns all workflow instance variables. The Camunda Community forums contain threads about this unresolved issue but for this work a solution needs to be found since form variables are essential for the process of generating QHAna user forms. To fix the issue two options have been identified:

1. The first approach involves marking variables used in a form with a prefix, i.e., the prefix should not only identify a workflow instance variable as form variable but also show the form a variable belongs to. Thus, a possible format would be 'qform-humanTaskFoo', this however clutters variable naming where prefixes such as 'return.' or 'qinput.' and 'qoutput.' are already used.
2. The other option involves a different Camunda workflow instance endpoint, namely '/get-rendered-form' that returns the HTML containing form fields where form variable names are present. As part of this approach a simple method needs to be written that parses the variable names from the HTML form, thus extracting all form variables in the process. This works since the rendered form is task specific and therefore the extracted variables belong to the task in question.

For this work the second approach was chosen since it does not clutter variable naming and also results in less work for QHAna users who want to create workflow models using BPMN, however a potential downside of this approach is that the implementation may break if Camunda chooses to

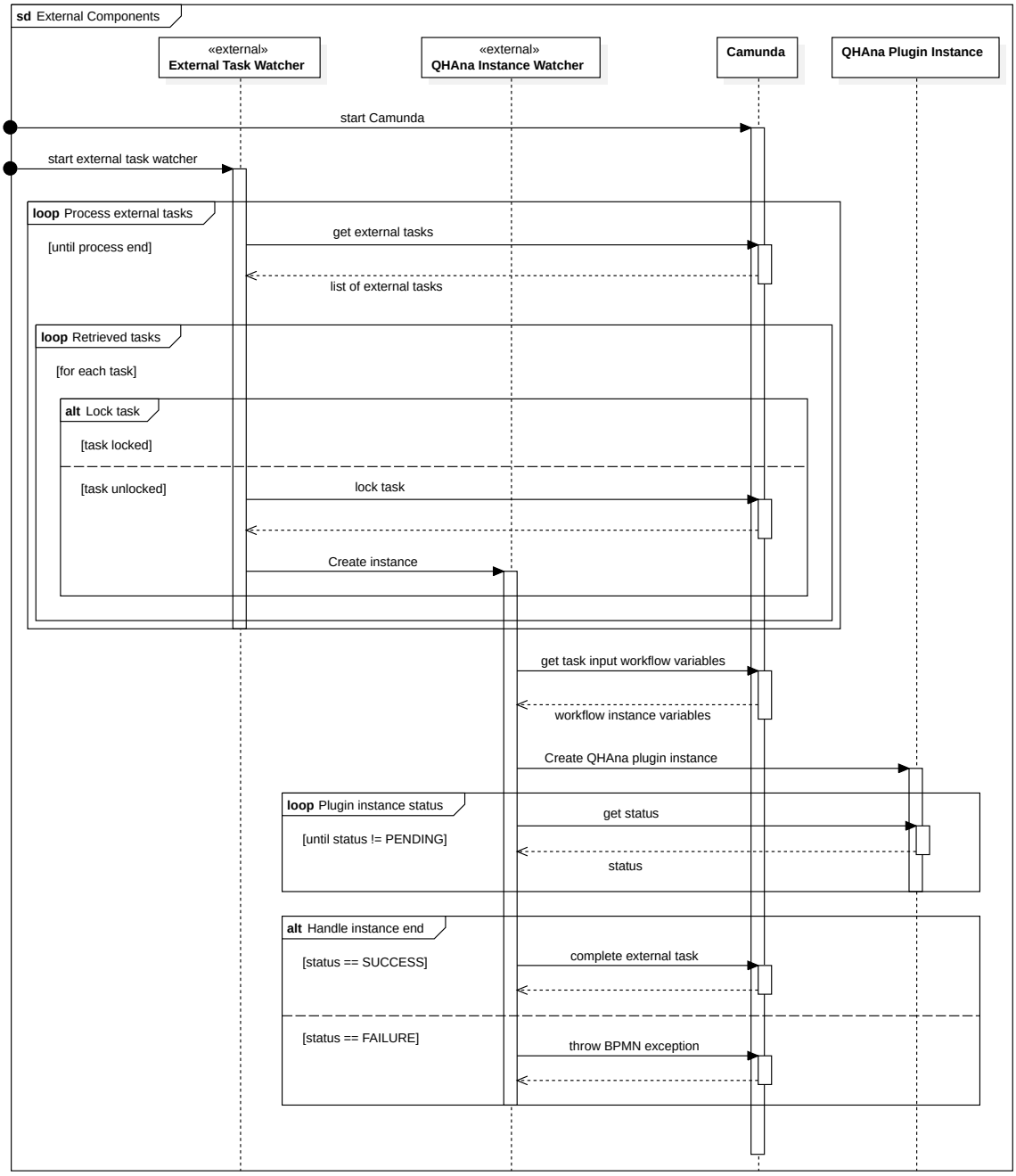

Figure 5.2: Sequence diagram for external components

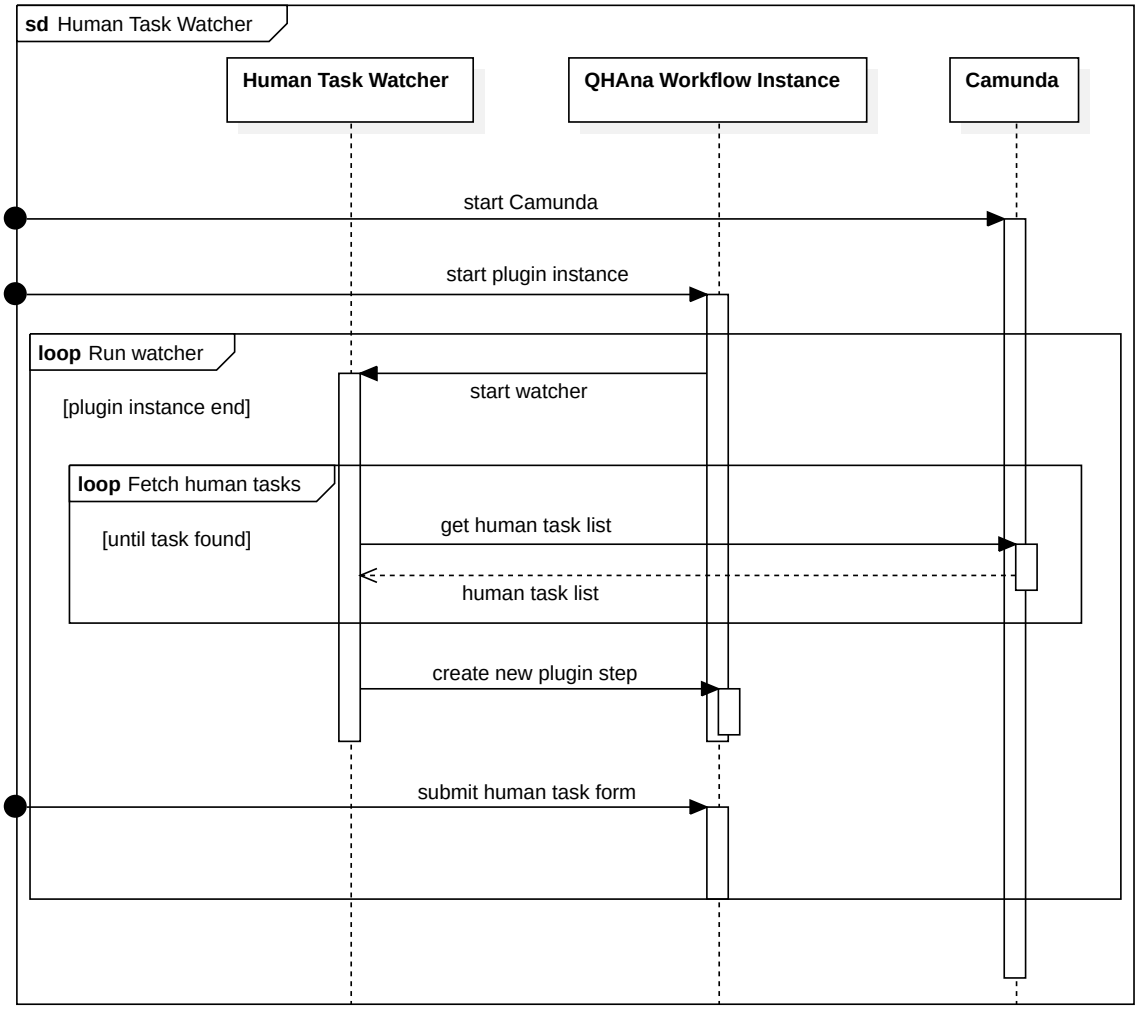

Figure 5.3: Sequence diagram for the human task watcher

change the contents of rendered forms in future updates. After extracting the form variables a few entries are written to the database of the QHAna backend, i.e., the BPMN properties for the human task are added. Such properties contain the BPMN XML string representing the used workflow model so that the model can be shown in a QHAna form using BPMN.IO and a task definition key that is used to highlight the human task in BPMN.IO that the user is currently completing (ref. Figure 5.5). It should be noted that the workflow presented in Figure 5.5 is only a demo workflow used for validation purposes and does not represent a real-world use-case. The human task id and form variables are also saved since they are needed when the user fetches the micro frontend for the next workflow meta-plugin step and are used to create the frontend schemas. After saving aforementioned values a new workflow meta-plugin instance step is created, i.e., this step points to the endpoint '/human-task-ui' for the micro frontend and '/human-task-process' to process the received input from the micro frontend.

