ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS

FACULTY OF INFORMATICS

THESIS

OE-NIK	Student's name:	**Balázs Szendrődi**
2024	Student's registration number:	**T/008504/FI12904/N**

Óbuda University
John von Neumann Faculty of Informatics
Institute of Biomatics and Applied Artificial Intelligence

THESIS

Name of the student: **Balázs Szendrődi**
Registration number: T/008504/FI12904/N
Neptun's code: [redacted]

Title of the thesis:

Workflow solution implementation for the cloud services framework
Munkafolyamat lehetőségének megvalósítása felhőszolgáltatási keretrendszerhez

Internal supervisor: Milán Balázs
External supervisor:

Deadline: 15 December 2024

Subjects of the final exam: Computer Architectures
Cloud service technologies and IT security specialization

The task:

Today, cloud technologies and systems are the main trend in the development of systems that enable virtualization over physical layers. In a diverse range of solutions, there is a lack of systems on the market that are specifically optimised for research and development and that users/researchers without deep development experience can use comfortably and efficiently.
The BlackAnt service platform under development at the University of Óbuda's Centre of Excellence in Cyber Medicine could provide a potential solution to this gap. The aim of the development is to create a system that can efficiently serve the R&D&I activities of the University of Óbuda by developing a suitable cloud-based service system.
Complex tasks and processes can be solved by workflows that allow full functionality to be achieved through the collaboration of separate entities.
In this thesis, the candidate is responsible for implementing workflows into the existing structure of the cloud services framework.

The thesis must contain:

- literature and development technology review,
- overview of the BlackAnt framework,
- design, implementation and testing of workflow components,
- integration of the workflow component into the existing cloud system,
- evaluation of the results.

Place of stamp

ÓBUDAI EGYETEM * Neumann János Informatikai Kar * 1. Budapest

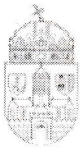
.....
Dr. György Eigner
head of institute

This specification is valid until: **15 December 2026**
ÓE HKR Section 54, paragraph (10)

I consider the thesis suitable for submission:

.....
external supervisor

.....
internal supervisor



ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
FACULTY OF INFORMATICS

STUDENT'S DECLARATION

I the undersigned student hereby declare that this degree project / thesis is the result of my own work; I have disclosed the references and tools used in an identifiable manner. The results indicated in my degree project / thesis completed may be used by the university and the institution announcing the task for their own purposes free of charge.

Dated: Budapest,.. 2024.12.13.

Szendrői Balázs

student's signature

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

CONSULTATION LOG

Student's name: SZENDRŐDI BALÁZS

Neptun Code: [redacted]

Speciality: FULL-TIME

Phone number: [redacted]

Mailing address (e.g. domicile): [redacted]

Thesis / thesis work[1] title in Hungarian:

MUNKAFOLYAMAT LEHETŐSÉGÉNEK MEGVALÓSÍTÁSA FELHŐSZOLGÁLTATÁSI KERETRENDSZERHEZ

Thesis / Thesis work[2] title in English:

WORKFLOW SOLUTION IMPLEMENTATION FOR THE CLOUD SERVICE FRAMEWORK

Internal supervisor: BALÁZS MILÁN

External Supervisor: .....

Please write the data in printed capital letters!

Occ.	Date	Content	Signature
1.	2024.02.05	Cloud framework infrastructure overview	
2.	2024.03.07	Local developement setup	
3.	2024.04.25	Literature review consultation	
4.	2024.05.02	Thesis finalization	

The Consultation log is required to be countersigned by either supervisor on a total of 4 occasions of consultation.

The student has complied with the requirements of the subject titled Tesis I. / Thesis II. / Thesis work I / Thesis work II / Thesis work III / Thesis work IV[3] ,(MSc) and is allowed to proceed for reporting / defense[4].

Supervisor suggested grade: 5

Dated: Budapest 2024. 05 15

.....

Internal supervisor

[1] Underline as appropriate
[2] Underline as appropriate
[3] Underline as appropriate
[4] Underline as appropriate

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

CONSULTATION LOG

Student's name: SZENDRŐDI BALÁZS

Neptun Code: [redacted]

Speciality: FULL-TIME

Phone number: [redacted]

Mailing address (e.g. domicile): [redacted]

Thesis / thesis work[1] title in Hungarian:

MUNKAFOLYAMAT LEHETŐSÉGÉNEK MEGVALÓSÍTÁSA FELHŐSZOLGÁLTATÁSI KERETRENDSZERHEZ

Thesis / Thesis work[2] title in English:

WORKFLOW SOLUTION IMPLEMENTATION FOR THE CLOUD SERVICE FRAMEWORK

Internal supervisor: BALÁZS MILÁN

External Supervisor:

Please write the data in printed capital letters!

Occ.	Date	Content	Signature
1.	2024.10.10	Workflow design overview	
2.	2024.10.24	Technology stack justification	
3.	2024.11.14	Development tasks and processes	
4.	2024.12.05	Thesis finalization	

The Consultation log is required to be countersigned by either supervisor on a total of 4 occasions of consultation.

The student has complied with the requirements of the subject titled Tesis I. / Thesis II. / Thesis work I / Thesis work II / Thesis work III / Thesis work IV[3] ,(MSc) and is allowed to proceed for reporting / defense[4].

Supervisor suggested grade: 5

Dated: Budapest 2024. 12. 13

Internal supervisor

[1] Underline as appropriate
[2] Underline as appropriate
[3] Underline as appropriate
[4] Underline as appropriate

Contents

ABSZTRAKT A dolgozat egy munkafolyamat-kezelő rendszer fejlesztésére összpontosít felhőinfrastruktúrában, különös figyelmet fordítva a backend megvalósítására. A kutatás a BlackAnt keretrendszer és a meglévő munkafolyamat felhasználói felületének bemutatásával kezdődik. Ezt követően a releváns szakirodalom áttekintése következik, amely a meglévő pipeline implementációk és azok korlátai köré összpontosít. A rendszer különböző technológiákat integrál, beleértve a Flask-ot a backend számára, MongoDB-t a felhasználó specifikus metaadatok tárolására, a Zeebe-t a munkafolyamatok kezelésére, valamint Elasticsearch-öt a keresési funkciókhoz. Emellett integrálódik a CI/CD pipeline-okkal, biztosítva az automatizált tesztelést, buildelést és telepítési folyamatokat. Az elsődleges cél egy robusztus, magas rendelkezésre állású megoldás biztosítása a komplex munkafolyamatok orkestrálására és monitorozására egy skálázható, konténerizált környezetben.

ABSTRACT This thesis focuses on the development of a workflow management system within a cloud infrastructure, with an emphasis on backend implementation. The research begins with an introduction into the BlackAnt framework and the existing workflow user interface. Then it explores the relevant literature review for existing pipeline implementations and the technologies used to store the workflow schema and their limitations. The system integrates various technologies including Flask for the backend, MongoDB for user specific metadata storage, Zeebe for workflow orchestration, and Elasticsearch for search functionality. It also integrates with CI/CD pipelines, ensuring automated testing, build, and deployment processes. The primary goal is to provide a robust, high-availability solution for orchestrating and monitoring complex workflows in a scalable, containerized environment.

1 Introduction

1.1 About BlackAnt

During my thesis work, I participate in the development of the BlackAnt platform. BlackAnt is a high-availability cloud framework with dynamic scaling, that uses micro-service architecture. The system is designed to provide a cloud solution not just for developers but researchers or those without the necessary knowledge. Users with the need for artificial intelligence, deep learning, machine learning algorithms or other resource-intensive computations should be able to utilize the framework.

The system consists of two main components, namely an environment server cluster and an execution server cluster. The environment component hosts the GitLab source code manager and the Docker Registry. Developers store their service implementations in GitLab, from which a Docker Image is automatically created for each merge using GitLab DevOps and stored in the Docker Registry. The running component of BlackAnt, when it receives a request from a user to use a service, fetches the Docker Image from the Docker Registry, which it will then launch on one of the working nodes as configured. During the process, the user will be kept informed about the status of his service via the web management interface, as well as via the Representational State Transfer (REST)-ful Application Programming Interface (API). [1]

The execution server cluster includes three underlying components. The runner server responsible for authentication of the clients, handles communication of the other server processes. The service server facilitates the main functionality of the platform such as schedules, resource monitoring, file server. The third component is the database server. The platform uses different database services for different tasks. MariaDB main database in conjunction with Galera Cluster for high availability, MongoDB , MinIO object storage.

There are three types of users in the platform, the 'user' role with the smallest level of privileges they have basic functionality available to them such as scheduling tasks in the system, 'developer' role with medium privileges they can utilize the platform to develop different applications, 'admin' role with the highest amount of privileges, have all of the settings and options available to them, and they monitor and maintain the system.

Authentication and authorization is achieved with Keycloak an open source identity and access management system. Authorization is implemented using Role Based Access Controll (RBAC) which restricts system access based on their roles within the platform. Users are allowed to access different resources based on their roles. Keycloak has a fully customizable login and registration interface using HyperText Markup Language (HTML) files.

Administrators can manage the users with integrated admin console. It also supports external providers for authentication such as Google or Facebook.
The back-end is implemented using the Flask micro web-framework written in Python. The interface developers have to implement, is also in python. The front-end is Angular, which is a TypeScript-based free and open-source single-page web application framework.

1.2 Workflow

The specific part, I will be working on is the implementation of the workflow component. The workflow is a directed graph representation of a more complex task or schedule. The workflow can be constructed from multiple components called nodes. These nodes can be, in the current stage: Schedule, Trigger, DataCollector, Notificator. Schedule is responsible for the configuration of the task, schedule time which can be one-time or recurring. The Trigger can be used to trigger the start of the workflow. DataCollector can be used to retrieve the result of the schedule. The Notificator node is responsible for sending the user email, or other kind of message in case a long running task is done or there is some error during the execution of the pipeline.
The workflow generation web user interface is also under development at this time. Functionality behind the workflows created on the visual editor will be my task. This includes database structure creation, API server implementation to receive changes from the front-end and a way to store the workflow details in a database.