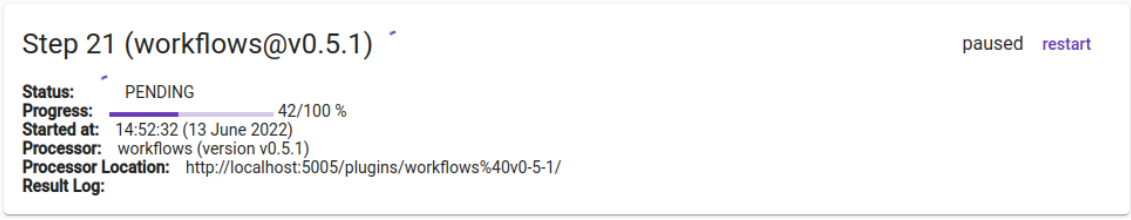

Figure 5.4: QHAna workflows meta-plugin instance status

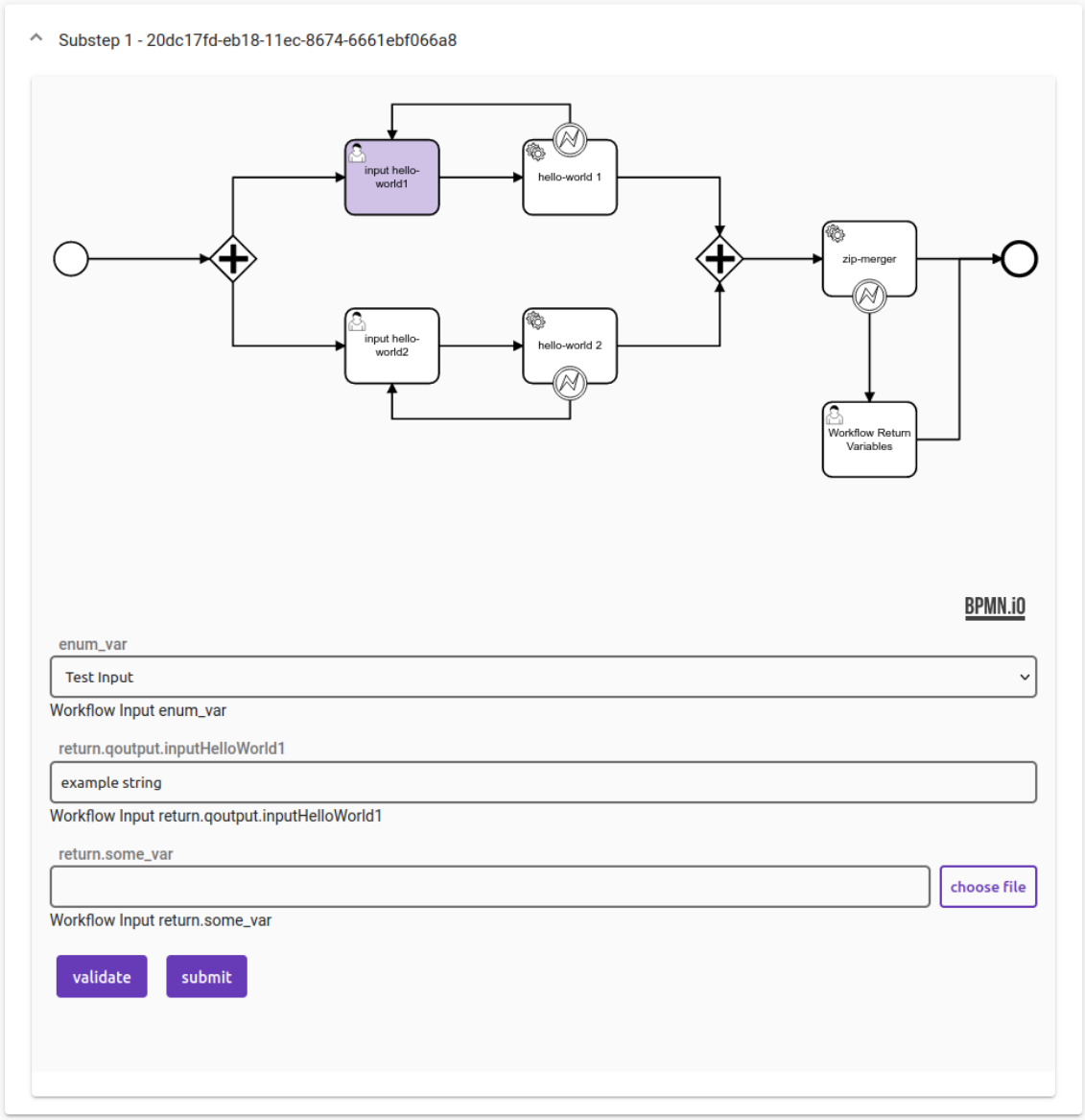

Figure 5.5: Human task micro frontend of a demo workflow for the QHAna workflows meta-plugin

5.4 Routes and Schemas

The workflow meta-plugin defines six endpoints listed in Table 5.1 as part of the plugin REST API, following endpoints provide general information, micro frontends or invoke plugin Celery tasks. As part of this work four different marshmallow schemas are specified that are used to generate responses and micro frontends for endpoints defined by the workflow meta-plugin. All schemas used by the workflow meta-plugin contain fields with following marshmallow field properties: a property of type boolean to specify whether a field is required and a boolean property 'dump_only' which if set to true will skip the field during deserialization.

- The plugin root endpoint '/' creates a response by using the workflows response schema and when invoked will return general information about the plugin. Information included are the title of the plugin, i.e., in this case "Workflows", a description specifying that this plugin is used to run workflows, the plugin name and version which together are used to create the identifier with the format 'plugin_name@version', the type of the plugin and plugin tags. The endpoint furthermore returns available entry points, i.e., the URLs to invoke the entry task and to fetch the micro frontend for the entry task.
- To start a new workflow plugin instance the user selects the workflow plugin in the QHAna UI that then fetches the micro frontend by calling the '/ui/' endpoint. The QHAna plugin runner provides a helper method 'render_template' that can create a micro frontend using following arguments: The name of a