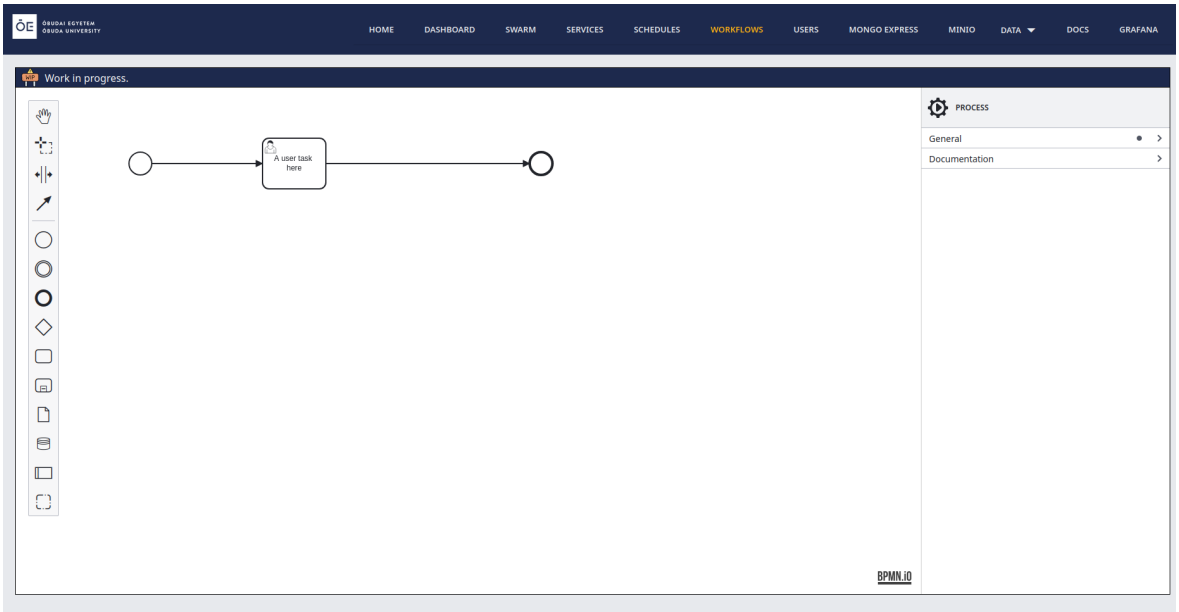


Figure 1-1: Workflow web user interface

2 Existing pipeline implementations

Developing efficient and maintainable data pipelines is crucial for modern data-driven applications. However, the complexity of these pipelines can make it challenging to understand the flow of data, identify bottlenecks, and troubleshoot issues. Pipeline visualization tools address this challenge by providing a clear graphical representation of the data processing steps.
This section dives into an exploration of the diverse array of pipeline implementations found industrial applications. By examining the existing literature and real-world deployments, we aim to elucidate the spectrum of approaches, architectures, and tools employed to construct and visualize pipelines.

2.1 Jenkins Pipeline

Jenkins Pipeline (or simply "Pipeline" with a capital "P") is a suite of plugins which supports implementing and integrating Continuous Delivery (CD) pipelines into Jenkins. A CD pipeline is an automated expression of your process for getting software from version control right through to your users and customers. Every change to your software (committed in source control) goes through a complex process on its way to being released. This process involves building the software in a reliable and repeatable manner, as well as progressing the built software (called a "build") through multiple stages of testing and deployment. The definition of a Jenkins Pipeline is written into a text file (called a Jenkinsfile) which in turn can be committed to a project's source control repository. This is the foundation of "Pipeline-as-code"; treating the CD pipeline as a part of the application to be versioned and reviewed like any other code. [2]
A Jenkinsfile can be written using two types of syntax — Declarative and Scripted. Declarative and Scripted Pipelines are constructed fundamentally differently. Declarative Pipeline is a more recent feature of Jenkins Pipeline which:

- provides richer syntactical features over Scripted Pipeline syntax, and
- is designed to make writing and reading Pipeline code easier.

Many of the individual syntactical components (or "steps") written into a Jenkinsfile, however, are common to both Declarative and Scripted Pipeline. [2]
Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that

span from simple Continuous Integration (CI) to comprehensive CD pipelines. By modeling a series of related tasks, users can take advantage of the many features of Pipeline:

- **Code**: Pipelines are implemented in code and typically checked into source control, giving teams the ability to edit, review, and iterate upon their delivery pipeline.
- **Durable**: Pipelines can survive both planned and unplanned restarts of the Jenkins controller.
- **Pausable**: Pipelines can optionally stop and wait for human input or approval before continuing the Pipeline run.
- **Versatile**: Pipelines support complex real-world CD requirements, including the ability to fork/join, loop, and perform work in parallel.
- **Extensible**: The Pipeline plugin supports custom extensions to its Domain Specific Language (DSL) and multiple options for integration with other plugins.

[2]
One example of a CD pipeline is the following where

- A **Pipeline** is a user-defined model of a CD pipeline. A Pipeline's code defines your entire build process, which typically includes stages for building an application, testing it and then delivering it.
- A **node** is a machine which is part of the Jenkins environment and is capable of executing a Pipeline.
- A **stage** block defines a conceptually distinct subset of tasks performed through the entire Pipeline (e.g. "Build", "Test" and "Deploy" stages), which is used by many plugins to visualize or present Jenkins Pipeline status/progress.
- A single **task**. Fundamentally, a step tells Jenkins what to do at a particular point in time (or "step" in the process). For example, to execute the shell command make, use the sh step: sh 'make'. When a plugin extends the Pipeline DSL, that typically means the plugin has implemented a new step.

[2]

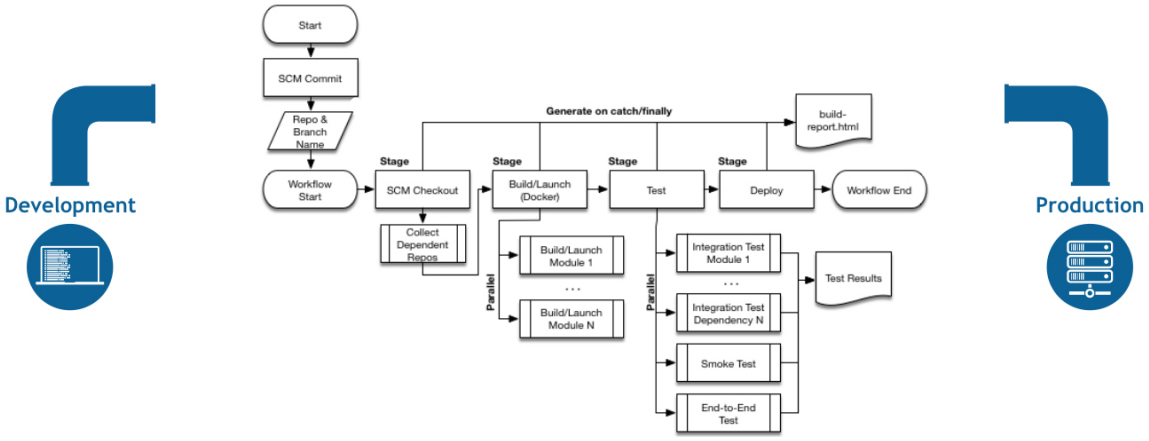


Figure 2-2: Pipeline-flow ([3])

2.2 Blue Ocean Pipeline Editor

Blue Ocean is built as a collection of Jenkins plugins. The key difference from Jenkins User Interface (UI) is that Blue Ocean provides both its own endpoint for HyperText Transfer Protocol (HTTP) requests, and it delivers HTML/JavaScript via a different path, without using the existing Jenkins UI markup or scripts. React.js and ES6 are used to deliver the JavaScript components of Blue Ocean. [4]
Blue Ocean as it stands provides easy-to-use Pipeline visualization. It was intended to be a rethink of the Jenkins user experience, designed from the ground up for Jenkins Pipeline. Blue Ocean was intended to reduce clutter and increases clarity for all users. However, Blue Ocean will not receive further functionality or enhancement updates. It will only receive selective updates for significant security issues or functional defects. To sum up, Blue Ocean's main features include:

- **Sophisticated visualization** of CD Pipelines, allowing for fast and intuitive comprehension of your Pipeline's status.
- **Pipeline editor** makes the creation of Pipelines more approachable, by guiding the user through a visual process to create a Pipeline.
- **Personalization** to suit the role-based needs of each member of the team.
- **Pinpoint precision** when intervention is needed or issues arise. Blue Ocean shows where attention is needed, facilitating exception handling and increasing productivity.
- **Native integration for branches and pull requests**, which enables maximum developer productivity when collaborating on code in GitHub and Bitbucket.

[4]

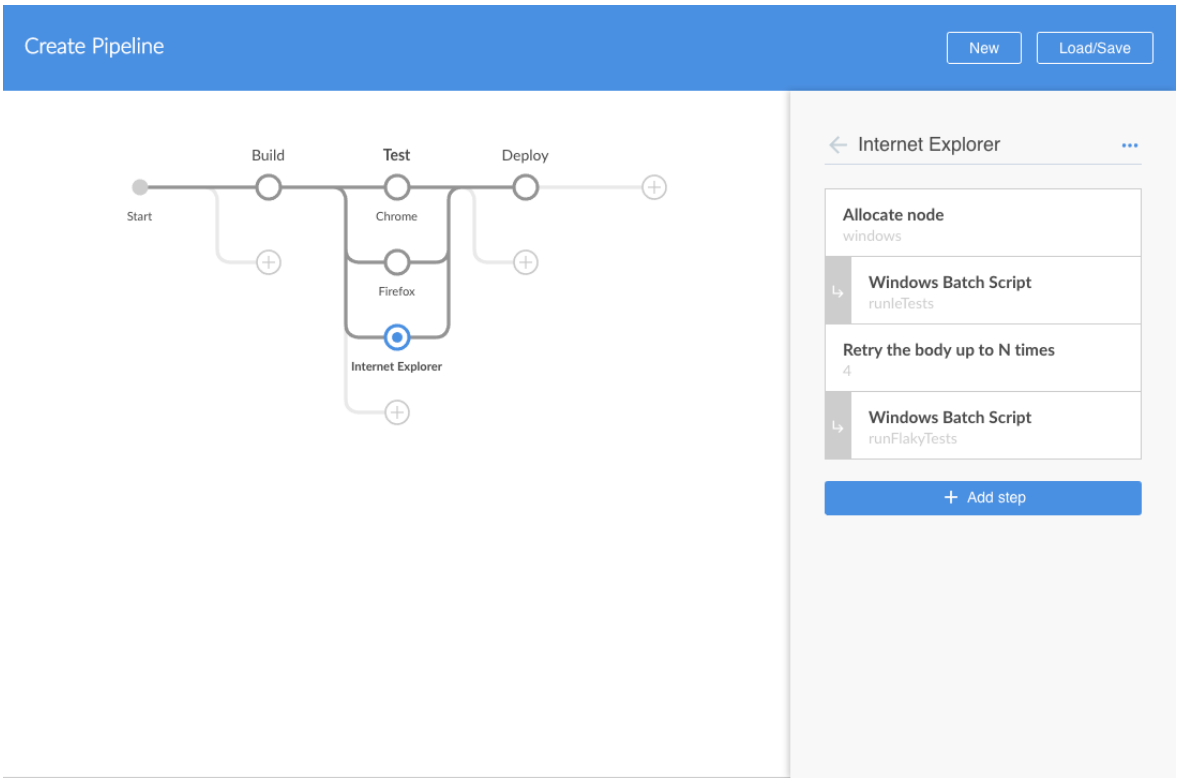


Figure 2-3: Blue ocean pipeline editor ([5])

In this example we can see a pipeline created to build, test, deploy a front-end application on different web browsers. The testing is done in parallel on the different browsers. New nodes can be added by clicking on the plus icons. The pipeline created on the canvas can be configured further by clicking on each node and then adding additional steps on the configuration menu on the right of the interface.

2.2.1 Pipeline Graph View Plugin

The Pipeline Graph View plugin is an alternative to Blue Ocean. It implements features for pipeline graph, summary of runs in a job, modern logs view. Actively maintained and developed, but the graph cannot be edited, it renders existing pipeline configuration.

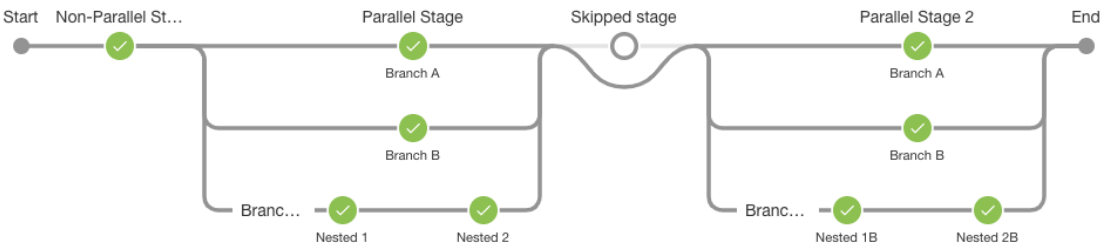


Figure 2-4: Semi-complex pipeline with graph view plugin ([6])

2.3 KubeSphere

KubeSphere is a distributed operating system for cloud-native application management, using Kubernetes as its kernel. It provides a plug-and-play architecture, allowing third-party applications to be seamlessly integrated into its ecosystem. [7]

One of the features of KubeSphere is a graphical editing panel. This panel allows the user to configure a pipeline using a web based graphical user interface. The user interface consists of two parts the canvas located on the left side and the content on the right. We can create the pipeline using the arrows in the canvas, and add steps to the stages in the content part. This operation creates a Jenkins declarative pipeline.

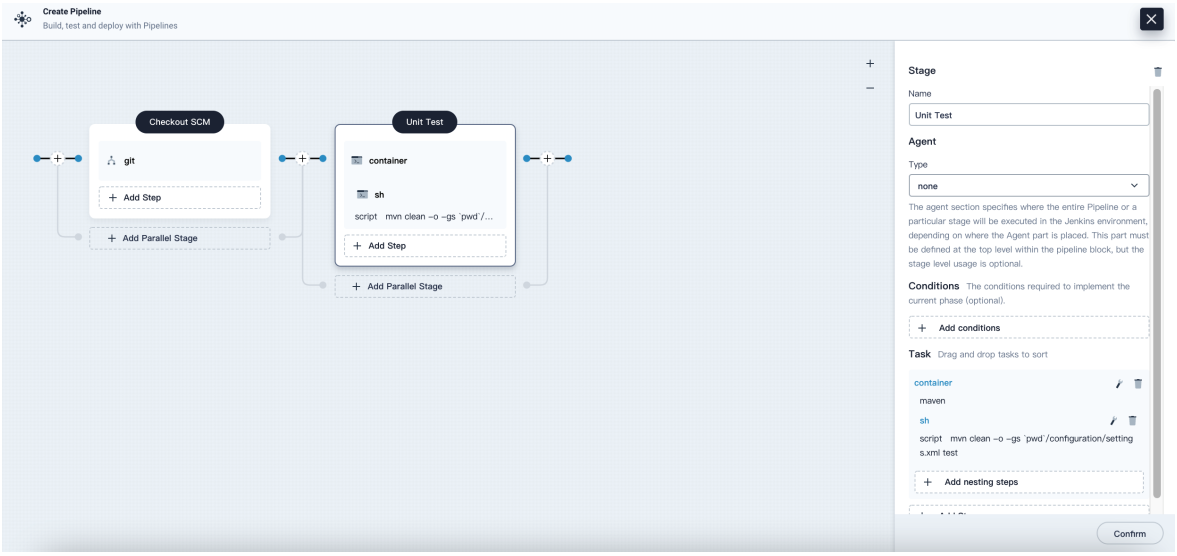


Figure 2-5: Making a pipeline with KubeSpehere ([8])

2.4 Camunda (8)

Camunda 8 orchestrates complex business processes that span people, systems, and devices. With Camunda, business users collaborate with developers to model and automate end-to-end processes using Business Process Model and Notation (BPMN)-powered flowcharts, alongside Decision Model and Notation (DMN) decision tables that promote speed, scale, and decision logic. [9]
Camunda 8 consists of six components

- **Modeler** - Model and deploy business process diagrams with BPMN and DMN.
- **Workflow engine** - Powered by **Zeebe**, Camunda's cloud-native workflow engine with speed, scale, and security without the overhead of building and maintaining a complex infrastructure.
- **Tasklist** - With Tasklist, process owners can achieve end-to-end process automation by orchestrating human tasks. When a user needs to work on a task, they'll observe it appear in Tasklist.
- **Operate** - Operate provides transparency and real-time visibility to monitor, analyze, and resolve problems with processes running in Camunda 8.
- **Optimize** - Optimize leverages process execution data to continuously provide actionable insights. Optimize specializes in BPMN-based analysis and can show users exactly what their process model needs for successful execution.
- **Console** - With Console, teams can create, configure, manage, and monitor clusters for all environments from development to production. Additionally, Console offers control over organizational settings such as user management, roles, and insights into usage metrics.

[9][10]

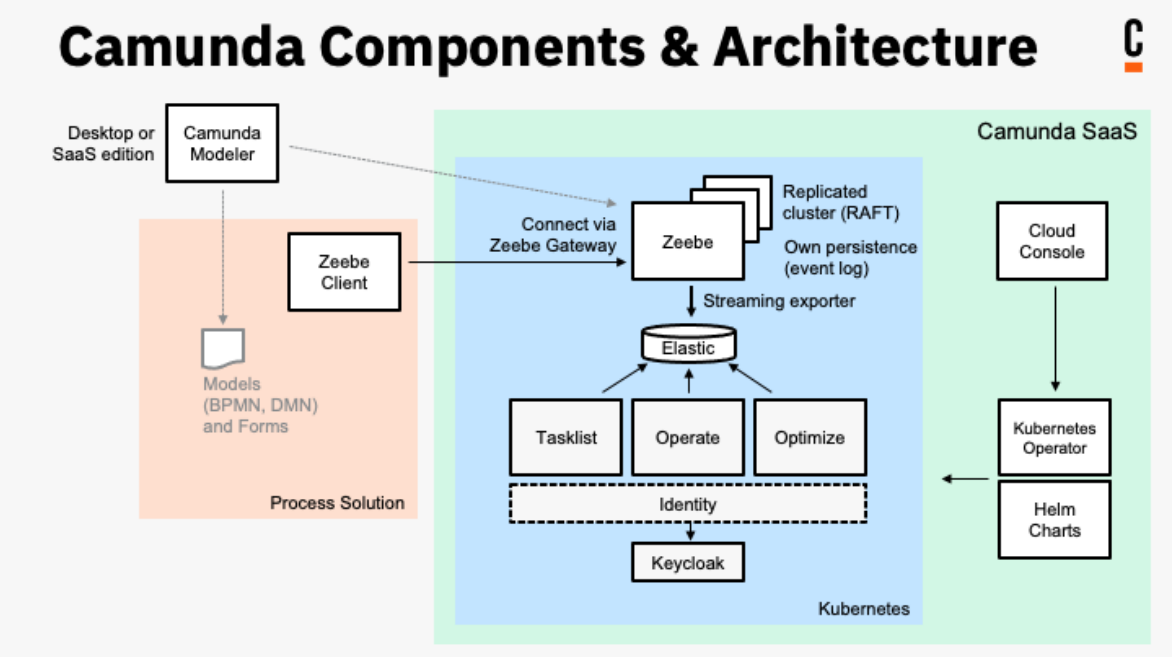


Figure 2-6: Camunda components and architecture ([10])

2.4.1 BPMN

The Object Management Group (OMG) has developed a standard BPMN. The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation.[11]

Another goal, but no less important, is to ensure that eXtensible Markup Language (XML) languages designed for the execution of business processes, such as Web Services Business Process Execution Language (WSBPEL), can be visualized with a business-oriented notation.[11]

This International Standard represents the amalgamation of best practices within the business modeling community to define the notation and semantics of Collaboration diagrams, Process diagrams, and Choreography diagrams. The intent of BPMN is to standardize a business process model and notation in the face of many different modeling notations and viewpoints. In doing so, BPMN will provide a simple means of communicating process information to other business users, process implementers, customers, and suppliers.[11]

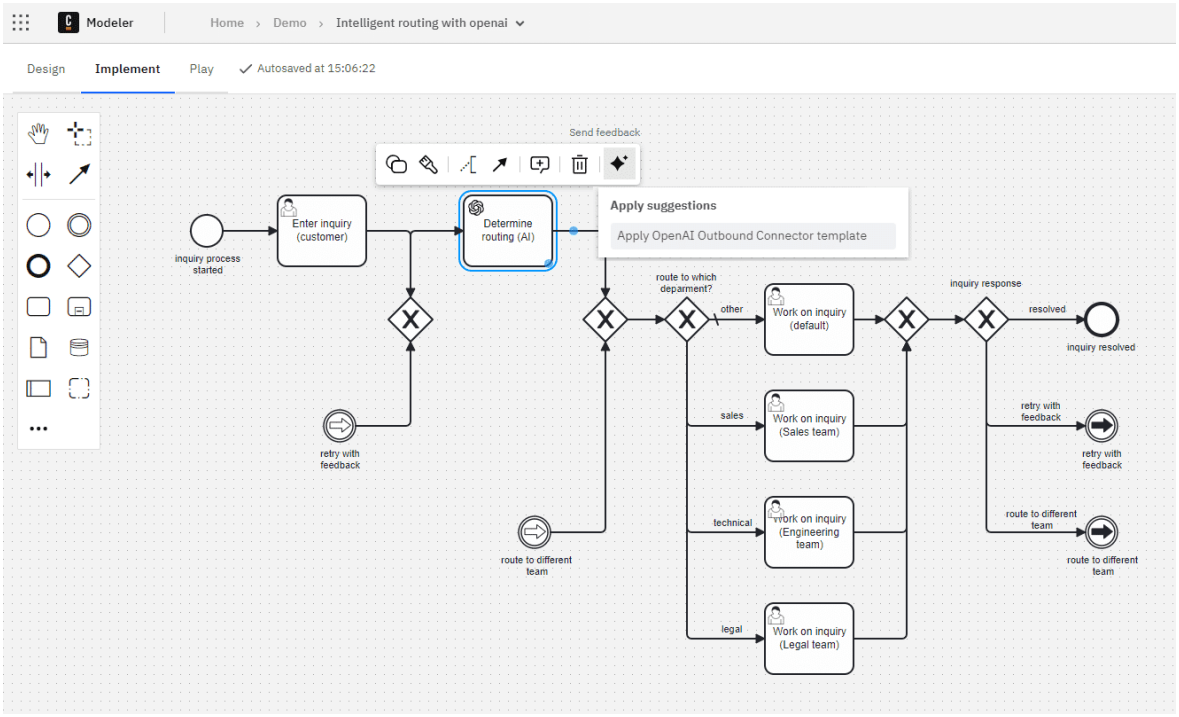


Figure 2-7: BPMN diagram on Camunda modeler([8])

3 Storing a workflow

Effective pipeline/dataflow visualization hinges not only on clarity of presentation but also on the underlying structure that holds the workflow information. This section delves into the various techniques for storing workflows within the visualization implementation. We will explore how different approaches can impact scalability, efficiency, and the overall user experience.

Our exploration encompasses various strategies and paradigms for storing workflows, ranging from traditional database-centric approaches to emerging technologies like graph databases. By examining the strengths and limitations of each approach, we aim to provide insights into selecting the most suitable storage solution for visualizing complex workflows.

3.1 Relational database

The existing setup uses MariaDB as the main database, but storing a Directed Acyclic Graph (DAG) in a Relational Database Management System (RDBMS)-s is not the most straight forward task.