template HTML file containing the structure of the micro frontend, i.e., the file contains `jinja`[1] special placeholders that the QHAna plugin runner provides, these placeholders can be used to specify, e.g., where the form fields, validate and submit buttons should be placed in the micro frontend view. For the '/ui/' endpoint QHAna provides a simple template that is used containing placeholders for, e.g., form fields. To replace placeholders in a template file the helper method 'render_template' uses a schema that contains information about the fields that should be present in the form. The workflows parameters schema defines a single field 'input_bpmn', i.e., this form field can be used to specify the name of the BPMN file to be used for a workflow instance. The name must match a file available in the 'bpmn' folder directory of the workflow meta-plugin. Furthermore, the helper method specifies a 'process' parameter that is used to set the URL for the process endpoint, i.e., in this case the URL to the entry process endpoint is specified. With the provided parameters the helper method then returns the micro frontend that is displayed in an iframe in the QHAna UI.
- After fetching the micro frontend for the workflow plugin entry task the form is submitted to the '/process/' endpoint, the arguments that are accepted by this endpoint are specified using a schema, in this case the workflows parameters schema is used. Thus, the endpoint expects a value for 'input_bpmn'. After invoking the endpoint a new processing task is declared that contains the Celery task name "start workflow" and the 'input_bpmn' parameter, the processing task is then saved to the QHAna backend. The Celery task is invoked and the user is redirected to the task view where the status and progress of the workflow meta-plugin