The relational paradigm is appropriate for well-defined data structures that are unlikely to

change and translate naturally to tables, and the relations among its entities are not numerous and not as relevant as the entities' attributes. Hence, given its maturity and technological development, RDBMS-s are widely used for data storage, with countless examples experienced in everyday life, like user data, inventory tracking, blog posts and many more. However, when most relationships are many-to-many, prevalent in densely connected data, querying the database requires multiple expensive JOIN operations, impacting the performance. [12] Although graphs can be modeled with tables representing vertices and edges, complex queries or graph algorithms (like path traversals) are challenging to optimize without implementing complementary structures, such as adjacency lists.

Figure 3-8: Graph representation in relational database

In graph theory, an adjacency list 3-9 is the representation of all edges or arcs in a graph as a list. If the graph is undirected, every entry is a set (or multi-set) of two nodes containing the two ends of the corresponding edge; if it is directed, every entry is a tuple of two nodes, one denoting the source node and the other denoting the destination node of the corresponding arc. [13]

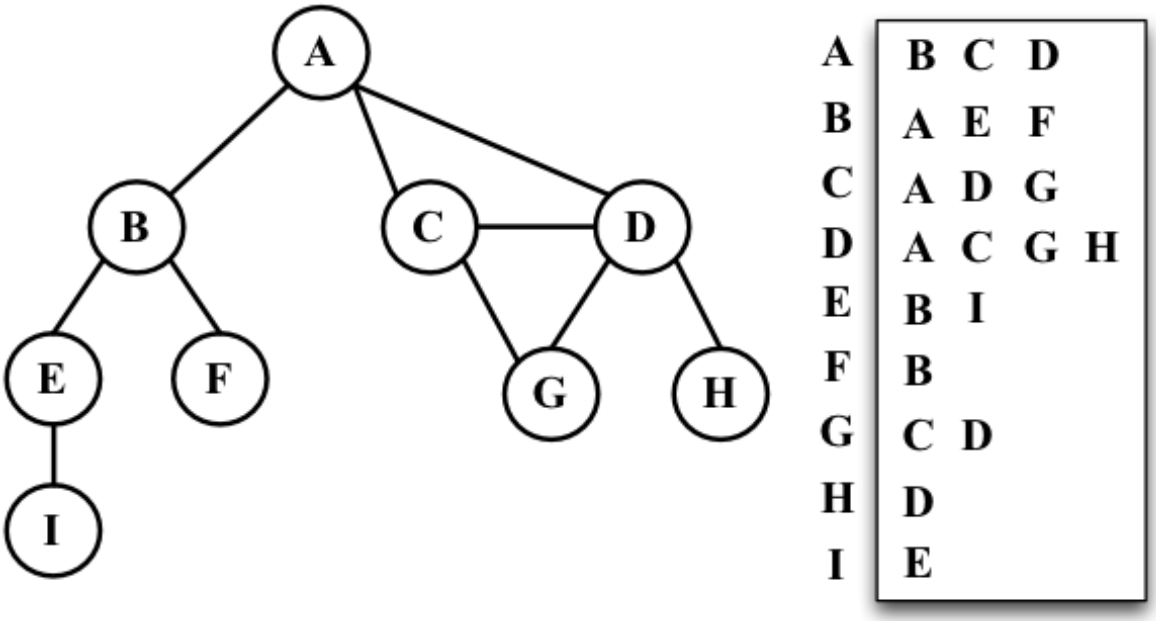


Figure 3-9: Adjacency list ([14])

3.2 Graph database

An other approach is to store the workflow in a graph database such as Neo4j.
Neo4j uses a property graph database model. A graph data structure consists of nodes (discrete objects) that can be connected by relationships. The Neo4j property graph database model consists of:

- **Nodes** describe entities (discrete objects) of a domain.
- **Nodes** can have zero or more **labels** to define (classify) what kind of nodes they are.
- **Relationships** describe a connection between a source node and a target node.
- **Relationships** always have a direction (one direction).
- **Relationships** must have a **type** (one type) to define (classify) what type of relationship they are.
- Nodes and relationships can have **properties** (key-value pairs), which further describe them.

[15]

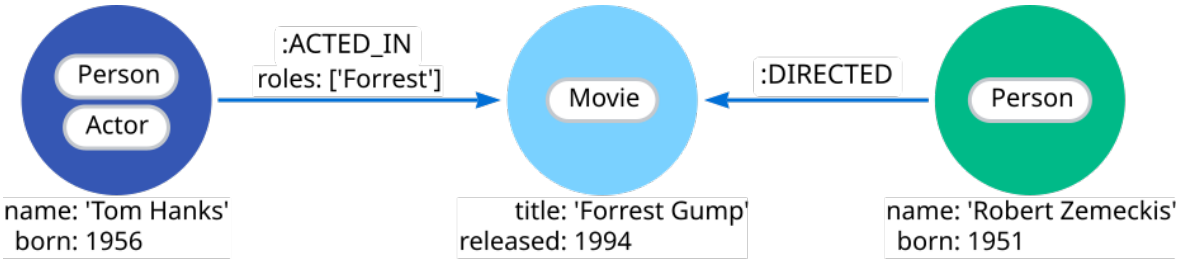


Figure 3-10: Example graph ([16])

To query the database Neo4j uses a language called Cypher query language. Cypher is Neo4j's graph query language that lets you retrieve data from the graph. It is like Structured Query Language (SQL) for graphs, and was inspired by SQL so it lets you focus on what data you want out of the graph (not how to go get it). It is the easiest graph language to learn by far because of its similarity to other languages and intuitiveness.

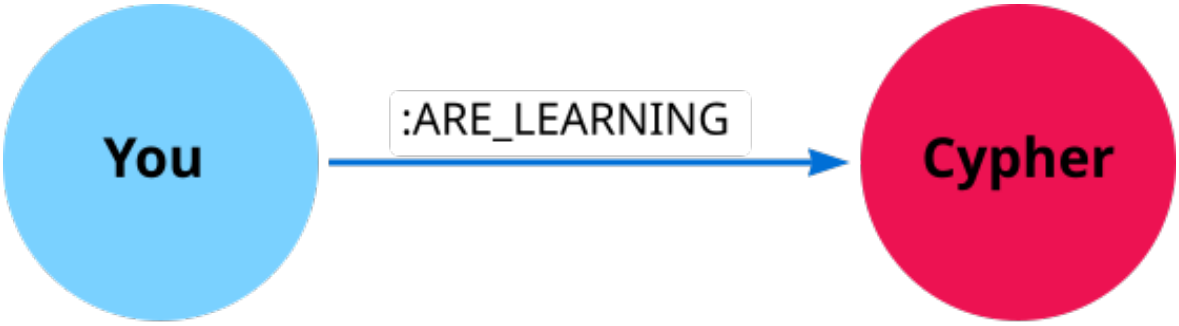


Figure 3-11: Cypher example graph ([17])

Cypher is unique because it provides a visual way of matching patterns and relationships. Cypher was inspired by an American Standard Code for Information Interchange (ASCII)-art type of syntax where **(nodes)-[:ARE_CONNECTED_TO]->(otherNodes)** using rounded brackets for circular **(nodes)**, and **-[:ARROWS]->** for relationships. When you write a query, you draw a graph pattern through your data. Neo4j users use Cypher to construct expressive and efficient queries to do any kind of create, read, update, and delete (CRUD) on their graph, and Cypher is the primary interface for Neo4j. [18]
The cypher representation of the graph 3-10:

```
CREATE (:Person:Actor {name: "Tom Hanks", born: 1956})-[:ACTED_IN
{roles: ["Forrest"]}]->(:Movie {title: "Forrest Gump", released:
1994})<-[:DIRECTED]-(:Person {name: "Robert Zemeckis", born:
1951})
```

3.3 Configuration file

Another potential way to store the workflow in the database is to describe the structure of the graph in a plain-text configuration file format such as JavaScript Object Notation (JSON), YAML Ain't Markup Language (YAML) or even a DSL to describe the structure of the database. The advantage of this approach is that the existing database setup could be used with a minor modification, adding a configuration file column to the database. The question arises however, that what format should the configuration file be in?

3.3.1 Jenkins declarative pipeline

A similar solution to this is used by Jenkins. The pipeline configuration is stored in a Jenkinsfile, a text file that contains the definition of a Jenkins Pipeline and is checked into source control. The following is an example of a three-stage CD pipeline.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

Figure 3-12: Jenkins declarative pipeline example ([19])

- **pipeline** defines a "block" containing all content and instructions for executing the

entire Pipeline.

- **agent** instructs Jenkins to allocate an executor (on a node) and workspace for the entire Pipeline.
- **stages** Containing a sequence of one or more stage directives, the stages section is where the bulk of the "work" described by a Pipeline will be located.
- **stage** (Build,Test,Deploy in this case) describes a stage of this Pipeline.
- **steps** describes the steps to be run in this stage.
- **echo** prints a message to console output.

This feature allows the user to define pipelines with a declarative syntax.
In computer science, declarative programming is a programming paradigm—a style of building the structure and elements of computer programs—that expresses the logic of a computation without describing its control flow. [20]

3.4 Kubeflow pipeline, pipeline as code

Kubeflow Pipeline (KFP) is a platform for building and deploying portable and scalable Machine Learning (ML) workflows using Docker containers. With KFP you can author components and pipelines using the KFP Python Software Development Kit (SDK), compile pipelines to an intermediate representation YAML, and submit the pipeline to run on a KFP-conformant backend such as the open source KFP backend or Google Cloud Vertex AI Pipelines. [21]
KFPs are defined with a pipeline function decorated with the `@dsl.pipeline` decorator. Take the following pipeline, `pythagorean`, which implements the Pythagorean theorem as a pipeline via simple arithmetic components:

```
from kfp import dsl

@dsl.component
def square(x: float) -> float:
    return x ** 2

@dsl.component
def add(x: float, y: float) -> float:
```

```
    return x + y

@dsl.component
def square_root(x: float) -> float:
    return x ** .5

@dsl.pipeline
def pythagorean(a: float, b: float) -> float:
    a_sq_task = square(x=a)
    b_sq_task = square(x=b)
    sum_task = add(x=a_sq_task.output, y=b_sq_task.output)
    return square_root(x=sum_task.output).output
```

Although a KFP pipeline decoratored with the `@dsl.pipeline` decorator looks like a normal Python function, it is actually an expression of pipeline topology and control flow semantics, constructed using the KFP DSL. [22]
A pipeline definition has four parts:

1. The pipeline decorator
2. Inputs and outputs declared in the function signature
3. Data passing and task dependencies
4. Task configurations
5. Pipeline control flow

[22]
KFP pipelines are defined inside functions decorated with the `@dsl.pipeline` decorator. The decorator takes three optional arguments:

- **name** is the name of your pipeline. If not provided, the name defaults to a sanitized version of the pipeline function name.
- **description** is a description of the pipeline.
- **pipeline_root** is the root path of the remote storage destination within which the tasks in your pipeline will create outputs. pipeline_root may also be set or overridden by pipeline submission clients.

- **display_name** is a human-readable for your pipeline.

You can modify the definition of *pythagorean* to use these arguments:

```
@dsl.pipeline(name='pythagorean-theorem-pipeline',
              description='Solve for the length of a hypotenuse of
              a triangle with sides length `a` and `b`.',
              pipeline_root='gs://my-pipelines-bucket',
              display_name='Pythagorean pipeline.')
def pythagorean(a: float, b: float) -> float:
    ...
```

[22]

4 Workflow Design and Development Overview

The workflow component micro-service is responsible for the creation, management, and running of pipelines. It consists of two main componenets a frontend and a backend component.
The following diagram illustrates how the components are interconnected:

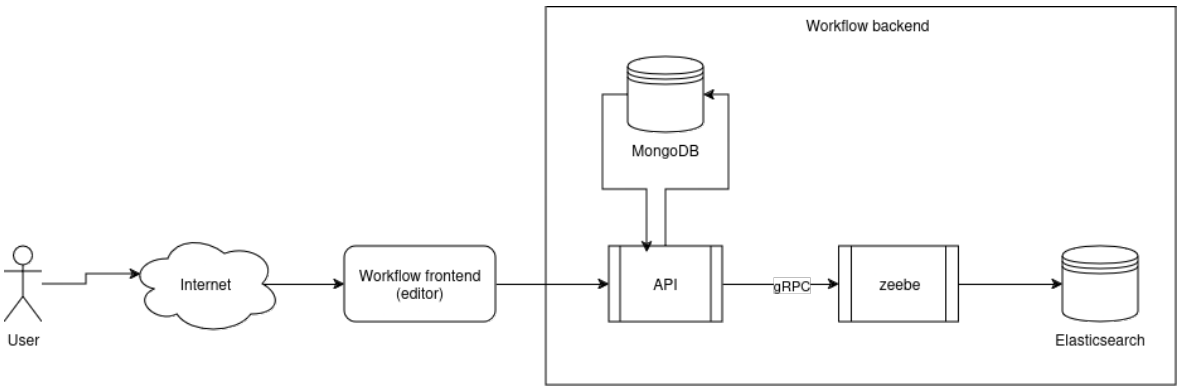


Figure 4-13: Workflow design

4.1 Frontend Component

The front-end is designed with an integrated workflow editor tool that uses *bpmn.js* to create industry-standard BPMN 2.0-compliant workflow files with the *.bpmn* extension. These files have an underlying XML structure with predefined elements. Once a workflow is created,

the frontend sends the corresponding BPMN file over a REST API to the backend, along with metadata about the workflow's parameters.

4.2 Backend Component

The backend handles data storage and provides REST API endpoints implemented using Flask and the Flask-RESTful package. The following endpoints are used to manage workflows:
Each endpoint is configured with a system wide role check, which is checked against the Keycloak authentication service. If the user does not have permissions to do a certain action the request is denied and a **403** response code is returned indicating that the user is not allowed to execute that action.

POST /workflow/upload

Upload workflow (.bpmn) files and additional metadata. The request body should conform to the following JSON document format:

```
{
  "name": "Workflow name"
  "description": "Workflow description"
  "bpmn": "bmpn file base64 encoded string",
  "variables": [
    {
      "name": "name of the variable",
      "description": "variable description",
      "type": "string|number",
      "default": "Default value for variable"
    }
  ]
}
```

If the request body does not have these field a **400** response code is returned and an error message indicating that the request body is incorrect. The uploaded workflow name is first checked against the intermediary MongoDB database, if the user has uploaded a workflow with that name already, the request is denied with a **409** status code conflict and an error message explaining the error. Next up the google Remote Procedure Call (gRPC) connection is established with the Zeebe component, trying to create a workflow. If this process fails **400**

error code is return with an error message to indicate that the workflow creation is unsuccessful. If the workflow creation is successful the following structure is added to MongoDB.

```
{
    "user_id": "UUID",
    "key": 2251799813685248,
    "deployments": [
        {
            "bpmn_process_id": "ECHO",
            "version": 1,
            "process_definition_key": 2251799813685250,
            "resource_name": "echo.bpmn",
        }
    ],
    "variables": [
        {
            "default": "Alapértelmezett érték",
            "description": "Leírás",
            "name": "A változó neve",
            "type": "string|number",
        }
    ],
}
```

This structure is created using the data returned from Zeebe and the `user_id` and the variables are appended to this. The `user_id` is created by Keycloak and is needed to connect the user with the workflows. The `resource_name` and the `user_id` is a composite system wide key that makes the workflow unique. Then the following structure and **200** status code is returned to the user indicating a successful request

```
{
  "process_definition_key": "string",
  "bpmn_process_id": "string",
  "version": 0
}
```

GET /workflows

Get all of the workflows for the user. This endpoint fetches the metadata associated with each workflow from MongoDB. This endpoint does not return the bpmn file content. The metadata information is returned as a list of objects where each object is a documnet in the workflow MongoDB collection. If the user does not have any workflows it simply returns an empty list.

GET /workflow/{name}

Get workflow by name. This endpoint returns a single workflow. If the workflow does not exist **404** status code is returned with an error message indicating that the workflow with that name does not exist. In case a workflow with the provided name path parameter exists, the corresponding bpmn content is fetched from the Elasticsearch database and appended to the standard MongoDB workflow collection schema and returned with the status code set to **200**.

DELETE /workflow/delete/{name}

Delete workflow by name. First the MongoDB is checked if the workflow with that name exists if there is such a workflow it is deleted from MongoDB and then delted form Elasticseach and a message is returned indicating that the delete was successful and a **200** status code is set for the respones. If there is no document matching the provided name path parameter in the collection a message indicating that the workflow is not found is returned with a **404** status code.

POST /workflows/run/{bpmn_process_id}

Run workflow by id. First the workflow with the bpmn_process_id is fetched from the MongoDB. If such document does not exist an error message indicating that there is no such workflow with a status code of **404** is returned. If the document exists the next check is if the request body conforms with the following JSON structure:

```
{
    "id": "UUID",
    "variables": {
        "message": "This is a Message"
    }
}
```

If the body is incorrect an error message describing this is returned with a **400** status code. Next the process is started through the Zeebe interface. If this operation fails an error message is returned with a status code of **400**. If the operation is successful a message stating that the specified workflow is started is returned with the status code set to **200**.

4.2.1 Flask

Flask is a lightweight Web Server Gateway Interface (WSGI) web application framework designed for quick and scalable application development. [23]

4.2.2 Flask-RESTful

Flask-RESTful extends Flask by adding support for quickly building REST APIs, promoting best practices with minimal setup. [24]

4.3 Integration with Zeebe

To enhance the workflow component, Zeebe is integrated into the system for handling workflow execution. Zeebe's capabilities include running workflows and storing process data in its internal Elasticsearch database.

4.3.1 Workflow Creation and Integration

Users can create workflows using the frontend's visual editor, which produces BPMN 2.0-compliant files. These files are then uploaded to Elasticsearch through Zeebe, which exposes an API using the gRPC protocol. The API's structure is defined in the `gateway.proto` file, enabling seamless communication. For retrieving processes and their associated metadata, direct interaction with Zeebe's Elasticsearch database is employed.

4.3.2 gRPC Protocol

The gRPC protocol allows a client application to directly call methods on a server application running on a different machine. The protocol is based on defining a service and specifying methods with their parameters and return types. Communication is facilitated by Protocol Buffers, which serialize structured data efficiently. [25]
The following snippet demonstrates a sample definition from Zeebe's `gateway.proto` file:

```
message DeployResourceRequest {
  repeated Resource resources = 1;
  string tenantId = 2;
}
message Resource {
  string name = 1;
  bytes content = 2;
}
```

[26]

4.3.3 Elasticsearch

Elasticsearch is utilized for searching and analyzing data stored by Zeebe. It supports near real-time search and provides a REST API for storing and retrieving data efficiently. Elasticsearch is an open source distributed, RESTful search and analytics engine, scalable data store, and vector database capable of addressing a growing number of use cases.[27]

- **Observability**: Logs, metrics, traces, and application performance monitoring.
- **Search**: Full-text search, semantic search, hybrid search, and geospatial queries.
- **Security**: Threat hunting and Security information and event management (SIEM).

[28]

4.4 Local and Production Deployment

For local development, Docker Compose is used to set up the required components. For production, the `docker-compose.yml` file is modified to ensure compatibility with Docker Swarm and the deployment environment.

4.5 Storing Metadata

User-specific metadata is stored in an intermediary MongoDB instance. This setup also supports implementing additional CRUD operations for managing workflows.

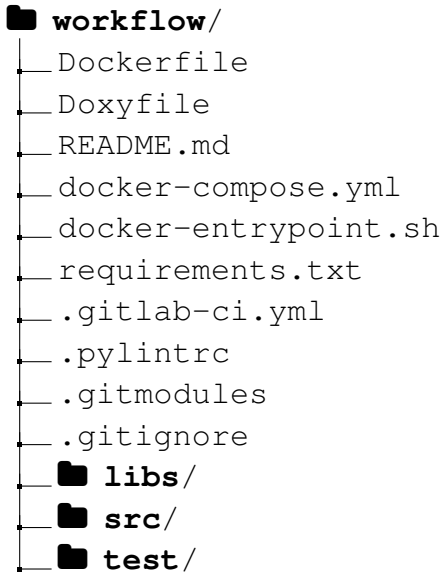
4.5.1 MongoDB

MongoDB serves as the intermediary database for storing user-specific metadata. Its document-based structure allows for flexible data modeling and high performance. Features include:

- Embedded data models that reduce I/O activity.
- Dynamic schema for polymorphism.
- High availability through replica sets.
- Horizontal scalability using sharding.

[29]

4.6 Root Directory structure



The `Dockerfile` and `docker-compose.yml` define the containerization setup, enabling consistent environments across deployments, while `docker-entrypoint.sh` initializes the container upon startup. Documentation is facilitated by `Doxyfile` for generating code documentation and `README.md`, which provides an overview of the project. Dependencies are managed through `requirements.txt`, and the `.gitlab-ci.yml` file automates tasks like testing and deployment via GitLab CI/CD. Configuration files such as `.pylintrc`, `.gitignore`, and `.gitmodules` handle coding standards, ignored files, and submodule tracking, respectively. The **libs/** folder contains reusable libraries shared

across microservices such as authentication related code, **src/** holds the main source code, and **test/** includes scripts and files for automated testing to ensure code quality and functionality. Further explanation will be provided in the relevant sections about the subdirectories and relevant files.

5 Implementation Details

The implementation details are illustrated using the directory structure and explained component by component, starting from the root or `src/` directory.

- **src/**
  - config.py
  - mongo_client.py
  - logging.ini
  - server.py
  - **camunda_interface/**
  - **resources/**
  - **workflow/**

5.1 Core Components

5.1.1 Configuration

The `config.py` file manages environment variables essential to the workflow, such as MongoDB connection strings and server port configurations. This ensures flexibility and security when deploying the application in different environments. Environment variables are accessed using Python's `os` module:

```python
# config.py
import os
INTERNAL_MONGO_DB_CONNECTION_STRING = os.environ.get(
    "INTERNAL_MONGO_DB_CONNECTION_STRING"
)
```

5.1.2 Logging

The `logging.ini` file configures custom loggers for internal workflow operations using Python's `logging` module. This allows for detailed tracking and debugging during execution.

5.1.3 MongoDB connection

The `mongo_client.py` file maintains a global MongoDB client object, created using the `pymongo` library. PyMongo is the official MongoDB driver, a Python package that can be used to connect to and communicate with MongoDB. [30] Using a global client object allows the reuse of connection pools, which enhances performance and resource utilization.