[1] https://jinja.palletsprojects.com/en/3.1.x/

Path	Schema	Methods GET	Methods POST	Description
/	workflows response	GET		Contains plugin meta-data.
/ui	workflows parameters	GET	POST	Returns the micro frontend, contains pre-rendered inputs in case of a POST request.
/<db_id>/human-task-ui	any input	GET	POST	Returns the human task micro frontend, contains pre-rendered inputs in case of a POST request.
/<db_id>/human-task-ui/bpmn_io	-	GET		Provides the rendered workflow model as response.
/process	workflows parameters		POST	Starts a new workflow plugin instance when invoked.
/<db_id>/human-task-process	any input		POST	Completes a human task with the input gathered.

Table 5.1: Structure of the workflow meta-plugin REST API

instance is shown. The response sent to the QHAna UI when redirecting the user contains the database id of the plugin instance, this identifier is needed to fetch, e.g., the micro frontend for a human task during workflow execution.

- Besides providing a micro frontend for creating a workflow meta-plugin instance the plugin defines a REST API endpoint '/<int:db_id>/human-task-ui/'. When provided with the identifier received from the '/process/' endpoint the human task ui endpoint returns the micro frontend for the human task that did not yet receive input from the user. Note that this endpoint is used only for forms that are required by the workflow instance and does otherwise not gather input that is required by the workflow meta-plugin, for this the '/ui/' endpoint is used. The QHAna UI is aware that a new human task requires input since the workflow meta-plugin will suspend execution and create a new plugin step when a new human task is found in the Camunda task queue. The 'AnyInputSchema' dynamic marshmallow schema is used to create a schema that does not contain hardcoded fields but rather defines fields given the human task form parameters where each parameter contains the name and parameter form input type, i.e., the type can be that of a plain text input, choice input, enum input or file input. A choice input differs form an enum input in that an enum field shown to the user contains values that correspond to the values the workflow instance variable will receive. For example a choice input may contain the string "Option A" that the user sees and if selected the workflow instance variable may, e.g., receive the string value "OPTION_A", for an enum input these two string values are always identical. The reason for including both choice and enum inputs when a choice input could be used as enum input is that it requires the user

less work to define an enum input when the aforementioned values need to be identical. The schema is then used to generate the partial micro frontend for the human task, the micro frontend is then sent to the user for completion and is integrated in an iframe that can be viewed in the QHAna UI when selecting the task details of a workflow meta-plugin instance. This is a partial micro frontend since the frontend is still missing a component that is fetched from the endpoint discussed in the following.

- To provide a complete micro frontend for gathering input that is used to complete human tasks the user should not only be able to view the form fields but also understand which human task is currently being processed and the context that this human task belongs to. Thus, a view is shown to the user where the workflow model is displayed, i.e., the view of the model is zoomed near the current human task and the task is highlighted in a different color compared to all other model components. Therefore, this allows for fully integrated human task processing where external tools, such as the Camunda Web Interface, are not necessary and thus lowering the barrier of entry for the use of QHAna. The view of the workflow model used by the workflow meta-plugin instance is integrated inside the form iframe as another iframe and placed above the form field contents. For the integration of a workflow model viewer BPMN.IO was chosen for its simple to use interface, i.e., by providing the model XML string and the task definition key for highlighting the current human task a viewer is returned. Placing the workflow model viewer camera near the current human task can be done through setting the focus by providing the task definition key of the human task. The '/<int:db_id>/human-task-ui/bmpn_io' endpoint is called by the partial micro frontend, received from the '/<int:db_id>/human-task-ui/' endpoint, and fetches the bpmn properties, i.e., the workflow model XML string and task definition key to then generate the micro frontend. The generated micro frontend is then sent back to the user and integrated into the existing partial micro frontend to complete the human task QHAna form.
- Once the user has filled out all form fields the contents are sent to the '/<int:db_id>/human-task-process/' when the submit button is pressed. The endpoint then spawns a new Celery processing task 'process_input' that takes the contents as argument and then completes the Camunda human task, i.e., the Camunda human task is removed from the corresponding queue and the workflow instance variables receive the values from the form contents.