5.1.4 Entry point

The `server.py` file serves as the application's entry point. It initializes the Flask application and Flask-RESTful API.

```
# server.py
from flask import Flask, jsonify
from flask_restful import Api
app = Flask(__name__)
api = Api(app)
```

Resource classes are mapped to endpoints with type-checked path parameters. An example of resource mapping:

```
# server.py
from resources.workflows import WorkflowGetResource
api.add_resource(WorkflowGetResource, "/workflow/<string:name>")
```

A default error handler is implemented to prevent stack traces from being exposed to users while logging errors for debugging purposes:

```
@app.errorhandler(Exception)
def default_error_handler(error):
    logging.exception(error)
    return jsonify(message=str(error)), getattr(error, "code", 500)
```

When running the application, environment variables defined in the configuration file are utilized, ensuring a modular and flexible deployment. This is achieved as follows:

```
import config
if __name__ == "__main__":
    app.run(debug=config.DEBUG_MODE, host="0.0.0.0",
    ↪  port=int(config.SERVER_PORT))
```

5.2 Workflow Endpoints

5.2.1 Resources

```
resources/
├── __init__.py
├── workflow_delete.py
├── workflow_get.py
├── workflow_run.py
├── workflow_upload.py
└── workflows.py
```

The `resources/` folder contains resource classes that represent RESTful endpoints. Resources are the main building block of an application based on Flask-RESTful. Resources are built on top of Flask pluggable views, giving easy access to multiple HTTP methods just by defining methods on the resource. [31]
Each class corresponds to a specific operation, such as uploading or retrieving workflows. For example, the `WorkflowsResource` retrieves workflows from MongoDB:

```
class WorkflowsResource(Resource):
    def __init__(self):
        self.__logger = logging.getLogger("workflow")
        self.__role_checker = RoleChecker()
        # set access control details here ...
    def get(self):
        auth_request = AuthorizationRequest()
        # set request details
        has_permission =
        self.__role_checker.check_permission(auth_request)
        if not has_permission:
            return Response(status=HttpStatusCodes.FORBIDDEN)
        # retreive data from MongoDB
        workflow_store = WorkflowStore(token_parser.user_id)
        workflows = workflow_store.get_all_workflows_for_user()
        response = jsonify(workflows)
        response.status_code = HttpStatusCodes.OK
        return response
```

5.3 External Interfaces

5.3.1 MongoDB abstraction layer

```
workflow/
├── __init__.py
└── db.py
```

The `workflow/` directory contains modules interfacing with MongoDB, abstracting the underlying database operations through the `MongoStore` class. This class provides a lightweight wrapper around the `pymongo` driver, centralizing CRUD operations.

5.3.2 Camunda Interface

```
camunda_interface/
├── __init__.py
├── elastic_interface.py
├── zeebe_interface.py
├── zeebe_grpc/
└── internal_models/
```

The `camunda_interface/` folder bridges the application to external orchestration services like Zeebe. It includes gRPC-generated files and high-level abstractions for managing workflows. For example, the `ZeebeGrpcManager` class simplifies the lifecycle of gRPC connections:

```
with ZeebeGrpcManager(logger=self.__logger, target="zeebe:26500") as zm:
    # gRPC connection is created automatically
    deployed_res = zm.create_resource(name=wfi.name, content=wfi.bpmn)
# gRPC connection is closed automatically
```

This class is implemented as a context manager. This allows the automatic opening and closing of the connection to the specified service. Under the hood the connection is established in the `__enter__()` method and closed in the `__exit__` method. These methods are called automatically by Python (even in case of an exception).
A context manager is an object that defines the runtime context to be established when executing a with statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the with statement, but can also be used by directly invoking their methods.[32]

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files[32]

Generated gRPC files

```
zeebe_grpc/
├── README.md
├── __init__.py
├── gateway.proto
├── gateway_pb2.py
├── gateway_pb2.pyi
└── gateway_pb2_grpc.py
```

The `zeebe_grpc/` directory contains the files generated from the `gateway.proto` file by the following command:

```
python -m grpc_tools.protoc -I./ --python_out=. --pyi_out=.
--grpc_python_out=. gateway.proto
```

This generates `gateway_pb2.py` which contains our generated request and response classes and `gateway_pb2_grpc.py` which contains our generated client and server classes and the `gateway_pb2.pyi` contains type annotations for the Python generated module.

Internal Models

```
internal_models/
├── __init__.py
├── elastic/
│   ├── __init__.py
│   ├── deployment.py
│   ├── job.py
│   ├── process.py
│   ├── process_instance.py
│   ├── process_instance_creation.py
│   └── variable.py
└── zeebe/
    ├── __init__.py
    └── deployed_resource.py
```

Elasticsearch models are implemented using the `elasticsearch-dsl` library. These models map Elasticsearch documents to Python classes, facilitating seamless data index-

ing and retrieval. For instance, the `ElasticProcessRecord` class represents a Zeebe process record:

```
# process.py
from elasticsearch_dsl import ( Boolean, Date, Document, InnerDoc,
Integer, Keyword, Long, Nested, Text)

class Authorization(InnerDoc):
    authorized_tenants = Keyword(multi=True)

class ProcessValue(InnerDoc):
    checksum = Text()
    versionTag = Text()
    deploymentKey = Long()
    duplicate = Boolean()
    version = Integer()
    resource = Text()
    tenantId = Keyword()
    resourceName = Text()
    bpmnProcessId = Text()
    processDefinitionKey = Long()

class ElasticProcessRecord(Document):
    partitionId = Integer()
    value = ProcessValue
    key = Long()
    timestamp = Date()
    position = Integer()
    valueType = Keyword()
    brokerVersion = Text()
    authorizations = Authorization
    intent = Keyword()
    rejectionType = Keyword()
    rejectionReason = Text()
    recordType = Keyword()
    operationReference = Integer()
    sourceRecordPosition = Integer()
    recordVersion = Integer()
    sequence = Long()

    class Index:
        name = "zeebe-record-process"
```

The Zeebe internal models are pretty simple dataclass models to represent the structure returned after process deployment creation from Zeebe.

6 Integrating workflow to the existing infrastructure

To create the Docker file to containerize the API component of the workflow. For that Since BlackAnt uses docker swarm mode for deploying the whole infrastructure. The file describing the state of the cluster is in a file `docker-compose-dev.yml` file for development and the `docker-compose-prod.yml` file for production use. In order to integrate the workflow micro-service into this infrastructure each workflow component has to configured in these files. The next snippet shows the additional configuration required for the workflow component. Each of these components will use the same network: `workflow_net` which is added to the rest of the networks.

6.1 API(workflow) configuration:

Since there is a MongoDB configured already for internal use inside the swarm. There is no need to add an additional configuration for that. A port mapping can be configured for local access outside the stack, which makes testing running instances more easy. The API will act as a gateway for the workflow component so it needs access to the `api_net` network which is used by the internal NGINX reverse proxy and Transport Layer Security (TLS) termination service to forward requests to corresponding micro-services. The `ENVIRONMENT_HOST` environment variable should be set to the BlackAnt docker registry Uniform Resource Locator (URL). `IMAGE_TAG` environment variable is responsible for setting the tag latest for development and stable for production.

```
# docker-compose-dev.yml
...
  workflow:
    image: ${ENVIRONMENT_HOST}:5000/system/workflow:${IMAGE_TAG}
    environment:
      INTERNAL_MONGO_DB_CONNECTION_STRING_FILE:
      /run/secrets/internal_mongo_db_connection_string
    networks: [api_net, mongo_internal_net,workflow_net]
    secrets: [internal_mongo_db_connection_string]
    deploy:
```

```
      mode: replicated
      replicas: 1
      placement:
        constraints: [node.role == manager]
    logging:
      driver: json-file
      options:
        max-size: 1024m
        max-file: '5'
        labels: workflow
    depends_on: [zeebe, mongo1-internal, mongo2-internal]
...
```

6.2 Zeebe configuration:

Recommended Java related environment variables are added for optimal operation.

```
# docker-compose-dev.yml
...
  zeebe:
    image: camunda/zeebe:${CAMUNDA_PLATFORM_VERSION}
    environment:
      # allow running with low disk space
      - ZEEBE_BROKER_DATA_DISKUSAGECOMMANDWATERMARK=0.998
      - ZEEBE_BROKER_DATA_DISKUSAGEREPLICATIONWATERMARK=0.999
      - JAVA_TOOL_OPTIONS=-Xms512m -Xmx512m
    restart: unless-stopped
    healthcheck:
      test:
        - CMD-SHELL
        - timeout 10s bash -c ':> /dev/tcp/127.0.0.1/9600' || exit 1
      interval: 30s
      timeout: 5s
      retries: 5
      start_period: 30s
    volumes: [zeebe:/usr/local/zeebe/data]
    networks: [workflow_net]
    depends_on: [elasticsearch]
...
```

6.3 Elasticsearch setup

Since Elasticsearch is a database, volumes needs to be configured with persistent storage. Some additional java related environment variables are set for optimal performance.

```
# docker-compose-dev.yml
...
  elasticsearch:
    image:
    docker.elastic.co/elasticsearch/elasticsearch:${ELASTIC_VERSION}
    environment:
      - bootstrap.memory_lock=true
      - discovery.type=single-node
      - xpack.security.enabled=false
      # allow running with low disk space
      - cluster.routing.allocation.disk.threshold_enabled=false
      - ES_JAVA_OPTS=-Xms512m -Xmx512m
    ulimits:
      memlock:
        soft: -1
        hard: -1
    restart: unless-stopped
    healthcheck:
      test:
        - CMD-SHELL
        - curl -f http://localhost:9200/_cat/health | grep -q green
      interval: 30s
      timeout: 5s
      retries: 3
    volumes: [elastic:/usr/share/elasticsearch/data]
    networks: [workflow_net]
...
```

6.4 Balack Ant docker registry

Black Ant has a dedicated self hosted docker registry which stores the images that are needed to deploy the whole infrastructure. When developing a new micro-service, the newly created custom docker images and the prebuilt images used by the service, need to be stored in the "local" docker registry.

6.5 Deployment

The BlackAnt infrastructure is deployed using docker swarm mode from a single docker compose file. This allows the infrastructure to be distributed and scaled according to business needs.

6.5.1 Docker Swarm mode

Swarm mode, an advanced feature integrated into the Docker Engine, is designed for managing clusters of Docker daemons (known as swarms) and serves as a robust production runtime environment. It enables seamless cluster management, allowing users to create swarms, deploy application services, and manage their behavior using the Docker Command Line Interface (CLI) without additional orchestration tools. With its decentralized design, the Docker Engine handles role specialization at runtime, simplifying the deployment of manager and worker nodes. The declarative service model lets users define the desired state of services, including scaling, where the swarm manager dynamically adjusts the number of tasks to maintain the specified state. The swarm manager ensures desired state reconciliation by constantly monitoring the cluster, replacing failed replicas, and redistributing tasks as needed. Features like multi-host networking with overlay networks, service discovery through DNS names, and load balancing enhance usability and scalability. Swarm mode prioritizes security with TLS mutual authentication and encryption for node communication and supports rolling updates for controlled service deployment with rollback options. It provides a modern and secure solution for distributed application management, distinct from the deprecated Docker Classic Swarm. [33]

7 CI/CD

Continuous Integration/Continuous Delivery (CI/CD) are fundamental practices in modern software development, enabling teams to deliver high-quality applications quickly and efficiently. CI/CD automates the software life-cycle, from integrating code changes to deploying applications, fostering rapid and frequent releases that respond to user needs. By catching errors early through continuous integration, teams can reduce downtime and maintain software quality. Continuous delivery extends this process by automating infrastructure provisioning and application deployment, ensuring that software is always in a deployable state. This automation accelerates time-to-market, enhances stakeholder feedback loops, and simplifies collaboration across development and operations teams. [34]

A CI/CD pipeline streamlines these processes by automating builds, testing, and deployments, minimizing human intervention and reducing errors. Continuous testing within the pipeline ensures bugs are caught early through automated tests such as unit, integration, and regression testing. Additionally, CI/CD promotes practices like frequent code integration, automated builds, and deployment in stable environments, maximizing visibility and enabling predictable releases. Integrating CI/CD with DevOps workflows bridges development and operations, fostering collaboration, improving efficiency, and encouraging continuous improvement. As a cornerstone of DevOps, CI/CD helps organizations achieve a faster, more reliable, and user-focused development process. [34]

7.1 Steps to create a GitLab CI/CD pipeline

1. **Create a `.gitlab-ci.yml` file:** Define stages, jobs, and scripts for the CI/CD pipeline. This YAML file allows configuration of variables, job dependencies, and execution conditions.

2. **Find or create runners:** Runners execute jobs. Use available runners on GitLab, register your own, or create one locally to run jobs either locally or in a container.

3. **Define your pipelines:** Use the `.gitlab-ci.yml` file to specify pipelines consisting of stages (e.g., build, test, deploy) and jobs triggered by events like commits or schedules.

4. **Use CI/CD variables:** Store configuration settings or sensitive information using key-value pairs. Define variables in the configuration file, project settings, or dynamically, and manage their security with protections and masking.

5. **Use CI/CD components:** Include reusable pipeline configuration units to reduce duplication and improve maintainability. Share components via the CI/CD Catalog or use templates for common tasks.

[35]

7.2 Black Ant CI/CD

Developers store their service implementations in GitLab, from which a Docker Image is automatically created for each merge using GitLab CI/CD and stored in the Docker Registry.

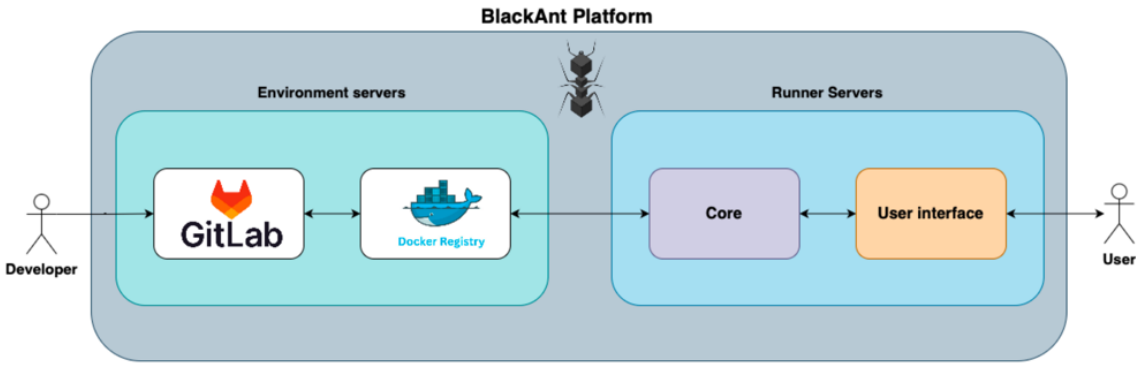


Figure 7-14: Black Ant platform
([36])

7.3 Workflow component CI/CD

The GitLab CI/CD pipeline for this project is structured into three stages: `test`, `build`, and `deploy`. Each stage contains one or more jobs that define specific tasks to ensure code quality, build consistency, and proper deployment workflows. The following subsections detail the steps involved in each stage.

7.3.1 Variables

The pipeline leverages predefined variables to manage its environment:

- `DOCKER_TLS_CERTDIR`: Set to an empty string to disable Docker TLS certificate directory creation.
- `GIT_SSL_NO_VERIFY`: Set to `true` to bypass Secure Sockets Layer (SSL) certificate verification when interacting with Git repositories.
- `GIT_SUBMODULE_STRATEGY`: Configured as `recursive` to ensure all nested Git submodules are fetched.

7.3.2 Test Stage

The `test` stage includes jobs to validate the project's code and configuration files:

Check Compose File: This job verifies the correctness of the `docker-compose.yml` file using the following steps:

1. Build Docker images specified in the compose file with `docker-compose -f docker-compose.yml build`.
2. Start the services in detached mode using `docker-compose -f docker-compose.yml up -d`.

This job runs only if `docker-compose.yml` is modified.

Run Pylint: This job conducts static code analysis for Python files, enforcing a minimum quality score of 9.0. The steps include:

1. Installing dependencies from `requirements.txt`.
2. Installing specific versions of Pylint and related tools.
3. Running `pylint-fail-under` with the project's configuration from `.pylintrc`.

It triggers only when files under the `src` directory are changed.

7.3.3 Build Stage

The `build` stage includes jobs for constructing and managing Docker images:

Build: This job builds a Docker image named `workflow`:

1. Logs into the Docker registry using CI/CD credentials.
2. Builds the image from the repository's `Dockerfile`.

It is excluded for the `master` and `main` branches.

Build and Push Image: For the `master` and `main` branches, this job builds the Docker image, tags it as `latest`, and pushes it to the Docker registry.

Generate Documentation: This manual job uses Doxygen to generate documentation:

1. Runs Doxygen with the provided `Doxyfile`.
2. Compiles LaTeX-based documentation within the `docs/latex` directory.

Artifacts, including a PDF and HTML documentation, are stored for 1 day.

7.3.4 Deploy Stage

The `deploy` stage consists of a manual job for tagging and managing Docker images:

Tag Stable Image: This job tags the `latest` Docker image as `stable` and pushes it to the Docker registry:

1. Logs into the registry using CI/CD credentials.
2. Pulls the `latest` image and retags it as `stable`.
3. Pushes the `stable` tag to the registry.

This pipeline is designed to ensure reliable testing, building, and deployment of the project while adhering to quality standards.

8 Testing

```
test/
├── __init__.py
├── base.py
├── initial_data.json
├── test.sh
└── test_workflow.py
```

For testing purposes, primarily functional and integration testing approaches are employed to ensure that each endpoint operates as intended. The test definitions reside in the `test_workflow.py` file. Each endpoint is represented by its own class, and within each class, test methods are implemented to cover various edge cases encountered during requests. The tests are designed to be executed within the local Docker Compose setup, enabling comprehensive testing of the workflow components' functionality. This setup includes the following services:

- Workflow API server
- Zeebe (orchestration engine)
- Elasticsearch
- MongoDB (internal database)

The `test.sh` script facilitates the testing process by initializing the database with dummy data from the `initial_data.json` file. This data is subsequently used by the tests to simulate realistic scenarios. After initialization, the script executes the test suite.
Testing is conducted using the Python `unittest` framework, a robust tool included in the standard library. Within `unittest`, each class serves as a test case, inheriting from `TestCase`. Test methods are prefixed with *test_* and are named descriptively to reflect the specific scenarios they validate. These methods utilize assertion statements to verify expected outcomes.
The results of the tests are evaluated and reported by `unittest`, with detailed output printed to the console. This structured approach ensures clarity and efficiency in verifying the behavior and reliability of the workflow implementation.

9 Evaluation

The implementation of the workflow system into the BalckAnt cloud services framework demonstrates significant progress in backend development, providing a robust and scalable solution for task orchestration. The backend offers comprehensive functionality through clearly defined endpoints, enabling effective management of workflows by users. The **POST /workflow/upload** endpoint ensures secure and conflict-free workflow creation, integrating with MongoDB for metadata storage and Zeebe via gRPC for workflow deployment. This feature highlights the system's robustness, as it includes extensive error handling to guide users in case of invalid inputs, workflow naming conflicts, or workflow creation failures. The **GET /workflows** and **GET /workflow/{name}** endpoints enhance user access by retrieving workflows and metadata efficiently, ensuring accurate and user-specific data retrieval.
Additionally, the backend supports workflow lifecycle management through the **DELETE /workflow/delete/{name}** endpoint, which ensures synchronized deletion across MongoDB and Elasticsearch, maintaining data consistency. The ability to execute workflows via the **POST /workflows/run/{bpmn_process_id}** endpoint adds dynamic functionality, allowing users to interact with deployed workflows seamlessly. Extensive error handling across all endpoints ensures that users receive clear feedback, improving the system's reliability and usability. Further achievements include the integration of Role-Based Access Control (RBAC) using Keycloak. This ensures secure, user-specific access to workflow management features and allows administrators to enforce granular access policies seamlessly. The system supports workflows tailored to specific users by utilizing Keycloak-generated user IDs as part of composite keys in MongoDB. Additionally, Docker Swarm mode provides a reliable and scalable backbone for deploying and managing the platform, ensuring high availability and load balancing across services. These architectural decisions align with modern practices in microservices and cloud computing, demonstrating a strong focus on performance and maintainability. These features demonstrate the backend's capability to fulfill complex R&D&I requirements while establishing a solid foundation for future enhancements. Although the frontend remains incomplete, the backend provides all essential functionality for workflow management, positioning the system as a critical component in advancing the University's research infrastructure.

9.1 Future Work

While the backend implementation effectively addresses core workflow management needs, several enhancements could improve functionality and user experience. Developing a fully

functional frontend with features like drag-and-drop workflow design, real-time visualization, and error notifications would make the system more accessible, especially for non-technical users. Incorporating workflow versioning would allow users to manage multiple iterations of a workflow, enabling rollbacks and comparisons for iterative development. Additionally, automated workflow validation and testing could ensure logical consistency and compliance with system requirements before deployment.

Further improvements could focus on advanced monitoring and analytics capabilities. Integrating Prometheus and Grafana for workflow metrics such as execution times, failure rates, and completion rates would provide valuable insights, while centralized logging with tools like ELK could streamline error tracking and diagnostics. Supporting multiple workflow engines, would increase flexibility and broaden the system's applicability. Additional features, such as dynamic updates to workflows and export functionality, could further enhance backend versatility.

Docker Swarm mode provides a solid foundation for the platform, but exploring more advanced orchestration solutions like Kubernetes could further enhance scalability and fault tolerance. Kubernetes' robust ecosystem and features, such as auto-scaling and more dynamic service discovery, align well with the system's high-availability requirements.

Many of these proposed enhancements have already been documented as issues in the project's GitLab repository. This ensures a structured approach to their implementation and provides a clear roadmap for future development. By prioritizing and addressing these issues systematically, the platform can continue evolving in alignment with its long-term objectives and user needs.

10 ABSTRACT

The goal of this project is to design and implement a workflow management system within the BlackAnt high-availability cloud infrastructure. The workflow system is primarily focused on backend development, with the frontend being managed separately. The workflow management utilizes Zeebe, a distributed workflow engine that allows for scalable and efficient orchestration of tasks. In addition, Elasticsearch is employed to provide advanced search capabilities across workflow data.

A critical part of the project involves automating the deployment process through CI/CD pipelines, ensuring smooth development cycles, automated testing, and continuous delivery. The CI/CD pipelines are implemented using GitLab and Docker, allowing for streamlined builds, testing, and deployment stages, ensuring that the workflow system is robust and resilient to failure.

The system's backend is built using Flask, with an emphasis on RESTful APIs and integration with external services such as MongoDB for metadata storage and Camunda for additional business process management capabilities. This architecture also ensures integration with existing infrastructure, allowing the workflow management system to scale across multiple environments, including local and production deployments.

The project also explores various methods for storing workflow data, comparing relational databases, graph databases, and configuration file approaches, with a focus on providing optimal scalability and flexibility. Relational databases, though widely used, often struggle with representing complex, interdependent workflows due to their rigid schema and normalization requirements. In contrast, graph databases like Neo4j offer superior flexibility for modeling workflows with dynamic task dependencies and relationships, enabling more efficient querying and real-time tracking. Additionally, the use of configuration files for defining workflows—common in tools like Jenkins—provides a human-readable and easily maintainable format but may lack the scalability required for more complex workflows. The study investigates how these approaches can be adapted to the project's needs, particularly in high-availability microservice architectures, to balance performance and ease of management.

11 ABSZTRAKT

A szakdolgozat célja egy munkafolyamat-irányítási rendszer megtervezése és megvalósítása egy magas rendelkezésre állású felhőinfrastruktúrában. A munkafolyamat-rendszer elsősorban a backend fejlesztésére összpontosít, míg a frontend külön van kezelve. A munkafolyamat-irányítás a Zeebe nevű elosztott munkafolyamat-motorra épít, amely lehetővé teszi a feladatok skálázható és hatékony megszervezését. Ezen kívül az Elasticsearch alkalmazása biztosítja a fejlett keresési lehetőségeket a munkafolyamat-adatokon keresztül.

A szakdolgozat kulcsfontosságú része a telepítési folyamat automatizálása CI/CD pipeline-ok segítségével, biztosítva a zökkenőmentes fejlesztési ciklusokat, automatizált tesztelést és folyamatos szállítást. A CI/CD pipeline-ok a GitLab és Docker segítségével van megvalósítva, lehetővé téve az egyszerűsített építési, tesztelési és telepítési szakaszokat, így biztosítva, hogy a munkafolyamat-rendszer robusztus és hibatűrő legyen.

A rendszer backendje Flask keretrendszerben valósult meg, amely a RESTful API-kra és az olyan külső szolgáltatásokkal való integrációra helyezi a hangsúlyt, mint a MongoDB a metainformációk tárolására, valamint a Camunda, amely további üzleti folyamat-irányítási lehetőségeket biztosít. Ez az architektúra biztosítja az integrációt a meglévő infrastruktúrával, lehetővé téve a munkafolyamat-irányítási rendszer skálázását több környezetben, beleértve a lokális és éles telepítést.

A szakdolgozat a munkafolyamat-adatok tárolásának különböző módszereit is vizsgálja, összehasonlítva a relációs adatbázisokat, gráf adatbázisokat és konfigurációs fájlokat, különös figyelmet fordítva az optimális skálázhatóság és rugalmasság biztosítására. A relációs adatbázisok, bár széles körben alkalmazottak, gyakran nem képesek jól reprezentálni a bonyolult, egymásra épülő munkafolyamatokat a merev sémájuk és normalizációs követelményeik miatt. Ezzel szemben a gráf adatbázisok, mint a Neo4j, kiváló rugalmasságot kínálnak a dinamikus feladatfüggőségek és kapcsolatok modellezésére, így hatékonyabb lekérdezéseket és valós idejű nyomon követést tesznek lehetővé. Továbbá, a munkafolyamatok meghatározására szolgáló konfigurációs fájlok alkalmazása—amely elterjedt olyan eszközökben, mint a Jenkins—emberi olvasásra alkalmas és könnyen karbantartható formátumot biztosít, de előfordulhat, hogy nem rendelkezik a szükséges skálázhatósággal a bonyolultabb munkafolyamatokhoz. A tanulmány azt vizsgálja, hogyan alkalmazhatók ezek a megközelítések a projekt igényeire, különösen a magas rendelkezésre állású mikroszolgáltatás-architektúrákban, hogy egyensúlyt teremtsenek a teljesítmény és a kezelhetőség között.

References

[1] "Introduction - blackant." https://dev.blackant.app:9002/introduction/. (Accessed on 03/28/2024).

[2] "Pipeline." https://www.jenkins.io/doc/book/pipeline/. (Accessed on 04/25/2024).

[3] "realworld-pipeline-flow.png (1322×499)." https://www.jenkins.io/doc/book/resources/pipeline/realworld-pipeline-flow.png. (Accessed on 04/25/2024).

[4] "Blue ocean." https://www.jenkins.io/doc/book/blueocean/. (Accessed on 04/25/2024).

[5] "editor-1.png (1095×721)." https://www.jenkins.io/images/blueocean/editor-1.png. (Accessed on 04/25/2024).

[6] "semi-complex-pipeline.png (1000×237)." https://raw.githubusercontent.com/jenkinsci/pipeline-graph-view-plugin/main/docs/images/semi-complex-pipeline.png. (Accessed on 04/25/2024).

[7] "What is kubesphere." https://v3-1.docs.kubesphere.io/docs/introduction/what-is-kubesphere/. (Accessed on 04/25/2024).

[8] "image15.png (1324×802)." https://cdn-ilcajnh.nitrocdn.com/wREzyOgRjKTvogDiDzGmbmBSDhJQxYLj/assets/images/optimized/rev-50e05c4/camunda.com/wp-content/uploads/2024/06/image15.png. (Accessed on 10/24/2024).

[9] "What is camunda 8? | camunda 8 docs." https://docs.camunda.io/docs/components/concepts/what-is-camunda-8/. (Accessed on 10/24/2024).

[10] "Introduction to camunda 8 | camunda 8 docs." https://docs.camunda.io/docs/guides/. (Accessed on 10/24/2024).

[11] "Business process model and notation (bpmn), version 2.0." https://www.omg.org/spec/BPMN/2.0.2/PDF. (Accessed on 10/23/2024).

[12] J. Hsu, C. Hsu, S. Chen, *et al.*, "Correlation aware technique for sql to nosql transformation," in *2014 7th International Conference on Ubi-Media Computing and Workshops*, (Ulaanbaatar, Mongolia), pp. 43–46, Institute of Electrical and Electronics Engineers Inc., 2014.

[13] "Role-of-adjacency-matrix-adjacency-list-in-graph-theory.pdf." https://www.researchgate.net/profile/Harmanjit-Singh-2/publication/274362141_Role_of_Adjacency_Matrix_Adjacency_List_in_Graph_Theory/links/551be5540cf2fe6cbf75f9c7/Role-of-Adjacency-Matrix-Adjacency-List-in-Graph-Theory.pdf. (Accessed on 04/09/2024).

[14] "268857bd-bb32-4fa5-88c9-66d7787952e9.png (580×315)." https://www.oreilly.com/api/v2/epubs/9781788623872/files/assets/268857bd-bb32-4fa5-88c9-66d7787952e9.png. (Accessed on 04/25/2024).

[15] "Graph database concepts - getting started." https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/. (Accessed on 04/10/2024).

[16] "neo4j.com/docs/getting-started/_images/graph_simple-arr.svg." https://neo4j.com/docs/getting-started/_images/graph_simple-arr.svg. (Accessed on 04/25/2024).

[17] "neo4j.com/docs/getting-started/_images/cypher_learning_arr.svg." https://neo4j.com/docs/getting-started/_images/cypher_learning_arr.svg. (Accessed on 04/25/2024).

[18] "Query a neo4j database using cypher - getting started." https://neo4j.com/docs/getting-started/cypher-intro/. (Accessed on 04/10/2024).

[19] "Using a jenkinsfile." https://www.jenkins.io/doc/book/pipeline/jenkinsfile/. (Accessed on 04/25/2024).

[20] J. Lloyd, "Practical advantages of declarative programming," in *Unknown*, pp. 3 – 17, 1994. Conference Proceedings/Title of Journal: Joint Conference on Declarative Programming.

[21] "Introduction | kubeflow." https://www.kubeflow.org/docs/components/pipelines/v2/introduction/. (Accessed on 05/09/2024).

[22] "Pipeline basics | kubeflow." https://www.kubeflow.org/docs/components/pipelines/v2/pipelines/pipeline-basics/. (Accessed on 05/09/2024).

[23] "Welcome to flask — flask documentation (3.1.x)." https://flask.palletsprojects.com/en/stable/. (Accessed on 12/02/2024).

[24] "Flask-restful — flask-restful 0.3.10 documentation." https://flask-restful.readthedocs.io/en/latest/index.html. (Accessed on 12/02/2024).

[25] "Protocol buffers documentation." https://protobuf.dev/. (Accessed on 10/31/2024).

[26] "camunda/zeebe/gateway-protocol/src/main/proto/gateway.proto at main · camunda/camunda · github." https://github.com/camunda/camunda/blob/main/zeebe/gateway-protocol/src/main/proto/gateway.proto. (Accessed on 12/02/2024).

[27] "Elasticsearch: The official distributed search & analytics engine | elastic." https://www.elastic.co/elasticsearch. (Accessed on 11/14/2024).

[28] "What is elasticsearch? | elasticsearch guide [8.16] | elastic." https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro-what-is-es.html. (Accessed on 11/14/2024).

[29] "Introduction to mongodb - mongodb manual v8.0." https://www.mongodb.com/docs/manual/introduction/. (Accessed on 11/14/2024).

[30] "Mongodb pymongo documentation - pymongo v4.10." https://www.mongodb.com/docs/languages/python/pymongo-driver/current/. (Accessed on 12/08/2024).

[31] "Quickstart — flask-restful 0.3.10 documentation." https://flask-restful.readthedocs.io/en/latest/quickstart.html#resourceful-routing. (Accessed on 12/08/2024).

[32] "3. data model — python 3.13.1 documentation." https://docs.python.org/3/reference/datamodel.html#context-managers. (Accessed on 12/09/2024).

[33] "Swarm mode | docker docs." https://docs.docker.com/engine/swarm/. (Accessed on 12/02/2024).

[34] "What is ci/cd? | gitlab." https://about.gitlab.com/topics/ci-cd/. (Accessed on 12/04/2024).

[35] "Get started with gitlab ci/cd | gitlab." https://docs.gitlab.com/ee/ci/. (Accessed on 12/04/2024).

[36] "high_arch.png (1018×355)." https://dev.blackant.app/docs/images/high_arch.png. (Accessed on 12/04/2024).

Images

Acronyms

DSL Domain Specific Language

KFP Kubeflow Pipeline

CD Continuous Delivery

CI Continuous Integration

API Application Programming Interface

REST Representational State Transfer

RBAC Role Based Access Controll

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

UI User Interface

BPMN Business Process Model and Notation

DMN Decision Model and Notation

OMG Object Management Group

WSBPEL Web Services Business Process Execution Language

XML eXtensible Markup Language

DAG Directed Acyclic Graph

RDBMS Relational Database Management System

SQL Structured Query Language

ASCII American Standard Code for Information Interchange

CRUD create, read, update, and delete

JSON JavaScript Object Notation

YAML YAML Ain't Markup Language

ML Machine Learning

SDK Software Development Kit

WSGI Web Server Gateway Interface

gRPC google Remote Procedure Call

SIEM Security information and event management

CI/CD Continuous Integration/Continuous Delivery

URL Uniform Resource Locator

TLS Transport Layer Security

SSL Secure Sockets Layer

CLI Command Line Interface