5.5 Exceptions

The workflow meta-plugin currently implements three different BPMN execeptions that can be thrown during plugin runtime. Such exceptions can be used to alter the workflow process instance flow depending on the type of the exception, i.e., existing features provided by Camunda and BPMN such as transactions can be rolled back or compensation activities can be executed if no rollback is possible.

- Should a QHAna plugin invoked by the QHAna instance watcher terminate with the status 'FAILURE' then a BPMN error of type 'qhana-plugin-failure' is thrown.
- A BPMN error of type 'qhana-unprocessable-entity-error' is thrown by the QHAna task client if a QHAna plugin invocation received unprocessable entities as input parameter and could thus not start the entry task.

- The BPMN error with type 'qhana-mode-error' is thrown when the workflow model contains a service task used to invoke a qhana plugin where the defined inputs are specified in a wrong manner. More specific, this error is thrown when an existing result of a QHAna plugin is used as input for another QHAna plugin. In this case the result of the previous invocation may contain multiple outputs, each containing the file name, content type, data type and reference. Thus, to use this result as input one specific output from the output needs to be selected, this can be done by specifying the file name, content type or data type as selection criteria. If a different unknown criteria is specified then the BPMN error is thrown.

To throw BPMN errors the workflow meta-plugin simply calls the '/bpmnError' endpoint of the Camunda REST API and passes the corresponding external task, error code and error message as arguments.

6 Conclusion and Outlook

Requirements for implementing a workflow meta-plugin were examined in this work, i.e., main criteria were outlined such as the deployment for workflow models and instance creation. When creating workflow models that need to be run by the meta-plugin all constructs of the workflow language should be supported and workflow instance variables need to be updated by the meta-plugin during runtime. QHAna users interact with the QHAna UI during workflow execution and do not need to utilize external components. As part of this work Camunda was chosen as the workflow automation platform and therefore workflow models are created using BPMN as the workflow language and subsequently run by the Camunda Engine upon deployment. Advantages for using BPMN include being able to use transactions and exceptions.

Design decisions were made such that the process of creating workflow models for the use with the workflow meta-plugin is easy and intuitive, this includes avoiding variable name cluttering by not introducing, e.g., unnecessary prefixes to variable names. Accessibility to novice users is further increased by only requiring new code to be written when gateway conditions depend on results that stem from QHAna plugin invocations. Maintainability of the plugin is sustained by following existing decisions from other plugins such as using Python as the programming language and Celery for implementing plugin tasks. With Camunda a popular workflow automation platform was chosen that provides extensive documentation. In the further course of this work components of the workflow meta-plugin were discussed where parts such as the external task watcher or qhana instance watcher are implemented with reusability in mind, i.e., such components run independently from a workflow meta-plugin instance as periodic Celery tasks. Subsequently, plugin routes, corresponding plugin schemas and plugin exception types were specified, for this the implemented functionality of each API endpoint and exception type was addressed.

Outlook

Whilst the presented workflow meta-plugin supports three different BPMN exceptions that can be thrown during runtime new exception types may be added in future, e.g., common failures in plugins could be added as standalone exception types rather than utilizing a generic plugin failure. QHAna human task forms currently include different field types, namely plain text, choice, enum or file inputs (Section 5.4). Form fields that represent a file input are displayed to the user, however when trying to select a file the pop-up that should list all available files with matching content type does not show any entries, therefore this issue should be addressed by a future update. The workflow meta-plugin currently contains two workflow models that are used as an introduction guide for developers, this includes a workflow model that represents a partial implementation of the MUSE data analysis workflow (Section 2.4), following updates to the plugin may include a complete implementation and additional workflow models that are of interest to QHAna users.

Bibliography

[Agn] Agnostiq. *Agnostiq Covalent*. URL: https://agnostiq.ai/covalent/ (cit. on pp. 23, 26).

[Ant20] K. K. Antonio Brogi Wolf Zimmermann. *Service-Oriented and Cloud Computing*. Springer Cham, 2020, pp. 87–88. ISBN: 978-3-030-44769-4. DOI: 10.1007/978-3-030-44769-4 (cit. on p. 13).

[Bar22] J. Barzen. "From Digital Humanities to Quantum Humanities: Potentials and Applications". In: *Quantum Computing in the Arts and Humanities: An Introduction to Core Concepts, Theory and Applications*. Ed. by E. R. Miranda. Cham: Springer International Publishing, 2022, pp. 1–52. ISBN: 978-3-030-95538-0. DOI: 10.1007/978-3-030-95538-0_1 (cit. on p. 11).

[Ber12] D. M. Berry. "Introduction: Understanding the Digital Humanities". In: *Understanding Digital Humanities*. Ed. by D. M. Berry. London: Palgrave Macmillan UK, 2012, pp. 1–20. ISBN: 978-0-230-37193-4. DOI: 10.1057/9780230371934_1 (cit. on p. 15).

[BFL18] J. Barzen, M. Falkenthal, F. Leymann. "Wenn Kostüme sprechen könnten: MUSE -Ein musterbasierter Ansatz an die vestimentäre Kommunikation im Film". In: Digital Humanities. Perspektiven der Praxis. Berlin: Frank und Timme, Mai 2018, pp. 223–241. ISBN: 978-3-7329-0284-2 (cit. on pp. 11, 15, 16).

[BL20] J. Barzen, F. Leymann. "Quantum humanities: a vision for quantum computing in digital humanities". In: *SICS Softw.-Inensiv. Cyber-Phys. Syst.* 35 (Aug. 2020), pp. 153–158. DOI: 10.1007/s00450-019-00419-4 (cit. on pp. 3, 5, 15).

[C D19] C. Dickel. *A Cloud Quantum Computer Busi-ness Plan*. 2019. URL: https://blog.qutech.nl/2018/07/18/a-cloud-quantum-computer-business-plan/ (cit. on p. 14).

[CCA+10] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, R. Sears. "MapReduce online". In: *Nsdi* 10 (Apr. 2010), p. 22 (cit. on p. 23).

[EBL18] S. Endo, S. C. Benjamin, Y. Li. "Practical Quantum Error Mitigation for Near-Future Applications". In: *Phys. Rev. X* 8 (3 July 2018), p. 031027. DOI: 10.1103/PhysRevX.8.031027 (cit. on p. 14).

[Ell99] C. A. Ellis. *Workflow Technology*. Vol. 7. Computer Supported Cooperative Work, Trends in Software Series, 1999, pp. 29–54 (cit. on pp. 13, 22).

[FBFL15] C. Fehling, J. Barzen, M. Falkenthal, F. Leymann. "PatternPedia – Collabroative Pattern Identification and Authoring". In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC)*. epubli GmbH, June 2015, pp. 252–305 (cit. on p. 16).

[GSK+11] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, M. Reiter. "Conventional Workflow Technology for Scientific Simulation". In: *Guide to e-Science: Next Generation Scientific Research and Discovery*. Ed. by X. Yang, L. Wang, W. Jie. London: Springer London, 2011, pp. 323–352. ISBN: 978-0-85729-439-5. DOI: 10.1007/978-0-85729-439-5_12 (cit. on p. 22).

[HCT+19] V. Havlíček, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala, J. M. Chow, J. M. Gambetta. "Supervised learning with quantum-enhanced feature spaces". In: *Nature* 567.7747 (Mar. 2019), pp. 209–212. ISSN: 1476-4687. DOI: 10.1038/s41586-019-0980-2 (cit. on p. 14).

[IEE22] IEEE Spectrum. *Quantum Computers Getting Smarter at Simulating Chemistry*. Mar. 2022. URL: https://spectrum.ieee.org/quantum-chemistry-largest (cit. on p. 13).

[KH13] A. Khrennikov, E. Haven. *Quantum social science*. Cambridge University Press, Jan. 2013 (cit. on p. 15).

[LBF+20] F. Leymann, J. Barzen, M. Falkenthal, D. Vietz, B. Weder, K. Wild. *Quantum in the Cloud: Application Potentials and Research Opportunities*. 2020. DOI: 10.48550/ARXIV.2003.06256 (cit. on pp. 14, 21).

[LBF19] F. Leymann, J. Barzen, M. Falkenthal. "Towards a Platform for Sharing Quantum Software". Englisch. In: *Proceedings of the 13th Advanced Summer School on Service Oriented Computing (2019)*. IBM Technical Report (RC25685). IBM Research Division, Sept. 2019, pp. 70–74. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2019-24&engl= (cit. on p. 23).

[Ley19] F. Leymann. "Towards a Pattern Language for Quantum Algorithms". In: *Quantum Technology and Optimization Problems*. Ed. by S. Feld, C. Linnhoff-Popien. Cham: Springer International Publishing, 2019, pp. 218–230. ISBN: 978-3-030-14082-3 (cit. on p. 14).

[LR99] F. Leymann, D. Roller. *Production workflow: concepts and techniques*. Upper Saddle River, N.J.: Prentice Hall PTR, 1999 (cit. on p. 22).

[Mos08] M. Mosca. *Quantum Algorithms*. 2008. DOI: 10.48550/ARXIV.0808.0369 (cit. on p. 21).

[MZO20] F. B. Maciejewski, Z. Zimborás, M. Oszmaniec. "Mitigation of readout noise in near-term quantum devices by classical post-processing based on detector tomography". In: *Quantum* 4 (Apr. 2020), p. 257. DOI: 10.22331/q-2020-04-24-257 (cit. on p. 14).

[NC02] M. A. Nielsen, I. Chuang. "Quantum Computation and Quantum Information". In: *American Journal of Physics* 70.5 (2002), pp. 558–559. DOI: 10.1119/1.1463744. eprint: https://doi.org/10.1119/1.1463744 (cit. on pp. 11, 14).

[Pre18] J. Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* 2 (Aug. 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79 (cit. on p. 14).

[QHA] QHAna authors. *QHAna-Plugin-Runner documentation*. URL: https://github.com/UST-QuAntiL/qhana-plugin-runner/tree/main/docs (cit. on p. 16).

[RP11] E. Rieffel, W. Polak. *Quantum Computing: A Gentle Introduction*. Cambridge, Mass.: MIT Press, 2011 (cit. on p. 14).

[RT19] R. Raz, A. Tal. “Oracle Separation of BQP and PH”. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 13–23. ISBN: 9781450367059. DOI: 10.1145/3313276.3316315 (cit. on p. 14).

[RWJ+14] T. F. Rønnow, Z. Wang, J. Job, S. Boixo, S. V. Isakov, D. Wecker, J. M. Martinis, D. A. Lidar, M. Troyer. “Defining and detecting quantum speedup”. In: *Science* 345.6195 (2014), pp. 420–424. DOI: 10.1126/science.1252319. eprint: https://www.science.org/doi/pdf/10.1126/science.1252319 (cit. on pp. 11, 14).

[TQ19] S. S. Tannu, M. K. Qureshi. “Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 987–999. ISBN: 9781450362405. DOI: 10.1145/3297858.3304007 (cit. on p. 23).

[WBL+20] B. Weder, J. Barzen, F. Leymann, M. Salm, D. Vietz. “The Quantum Software Lifecycle”. In: *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software*. APEQS 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 2–9. ISBN: 9781450381000. DOI: 10.1145/3412451.3428497 (cit. on pp. 14, 21).

[WBLV21] B. Weder, J. Barzen, F. Leymann, D. Vietz. *Quantum Software Development Lifecycle*. 2021. DOI: 10.48550/ARXIV.2106.09323 (cit. on pp. 21, 22).

[WBLW20] B. Weder, U. Breitenbücher, F. Leymann, K. Wild. “Integrating Quantum Computing into Workflow Modeling and Execution”. In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020, pp. 279–291. DOI: 10.1109/UCC48980.2020.00046 (cit. on pp. 14, 21, 22).

[Zap] Zapata Computing. *Zapata Orquestra*. URL: https://www.orquestra.io/ (cit. on pp. 23, 24).

All links were last followed on July 14, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature