

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Towards a framework for a development environment based on
Kubernetes in the context of the FactCheck project

verfasst von | submitted by
Lukas Greiner BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 066 921

Studienrichtung lt. Studienblatt | Degree
programme as it appears on the student
record sheet:

Masterstudium Informatik

Betreut von | Supervisor:

Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas

Acknowledgements

First of all, I would like to express my deep gratitude for the opportunity to attend this university and, in particular, this institute, as it has provided me with an exceptional environment to advance my understanding and skills in computer science.

First and foremost, I would like to express my deepest gratitude to my thesis supervisors Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas for his continuous expertise, guidance, and patience that have been invaluable throughout this research project. His constructive feedback and insightful suggestions helped shape this thesis and significantly contributed to its completion. Also, I would like to thank Adrian Hofer, BSc MSc. for his technical support, especially regarding the FactCheck software projects, hardware setup, and Azure cloud infrastructure setup.

I would also like to thank Ingeborg Sziederits and Renate Mercsanits for their careful proofreading and valuable suggestions. Their attention to detail and thoughtful feedback greatly contributed to the quality of this work.

And, of course, I would like to thank my family, my friends, and all the people I had the pleasure to meet and learn from throughout these years. Especially, I would like to thank my late father, Manfred Greiner, who sadly passed away in the first semester of my bachelor studies, without whom I would not have been able to study here at this institute and in this city.

Abstract

This thesis addresses the need for a comprehensive development environment for software developers, especially for student developers. This endeavor not only requires the setup of a dedicated development server but also the introduction of deployment processes, configuration management, production data replication, cloud integration, and evaluation of the deployability of the current project. Although these requirements could be achieved by a single developer, an enormous amount of time and effort would divert significantly from active software development.

This thesis aims to address these issues in the context of the FactCheck project by providing the necessary software, a Kubernetes-based deployment infrastructure, and deployability evaluation metrics. Given that most FactCheck projects are implemented in Python, direct support for creating already compatible Python projects is provided, which in the best-case scenario can be deployed in one click. With this framework in place, developers that work on the FactCheck project should be enabled to quickly deploy their software, set up their development environment, and improve their compatibility with the production environment.

Targeting student developers, this framework is tailored to a university environment, providing each student developer with a dedicated environment that prevents interference from others. Each of these environments also allows access to web resources via the internal network. To allow this in a secure manner, HTTPS communication and basic authentication methods are automatically provided. With these features in place, student developers are enabled to develop higher quality and production-ready software.

Furthermore, the deployability metrics provided make it possible to evaluate the current deployability. These make it possible to compare and evaluate the progress of the software on the way to publication in the production system.

Kurzfassung

Diese Arbeit befasst sich mit dem Bedarf an einem umfassenden System zur Unterstützung von Softwareentwicklern, insbesondere von studentischen Softwareentwicklern. Dieses Unterfangen erfordert nicht nur die Einrichtung eines dedizierten Entwicklungsservers, sondern auch die Einführung von Deployment-Prozessen, Konfigurationsmanagement, Produktionsdatenreplikation, Cloud-Integration und die Evaluierung der Deployability des aktuellen Projekts. Obwohl diese Anforderungen von einem einzelnen Entwickler umgesetzt werden könnten, würde ein enormer Zeit- und Arbeitsaufwand von der aktiven Softwareentwicklung ablenken.

Diese Arbeit zielt darauf ab, diese Probleme im Kontext des FactCheck-Projekts zu behandeln, indem die notwendige Software, eine Kubernetes-basierte Deployment-Infrastruktur und Metriken zur Bewertung der Deployability bereitgestellt werden. Da die meisten FactCheck-Projekte in Python implementiert sind, wird direkte Unterstützung für die Erstellung bereits kompatibler Python-Projekte zur Verfügung gestellt, die im besten Fall mit einem Mausklick bereitgestellt werden können. Mit diesem Framework sollten Entwickler, die an dem FactCheck-Projekt arbeiten, in der Lage sein, ihre Software schnell zu veröffentlichen, ihre Entwicklungsumgebung einzurichten und ihre Kompatibilität mit der Produktivumgebung zu verbessern.

Dieses Framework ist auf eine Universitätsumgebung zugeschnitten und bietet studentischen Softwareentwicklern eine eigene Umgebung, die Interferenzen durch andere vermeidet. Jede dieser Umgebungen ermöglicht auch den Zugriff auf Webressourcen über das interne Netzwerk. Um dies auf sichere Weise zu ermöglichen, werden automatisch HTTPS-Kommunikation und grundlegende Authentifizierungsmethoden bereitgestellt. Mit diesen Funktionen sind die studentischen Entwickler in der Lage, qualitativ hochwertige und produktionsreife Software zu entwickeln.

Des Weiteren wird durch die bereitgestellten Deployability-Metriken die Möglichkeit zur Evaluierung der aktuellen Deployability bereitgestellt. Diese ermöglichen den Vergleich und die Auswertung des Fortschritts der Software auf dem Weg zur Veröffentlichung im Produktivsystem.

Contents

Acknowledgements	i
Abstract	iii
Kurzfassung	v
List of Tables	xiii
List of Figures	xv
1. Introduction	1
1.1. Motivation	1
1.2. Background and Context	2
1.2.1. Software Development Lifecycle	2
1.2.2. Software Development Workflows	3
1.2.3. The FactCheck Project	3
1.2.4. Python Software Development	4
1.2.5. Software Quality and Deployability	4
1.2.6. The Kubernetes Environment	5
1.2.7. The Azure Environment	6
1.3. Research Questions	7
1.4. Objectives of this Thesis	8
1.4.1. Standardized Development Projects and Processes	8
1.4.2. Adaptable and Intuitive Deployment Tools	8
1.4.3. Deployability Metrics	9
1.4.4. Synchronizing Data between Production and Development Environments	10
1.4.5. Authorization of Developers	11
1.4.6. Integrating with the Azure Environment	11
1.5. Topic Scope	12
1.5.1. Deployability	12
1.5.2. Development Environment	12
1.5.3. Project Standardization	13
1.5.4. Cloud Integration	13
1.6. Thesis Structure	13

2. Related Work	15
2.1. Deployment of Student Projects	15
2.1.1. Designing a PaaS solution for large-scale deployment of student projects	15
2.1.2. Designing and implementing a private cloud for student and faculty software projects	15
2.2. Software Setup and Architecture	16
2.2.1. Using DevContainers to Standardize Student Development Environments: An Experience Report	16
2.2.2. Realization of best practices in software engineering and scientific writing through ready-to-use project skeletons	16
2.3. Software Deployability	17
2.3.1. Designing software architecture to support continuous delivery and DevOps: A systematic literature review	17
2.4. Evaluation and Relevance	19
3. Conceptional Foundation and System Design	21
3.1. Framework Design	21
3.1.1. Overall Structure and Deployment	21
3.1.2. FactDeploy	23
3.1.3. Docker Registry	23
3.1.4. FactCheck-Config-Service	24
3.1.5. Kubernetes Environment	24
3.1.6. Azure Environment	24
3.1.7. Data Replication Service	25
3.2. System Design	25
3.2.1. Project Structure	25
3.2.2. User Interface	26
3.2.3. Storage and Data Manipulation	27
3.3. Dynamic Local Development Environment	27
3.3.1. Docker Containers and Images	27
3.3.2. Docker Compose	28
3.4. Dynamic Project Creation	29
3.4.1. Project Structuring	29
3.4.2. Modular Definition	30
3.5. Dynamic Dockerfile Creation	30
3.6. Configuration Management	30
3.6.1. Server-Side Application for Configuration File Storage	31
3.6.2. Client-Side Integration for retrieving Configuration Files	32
3.7. Building and Pushing Images to a Registry	33
3.7.1. Version Upgrade	33
3.7.2. Building Process	34
3.7.3. Pushing Process	34

3.8. Deployment to Kubernetes	34
3.8.1. Software Deployment	34
3.8.2. Role-Based Authentication	35
3.8.3. Client Side Access	35
3.8.4. Templating	36
3.8.5. Docker Registry Access	37
3.8.6. External Access	37
3.8.7. Viewing Deployments	38
3.9. Quick Deploy Process	39
3.10. Data Replication Process	39
3.10.1. Architectural Choices	39
3.10.2. Efficient Replication	40
3.11. Integrating Azure Services	40
3.11.1. Migrating to Azure Kubernetes Service	40
3.11.2. Connecting to Azure	41
3.11.3. Connecting two Kubernetes nodes in a hybrid environment	42
4. Implementation	43
4.1. Technology Stack and Environments	43
4.1.1. FactDeploy	43
4.1.2. Factcheck-Config-Service	44
4.1.3. User Generator	45
4.1.4. FactCheck Backend	45
4.1.5. FactCheck Factbase	45
4.1.6. FactCheck Factserver	45
4.2. FactDeploy	46
4.2.1. Project Architecture	46
4.2.2. System Layer	47
4.2.3. Project Generation	50
4.2.4. Docker Compose Generation	53
4.2.5. Dockerfile Generation	57
4.2.6. Configuration Upload	60
4.2.7. Building and Pushing Docker Images	63
4.2.8. Kubernetes Deployment	66
4.2.9. Kubernetes Viewer	71
4.2.10. Quick Deploy	72
4.3. FactCheck Config Service	75
4.3.1. REST Endpoints	76
4.3.2. Data Management	76
4.3.3. Deployment and External Access	77
4.4. FactCheck Services	78
4.4.1. Changes made to all FactCheck services	78
4.4.2. FactCheck Backend	79

4.4.3. FactCheck Factserver	79
4.4.4. FactCheck FactBase	79
4.5. User Generator	79
4.5.1. Usage	80
4.5.2. User Creation	80
4.5.3. Kubernetes User Setup	80
4.5.4. User Deletion	81
4.5.5. User Certificate Renewal	82
4.6. Development Container	82
4.7. Private Docker Registry	83
4.8. Kubernetes	84
4.8.1. Setup	84
4.8.2. External access to services	85
4.9. Production Data Replication	87
4.9.1. Configuring Replication Processes	88
4.9.2. Adapting to Environmental Constraints	88
4.9.3. Replication Workflow	89
4.9.4. Deployment	90
4.10. Microsoft Azure	90
4.10.1. Creating a development environment in Azure	90
4.10.2. Adaptation to FactDeploy	91
4.10.3. Connecting on-premise infrastructure to Azure	92
5. Evaluating Deployments	95
5.1. Background and Goals	95
5.2. Defining Deployability	95
5.2.1. Requirements	95
5.2.2. Scope	96
5.3. Metric Definitions	96
5.3.1. Structure	96
5.3.2. Symbols	97
5.3.3. Phrases	98
5.3.4. Impact	98
5.4. Actors	100
5.4.1. Main Actors	100
5.4.2. Supporting Actors	100
5.4.3. Supplementary Actors	101
5.5. Environment Constraints	101
5.6. The Deployability Metric	103
5.6.1. Main Actors	103
5.6.2. Supporting Actors	112
5.6.3. Supplementary Actors	128
5.7. Usage	133

6. Methodology	135
6.1. Prototypical Implementation	135
6.1.1. Requirement Choices	135
6.1.2. Development Methods	135
6.1.3. Architectural and Technological Choices	135
6.2. Deployability Metric	136
6.2.1. Methods for choosing features and best practices	136
6.2.2. Writing Tools	137
7. Conclusion	139
7.1. Results	139
7.2. Discussion and Limitations	140
7.2.1. Evaluation of FactDeploy	140
7.2.2. Evaluation of Auxiliary Services	140
7.2.3. Evaluation of the Hybrid Cloud Architecture	141
7.2.4. Evaluation of the Deployability Metric	141
7.3. Future Work	142
Bibliography	145
Acronyms	150
A. Appendix	151
A.1. Source Code	151
A.2. FactDeploy	151
A.2.1. Prerequisites	151
A.2.2. Install	151
A.2.3. Quick Start	152
A.2.4. Description	152
A.2.5. PyCharm Quirk	152
A.3. Main Features	153
A.3.1. Quick Deploy	153
A.3.2. Project Generation	153
A.3.3. Configuration Upload	154
A.3.4. Dockerfile Generation	154
A.3.5. Docker-Compose Generation	155
A.3.6. Docker Build and Push	155
A.3.7. Kubernetes Deployment	156
A.4. Additional Features	156
A.4.1. Kubernetes Viewer	156
A.4.2. Configuration Backup, Deletion, and Restore	157
A.5. Useful Kubernetes Scripts	157
A.5.1. On-Premise Kubernetes Installations	157
A.5.2. Azure Kubernetes Services	158

Contents

A.5.3. Azure Arc Kubernetes	159
A.6. Known Issues	159
A.6.1. Private Link Connection	159
A.6.2. Possible DNS Issues	159
A.7. User Generator	160
A.7.1. Installation	160
A.7.2. Usage	161
A.8. FactCheck Config Service	161
A.8.1. Installation	161
A.8.2. Building and Pushing to a Docker Registry	162
A.8.3. Usage	162
A.9. FactCheck Development Container	162
A.9.1. Usage	163
A.10. Docker Registry	163
A.10.1. Installation	163
A.10.2. Usage	164
A.10.3. Recommended:	164
A.11. FactCheck Sync Service	164
A.11.1. Prerequisites	164
A.11.2. Local Setup	164
A.11.3. Kubernetes Setup	164
A.12. Factcheck VPN Relay	165
A.12.1. Building a Docker Image of the VPN Relay	165
A.12.2. Running the VPN	165
A.12.3. Check if its working	165

List of Tables

5.1. Illustration of the layer structure in the deployability metric.	97
5.2. Software Deployment Definitions - Dockerfile Definition	104
5.3. Software Deployment Definitions - Kubernetes Definition	110
5.4. Software Deployment - Helm Structure Definition	112
5.5. Software Deployment - Docker Compose Definition	115
5.6. Software Deployment - Python Software Structure	118
5.7. Software Availability Features	119
5.8. Software Upgradability	122
5.9. Software Configuration Structure	124
5.10. Software Security	126
5.11. Metrics Aggregation	128
5.12. Software Availability	130
5.13. Software Management	132

List of Figures

3.1. Deployment diagram of the development environment.	22
3.2. Flowchart for Configuration Upload Mechanism	32
4.1. FactDeploy Architecture	46
4.2. Classes available in the system module.	48
4.3. Classes available in the project module.	52
4.4. Classes available in the Docker-Compose module.	55
4.5. Classes available in the Dockerfile module.	58
4.6. Classes available in the configuration-upload module.	62
4.7. Classes available in the push and build module.	65
4.8. Classes available in the Kubernetes module.	68
4.9. Classes used by the Quick Deploy feature.	73

1. Introduction

1.1. Motivation

A software engineering development environment is the foundation of every software project ever created. This includes the early beginnings of software development in general. In today's context, the development environment can be vastly different from project to project and developer to developer. These variations allow developers to efficiently craft software tailored to their specific needs and expand their area of expertise [Int25]. Although this is a noble approach in the field of software engineering, larger and overall more complex systems provide a particular challenge as these systems can consist of several different software products, several different programming languages, and even several different hardware products on which the system should run [Sin22]. Therefore, a development environment, in its current form, has to be customized to its specific platform and for its specific use case.

This complexity and the wide variety of systems, with all their advantages and disadvantages, allow advanced users, particularly developers, to contribute their work to their projects via their respective development environments. Although contributing to such a project is relatively easy, each contribution should be based on an understanding of the current project state and, in the best case, the stakeholders, the environments, and the systems around it. Furthermore contributions can and shall change the current behavior of the project which can, due to human error or unforeseen technical limitations, leave the system in a non-working, desolate or in the worst case completely destroyed state [MGvdHW10]. Even though these states should be relatively rare and should be avoided at all costs, they will eventually happen. These kinds of states are to a greater degree found in student projects, which happen due to their limited experience in software development and can be expected and/or even encouraged depending on the preferred learning method [AG20]. Several configurations, deployments, and software adaptations are necessary to transform these projects into a production system-ready or publicly accessible state.

Managing vastly different student projects in various states of completion, which may interact with other software and may be in erroneous states, is a non-trivial task. These characteristics lead to the need to develop a framework for a development environment that respects these states, interactions, and different operating environments while enhancing the overall quality of the software and bringing these projects closer to the goal of being part of a production system and/or to be publicly available.

1. Introduction

1.2. Background and Context

The following section provides various insights into each concept touched on in this thesis and is aimed at understanding and clarifying the environment and assumptions used in its development.

1.2.1. Software Development Lifecycle

The software development lifecycle has evolved since the first programs were created. The following stages are used regularly in modern development methods and can be applied in a variety of ways while developing a software product [Lem18].

1. Planning: This involves defining the project goals, identifying the requirements and determining the scope of the project.
2. Design: This involves creating a detailed plan for the software's architecture, including its components, interfaces and data flows.
3. Development: This involves writing the code, creating the user interface and testing the software to ensure that it meets the requirements.
4. Testing: This involves running various tests to ensure that the software is free of bugs and meets performance requirements.
5. Deployment: This involves deploying the software to the production environment, where it can be used by potential users.
6. Maintenance: This involves fixing bugs, updating the software and providing support to users.

Each stage may differ depending on the development technique used, such as agile or waterfall development, and can be tailored to the demands of a specific project [Lem18]. The main focus of this thesis lies on the development, testing, and deployment stages, which are shared by both the agile and waterfall development methods. Although these are distinct stages, there has been a recent push to narrow the gap between them in order to boost productivity, detect defects faster and provide software to users in a more reliable manner. According to [Ser21], in an ideal case, this would result in NoOps, where the operation and development actions would not overlap. Before this concept came into existence, each of these stages would have its own given time frame and dedicated team. This resulted in long development, testing, and deployment stages, even in agile projects.

One of the main influencing factors in this thesis is the combination of these stages and the potential results of increased productivity and defect detection, since a well-defined software development framework must provide the necessary support to achieve these goals. Furthermore, it provides a reliable ground for student projects to traverse each of these stages in a timely manner, which in the context of a university project, is essential for success and possibly even motivates each student to drive their work further.

1.2.2. Software Development Workflows

In the software development domain, workflows are structured methodologies that guide the creation of software applications. These workflows are a central point of the development process, ensuring that developers follow a systematic approach to building software solutions [Lem18]. Coding and testing are commonly done iteratively, allowing continuous improvement, bug fixing and therefore better software. Collaboration and version control systems are essential in this process as they allow multiple developers to work on the same project simultaneously while maintaining a common code base [PD22]. The development workflow ultimately culminates in the application's deployment, which publishes the software to its intended users.

All of these previously described steps are common, but their implementation depends on the software development team, the development environment and especially the developers themselves [PD22]. These steps are often focused differently in the context of a university project, mostly due to time constraints and the project's aim. In projects that focus mainly on developing software and presenting the results to a limited audience, other parts are at risk of being unused and/or implemented in a limited fashion [HRJ21].

This shift in focus and the surrounding limitations in university projects demonstrate that students need the necessary tools to be able to focus on their main components in their projects while still being able to ensure that supporting components are not falling behind [HRJ21]. The lack of these tools, in conjunction with the FactCheck project, has been a major influence on the software developed in conjunction with this thesis.

1.2.3. The FactCheck Project

The FactCheck Project, introduced and initially developed by Dipl.-Ing. Dr. Peter Kalchgruber [Kal18], is the main project in collaboration with this thesis. Currently, several student developers are actively developing and expanding the FactCheck Project, adding various features across different fields. Its primary goal is to enable users to compare, analyze and verify information on the web using (un)structured data.

The project itself is based on several components, each of which has its own use case. Currently, the FactCheck Project comprises a number of software components developed by a number of separate developers. Its fundamental architecture consists of the "IdaFix", "Factserver", "Factbase" and "Backend" projects, whereby the first refers to the browser add-on and the other three are backend services based on various versions of Python. Several other services are available, which include a feedback framework and an analytics dashboard. Furthermore, data is also stored in numerous databases, the majority of which are based on MySQL and CouchDB. All of these various software components, in various versions, are currently deployed manually to the Azure cloud environment or on servers of the University of Vienna. From this collection of services, databases and environments, a highly interconnected microservice architecture is identified, where services depend on each other's availability.

1. Introduction

As several student developers are and will be working on further projects related to the FactCheck project, testing and deployment strategies relying on production data and other services require a highly coordinated effort to prevent data corruption and reduce costs for running cloud services. Such requirements have a crucial influence on the implementation of a framework for a development environment, as they form the foundation of the final product. Furthermore, collaboration with the FactCheck project was essential as these requirements and environments exist, which show direct practical usage of the developed software.

1.2.4. Python Software Development

In the software development community, the saying "using the right tool to do the right job" also applies to the choice of programming language. One of these tools is the Python programming language which has its own feature set and applicability, as is the case with all programming languages. Throughout its history, Python has gained popularity for its simplicity, readability and versatility. It has been widely adopted in various fields, including web development, data science, artificial intelligence and scientific research [Ame24]. Due to the large and active community that has developed over time and contributed to an extensive ecosystem of libraries and frameworks, Python has solidified its position as one of the most popular programming languages in the world [Ove24]. This cross-disciplinary appeal makes it an ideal choice for scientific software projects. The strength of its ecosystem lies in data analysis, visualization and machine learning ensure that Python is at the forefront of scientific research and innovation [Ame24]. Furthermore, due to its simplicity, it has a history of scripting and, therefore, a fast-paced development environment.

These characteristics are essential for exploring the possibilities in the realm of adaptable development environments, which rely on their versatility, simplicity and interactivity to allow fast-paced development in combination with cross-platform executability and advanced user interaction. Furthermore, the collaboration with the FactCheck framework, which has several services based on Python, allows for a consistent technological stack and easier interaction and integration where necessary.

Python provides flexibility in initializing and structuring software projects. Therefore, defining and adhering to best practices during the development phase is crucial for minimizing potential errors and ensuring a high deployability of a project. Furthermore, the application of these structures and metrics allows student developers to evaluate their software, refactor it and employ new technologies to move forward on their path to deployment in a production environment.

1.2.5. Software Quality and Deployability

The evaluation of the quality of the software is, although it can be measured by several metrics, always up to the current developer to interpret the code and the overall structure

[MNM20]. Additionally, the user of the software might have their own view on the quality of the software, however well it is coded. This kind of person-specific view on software quality, although valid, does not ensure that the result of a project always conforms to its expectations in terms of software quality [SMFS⁺16]. The same argument can be made for the deployability of a software product, with several options for an ideal deployment, configuration, or environment available, where the ideal selection is ultimately up to the person deploying the application. Certainly, automated tools, experience, clear documentation and straightforward APIs can make software development and deployment easier, but still the characteristics of each software product can influence the applied development and deployment method [BD19]. To keep all these different opinions, options and tools in check, the software development community introduced several examples, predefined structures and tutorials, all contained under the term "best practices" [Bar12]. These practices should, in most cases, help developers make an informed decision on the possible solutions to their occurring problem. Although this is a novel idea, to identify if it is a best practice, the knowledge on where to find such practices, how to apply them and how to filter them for applicability is not always available. More often than not, they are scattered or behind a paywall which hinders their accessibility to other developers.

Finding a way to collect and set all these best practices in relation to a complete overview is an enormous endeavor. As this endeavor is far out of the scope of this thesis, the selection of best practices is limited to software development in Python, Kubernetes deployments and cloud computing patterns. These best practices are essential to the implementation of the developed software and are one of the important resources to define the deployability metric in later chapters.

1.2.6. The Kubernetes Environment

The Kubernetes project, which is often abbreviated as K8s, is an open-source container orchestration platform that has, as described by Carrión [Car22], gained widespread adoption in the world of cloud computing and containerization. Initially, it was developed by Google and is now maintained and improved by the Cloud Native Computing Foundation. It provides a scalable, portable and extensible framework for automating the deployment, scaling and management of containerized applications. Such features allow that software to be scaled horizontally by providing more computing units instead of faster computing units. For user authorization an RBAC (Role-Based Access Control) system is available, which allows administrators to define the necessary access to resources and functionality for users based on roles. This results in the option to completely separate users from each other, as done by Kjorvezirski et al. [KFMJ18], while also providing flexibility for shared access between users. Additionally, advanced networking capabilities via so-called ingresses and load balancers can be provided. Such networking capabilities also allow for TLS termination, in such a way that traffic from and to the cluster can be securely done via, e.g., HTTPS and internal communication can be based on HTTP.

Whereas, a Kubernetes instance can be deployed on the Azure cloud, allowing for easy

1. Introduction

switching between an on-premises and a cloud solution. This kind of portability is made possible since most components in Kubernetes are cloud agnostic and can therefore be used to deploy components without prior knowledge on underlying cloud platform [GB23]. This also creates the possibility of letting developers directly deploy to a development or test system and still use the same architecture as on the live system.

Kubernetes is currently not used in conjunction with the FactCheck project, but it is seen as a well-established on-premise solution for deployments and an adequate alternative for deploying to the Azure cloud platform. Although this system is fairly popular, using it for independent student projects as a development environment where separate but also collaborative work should be possible is explored in this thesis, as well as to bring advanced deployment workflows to student developers in a simplified manner.

1.2.7. The Azure Environment

Providing cloud computing infrastructure is a non-trivial task and is therefore more often than not provided by companies with an extensive background in software and hardware production and financial security. In the case of Azure, which is a cloud computing platform provided by Microsoft, these prerequisites are given. It offers a robust and reliable environment for production workloads and provides a wide range of cloud services and features essential to providing highly available and highly scalable software [Eku25]. Furthermore, services that allow for hybrid cloud integration and connection with external cloud infrastructure, such as the previously mentioned Kubernetes environment, are available [DB22], and are essential for this thesis.

The introduction of a hybrid cloud system in combination with a development framework stems from the idea that students should have the possibility to evaluate their software on an almost production-like system. Such a system, when derived from a cloud environment such as Azure, needs to support cloud services and allow access to production data. Therefore, the integration of cloud services as well as one-way data synchronization methods is essential for the successful operation of such a development environment.

1.3. Research Questions

After introducing the proposed environments and workflows, the central question of "How can the creation of software and the subsequent deployment on Kubernetes be abstracted for computer science students to utilize its essential features, even if they have no prior experience with Kubernetes and the FactCheck production environment?" still stands. To successfully answer this question, there are several sub-questions that have to be answered:

1. How can students develop their software while ensuring that their work is compatible with the production environment? To avoid compatibility issues and related errors, developing and testing software on a local system should have almost no impact on deploying it on a production system.
2. What are the current challenges faced by students when deploying applications to the Azure environment, and can they be fixed by deploying to Kubernetes? Deployments to a development system are a crucial part of software development and need, especially for student developers, to be executed in a stable, reliable and easy-to-use way as deployments and tests happen regularly.
3. How can student developers check if their software is ready for a deployment? Evaluating the current state of their software product for production or deployment readiness is a challenging task that requires prior knowledge and/or experience.
4. How can production data be partially synchronized to the development environments while facing data size and performance limitations? As students develop in their development environment, they need to be able to test their implementation against a production dataset to ensure maximal compatibility.
5. How can students be authorized to deploy their own software without interfering with other students data or software? Development systems commonly rely on a single software set or data set, which can change but should stay stable due to upgrade mechanisms throughout the development process. As this cannot be required for student projects, the complete separation of student environments is a necessity.
6. How can the current infrastructure on Azure be partially reused to combine the advantages of an on-premise and a cloud-based environment? Combining the flexibility of Azure products with the advantages of an on-premise development system allows for a production-like environment even in the development phase and furthermore improves stability.

1. Introduction

1.4. Objectives of this Thesis

To successfully answer the previously mentioned sub-questions and then create an appropriate framework, several objectives have to be accomplished. These objectives and their requirements for a successful implementation are described in the following sections.

1.4.1. Standardized Development Projects and Processes

Standardization has long been acknowledged as a crucial element in the development of production-ready software, including software development projects and processes [VGP20]. Some of the FactCheck core projects are based on the Django framework which provides a basic standardized project structure that can be highly customized. However, there are also projects in FactCheck that do not follow the Django framework and as a result, their structures can vary widely, which is a valid approach, as not every project requires the feature set or it can even be based on a completely different framework. This customizability from project to project allows students to be flexible, but it also increases the complexity of the automation process, making it difficult for external services to configure software based on the environment it runs in.

This objective therefore aims to resolve these challenges and enables students to employ any framework, or even plain Python, via a clear standardized base project for both current and future iterations of the FactCheck program. This requires the fulfillment of the following sub-objectives:

- The folder and file structure of the software have to follow a minimal, predefined layout.
- Essential endpoints for detecting the software’s liveliness and readiness have to be provided.
- The software configuration must be changeable from outside the software and automatically applied upon restart.
- A single predefined definition of the software’s dependencies must be available.
- The operating system and its Python version have to be predefined and checked on startup.

By successfully implementing these sub-objectives, the portability and compatibility of the software can be enhanced, thereby reducing environment-based and developmental errors.

1.4.2. Adaptable and Intuitive Deployment Tools

Customized deployment tools tailored for specific workflows are common in the software development industry, as each system has its own unique requirements. According to Shahin et al. [SBZ17], user friendliness and intuitiveness are often overlooked in these tools,

especially those designed for professional usage. In the best-case scenario a deployment to a development system should be a single-click action to test software quickly and timely. This goal is not often achieved, as constraints set by the underlying processes, such as CI/CD pipelines, verification of pull requests, executed unit tests and even company politics, can hinder the deployment process. Therefore, it is worthwhile to explore the possibility of achieving a fast-paced development and deployment cycle, as the reduction of overhead can enhance software quality by reducing downtime until developers can test their software and conduct direct testing in a production-like environment.

To achieve the previously described objective for this thesis, such as a one-click deployment tool, several requirements have to be met:

- Each project has to have all its necessary deployment information pre-configured and stored in its folder structure. If no such structure exists, a minimally standardized deployment structure should be generated.
- To configure the necessary custom environmental data the deployment information must be customizable via the deployment tool and each input should be stored for future deployments.
- The deployment tool should be included via a library, a project plugin or an external tool.
- Environmental-based configuration files should be automatically detected, uploaded and then replaced with the local configuration file before deployment.
- Generated deployments should also be automatically applied to the Kubernetes environment and replace previous versions if necessary.
- The finally deployed image should be as small as possible as storage capacity is limited.

When implementing these requirements, a quick and non-bureaucratic deployment procedure should be the result, which enables developers to carry out rapid tests in the development environment while still maintaining stability in the production environment.

1.4.3. Deployability Metrics

When first implementing a software product the traditional approach is to implement the necessary functionality and features and when deemed ready for production use, to implement the necessary deployment methodology [Lem18]. Such an approach leads to the possibility of production-breaking issues after the development phase which, in turn, leads to longer or repeated development phases and deployment delays. Therefore, the introduction of continuous deployments and continuous integration while developing software has been essential in modern software development [SBZ17]. To achieve continuous deployment, the exact point for the first deployment has to be identified and also the

1. Introduction

exact point for a production deployment. Identifying these points, especially for student developers with limited prior experience, is not easy.

As a result of these undefined or vague deployment points, one objective of this thesis is to introduce a deployability metric, which should let the developer identify the point when their software is ready for a development or production deployment. Furthermore, as a metric, it should allow the developer to identify the capacity in which it is ready for deployment. To achieve this objective, the following requirements have to be met:

- A list of best practices with the necessary steps and paths for Python deployments on Kubernetes has to be defined.
- The list of best practices has to be split into development, production and additional steps that aim to identify which steps are necessary for the relevant phase.
- Each step and best practice must be associated with a value that defines the impact on the deployment.
- Steps and each best practice can have additional markers that will help to define if they are necessary for the deployment or optional.
- The final metric should be calculated using a formula that accounts for the impact of each step or practice and results in a value, or multiple values, that represent the software's deployability.

By identifying and creating such a deployability metric the developer should be able to identify and calculate if their current project state is viable for a deployment on the respective system.

1.4.4. Synchronizing Data between Production and Development Environments

In the FactCheck project the current production data is stored in a database that can be accessed by all students who have access to the Azure production environment. This sharing of a single database could potentially lead to a student changing or manipulating data, which can be malformed and therefore break other services. Furthermore, changes in the data model can lead to versioning problems when several students are working with the same dataset. These challenges necessitate the following steps to be taken to implement a well-functioning database synchronization tool:

- A subset or even the complete dataset should be manually or automatically synchronized to a development environment.
- Local data should be continuously updated to reflect current changes in the production dataset.
- A reset mechanism should be available to return to the last downloaded dataset.

Allowing students to always work with production data locally leads to an improvement in productivity and reduces the risk of data loss on the production system.

1.4.5. Authorization of Developers

Authorization for users, particularly in a university setting, can cause a range of problems. These include credential storage, resource control, role administration and certificate management [Aut25b]. Furthermore, in a per-user-separated environment, which should also provide the possibility to collaborate when necessary, the problem of authorized resource sharing arises. As a result, an authorization concept should be implemented that meets the following conditions:

- If possible, users should be authorized and created by an external service, such as the university's LDAP server. Alternatively, the "Basic Authentication" method should be utilized.
- A default set of Kubernetes resources should be available for users and administrators should be enabled to extend this resource set.
- Confidential data should not be directly stored on the system or at least be encrypted and stored securely.
- Accessing resources via the web should only be possible with valid credentials.
- Administrators must be able to manually allow resource sharing between users.

Ensuring appropriate authorization, especially for the FactCheck project and a university-based infrastructure, is a necessity, since data integrity and non-interference in student projects are essential.

1.4.6. Integrating with the Azure Environment

The FactCheck project's current production environment is primarily based on a manually installed architecture on the Azure cloud and university on-premise servers. This is not ideal for development purposes due to the fact that costs rise with an increasing number of developers and students. As a result, the heavy objects are moved to an on-premises infrastructure, which in turn means that many of the cloud provider's capabilities are no longer available. Elastic load balancing, data storage capacity, DNS resolution and other functions are some examples [Eku25]. Even when subjected to severe loads, many of these are still useful or required for a functional system. To ensure that these features remain available, the following procedures must be followed:

- The currently used cloud services requiring migration have to be identified.
- Conceptualizing ways to replace existing services with on-premise resources.
- Integrating non-replaceable services for a hybrid cloud approach.

1. Introduction

In accordance with these steps, the FactCheck project can ensure that the features provided by the cloud provider are still accessible in the on-premise environment, allowing for a functional and reliable system with a similar feature set. This approach, which is commonly called hybrid cloud, can therefore enable the interaction between cloud and on-premise infrastructure by providing features such as connecting databases and DNS resolution while reducing dependence on each environment.

1.5. Topic Scope

For this master thesis, the precise scope of research has to be defined, aiming to provide an overview of deployability, development environments, project standardization and cloud integration in the context of the FactCheck project. Since this topic encompasses many areas and many technologies, each of these topics has to be limited to specified subjects and technologies.

1.5.1. Deployability

Over the course of this thesis, the deployability of software projects will be defined, bearing in mind that the projects may be based on different technologies and subsequently an exact scope has to be defined. As most software components in the FactCheck project rely on Python as their primary programming language, the definition of deployability will focus on this language. Other languages within the scope of this thesis will not be supported, however future individuals can easily add them. The same rationale applies for using the Kubernetes deployment platform, given the multitude of diverse of deployment platforms available.

As the deployability metric is defined by a collection of best practices and their impact on the overall deployment of the software, its completeness cannot be guaranteed. This is due to the ever-changing environment, APIs and technologies used when deploying software. Therefore, it aims to use the most stable and commonly employed best practices. Applying these best practices is optional and can be extended or updated by individuals in the future.

1.5.2. Development Environment

The constraints for development environments are similar to the deployability constraints, as technology and its related best practices are subject to change and are developed further. Therefore, the use of development environments will only include, from a technical standpoint, support for the current Docker and Kubernetes versions. From a conceptualization standpoint only the creation of a local development environment in Docker and an environment instance via Kubernetes is elemental. Other development environments, such as IDEs, text editors, additional plugins and external software, are not part of this thesis due to time constraints and relevance concerns.

1.5.3. Project Standardization

As already mentioned in the deployability section, the main focus lies on Python projects. The scope for standardization in such projects can vary widely, as external dependencies, IDEs and others bring their own template for a project structure. As a result, the project standardization is limited to only plain Python projects and must include software quality and deployment-relevant dependencies, as well as minimal code examples for best practices. It should therefore be viewed as a basis for further development and customization.

1.5.4. Cloud Integration

The introduction of cloud services into the Kubernetes development environment should be limited by requiring that custom software should not be replaced or by imposing constraints on existing software. Therefore, the main priority lies in achieving integration with cloud services that are already used by the FactCheck project. Furthermore, the data synchronization software used in this thesis will only synchronize between the Kubernetes instance and the FactCheck production instance.

1.6. Thesis Structure

This thesis is structured in several chapters, each of which has its own topics to examine.

In Chapter [1](#) an introduction into the topics can be found in Section [1.2](#), the objectives in Section [1.4](#), the scope definition in Section [1.5](#) and the fundamental research questions in Section [1.3](#).

Related work done by others can be found in Chapter [2](#). For each of the main topics of this thesis, information on the current state and work done by other teams can be found. This includes the work done to provide a development environment for students in Section [2.1](#), related architectural works found in [2.2](#) and research done to define deployability found in Section [2.3](#). An evaluation of these topics can be found in Section [2.4](#).

The concept for the implementation and the overall system design can be found in Chapter [3](#). The overall structure and communication pattern of the framework can be found in Section [3.1](#). For a detailed overview of the design and core principles of the developed software modules, the Section [3.2](#) can be referenced. For detailed concepts for each feature and module developed in this thesis, the Sections [3.3](#) and [3.11](#) can be referenced.

A description of the implementation process for each of the previously mentioned concepts can be found in Chapter [4](#). The technologies used while implementing all software products in the context of this thesis can be found in Section [4.1](#). Regarding the centerpiece of the framework, which is the FactDeploy software, the implementation details can be found in Section [4.2](#). Adaptations done to support the Azure cloud can be found in Section [4.10](#). All other sections in this chapter contain detailed explanations of their implementation process and choices with regards to their topics.

1. Introduction

In Chapter 5, which is called "Evaluating Deployments", the metric itself can be found in Section 5.6. Furthermore, definition thereof can be found in Section 5.3. For the necessary background information the Section 5.1 can be referenced. Also the definition of actors in Section 5.4 and the application of the metric in Section 5.7 can be found in this chapter.

In Chapter 6, the "Methodology" chapter, the used methods and decisions regarding the circumstances of the prototypically implemented software can be found in Section 6.1. The choices made and methods with regard to the deployability metric can be found in the Section 6.2.

Last but not least, in Chapter 7, the results can be found in Section 7.1, the discussion and limitations in Section 7.2 and the possible future work required in relation to this thesis are described in Section 7.3.

2. Related Work

2.1. Deployment of Student Projects

2.1.1. Designing a PaaS solution for large-scale deployment of student projects

Kjorveziroski et al. [KFMJ18] conducted a highly relevant project that explores the introduction of a university internal platform for student project deployment. The main focus lies on providing an easily scalable and user-isolated platform. The choice of a basis for a platform was Kubernetes, as it provided the necessary customizability, scalability and high versatility. Furthermore, the use of Kubernetes is tailored to an academic environment and allows for a 'one-click' deployment, reducing administrative effort.

To implement such a platform, a Kubernetes cluster was deployed in combination with the networking plugins Flannel and Calico. For the container runtime, Docker Runtime was chosen for running images and MetalLB was chosen as a load balancer. To allow users to connect to the cluster, the Apache Guacamole project was used, which allows them to manage the software in their dedicated namespace via SSH. For users to deploy their applications, they mainly focused on providing predefined templates that allowed them to add customized data. These templates are intended to be defined by an administrator and customized by the user via a web API. It was concluded that the project could be extended by providing advanced monitoring, automatic IP provisioning and application exposure to the public web.

2.1.2. Designing and implementing a private cloud for student and faculty software projects

Another relevant project is that of Le Fevre et al. [LFK22], which aims to deploy a private cloud for student and faculty software projects. Their cloud implementation necessitated the installation of physical hardware, including networking infrastructure. On the hardware they installed the Ubuntu distribution as the operating system and KVM as a VM hypervisor. The project involved transforming physical machines into CloudStack agents. For deploying containers, they chose Kubernetes. This allowed for deploying applications across several systems while managing those from a central point. To deploy student projects, they directly use the "kubectrl" command in combination with deployment files, but they suggest using a CI/CD workflow in the final deployment. Moreover, interviews with students were conducted in order to evaluate which principles

2. Related Work

they needed to focus on. It was concluded that simplicity, usability and repeatability were more important than configurability. Overall, the main outcome was the implementation of a secure and scalable IaaS platform.

2.2. Software Setup and Architecture

2.2.1. Using DevContainers to Standardize Student Development Environments: An Experience Report

In their paper, Valstar et al. [VGP20] evaluated the use of development containers in a student developer environment. Additionally, they conducted a study with 416 students to evaluate their use of development containers in a C++ environment. While they previously deployed centralized Linux servers to which students could connect and develop their software based on accessibility, the slow update cycle, unavailable GUIs and network connectivity issues compelled them to look for alternatives. They chose to deploy projects via the DevContainer project and focus on development inside the Visual Studio Code IDE.

The study was conducted in the course of four programming assignments in C++, where they allowed students to choose either the provided Linux servers or DevContainers. The results overwhelmingly showed a preference for DevContainers, with 71% developing in DevContainers and overall 70% recommending them to students for the following term. They also examined why they gave these preferences and concluded that VSCode usage, local development and portability were major factors. They also identified issues with students' less powerful workstations and errors when setting up the environment. The final conclusion was that using DevContainers resulted in fewer setup issues, faster updates to their tooling and overall faster turnaround times for improvements to the programming assignments. Also, the overwhelming support by students resulted in future usage of DevContainers in their courses.

2.2.2. Realization of best practices in software engineering and scientific writing through ready-to-use project skeletons

Haider et al. [HRJ21] collected a non-exhaustive list of best practices in scientific software engineering and used project skeletons to comply with these principles in their paper. In their work, they identified several issues regarding current practices and the environment in which scientific software is written. Their main issue is that high-quality scientific software is rarely rewarded, resulting in software without well-written documentation or support for software design patterns. To counteract these developments, they propose the use of project skeletons that adhere to best practices. Their list of best practices includes the following recommendations:

- Project management: management of code via a version control system and the usage of a ticket system to track bugs and responsibilities.
- Coding style: the readability and consistency of coding style for other developers and the usage of static code analysis software.
- Independence: The usage of open-source file formats and dependencies and interoperability with the most common operating systems.
- Automation: Automation or repeatable tasks such as building, testing and deploying the software.
- Documentation: well-defined documentation at different levels for a better understanding of the bigger picture.
- Tests: Regular testing of modules and interactions may result in fewer errors and bugs and therefore increase code quality.
- Deployment: Distribution of software in a ready-to-use package for easier usage by larger communities.

With these best practices in place, several projects having implemented these were evaluated. This includes projects like "CleanCppProject" by Kracejic, "Python package template project" by Ioannides and "Bertha" by Riesch and Jirauschek. They proposed their own template called "Bertha-TeX" for LaTeX-related projects. They on top observed that the use of project skeletons improved the adoption of best practices in both software development and scientific writing. Their main point is that they lower the barrier to implementing these practices by minimizing the required knowledge and automating the setup steps for new projects.

2.3. Software Deployability

2.3.1. Designing software architecture to support continuous delivery and DevOps: A systematic literature review

As software architecture and continuous integration and continuous delivery (CI/CD) in relation to deployment are two of the main topics in this thesis, the conference paper created by Bolscher et al. [BD19] is a fundamental cornerstone. They evaluated 13 papers regarding these topics between 2015 and 2019. In these papers, they discovered 17 advantageous software architecture characteristics and ten obstacles which software products have to overcome so they can adopt a CI/CD process. They also found a high correlation between software architecture and DevOps and, therefore concluded the need for implementing CI/CD practices. Five of the reviewed papers explicitly highlighted the significance of software architecture in relation to these concepts. Regarding this application, Bass states that *"the architect is critical for success in adopting DevOps practices"* where Chen et al. express the importance even more explicitly: *"one cannot realize DevOps in a scalable way without building this into the architecture"*. Moreover, Shahin et al. [SBZ17] argued that the adoption of DevOps practices necessitates modifications to

2. Related Work

software architecture and they proposed that software design should be given precedence in the implementation of CD. Therefore, Bolscher et al. refer to these statements and their implications as a primary resource for the importance of their work to the scientific knowledge base. Bolscher et al. also compiled a comprehensive list of known software architectural issues, which results in the following list:

- Scaling
- Decomposition
- Methods and Tools
- Highly coupled (monolithic) systems
- Ever-changing operational environments and tools
- Monolithic database
- Application level dependencies
- Testing
- Logging
- Monitoring

The authors identify and describe the reason for putting each of the listed issues. They especially mention the issue of lacking "methods and tools", those being the cornerstones of DevOps. The complexity of existing systems and the evolving CI workflow result in high maintenance and adaptation efforts. Bolscher et al. also identified beneficial software architecture characteristics. The list includes the following 17 characteristics that support the DevOps process:

- Agility/Modifiability
- Deployability
- Automation
- Traceability
- Stateless components
- Loosely coupled
- Production versioning
- Rollback
- Availability
- Performance
- Testability
- Security
- Loggability
- Monitorability
- Modularity
- Virtualization
- Less reusability

In this list, they identified "deployability" as the most relevant software architecture characteristic that, according to their opinion, would enable DevOps adoption. In this context, Chen established a compilation of "Architecturally Significant Requirements" (ASRs). Chen defines these ASRs as "requirements that have a measurable impact on a software system's architecture," where "deployability" is also included. More specifically, Chen defines deployability as one of the aspects of a deployable architecture that allows software to be deployed without downtime while moving quickly between environments. Based on this list, Bolscher et al. also identified the related software architecture styles. Their primary claim was that a microservice architecture is the most popular and most suitable style for fulfilling the previously mentioned software architecture characteristics.

2.4. Evaluation and Relevance

The identified related work to this thesis highlights the fundamental importance of software architecture, deployment environments and development setup in the deployment of student software projects. While Le Fevre et al. [LFK22] describe a reference architecture for the deployment of a private cloud via their university infrastructure, the work done by Kjorveziroski et al. [KFMJ18] more closely resembles the architecture intended for implementation in this thesis. In terms of software architecture and structure, the papers created by Valstar et al. [VGP20] and Haider et al. [HRJ21] resemble the basis for the architecture of a reference project for students. Their use of development containers and application of best practices laid the groundwork for the implementation in this thesis software prototype. The work done by Bolscher et al. [BD19] highly influenced the creation of the deployability metric in Section 5.6. They identified issues and characteristics necessary for designing software architecture for continuous delivery and pointed to the importance of further work in this regard. Additionally, Chen's definition of "deployability" as "requirements that have a measurable impact on a software system's architecture" established the foundation for measuring and comparing the created metric.

3. Conceptual Foundation and System Design

At the heart of any technological development lies a well-defined concept and a carefully crafted system design, both of which play essential roles in shaping the success and efficacy of the solution. In the context of this master thesis, the primary concern is the fulfillment of the objectives and research questions outlined in the introductory chapter. To achieve a well-defined implementation of these objectives the following sections will provide a number of implementation concepts defined for each component as well as concepts related to the overall project.

3.1. Framework Design

The conceptual framework serves as the foundation upon which the entire system is established. In this section, the structure of the framework will be examined and the interaction between services will be described. This establishes a high-level overview, while detailed design choices will be described in the "System Design" section.

3.1.1. Overall Structure and Deployment

In Figure [3.1](#), the current deployment structure and the interaction between important components are displayed via a deployment diagram. The diagram encompasses five major components.

- FactDeploy
- A Python Project
- On-Prem or Azure Kubernetes Cluster
- Docker Registry Server
- FactCheck production database

These components and their interactions with each other provide the overall functionality of the framework. It is important to note where each of these components is placed. FactDeploy and the Python Project are installed on the user's workstation and they can be modified and configured as needed. The on-premise Kubernetes cluster, the FactCheck

3. Conceptional Foundation and System Design

production database and the Docker Registry Server are all placed on university-provided servers.

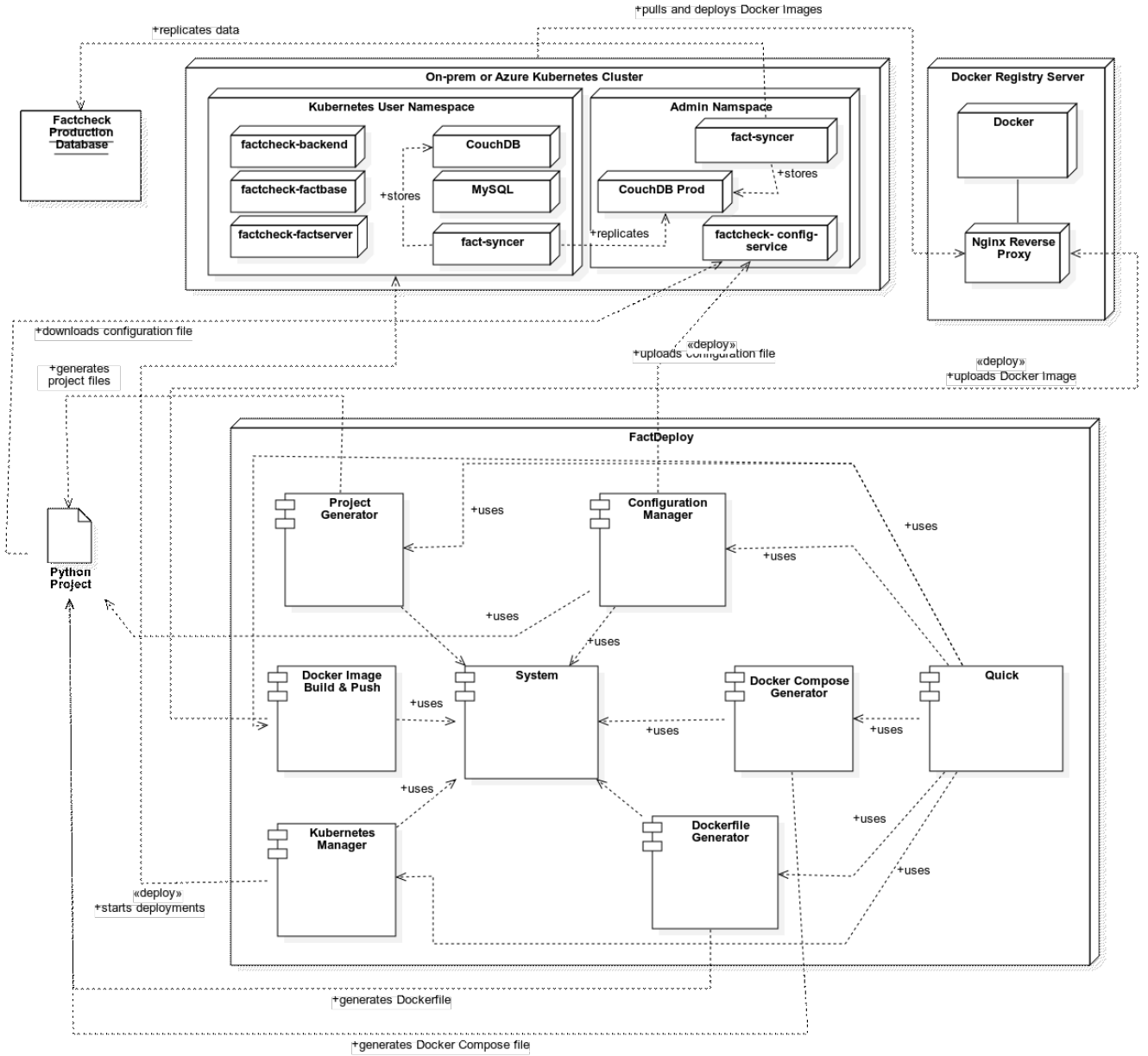


Figure 3.1.: Deployment diagram of the development environment.

As can be seen in Figure 3.1 the Azure environment is not a dedicated environment, as with the deployment of a Kubernetes cluster upon it, the additional abstraction layer is sufficient for the framework to be compatible with both environments.

3.1.2. FactDeploy

The FactDeploy software is the framework's central point. It implements the following features:

- Creation of deployment-ready Python projects
- Upload of project configuration
- Building and pushing Docker Images
- Docker Compose file generation
- Deployments of Python and FactCheck projects to on-premise and Azure Kubernetes instances
- Quick Deployments

With this feature set in place software projects may be deployed and the main components of the framework can be managed. It can connect to the Kubernetes instance, the Docker daemon, the Docker registry, the factcheck-config-service and an Azure Kubernetes cluster. FactDeploy is executed on the developer's workstation and can be run independently, but full functionality requires the Docker daemon and an accessible connection to a Kubernetes instance.

The Kubernetes instance's connection is based on the Kubernetes Python API and relies on HTTPS calls to the internal Kubernetes API Server. These HTTPS calls contain the YAML definitions of Kubernetes resources, meta-information and they initialize deployments.

Connecting to the Docker daemon and Docker registry is done via the Docker Python API which directly connects to the Docker instance running on the developer's workstation. Connecting to Docker is required to allow image building and pushing to a Docker registry.

The factcheck-config-service, which is the main storage service for storing configuration files in a secure manner, is done via HTTPS calls. These calls contain the contents of the configuration file and provide API-keys for accessing these files via another HTTPS call.

3.1.3. Docker Registry

A private or Azure-deployed Docker registry is accessed via the locally running Docker daemon. The FactDeploy software then initiates the building and pushing process, which uploads and stores the created images in a registry with a unique tag. These images are then available for both local and Kubernetes deployments via their specific tags.

3. Conceptual Foundation and System Design

3.1.4. FactCheck-Config-Service

This service securely holds all uploaded configuration files. This service can be deployed via a Docker Image on a supported platform. Based on a basic web-app architecture, this service provides a dedicated REST API for storing and retrieving configuration files. As these files can hold sensitive information, they are securely stored by being encrypted before storage. To retrieve these files an API-key is returned on each upload, which is then used for each file download. This API-key is also used to decrypt the configuration file, therefore careful and secure handling of it is of importance.

3.1.5. Kubernetes Environment

The Kubernetes environment, which can be either an on-premise or an Azure instance, hosts all deployed software. As multiple users may be deploying software, the need for user separation arises. Therefore, a dedicated namespace is available for each user which allows access to resources without colliding with other users.

In an on-premises environment user creation can be done via the "User Generator" project, which allows access to resources and creates the dedicated namespace.

For external communication from, for example, a browser, the users are able to access their software via HTTPS, as SSL termination is done via the centrally deployed "Ingress", which routes network packages to the correct destination. This is only available within the university network and may only be accessed by providing credentials for the "Basic Authentication" mechanism.

Services must be available through a Docker Image repository to enable users to deploy them. Therefore, the previously described registry is accessible via the Kubernetes instance and when a new service is deployed, the image is directly pulled from there. This approach is also used for the Azure Kubernetes instance.

3.1.6. Azure Environment

To achieve a hybrid cloud environment, which allows Azure Services to be accessed from an on-premise environment and the other way around, a connection between these environments has to be established. This can be done in several steps that expand the functionality. Either by creating a Point-To-Site or Site-To-Site VPN connection and then additionally connecting Azure ARC [DB22]. In this scenario, the VPN connection options have been chosen, as has the optional inclusion of Azure Arc services.

The Site-to-Site connection, which will be referred to as "S2S" from here on, is useful for a centralized deployment of the development environment. This allows for direct access from the Azure environment to the on-premise environment, as well as the other way around [McG23]. Although S2S VPNs allow for broader access to the university network,

at this point in time only a dedicated set of IP addresses is allowed to be connected to the Azure environment.

To circumvent the IP address limit and allow other servers to be accessible, the Point-To-Site VPN is a promising alternative. This Point-To-Site VPN will be from here on referred to as "P2S". It enables dedicated hosts to access the Azure environment and circumvents the IP address limit as a dedicated connection is established [McG24].

In this thesis, another option was also explored, which is to deploy the university's VPN as a sidecar container to a already running container. This would allow a single service running in an Azure Kubernetes instance to access applications running in the university's network. While this is possible, its usefulness is somewhat limited, it only allows for a single connection with two-way authentication.

3.1.7. Data Replication Service

To allow for an efficient data replication mechanism, in-cluster replication and a production replication mechanism are available. Production data replication entails transferring data from the production environment to a locally deployed CouchDB instance. As several users may access production data, the in-cluster replication mechanism comes into play. Here, the replicated data from the central CouchDB instance, previously described, is also replicated to a dedicated CouchDB instance for each user.

3.2. System Design

The system architecture is fundamental for providing an extensible and maintainable software product that can adapt easily to new features and change requests. Therefore, the next sections will describe the concept and design of the project's implementation to fulfill these requirements.

3.2.1. Project Structure

The Python programming language is the basis for this system, thus the main concepts required for a project and a programming structure are applied. The project structure is built on this foundation and extended with several external libraries that provide essential functionality and enable faster implementation of new features.

Due to the underlying concept of manipulating configuration, deployment and Docker Image definition files, the project structure can be partitioned onto an independent feature basis with a common core. Such partitioning provides the flexibility to extend the software easily without interfering with other domains of the software. Furthermore, the common core provides a foundation that reduces code duplication and provides a common interface for accessing external files and APIs.

3. *Conceptional Foundation and System Design*

To define the common core in this project called the "system" module, the scope and its functionality have to be defined. In this context, scope and functionality refer to the system module's exclusive use of functionality for other domains, as well as its provision of common structures for interdomain interaction and access to external APIs. It doesn't implement any objectives, features, or other requirements.

On the other hand separable domains, which are implemented in their own dedicated modules, contain the necessary code for building their feature set based on the common core module. Without directly interacting with other domains, these modules include their UI, services, data manipulation and internal interaction components. Interaction is only indirectly possible by storing relevant information in a common storage from which all domains can read through a common API provided by the system module.

These two main concepts enable a highly modular system that is easily extensible and testable without having large amounts of code duplication.

3.2.2. **User Interface**

User interfaces are, as the name already implies, interfaces by which a user can interact with the system. The primary user group for this system is projected to be computer science students working on their own software projects. Therefore, a background in computer science and familiarity with command-line interfaces can be expected. This is an effective way to interact with the user base and therefore this software will use a command-line interface (CLI).

When working with on the UI, the one concept that needs to be applied to this project is the separation of the UI and the business logic. This concept improves testability and simplifies the debugging process. As a result all UI logic should be broken down into individual components, each of which stores all results in a separate object that is then used by service components for their processes. The complete concept is initiated and executed via a separate component. This implementation will therefore be similar to the classical model-view-controller pattern, which has proven its reliability over time [GHJV95].

One of the main disadvantages of CLIs is that, in simple implementations, they often rely on defining the specific feature used in a run via a parameter and providing additional parameters prior to execution. It then only outputs the current state or the program's results. This approach, although possible, is not viable for this project as many different parameters are needed and functionality partially depends on previous input. Therefore, an advanced CLI interface is a necessity for this project to help the user guide the process through the decision trees and inputs. Additionally, it is necessary to display helpful information about the correct usage and the impact of each step on the final process, particularly for users who may not have previously used such software. To ensure the framework to be effective for this project's use cases, the input of parameters must be provided by effective methods for inputting paths, texts, numbers, selections, lists and

confirmation questions.

3.2.3. **Storage and Data Manipulation**

As a large number of parameters are needed for executing this software effectively and recurrently, the storage of these parameters is a necessity. The most efficient way to implement such behavior is by storing all information in a properties file. Although this approach is simple to implement and work with, its limitations arise when the number of parameters increases, lists are added, or complex objects need to be stored. To achieve the flexibility of property files while still being able to work with complex objects, the introduction of a YAML parser/emitter library is necessary. With such a library complex objects can be serialized and stored in a YAML file, which can be directly read back into a complex object. Such an approach enables easier usage and interaction with stored content, as objects can be directly used without additional parsing. Furthermore, as all data are stored in a YAML file, which is in a human-readable format, a user can alter it manually if any input was made erroneously.

3.3. **Dynamic Local Development Environment**

The conceptualization of a dynamic local development environment, in which a user can expand its functionality based on their needs, stems from the idea that this tool should be technologically agnostic to the best possible extent. As a result, the technological background for each project developed with this tool can be completely different from the previous one. One of the main implications of this concept is that extendability, customization and abstraction are critical to providing such a service. Furthermore, for this concept to work, the identification of tools, software, interactions and workflows between FactCheck services was essential. As this thesis is written in the context of the FactCheck Project, the integration of the required services in a dynamic way was a key requirement.

3.3.1. **Docker Containers and Images**

Developing software locally, especially server-side applications with a larger user base, initiates possible error sources based on the system and environment in which the software is executed. For this reason, the use of containerization technologies, such as Docker Images and the resulting Docker Containers, is the fundamental technology for implementing this concept. This technology has the advantage of being based on a lightweight virtual machine for each container [Mer14]. Therefore, Docker is able to run these containers on the most common operating systems without the software running inside the container needing to be adapted to the host environment.

In this thesis Docker Images will be used in different ways. The most common way of using existing images, changing their configuration and running them as containers. This

3. Conceptual Foundation and System Design

will be done for standard software like databases. This approach is quick to implement and relies on already (partially) pre-configured images. This in turn reduces the risk of faulty image definitions and possible security issues [\[RSO22\]](#).

The second method is based on using self-made images that are implemented specifically for a service and built locally. Such a case will be used primarily in relation to services from the FactCheck project that are not publicly available and require custom configuration and dependencies to function.

The third approach would be to use the feature which lets a user run commands inside a running container. By using this feature and endlessly running a Linux distribution like Ubuntu inside a container, the developer now has the possibility to execute their code inside the container. This is all possible without building a full Docker Image of their current project. This approach brings the advantage that the environment in which the project is executed is the same as deployed in production and also debugging is faster and is now fundamentally bound to a single operating system provided by the container and not by the developer's workstation. To implement this without manually copying source code into a container, the project (stored on the developer's workstation) needs to be synced inside the container. This behavior can be achieved using the concept of volumes, which maps an external folder to a folder inside a container. Therefore, exploitation of these concepts allows developers to work both locally and in a production-like environment without the need to start a building process and deploy a production server for debugging.

3.3.2. Docker Compose

After identifying and implementing the previously described strategies and concepts for using Docker Images in this thesis, the next step involves utilizing Docker Compose to automate the creation and management of containers [\[Doc25a\]](#). This automation is managed by defining the services and their behavior in relation to the overall system and itself via so-called docker-compose.yaml files. Services, networks, volumes and several other constructs can be defined in such files, enabling the creation of a complex system in which services can communicate with each other. By merging, extending, or including these files, several techniques can be applied to load external services or other docker-compose.yaml files.

The goal of this thesis is to dynamically create this type of files enabling developers to easily choose from pre-configured services to be used in their project and also gives them the chance to integrate external Docker-compose projects into their system. To achieve this, the structured data in the docker-compose.yaml file will be generated in a modular way. The strategy for the modularization of such a docker-compose.yaml file is to apply self-defined, pre-configured snippets of service definitions for common services used in relation to the FactCheck project. To dynamically integrate FactCheck services in their docker-compose.yaml files, available for each service, are read and a link is generated that connects all services defined in that file to the docker-compose.yaml file for the given project.

External connections to services, such as networks, volumes and environmental variables, will be defined via two different approaches. For example, the main network will be defined globally so that all services can connect to it, if needed. In addition, local networks can be created, which need to be defined by the user upon service creation. The same goes for volumes, as the most common use case is that a single service, such as a database, has access to a storage medium. As a result, volume definitions will also be defined when the service is created via user inputs. The same approach will be used for environmental variables, as these are also, in most cases, service-specific. For global definition, the user will have to customize their generated docker-compose.yaml file to fit their needs.

3.4. Dynamic Project Creation

In this thesis, the focus is on Python projects created by student developers, as they are heavily used in the FactCheck project and scientific software creation. Such projects are primarily focused on implementing a scientific concept rather than providing software designed based on software engineering best practices. To combine these two concepts, the introduction of a fundamental project structure and the use of modular software building blocks allows for easy integration of software engineering best practices into scientific software.

3.4.1. Project Structuring

Python projects on their own have a minimal structure, as the original use case of the Python programming language was intended for implementing small scripts to automate processes. To enable the use of Python for larger projects, a common project structure must be introduced. As a result, it is necessary to identify commonly used Python project structures. In the best case, best practices not depending on external dependencies have been selected and will be used as building blocks when generating a new project.

These best practices include structures defined in the Python-Packages Book by Tomas Beuzen and Tiffany Timbers [BT22]. This book establishes a standard package structure that encompasses configuration files, folder structure, documentation, and versioning. In addition usage of git-related services, such as CI-CD pipelines, tagging, and publishing, is available. Combining these structures with others commonly found in projects is then used to generate the final project structure based on user selections. Additionally, structures that are necessary for external dependencies, like health checkpoints for Kubernetes, will also be available in that selection.

All these included modules, when chosen carefully, result in a beneficial starting point for a scientific software project that will be positively accepted from a software engineering standpoint.

3. Conceptional Foundation and System Design

3.4.2. Modular Definition

Some of these prerequisites are only important for specific use cases, such as deployment to a Kubernetes cluster or releasing a project as a library for other software products. Furthermore, students' creativity should not be restricted by external dependencies. Therefore, the modularity of these packages and the avoidance of dependence on a specific framework, library or external software are of utmost importance.

The concept for such an approach is to use, as already proposed in the Docker Compose section, snippets, in this case files, and combine them into a final project. Because each snippet should be, in the best case, completely independent; adding or removing snippets should have no effect on other parts of the software. This enables the user to select and deselect modules based on their needs, requirements, and environment.

3.5. Dynamic Dockerfile Creation

Dynamically creating Dockerfiles is based on the same principle as docker-compose and already described principles in Section [3.3.1](#). The process involves creating YAML files and adhering to the modularity concept for using Docker containers. Furthermore, the first use of templating will be necessary.

First of all several base Dockerfiles are available as a basis for different use-cases; additional functionality, such as port definitions and non-root user creation, should be selected by a user. These functionalities will be added and configured via templating based on user preferences.

The templating approach will be based on classical text replacement via keywords. In this case each keyword will be directly replaced with user input. The intention is to directly store the Dockerfile inside the current project folder, which enables users to customize it and remove or update erroneous data. This ensures that best practices can be applied while allowing users to customize their Dockerfile and therefore the resulting Docker Image to their needs.

3.6. Configuration Management

A software configuration management system serves as the backbone of software projects. The ability of software to load its configuration on demand and adapt itself to its surroundings is essential, especially for server-side software with an ever-changing environment.

When starting up a service in a production-like environment, like the Kubernetes instance used in this thesis, each service should be executing a production configuration. There are three common ways to achieve this. First and foremost, the simplest form is to

use a production configuration switch in the software when building it for a production environment. The main disadvantage of this approach is that it necessitates the redeployment of services every time a configuration change occurs due to the fact that it is inherently integrated into the software. The second approach is to inject the production configuration upon startup. To accomplish that, the service needs to know where the configuration is stored at build time. This method has the advantage that on each startup, the configuration file is read again whereby the most current configuration will be read. The main drawback is that a service restart is required. To counteract that, a third method can be introduced: polling and injecting the configuration at runtime. This implies that the service can update its current configuration at any time, but it must also be resilient enough to identify alterations in the configuration and adjust accordingly.

3.6.1. Server-Side Application for Configuration File Storage

The goal of this thesis is to give the user all three previously mentioned possibilities. Therefore a dedicated configuration service has to be developed that allows users to upload and store their configuration. Furthermore, access to each uploaded configuration file has to be possible. As such configuration files contain sensitive information, each file has to be accessible only for a specific user and, in the best case, only for a specific service. Accomplishing these requirements will be done through a simple mechanism. First and foremost the configuration file will be uploaded via the configuration service's public REST endpoint. This configuration file will be encrypted with a generated key and stored in a database, in this case a SQLite database. Each configuration file will be identified via another generated key with which it will be retrievable. Both of these keys are concatenated with a ":" as a separator which results in the final API-key that is returned to the user for further requests. As the key for encrypting the uploaded data is not stored, only the initial uploader can gain access to their data and decrypt it. The complete execution flow can be seen in Figure [3.2](#).

Each request for a configuration file needs to have an "x-api-key" header set with the API-key as value. If that requirement is met, the service searches for the configuration file decrypts it and returns it in plain text, which can then be used by the developer.

With this service in place and accessible to all services the startup injection and the runtime injection mechanism can be implemented by each service.

3. Conceptual Foundation and System Design

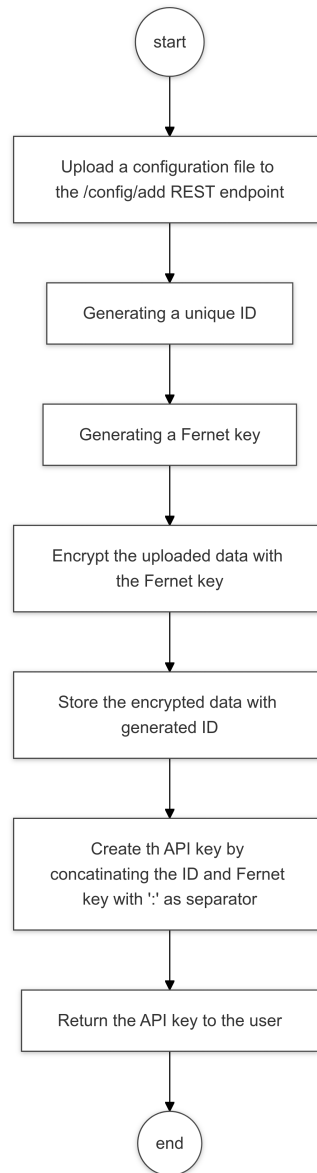


Figure 3.2.: Flowchart for Configuration Upload Mechanism

3.6.2. Client-Side Integration for retrieving Configuration Files

On the client side, all three possible configuration injection methods can be implemented. As each approach has its advantages and drawbacks, one has to be chosen. For this thesis, the startup injection method was chosen because it is completely agnostic to the project's implementation and has no implications for the program logic.

3.7. Building and Pushing Images to a Registry

Implementing this logic can be done by supplying a reference to the configuration service and the necessary API-key to the project. To do this agnostically, both the reference and the API-key will be supplied via environmental variables on deployment, which can then be used by the software to retrieve its configuration from the server. This mechanism can be used at startup.

For example, Docker Images require a command or a file to execute on startup. Therefore, a custom startup file can be provided for each project, which loads the production configuration file from the configuration server and then replaces the development configuration. This logic therefore allows startup configuration injection without any dependence on the programming language or environment of the deployed project.

3.7. Building and Pushing Images to a Registry

To build and push images to a Docker Registry [\[Doc25b\]](#), which manages uploaded software images, developers need to authenticate themselves to access their images. As images built by students should not be publicly available, a dedicated Docker registry was created. This registry provides authentication via the standard username and password method and as it is on university hardware, it can only be accessed from the university network. This also implies that it is completely separate from the public internet, thus the student's software remains in a private environment.

3.7.1. Version Upgrade

The Docker Registry [\[Doc25b\]](#) differentiates between images via their dedicated tags. This tag is a string that contains the registry hostname, the software name and a version identifier. To allow students to upload new software to the Docker Registry, each time the building process is started, the version identifier has to be increased. To accomplish this, the user will have numerous options. The first possibility relies on supplying a file with the current version number. If this file is available in the currently selected project and the user allows for an automatic version upgrade, the minor version number will be automatically increased by one. This means that the tag file and the version file will be in sync. If these requirements are not satisfied, the minor version of the image will still be increased by one, but based on the latest available image in the local Docker repository. This means that the version file and the image tag will not be in sync and the version string will start with "0.0.1". It is also necessary to mention that this version string will be preceded by the developer's username. This is necessary when, due to pure coincidence, the same project name is chosen twice. The Docker Registry and the developer can differentiate between images and not override images from other developers. As a result, the final version string for Docker Images will look like this: "max-0.0.1" and the complete tag will be "dockerregistry-lab.cs.univie.ac.at:443/example-project:max-0.0.1". With this tag, the correct image can be identified by the user, the Docker Registry and any system that executes that image.

3. Conceptional Foundation and System Design

3.7.2. Building Process

To build Docker Images, which are an abstraction layer around software projects, a building process has to be started. With this efficient process, which stores software in reusable layers and combines them into images, software can be executed via any software that supports the common image standard called "OCI" [Ini24]. As both the FactCheck projects and the currently selected project have to be built, the whole process has to be agnostic. To achieve this all the necessary parameters have to be provided. The path to a service's Dockerfile and a tag are essential components. With all necessary parameters provided the building process can be executed. To increase speed the user will be able to select when wanting to use caching in the process of building any Docker Image.

3.7.3. Pushing Process

The pushing process will happen directly after the building process. The only prerequisites are a previously described tag and an already-built image. As the tag identifies the image and also the registry to which the image will be pushed no additional information is necessary. To inform the developer of the upload progress the current percentage will be displayed. This is necessary as these images can have hundreds of megabytes when pushing the first Docker Image of software. For efficiency reasons and to speed up the pushing process only layers¹ of the Docker Image that are not found in the registry will be uploaded.

3.8. Deployment to Kubernetes

One of the main ideas of this thesis is to provide students with a dedicated development system that is as close to a production environment as possible. For such an idea to be implemented, a suitable platform has to be chosen as a basis. Kubernetes now comes into play, as it is widely used as a deployment platform in production environments and provides the necessary components to allow both on-site and cloud infrastructure deployments, as well as easy migration/interaction between them [GB23].

3.8.1. Software Deployment

To deploy a project currently implemented by a student developer, dedicated Kubernetes deployment files are needed, which provide necessary information about networking, deployment and storage. These files are frequently very similar between projects, so that they can be adapted with minimal changes and applied to other projects. Developers will have access to the most essential deployment types in the prototypical implementation.

¹<https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/>

This includes "Job" or "CronJob", which can be executed once or a specified number of times, or repeated at a specific time and "Deployment" which runs continuously.

"Job" and "CronJob" deployments, which cannot be directly accessed by other deployments due to their execution timeframe, require less configuration in their deployment files. Only the result of their processing is available to other services, most commonly in a storage unit like a database or a file. Therefore, developers who choose a "Job" or "CronJob" deployment need less overall configuration but will need to supply additional information on how often and when they want to execute their deployment.

On the other hand, services with the type "Deployment" need additional information on how external and internal networking will be handled. To make these kinds of services available inside the system for other deployments, a dedicated "Service" definition has to be supplied. And for externally available services, an "Ingress" definition has to be added. To make this easier for developers default values will be supplied based on previous inputs such as the project name and username. This means that developers only need to supply the port on which a service will be available.

3.8.2. Role-Based Authentication

Kubernetes allows for role-based authentication, which means that users have access to resources and namespaces via their current role. There are two types of roles: "Role" and "ClusterRole".

For this thesis, the use of namespaces and cluster resources is important to allow users to access all necessary data to deploy their systems. Users will deploy all their software on a dedicated namespace. Furthermore, they will have access to common deployment, networking and batch processing APIs in their namespace, ensuring that all essential functionality is available. To extend this functionality, an administrator will have to manually update their permissions. The username provided when creating a dedicated Linux user will identify a namespace. In the context of this thesis, cluster roles will be limited to allow for restricted storage management, as this is required for users to be able to create and manipulate databases and files without losing their data on restarting any of their storage services.

To automate this process of user creation and role configuration, a dedicated script will be implemented that creates a user with all necessary default role configurations and allows users access to the server, the namespace and their applications via a username and password.

3.8.3. Client Side Access

Accessing the Kubernetes cluster externally has many security and connectivity implications. Kubernetes itself has three methods for allowing access to internal services. The first method implies that the user is already logged into the host server. Allowing this

3. Conceptual Foundation and System Design

access also requires that an SSL certificate and a key be generated for this user and these have to be registered with the Kubernetes cluster. The second method allows access to the cluster externally by making the Kubernetes API Server accessible to the network. This method also requires users to be authenticated against the Kubernetes cluster with a certificate and key, which must have been previously registered. The third method directly loads the .kubeconfig file from the server and connects to the Kubernetes server via port 6443. Due to security policies, only ports 80 and 443 are accessible from the outside. This procedure had to be adapted to fit these circumstances.

In the context of these available methods, the only possible solution would be a combination of the first and third methods. This is because the university's security policy only allows access to the Kubernetes cluster from the outside via ports 80 (HTTP) or 443 (HTTPS), not via port 6443 (Kubernetes API Server). As the HTTPS ports are necessary for allowing users to access their services via a browser, the only option remains to access Kubernetes via SSL tunnels.

Consequently, to allow clients outside of the cluster to access it, SSL tunnels need to be established. These load the configuration of a registered user who can then access the Kubernetes API Server on Port 6443 via the Kubernetes client for Python. As a result, users can now deploy and manage their services from their own workstation by simply providing their credentials to the Kubernetes host server, eliminating the need for storing critical certificates and keys on their local workstation.

3.8.4. Templating

A popular templating engine for Kubernetes is called Helm. This software, with the correct configuration, can generate Kubernetes deployment files and package several components together to deploy all necessary components at once. Although this approach is excellent for packaging a production build, a development build needs the flexibility to disable, alter and select additional components.

Therefore a more dynamic approach is necessary. To allow this, a common pattern replacement method has to be implemented which is compatible with Kubernetes deployment, Dockerfile and Docker Compose files. This pattern replacement approach may use predefined keywords with the prefix symbol "«" and postfix symbol "»". This pattern will then be replaced by the actual value associated with the keyword and will therefore enable templating.

Several default Kubernetes files have been created, which are based on the selected type of software (Service/Job/CronJob). Additionally, dedicated files for FactCheck services and commonly used databases, such as CouchDB, MongoDB and MySQL, have been created. These files are necessary because initializing FactCheck services and databases requires additional configuration. Any of these files will contain the necessary patterns to apply the templating approach.

3.8.5. Docker Registry Access

When deploying software to Kubernetes access to a repository for Docker Images is necessary. In this thesis, the deployed Docker Registry will store all images created by student developers in a secure environment. For Kubernetes to access these images, the correct credentials have to be supplied. Therefore, a so-called "Secret" containing these credentials will be created for each developer and which is only accessible in their dedicated namespace. This "Secret" will be provided on every deployment so that a student developer can load and deploy the software correctly. Access to the default public Docker Registry is also available.

3.8.6. External Access

External access to services is commonly done via a dedicated "LoadBalancer" object in Kubernetes; this object can be created on-premise or as a cloud entity from a cloud provider. Both of these approaches are not possible, as this Kubernetes instance is running on a university server. This means a cloud load balancer is not available. An on-premise load balancer is also not an option as only one IP address is available but that on-premise load balancer requires an address pool to function properly.

To overcome this issue, another way had to be found. In this case a so-called "NodePort" will be used, which will make ports 80 (HTTP) and 443 (HTTPS) available to the Kubernetes cluster for receiving external requests. This approach leads to the next problem, which is internal routing, as network traffic is commonly routed through a single "Ingress" instance. This means that when a single Ingress is used all publicly available services must be defined there. As this development environment is intended for multiple users with multiple services running at the same time, all of which could be publicly available, a more dynamic way has to be found.

To implement such a dynamic routing structure a non-standard Ingress created by NGINX has to be used. This Ingress can be defined as having one main Ingress and multiple "minion" Ingresses, which can lie in different namespaces. As the previous requirement that only one Ingress can exist still holds, all "minion" Ingresses will be merged into the main Ingress. This means, in the context of this environment, that for each user there can be a dedicated routing structure without the need to change the main routing input. Furthermore, each user has the possibility to route traffic to any of their services and make them publicly available for anyone in the university network. While this allows for external access to a service, accidental or intentional access to someone else's service is now possible.

To reduce the risk of unintentional access to someone's services, security mechanisms have to be implemented. There are plenty of choices for secure access protocols: OIDC, LDAP, JWT, etc. While these are highly secure, they are also complicated to set up and may require additional steps by external developers to configure their software implementation in compliance with these protocols to allow access to their deployed services. Therefore,

3. *Conceptional Foundation and System Design*

a compromise between security and practicability has to be achieved. Because the Kubernetes instance is currently running on the University of Vienna premises, access to these services is only possible with valid authentication through a university account. This situation already mitigates potential security risks for a user group that includes both university students and individuals. Therefore, a simple user identification mechanism should be sufficient for the software development purposes of this thesis.

3.8.7. **Viewing Deployments**

An essential feature, especially for developers not familiar with Kubernetes, is the introduction of the so-called 'Kubernetes Viewer' feature. This feature allows users to view, delete and interact with a set of deployed Kubernetes objects that are directly related to the software they implemented. This set includes Pods, Deployments, Jobs and CronJobs.

Deployments, Jobs and CronJobs are higher-level Kubernetes objects that represent a collection of pods. These objects provide metadata about the current state, metadata and events that occurred during their lifetime. Pods, on the other hand, are essentially an abstraction of the executed program (which in this thesis is a Docker Image) and therefore contain the logs, metadata and necessary information regarding the program's state. All this information must be provided to the developer in order to gain a better understanding of possible issues that may occur after the deployment of their software. Furthermore, the deletion of these objects is necessary, as in worst-case scenarios, developers need to stop their programs manually before any data gets corrupted or too many resources, in terms of CPU time and storage space, are used.

Interaction, deletion and viewing of any of these Kubernetes objects have to be done in a concise way. Therefore, four essential actions were chosen to provide these necessary features:

- **Logs:** Displays the complete logs of a selected Pod
- **List:** Displays a list of all Kubernetes objects of a specific type (Deployments, Jobs etc.).
- **Describe:** Allows for a detailed introspection into the current and previous state, metadata and events of a Kubernetes object.
- **Delete:** Allows for the deletion of a Kubernetes object.

With these features in place, a developer will be able to have an overview of the current state of their running program without manually entering the Kubernetes server.

3.9. Quick Deploy Process

The quick deployment process is done simply by combining several of the previously described processes. This set of processes includes project creation, Dockerfile creation, Docker-Compose creation, configuration upload, the building and pushing workflow and the Kubernetes deployment process. When a project is already defined Dockerfile, configuration, and Docker Compose creation will be skipped, and only a new Docker Image will be built, pushed, and then deployed to Kubernetes. Previous input settings will be reused and if none are available, sensible default values are predefined and used.

Therefore, in the best-case scenario where all parameters are given, quick deployments will allow students to execute their deployments with a single click. In the case of any necessary but missing parameters, the user will be asked to provide them once. Upon the second usage, all necessary parameters should be available, allowing for a even faster deployment.

3.10. Data Replication Process

This thesis's fundamental requirement is to allow developers to develop their software with a copy of the production data. As the real production data is not directly stored in the Kubernetes instance, it has to be cloned from the original source. This data only includes publicly available data in the context of this thesis, which are so-called "facts" provided by the FactCheck project and stored in a CouchDB database via JSON structures.

3.10.1. Architectural Choices

Because this replication process is based on the CouchDB replication mechanism, it can be started from both the receiver and provider databases [Cou25]. This has implications for security, infrastructure, and performance, as based on the environment in which each database is placed, a different approach has to be taken.

The decision to shift the replication process from the production database to the development environment was restricted because the system only permits read access from a specific set of databases. Consequently the replication process has to be initiated in the development environment.

Another choice that had to be made was the placement of the replication process between the copies of the production database in the development environment and the ones for each user of the development environment. Placing it directly on the main production database may bring advantages in terms of security, as only admins can allow replication processes to be executed. The effort to start this process for each user would not be proportional to its effectiveness. This is because either each receiving database must be known beforehand, or the main database must search through all available databases to determine from which one to start a replication process. However, by placing each of

3. Conceptual Foundation and System Design

the replication processes on the receiving side, the user databases only have to know the location and credentials to the main database. Such a placement would allow users to dynamically create their own databases, and each user could replicate data from the main database without needing to know each one beforehand.

3.10.2. Efficient Replication

The production data which contains several gigabytes of files, requires an efficient replication process. Initially, the production data is replicated using CouchDB's replication mechanism and is stored centrally. To reduce memory usage, the replication process will be halted at an administrator-specified percentage. To allow developers to have their own dataset, each user will be supplied with a copy of the replicated data. When a user is initially created, a copy of this data will be available to their user, which they can then manipulate and use for their own projects. Due to the updates on the production data, the replication process has to happen at intervals between the production database and the central database on the Kubernetes instance. Additionally, a replication process from the central database to the users' database is required to keep their copy of the data up-to-date.

3.11. Integrating Azure Services

3.11.1. Migrating to Azure Kubernetes Service

The main objective of this thesis is to support or provide alternatives to currently used services by the FactCheck production system running in Azure. During the research for this thesis on the FactCheck project's use of Azure services, it became clear that only the following services were being used:

- Azure Functions: These are services that execute code in the Azure environment. These can be ported to Kubernetes via the KEDA Framework [\[Aut25a\]](#), but this is not necessary, as calls to these functions can be made through HTTPS as long as they are accessible via a network connection.
- Azure CosmosDB is a custom database with an API compatibility layer that allows users to access it, like MongoDB. A MongoDB instance is simple to deploy on Kubernetes, or the CosmosDB can be directly accessed via a network call.
- The Azure Data Lake is a hierarchical file system storage system with a focus on big data analytics. This service is also accessible via a network call.
- Azure Blob Store is a large file storage system that is directly usable through a PersistentVolumeClaim with "azureblob-nfs-premium" as the storage class. Therefore, there is no need for any migration.

- Service Bus: This message queue can be ported to Kubernetes via the KEDA Framework[Aut25a], but this is not necessary as calls to this bus can be done through a connection string and APIs as long as it is accessible via a network.
- Azure API Management is an abstraction layer that manages access to APIs in front of multiple applications.

Most of these services are directly related to either a network connection or accessing storage. These services can be directly accessed through Kubernetes, ported, or directly accessed via a network connection. This situation allows for simply concentrating on bringing the Kubernetes instance into the same network as the services running in the Azure cloud. This allows access to supported and/or already used Azure services.

Azure API Management is the only one on the list that cannot be easily ported or accessed via a network call, as this service would have to be placed in front of the Kubernetes instance. Because this service is primarily for routing API calls, its usefulness in a development environment is hardly given. This is due to the possibility that API calls can change at any time in the development process, resulting in a greater management effort. This service has no impact on the system's functionality, thus it will not be explored further in this thesis.

3.11.2. Connecting to Azure

One requirement for this thesis is access to Azure services through on-premise deployed services, and reverse. To achieve this, the two systems must interact by allowing a network connection between them. Azure has several options to connect these systems, but the two fundamental options are Azure Arc and VPN gateways or a combination of both.

For the implementation of a hybrid cloud approach, the choice was made to use VPN gateways to set up their connection. This choice is based on the requirement of Azure Arc to have dedicated ports open to the public internet to allow the connection or a private-link connection via the VPN gateway. The Kubernetes instance of the on-premise solution is placed in the university network, which is not accessible to the public, and the required ports would have to be opened to the public as well.

With VPN gateways these limitations can be circumvented. Depending on the chosen connection method there are other more manageable requirements. Site-to-Site connections require greater administrative rights and access to network configuration tools, but they allow for connecting complete on-premise networks with Azure cloud networks.

In the case of Point-to-Site, only a single server can be connected to the Azure cloud. The major downside of this approach is that certain ports have to be available to be bound by the VPN client software, and the server has to have access to the public internet.

Depending on the use case, both of these approaches are suitable for allowing a hybrid cloud environment. While a proof-of-concept will be done for a Point-to-Site connection,

3. Conceptional Foundation and System Design

the main connection method will be done via a Site-to-Site connection provided by the FactCheck team.

Another option was also explored while working on this thesis, which was based on using the university's VPN to connect to the Azure Kubernetes Service. Here, the idea was that any device could connect to the university network by using the university VPN client. Therefore, a VPN client could be deployed through which all traffic to university servers could be routed. While this approach seems useful for the use case of creating a hybrid cloud with a free connection method, however there are several limits. This includes the necessary two-way authentication method and prohibited multiple simultaneous connections. This means that the connections are not stable, at any point in time, as the connection could be closed by opening another. Therefore this approach cannot be used for the purpose of a development environment.

3.11.3. Connecting two Kubernetes nodes in a hybrid environment

To allow for on-premise Kubernetes to interact with the Azure cloud, both have to be accessible via a VPN connection.

If this connection is available the on-premise environment can be integrated via Azure Arc. Although Azure Arc does not allow to directly connect an on-premise cluster with a Kubernetes cluster, there are several advantages. It allows for detailed insights into the cluster and, most importantly, it allows for easier management of the on-premise cluster.

This connection between the on-premise and Azure Arc services enables the cluster to interact with the Azure environment showing knowing that they are currently communicating over a VPN connection.

4. Implementation

This chapter provides the necessary juncture where the conceptual foundations and requirements of the previous chapters are transformed into a practical reality. In the case of this thesis, several small software projects had to be implemented in a prototypical way. This chapter also serves to showcase the implementation and design decisions taken, offering a comprehensive overview of the tools and techniques used.

4.1. Technology Stack and Environments

The introduction of this development framework for the FactCheck project also partially established the technological stack for the prototype's implementation. Therefore all custom-programmed software for this thesis relies on the Python 3.12 programming language.

To allow file versioning and, consequently, tracking of the current state of the source code the version control system Git was used. The repository, where all source codes will be available, is hosted on the University of Vienna's GitLab platform. This repository contains FactDeploy, factcheck-config-service, factcheck-sync-service, factcheck-vpn-relay, user generator scripts, Kubernetes scripts, VPN setup files and the configuration of the Docker Registry all in separate folders.

The programs themselves are using a virtual environment called "venv" which means that the libraries used are only installed in that virtual environment. Since Python packages are installed globally, it was necessary to introduce this virtual environment to prevent other Python programs from being aware of these libraries, as version inconsistencies and overlap could potentially introduce unwanted behavior across different programs.

4.1.1. FactDeploy

FactDeploy enables fast and simple Kubernetes deployments. To achieve this and other features, several Python libraries were used to fulfill all necessary requirements.

The relevant Python libraries are:

- ruamel.yaml - allows for reading, writing manipulating YAML files.
- inquirerpy - is essential for creating a command-line interface with input verification and basic styling.

4. Implementation

- pyfiglet - is a small library for displaying ASCII art of the software name.
- virtualenv - needed for the generation of projects
- Flask - needed for the generation of projects, adds REST functionality.
- docker - needed for building and pushing Docker Images to a Docker registry.
- packaging - is needed for automatic version upgrades when building projects.
- kubernetes - needed for interactions and for deployments to Kubernetes.
- paramiko - provides the SSH client for connecting to the server Kubernetes is running on.
- sshunnel - is used to create a SSH tunnel to allow forwarding of the connection to the local host.
- requests - an HTTPS library that allows for calling external APIs over HTTPS.
- passlib - used for Httppasswd generation for basic authentication for users to access their public endpoints.

FactDeploy is designated to run on students personal computers; therefore, it is designed to run on Microsoft Windows, macOS and Ubuntu.

4.1.2. Factcheck-Config-Service

The main functionality of the factcheck-config-service is to allow for updating and downloading configuration files for services. Therefore, connectivity through a REST API and the storage of these files were crucial, necessitating the use of the following libraries:

- Flask - adds REST functionality to allow for updating and downloading configuration files.
- Gunicorn - adds a production WSGI-HTTP server for allowing requests to be executed via multiple workers simultaneously.
- Gevent - Adds better worker threads to Gunicorn to increase performance and utilization.
- sqlite3 - is a Python-included package to create and work with the file-based database software SQLite.
- cryptography - adds cryptographic functionality, used for encrypting and decrypting uploaded configuration files.

When deployed on Kubernetes, the factcheck-config-service runs inside a standard Ubuntu 24.04 container.

4.1.3. User Generator

As simple user accounts in Ubuntu are not able to access Kubernetes directly, a setup and generation process has to be started. To grant this permission, the project had to access the following libraries:

- kubernetes - needed for updating network configuration.
- ruamel.yaml - allows for reading, writing manipulating YAML files.

This software, which should only be executed by an administrator, does not rely on any deployment definition and should be executed via a simple bash script.

4.1.4. FactCheck Backend

The FactCheck project provides a service simply called "backend" that provides precision and information metrics to users or other FactCheck services. This service is also based on Python 3.12, but in addition, it uses the Django Framework, which allows for the creation of a web application. The database used is another important factor, especially in relation to this thesis. As Django allows a selection of databases, the following are available: SQLite, PostgreSQL, Oracle, MySQL, or MariaDB. Currently, the default configuration uses the MySQL database for production storage and, therefore will also be the same in the development environment. During the implementation of the prototype for this master's thesis, this part of Factcheck was marked as deprecated. Nevertheless, the necessary adjustments and measures to ensure functionality are described, as this application may be used again in the future.

4.1.5. FactCheck Factbase

Factbase is a simple service that provides the views and replication functionality of the CouchDB database for storing so-called "facts" (JSON data structures) and similar data. Therefore, the main dependency for this service is the availability of a CouchDB database and, when needed, an external CouchDB instance for replication.

4.1.6. FactCheck Factserver

This service is regarded as the most important in the fact-checking environment. This service is responsible for comparing entities with their respective representations. As sets of entities and other stats are stored in CouchDB, one of the main dependencies for this service is the availability of a CouchDB database correctly initialized and containing the necessary data. The other main dependency is the FactCheck Backend, which provides the API for the required precision metrics. It also provides a frontend and hosts an HTTPS server based on the Tornado framework.

4. Implementation

4.2. FactDeploy

FactDeploy is the central component of this framework, having core functionalities for generating standardized, deployable Python projects and allowing for interaction, manipulation and deployments with a Kubernetes cluster from an external device, such as a student's personal workstation.

4.2.1. Project Architecture

The FactDeploy software divides into three structural layers as can be seen in Figure 4.1. The first layer (also called the system layer) is the common core for and is used by the second layer (also called the feature layer). The feature layer contains, for each feature, a dedicated component that does not interact with any other component on this layer. The third layer (also called the user layer) includes the quick deploy component and the main menu component for user interactions. These user layer components can use and manage the components from the feature and system layer. Therefore only components in upper layers can use lower-level components. One of the main goals was to allow for extensions, such as feature additions.

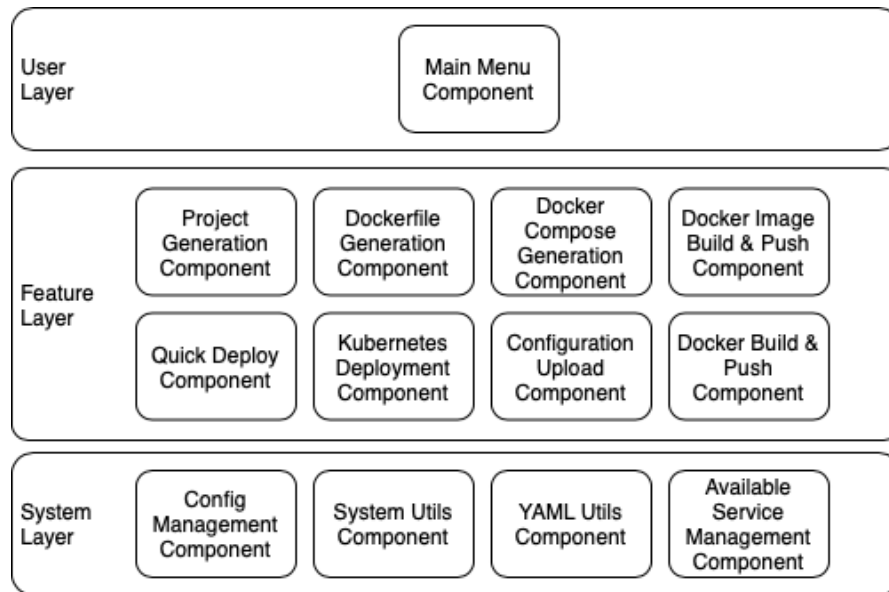


Figure 4.1.: FactDeploy Architecture

Each component in the feature layer is further divided into well-defined components that are similar across all of them. The communication between these components is inspired by the classical MVC (Model-View-Controller) pattern [GHJV95], where separation between UI, business logic and model entities is enforced.

In this thesis, the UI layers are defined by classes with the suffix 'UiCreator'. Their main responsibility is, as the name already suggests, letting users input information necessary for the implemented feature. Classes that are suffixed with 'Initializer' are related to the controller concept in the MVC pattern. It coordinates execution and communication between the UI and the model. The "model" in this software is a class called 'Config', which stores all necessary information. This class is used by all features to enable simple information sharing. The difference to the MVC pattern is that the model class will not directly update the UI, as this functionality was not necessary for the implemented workflows.

Additionally, classes were defined and are suffixed by 'TemplateGenerator'; thus contain the core logic for each feature, at the same time reducing the complexity of the 'Initializer' components by outsourcing functionality to them. Still, they are managed by the 'Initializer' classes. Therefore, the core functionality of the 'Initializer' and 'TemplateGenerator' classes can be capsuled into their associated spaces.

4.2.2. System Layer

Functionality, that is shared across several features, is implemented in the system layer. This layer contains several utility methods data manipulation methods and predefined information. Throughout the project the most common feature used is the configuration loader and the related 'Config' class. These classes are used by all features to save their necessary configuration after a feature is successfully executed. Furthermore, on each successful completion, the configuration is serialized into the 'config.yaml' file via the 'ruaml' library and de-serialized back into the 'Config' file on startup. This feature allows for storage of inputs between runs and reduces inputs overall as previous ones will be used to skip prompts for information that is already available.

External services that are used by Docker Compose and the deployment features have their own dedicated configuration. Since there are multiple services, whereby some may have varying configurations between Docker Compose and deployment definitions, they are stored in a dedicated 'set' with their respective data structures. These data structures contain dedicated definitions for services and databases to reduce the storage of unnecessary information (e.g., a service does not have a default database definition, but a database does). Overall, these sets are also included inside the previously mentioned 'Config' file and are used in the same way. The content of these classes can be concretely seen in Figure [4.2](#).

4. Implementation

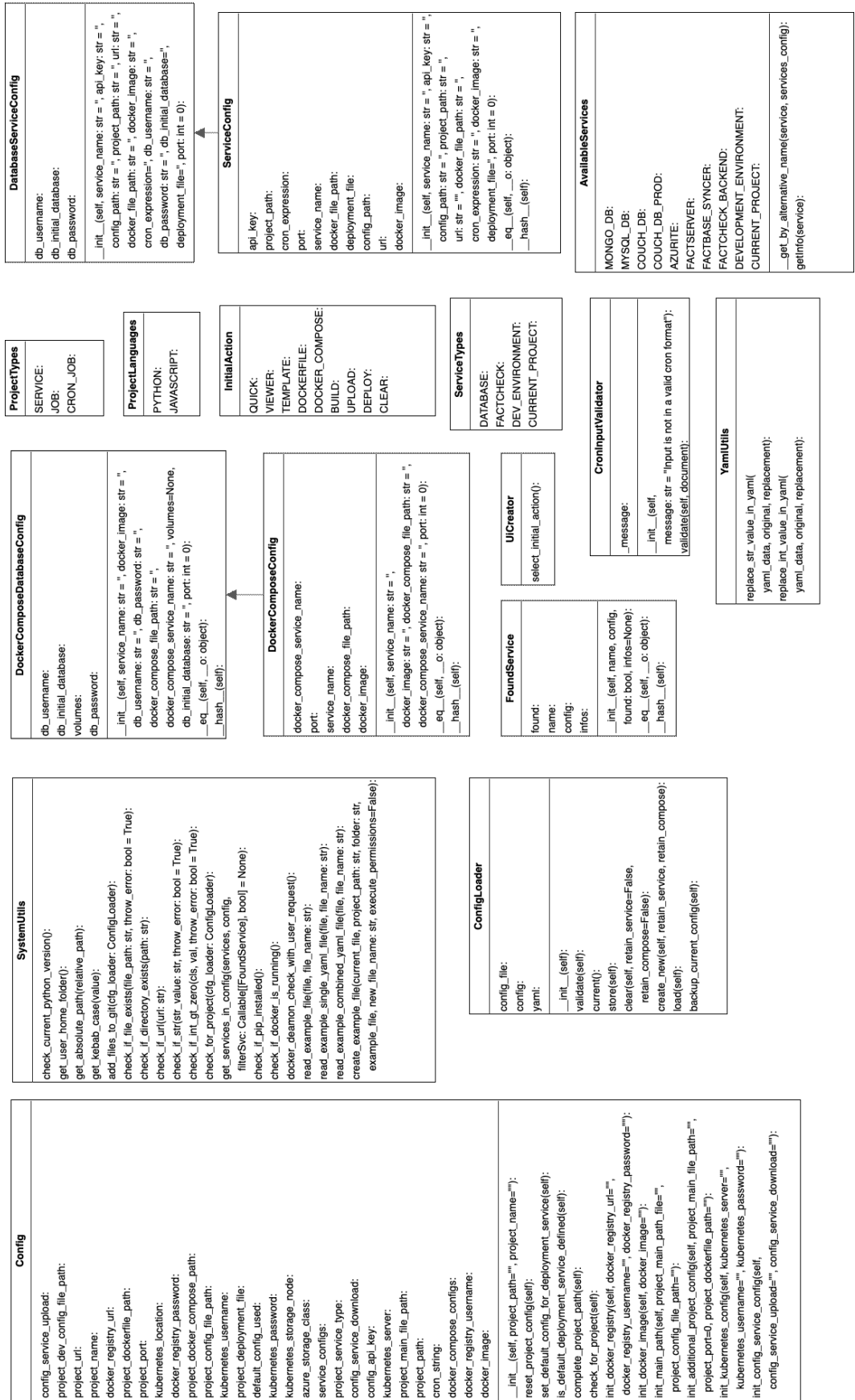


Figure 4.2.: Classes available in the system module.

To define which external services can be used by the system, an enum class called 'AvailableServices' contains a list of all services and default information for each enum entry. The following entities are currently given:

- Databases
 - CouchDB - necessary for the FactCheck-Backend/Factbase service
 - MySQL - necessary for the FactCheck-Backend
 - MongoDB
- FactCheck Services
 - FactCheck-Backend
 - FactCheck-Factbase
 - FactCheck-Factserver
- Development Tools
 - Development Environment (Ubuntu 24.04)
 - Current Project

Each of these services has its own optional default configuration and additional metadata, which is defined by the following fields:

- title - used for displaying names of services in the UI
- kebabTitle - corresponds to the 'kebab-format' and is identical to the unique identifier of each service for easier comparison. Initially only used as the Docker Registry required Docker Images to have their tags in this format.
- type - used for grouping of services currently the following types are available:
 - db - used for grouping database services.
 - factcheck - used for grouping FactCheck services.
 - dev-env - used for grouping development environments (currently contains only the FactCheck development environment)
 - current-project - a special type, only used to identify the currently loaded project.
- file - defines the name of the corresponding docker-compose YAML snippet.
- defaultImage - defines the Docker Image to be loaded by the corresponding docker-compose YAML snippet (not available for the current project).
- deploymentFile - defines the name of the corresponding Kubernetes deployment YAML file (not available for development environments and the current project).

4. Implementation

- defaultPort - default port for each service, only defined when the service supports listening on ports for data.
- defaultUsername - only defined for database services. It is the default username used by FactCheck services to connect to the database.
- defaultPassword - only defined for database services. It is the default password used by FactCheck services to connect to the database.
- defaultDatabase - this field is only defined for database services and corresponds to the name of the database that should be initially created. It also corresponds to the ones used by the FactCheck services and is optional, as in the case of CouchDB, do not support this functionality.

All of these or only a subset of these services, can ultimately be used by each implemented feature. This enables compatibility between features and, to an extent, the whole FactCheck project.

Utility methods are also provided and include several small methods, like extracting the absolute path or checking if the Docker daemon is running. It is important to mention here the 'get_service_in_config()' method and the 'replace_string/int_in_yaml()' methods. The first method allows to find information on several external services quickly. To do this, the information in the 'AvailableServices' enum is compared to a set of configurations (docker-compose or service configurations) and returns the found service, when available, with its default configuration from the enum and the one stored in the related configuration set. Additionally, information on services not found are returned too. This lets features find missing service configurations and add them to their related set. Furthermore, a filter can be added to remove services from the result when not needed in that present context. The concrete structure and content of these classes can be seen in Figure [4.2](#).

The 'replace_string/int_in_yaml()' methods are also important due to the widely used templating feature. Their main feature is to recursively iterate loaded YAML files. When that value, in this case a template pattern, is found in those YAML files, this pattern is then patched with the replacement value. This allows for files, especially deployment-related files, to be customized to user preferences and configurations.

4.2.3. Project Generation

The project generation functionality is regarded as the cornerstone for every other functionality in this project. With this feature users are able to generate a base project that fulfills the best practices for the project structure stated in the related chapter of the deployability metric. Still, users will be able to choose if they want or do not want to generate certain features. The UI component of these features allows users to choose between the following options:

- Project Type - This choice contains options regarding the execution style of the generated project. Currently the types Service, Job and CronJobs are available:

- Service - when choosing the 'Service' option, the generated project and all generated files will be constructed to accommodate endlessly running software, which is useful for server applications.
 - Job - for the 'Job' option a similar approach is taken, although with a slight modification, which is that the generated project will be designed to run only once. Such a project is especially useful for imports/exports or larger conversions.
 - CronJob - the last option 'CronJob' is an adapted 'Job' project for which a special deployment configuration is used. The user can specify via a 'Cron-Expression' a specific repeating time point when a 'CronJob' should be executed (e.g. every day at midnight).
- Folder Structure - a predefined structure will be used to generate the project; these structures are mainly folder structures for source, test and configuration files.
 - File Generation - this includes recommended project files like pyproject.toml, LICENSE, .gitignore and several more.
 - Virtual Environment - when this option is chosen a virtual environment based on 'virtualenv' will be generated and activated for the currently generated project, which allows for better dependency management. As dependencies might interfere with each other if multiple incompatible versions are installed.

In addition to these choices, the user will be asked if they want to use the default configuration for the deployment services. This functionality is included in this step because it is also a foundation for other features and has a high probability of being identical between projects. Selecting this option allows for the omission of several user inputs in subsequent features. These inputs would include the configuration for the connection to the Docker Registry, the URL of the Kubernetes instance and the endpoints for the factcheck-config-service to update or download configurations. If these default configurations were explicitly not chosen the related inputs will reappear for features where that information is required.

If the user wants to use already existing projects, there is the choice to supply the most essential parameters and use these to work with the other features provided by FactDeploy.

The implementation of these features is done as seen in Figure [4.3](#) with three different classes, where the ProjectInitializer class is responsible for managing the UI and the business logic. This allows for complete separation between these two components, resulting in better modularity and flexibility.

4. Implementation



Figure 4.3.: Classes available in the project module.

One feature that has not been mentioned yet is the backup feature. The user will only be able to access this feature when invoking the project generation functionality,

provided that a previously generated project has already been stored. Here, the user can choose which parts of that previously defined project's configuration should remain after the backup. This includes the docker-compose configuration and the external services configuration (such as metadata for the FactCheck projects). The complete configuration file will then be copied and stored with this pattern as filename: `<project-name>-backup-<current-date>-config.yaml`, which in turn can be restored by renaming the file back to `'config.yaml'`.

4.2.4. Docker Compose Generation

Development, without connection to the internet or to university servers, requires development directly on the workstation of the developer. To support these kinds of actions, a custom Docker Compose file generator was implemented. The detailed structure for this generator can be found in Figure 4.4. To make it work, as the name already implies, a Docker installation on the developer's system is required. The main goal is to have a completely local installation of all necessary services in order to run the FactCheck framework. This requires that the FactCheck projects, databases, development containers and tools are available to the developer.

Allowing customization and modularity in this process is essential, as integrating different external services has own requirements to work in the Docker environment. To do this, templating and the combination of YAML snippets are essential. In this context, snippets are essentially parts of a YAML file that fully defines their purpose and contains all necessary information regarding their functionality. By combining user-selected features with the related snippets that are selected, a program can create a customized docker-compose.yaml file for the user. At present there are snippets for the FactCheck services, databases such as MySQL, CouchDB, MongoDB, the development environment and the currently loaded project available.

Essential configuration options, such as database services, are also configured via environmental variables. This includes, for example, the username, password and name of the initial database. To allow users to customize these values when initializing their docker-compose.yaml file, templating was used. Templating is implemented by recursively iterating a YAML file and replacing values in these files that correspond to a specified pattern. Currently implemented are the following patterns:

- «project-name» - is replaced by the name of the currently selected user project
- «image» - is replaced by the default image defined in the AvailableService class
- «db-username» - is replaced by the user-specified database username for the corresponding database
- «db-password» - is replaced by the user-specified database password for the corresponding database

4. Implementation

- «db-initial-database» - is replaced by the user-specified name for the initial database (if the database supports this feature).
- «port» - is replaced by the port of the currently selected user project

Additionally, the user can select to use default values. These default values correspond either to the ones defined by the developers of the selected service or to values used by the FactCheck project. Although these values may appear arbitrary, developers can, if necessary, update them in their projects at all times, so they should be considered reference values and are not required. All default values in the system have a single source of truth, which is the AvailableService class. Furthermore, the system's set of services defines all associated default values.

The implementation of the FactCheck part of the generator is based on two different concepts, both of which are viable but require different prerequisites to work. The first one is a fully local installation, which can be selected in the UI using the 'Locally' option. This option only requires that the developer has already downloaded FactCheck projects via their dedicated Git repositories. In each of these projects, a separate docker-compose.yaml file is available. Originally these files were/are used by the 'FactDeploy Docker' project, which statically references these docker-compose.yaml files in its own docker-compose.yaml file. This means that to execute it, all services must be available and in a specific location. This generator takes a more dynamic approach and allows users select the docker-compose.yaml from whichever project they want, then loads it into memory and adds it to its internal docker-compose.yaml file. That means that not all FactCheck services must be selected, a subset is also possible. The main advantage of this approach is that FactCheck services can be customized by each developer for their specific needs. Although this approach allows for high customizability, it has however a major disadvantage, namely the building process. On the initial initialization and on each change to a FactCheck service the specific service has to be rebuilt locally on the machine, which can take several minutes and use a substantial amount of processing resources.



Figure 4.4.: Classes available in the Docker-Compose module.

4. Implementation

To circumvent these issues, a second approach is available. This approach is called "Prebuild" and can be selected via the same name in the UI. The main difference from any other approach is that the FactCheck services do not have to be previously available on the system and do not have to be manually built on the system. Allowing such behavior is enabled by providing pre-built Docker Images for FactCheck services, which are stored in the FactCheck Docker Registry via the "dockerregistry-lab.cs.univie.ac.at:443/factcheck-<service-name>-default:latest" tag. All FactCheck services follow the same pattern, thus "<service-name>" is simply replaced by the service's corresponding name. When using this approach, a default configuration is supplied via configuration injection, which is applied at startup. To customize this configuration, a modified configuration can be uploaded to the factcheck-config-service, and by supplying the new API-key to the service the new configuration will be used on the next startup. Furthermore, the factcheck-config-service URL can be customized, if desired. Both the API-key and the URL are supplied to the service via environment variables and can only be accessed from within the Docker Image.

Now, to allow developers to work as close as possible to a production system, a development container is necessary. The functionality and structure thereof will be defined and examined in Section [4.6](#). To give a short introduction, at its core, it is a customized, endlessly running Ubuntu 24.04 image with additional common software development tools installed. The idea behind this image is that developers enter it via the 'docker exec -ti <container-name> bash' command and execute their project there in order to get closer to the development or production environment.

For faster generation and repeated use cases, the selected options are stored inside the previously defined 'Config' class and its corresponding file. All of these options are stored inside a dedicated DockerComposeConfig and a derived DockerComposeDatabaseConfig class. Both of these classes store the following information:

- service-name - stores a uniquely identifiable name for a service in the system
- docker-compose-file-path - stores the docker compose file path for a service
- docker-compose-service-name - stores the name of the service from the externally loaded file.
- docker-image - stores the Docker Image tag which should be used for a particular service.
- port - stores the port from which the service must be accessible from outside Docker.

The DockerComposeDatabaseConfig stores additionally the following information:

- db-username - database username for connecting to the database
- db-password - database password for connecting to the database
- db-initial-database - database name for the intially created database

- volumes - mapping of the internal database storage to an external folder to allow for persistence of the data between shutdowns.

Storing this information is done inside of a 'set' in the 'Config' class. This ensures that, when generating docker-compose.yaml files, only one definition of the same service is accessible. This is crucial because different user interface interactions have the potential to either update or create new configuration information. If we were to use a 'list' or 'array', we would store data twice, as they lack a mechanism to verify if an object already exists in their data structure. To allow the 'set' to work properly and to check the existence of an already defined object, the incoming DockerComposeConfig and DockerComposeDatabaseConfig are compared via their service names, as any other information is subject to change in the creation process. The service name was also chosen because it is unique throughout the system, and when adding services to FactDeploy, they must adhere to this convention, as otherwise their data would, in the worst case, be overwritten or duplicated.

4.2.5. Dockerfile Generation

Dockerfiles are essential for the deployment process of projects generated with the FactDeploy tool, as those are the cornerstone for any deployment definition. To be precise, the generated Docker Image comes from these files and is loaded by Kubernetes to be executed inside a pod and then made available throughout the system.

The initial plan for these files to be generated was similar to the ones generated by the docker-compose.yaml file generation, where several snippets can be combined to generate a complete and correct file. While conceptualizing this process, the drawbacks of the initial idea came to light and this approach was therefore abandoned. First of all, the generated Dockerfiles can differ from use-case to use-case. For example, when using them locally on a developer's PC, the aspect of faster building times for repeated testing is important. Production and development deployments on the other hand require Docker Images to be reduced in size, as a large number of deployments can fill up a large amount of space rather quickly. While these use-cases can be accomplished by building rather complicated UIs, the final drawback was the order and pre-/post-requisite in which each element would have to be added, as snippets would rely on each other and furthermore have to be adapted when multiple building processes happen at the same time. This can happen, for example, when builder and runner images are used, which are very common for production deployments. As finding a path for each permutation of snippets that would work together and building an associated UI would be out of scope for this thesis, another path to allow customizability of Dockerfiles was chosen.

4. Implementation

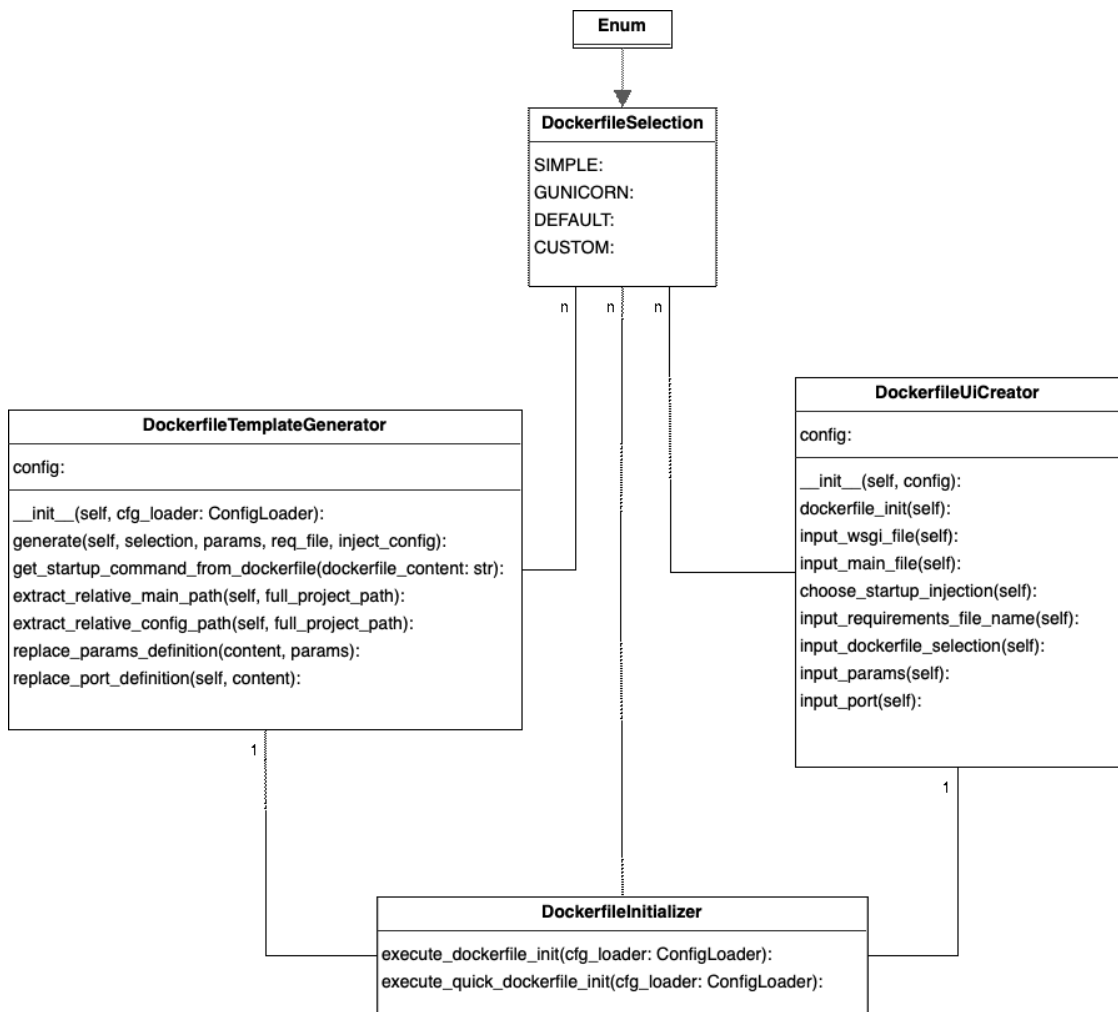


Figure 4.5.: Classes available in the Dockerfile module.

As the composition of snippets was not possible for Dockerfiles another approach was introduced. A selection of Dockerfiles will be provided for the developer, each of which has its own advantages, drawbacks and use-cases that are prepared for the previously introduced templating approach. Because these are common text files, the mechanism of recursively reading YAML files and replacing information accordingly is unnecessary. A simpler approach of directly using string replacement is applied, which reduces complexity and syntax errors. The detailed class structure to implement this workflow can be found in Figure [4.5](#).

Thus, values for the following templating patterns can be supplied by the developer:

- <project-name> - name of the currently, by FactDeploy, loaded project.

- `<main-file>` - path to the main file of the loaded project.
- `<params>` - optional parameters which can be supplied on startup and separated by a " " string.
- `<port>` - defines on which port Docker should allow the host system to access a container and defines on which port the program shall listen (only when using gunicorn image)
- `<requirements-file>` - defines the name of the requirements file to be used for building this image (useful when having different requirements in different environments)

These templating patterns are used throughout the available Dockerfiles. Currently, the following Dockerfiles are selectable via the UI:

- Default - the simplest Dockerfile which contains only essential development and runtime tools. This file is particularly useful for local development with repeated building processes.
- Quick - specialized Dockerfile for the "Quick-Deploy" feature, which will be described in the dedicated Section [4.2.10](#).
- Gunicorn - specialized Dockerfile for working with the Gunicorn WSGI HTTP server, especially useful for high performance applications and general production deployments.
- Custom - specialized Dockerfile for custom execution of a Python project, where a dedicated script for running developer software can be supplied. It is useful for projects that run only once or at specific times (Jobs / CronJobs) and with specialized setup workflows.

Except for the "Default" Dockerfile, all other options are based on the "builder" and "runner" image concepts. The "builder" image is used during the build process to compile or package the application code. In the case of Python builds, it is commonly used to install dependencies and development tools. Often, a dedicated compilation step is also defined, but it is not required in this case as it is intended for other programming languages. This is due to the fact that Python is an interpreted programming language and is therefore executed line by line in runtime [\[Fou25\]](#). Therefore, the main goal for defining this Dockerfile is to keep the builder's image as lightweight as possible while including everything needed for the application's building process.

A "builder" image must have a specific structure. This image is based on Ubuntu 24.04 and is intended for building a Python 3.12 project. So it installs Python 3.12 along with development tools, sets up a virtual environment and installs project dependencies specified in a requirements file. Additionally, it avoids interactive prompts during the build process by setting the "DEBIAN_FRONTEND=noninteractive" environment variable. The image includes several common repositories, such as "ppa:deadsnakes/ppa" and "universe", as well as additional though optional packages such as gcc, libcurl, libssl

4. Implementation

and libmysqlclient for building and running the project. In the final step, the virtual environment's location is stated to be in the `"/home/<project-name>/venv"` folder and the Python dependencies are installed within it.

The "runner" image, on the other hand, is the final image that will be used to run an application in a production or runtime environment. It usually contains only runtime dependencies and artifacts generated during the build process and excluding unnecessary build tools and intermediate files. The goal, in this case, is to also keep the runner's image minimal in order to reduce its size and potential security vulnerabilities. Using separate builder and runner images helps to achieve a clean separation between the build environment and the runtime environment. It allows to build an application in a controlled environment with specific tools and dependencies, and then package only the necessary artifacts into a smaller and more secure image for production.

In this case, the "runner" image is based on the same base image, namely, Ubuntu 24.04 and Python 3.12 installed for executing the application. The virtual environment from the builder image is copied into the runner image. The project code is then copied into the `"/home/<project-name>/code"` directory and the working directory is set accordingly. The image exposes a specified port and sets environment variables according to the selected Dockerfile option. Furthermore, the environment variable `"PYTHONUNBUFFERED"` is set to 1 to ensure that messages always reach the console. Additionally, the build-time and environmental variables `CONFIG_ENDPOINT` and `API_KEY` are set to provide support for the factcheck-config-service, if wanted by the developer. In the penultimate step the virtual environment is activated and the final step is the definition of a default command to execute a startup script when the container starts.

When selecting the "Default" image both the runner and builder images will be simplified and combined into one. It is based on the Python 3.12 base image rather than the previously used Ubuntu 24.04 image. It installs additional development tools, sets up a virtual environment and installs project dependencies specified in the requirements file. It sets the working directory to `"/<project-name>"` in which it copies the requirements file. Additionally, it ensures that messages always reach the console by setting `"PYTHONUNBUFFERED"` to 1. The requirements file is then installed, and the project's source code is copied to the `"/<project-name>"` directory. Finally, the user-defined main file is executed via the `CMD` command.

4.2.6. Configuration Upload

The idea behind this feature is to provide support for externalizing the configuration of multiple applications at once, as well as being agnostic of the programming language and framework features used.

This means that the user does not have to manually upload their production configuration and then update their API-key to accommodate their configuration injection. Therefore, when using the FactCheck services and the Dockerfiles provided by FactDeploy, both will

have an uploaded production configuration and an up-to-date API-key. The developer will only need to specify the location of their production configuration once. Once they update it, they can simply use the configuration upload feature to ensure the selected services receive the latest version of the configuration. Depending on the configuration injection method used by a service, the associated deployment has to be restarted, redeployed, or, in the best case, automatically loaded with the new configuration.

This feature is implemented by running a dedicated factcheck-config-service instance and communicating with it via REST calls. In this case the factcheck-config-service is running directly in the same Kubernetes instance as all other services, however in the "default" namespace, which is only accessible to administrators. The detailed implementation of the factcheck-config-service will be described later in Section 4.3 but the class structure can be seen in Figure 4.6. In this section, only its use in the FactDeploy tool will be discussed.

The feature itself is dependent on the user's selection of services and configuration files to upload. To achieve this, the developer will be able to select a subset of the services listed in the AvailableServices classes. This selection is done via InquirerPy and includes only the FactCheck Services and the currently loaded project. Configuration upload for other services, such as CouchDB, is not supported because their implementation does not support any kind of configuration injection and must therefore be done via environmental variables or custom Docker Images supplied by a developer. For the selected services the user must supply the path to the production configuration, define the upload and download endpoints of the factcheck-config-service and specify the location of these projects. All of this information must be given once and will be reused in subsequent runs.

4. Implementation

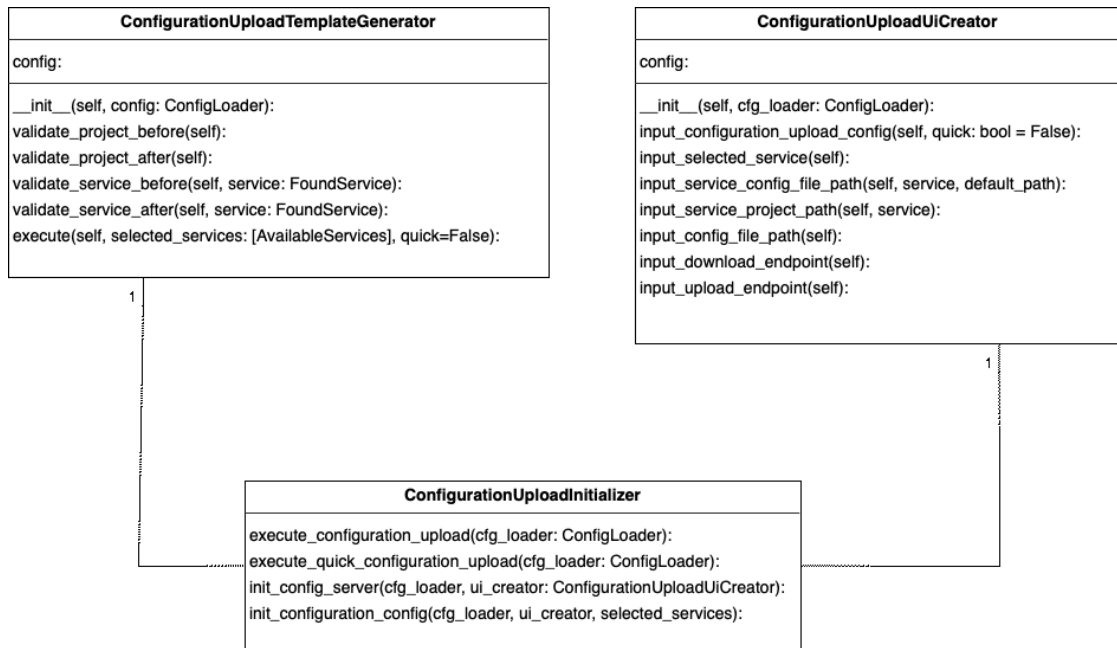


Figure 4.6.: Classes available in the configuration-upload module.

The configuration upload is then performed via a simple REST call to the factcheck-config-service for each selected service. To accomplish this, a simple protocol is defined that relies on the following HTTP request and its response:

- HTTP Request
 - Type: POST
 - URI: <https://kubclient3.cs.univie.ac.at/config/add>
 - Content-Type: text/plain
 - Body: content of the configuration file in plain text.
- HTTP Response
 - Content-Type: application/json
 - Body: a JSON structure with a key named 'api-key' and a value containing the api-key as a string.

The information contained in the response, the API-key, is then stored via the 'Config' class and later used in the deployment process. This API-key is unique for every configuration instance and can therefore be used to retrieve the uploaded content again. To use the aforementioned API-key the services have to call the download endpoint with an HTTP

GET request with an attached "x-api-key" header. As a result, the response object will contain in its body the complete configuration in plain text. This plain text can then be used to replace the internal configuration at this point being the development configuration, with a production configuration.

4.2.7. Building and Pushing Docker Images

Building and pushing Docker Images is the basis for every deployment made with FactDeploy. To allow this feature to work correctly, some basic prerequisites are required. Basically, Docker installation and a correctly defined Dockerfile are sufficient thereby disregarding any programming language constraints. This allows FactDeploy, with the Docker plugin for Python, to build any sufficiently defined project. In this thesis this plugin was therefore used to build and also push the FactCheck services and the currently loaded project to the private Docker repository created for the FactCheck project.

This feature also relies on the previously described service selection to implement this behavior. The set of services a developer can select is also limited to the current project and the FactCheck services. Support for other services would be possible if the AvailableService class and UI were expanded accordingly. In the context of this thesis these services were chosen as they are the only ones relevant to the FactCheck project and benefit from being built locally instead of downloading those from a Docker registry. For a detailed class structure see Figure [4.7](#).

The developer has to provide several parameters for this feature to work correctly. This includes the following list of parameters, which the developer can supply via the user interface:

- Automatic Version Upgrade - this allows developers to automatically increase the patch version number by one; otherwise the version number stays the same.
- Enable Docker Cache - allows developers to enable or disable the Docker Cache if needed.
- Kubernetes Username - will be added to the image tag to better distinguish between build services.
- Docker Registry URL - the Docker Registry's public endpoint for uploading or downloading images.
- Docker Registry Username - username for Docker Registry access.
- Docker Registry Password - password for Docker Registry access.
- Relative Path to Dockerfile - the path to the Dockerfile that should be used for building the Docker Image.

In addition, the developer must decide which services to build and push. This lets developers skip some processes if not needed in their use case or for debugging.

4. Implementation

With this information available, the local Docker instance will be logged in with the specified username and password in the Docker Registry. This allows the developer to upload or download images from the Docker Registry. Furthermore, the defined information is used to create a so-called "tag", which uniquely identifies a Docker Image.

The creation of this tag uses several sources, such as the Docker Registry name, the service name (in kebab-case format), the Kubernetes username and the version number. The final tag will be in the following format:

- Pattern: `<docker-registry-url>/<service-name>:<kubernetes-username>-<version-number>`
- Example: `dockerregistry-lab.cs.univie.ac.at/factcheck-backend:exampleUser-1.0.0`

Several resources can be used to generate the "`<version-number>`". When a version identifier is found in the current service's `pyproject.toml` file, it can be used as a basis of truth. Another version identifier could be specified by the developer while building an image, resulting in another source.

Given the possibility of both version numbers being correct, it is necessary to select one. If a version number is found in a `pyproject.toml` file, it has a higher priority than the Docker Image tag, as these can be inconsistent when, e.g., images are deleted. Otherwise a Docker Image tag is chosen, using the version specifier in the tag.

When using the configuration file or the Docker-Image-Tag option, the found version identifier can be increased based upon the previously mentioned version increment mechanism, if wanted by the developer. As several users may have the same version number for a service, e.g., `factcheck-backend`, using the username as a prefix is mandatory. Otherwise, users could override each other's generated Docker Images in the Docker Registry.

The building process itself is straightforward, where the combination of the path to the project, the path to the Dockerfile and the newly generated tag are supplied to the image building API provided by the Docker dependency. Here the activation of the Docker cache comes into play. As building images can be resource intensive, enabling it allows for reusing of already available Docker Image layers of other images. With this parameters in place, the building process is started. The progress of the complete building process including all warnings and error messages, is displayed to the developer when the action is complete.

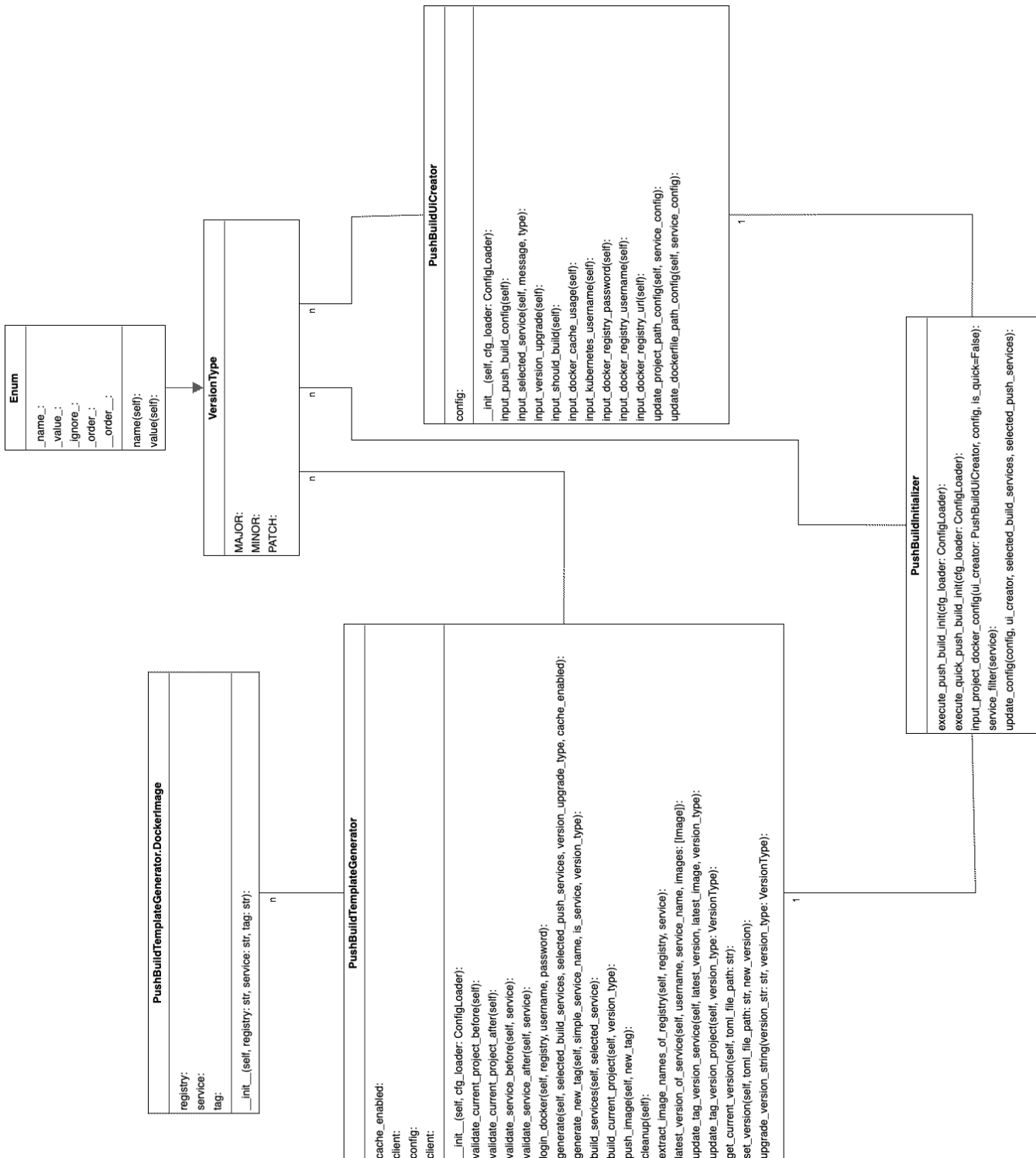


Figure 4.7.: Classes available in the push and build module.

4. Implementation

The last step in this whole process is to push the images of the selected service to the Docker registry. This step is simply done by specifying the complete Docker-Image-Tag as a parameter for the Docker Image push API. The uploading process will be streamed directly and displayed on the console so that the progress can be viewed in real-time. After finishing the build, the built images are available for download and used by Kubernetes in the deployment process.

4.2.8. Kubernetes Deployment

On to the final step in the entire process, the software artifacts are deployed to the Kubernetes instance. This section evaluates the deployment process and describes the dependencies used.

First of all, the necessary prerequisites have to be identified. For all features of the FactCheck services to work, the selection of services to be deployed is extended. They now include not only the FactCheck services themselves but also the available databases (MySQL, CouchDB and MongoDB), as well as the currently loaded project. If any configuration for these services is still required, the user will be asked to supply it (usernames, passwords, etc.). Additionally, the user will be asked to supply the Kubernetes-related configuration (server url, username, password), indicating if the deployment files should be stored in their respective project, and that type the current project has (service, job, cronJob). When selecting a "cronJob" the user will also be asked to supply the desired cron string.

With this information available the deployment process can start. The whole process happens inside the "KubernetesConnector" class. When the user starts a deployment, a connection to the Kubernetes server has to be established first. As the standard port (6443) for Kubernetes instances is not publicly available and only ports 80 (HTTP), 443 (HTTPS) and 22 (SSH) can be used to access the university server, the connection process has to happen in two steps.

In the first step, a connection to the server is established via an SSH tunnel. When the connection succeeds, the current user's configuration file for connecting to the Kubernetes instance is downloaded. This YAML file contains the necessary certificates, keys and information for further connections for each user located in the "\$HOME/.kube/config" directory. This file is then loaded into an instance variable, which will be used in the second step. Following this, the SSH tunnel is closed.

Now to the second step: Another SSH tunnel has been established here, but this time directly to the Kubernetes instance on the server, as the previously downloaded configuration is now available. With this configuration the Kubernetes Python API is initialized that connects to Kubernetes, it is then used to execute API calls inside the `execute()` method. To make development easier and have better separation of all functionalities, the `execute()` method is an "abstract method" that has to be implemented by a derived class. This derived class is the "KubernetesTemplateGenerator" and will, upon instantiation,

connect to the Kubernetes server. All of the workflow's functionality is then placed inside the `execute()` method implementation.

Before introducing the implementation details of the deployment workflow, it necessary to mention the Kubernetes Python API and its abstraction class. When using the Kubernetes Python API directly, each method call corresponds to an HTTP call to the Kubernetes REST API. While this approach allows the developers of the API to implement new features quickly, its usability suffers. To increase its usability an abstraction class is introduced called "KubernetesApi". It contains all necessary functionality for supporting the repeated deployment of basic Kubernetes objects, which include the following types:

- Ingress: for exposing software HTTPS endpoints to the public.
- Service: for inter-pod communication on the Kubernetes Server.
- Deployment: for the deployment definition on how to run endlessly running services.
- Job: for the deployment definition on how to run a pod once
- CronJob: for the deployment definition on how to run a pod on a specified repeated pattern.
- PersistentVolume: for defining a specific storage space to be available to pods.
- PersistentVolumeClaim: for defining which persistent volume should be used by a pod.

This abstraction is necessary because each object must be created or, if it already exists, replaced. It requires selecting an object and an action (create or replace) to work. This is necessary because the Kubernetes Python API has a dedicated method for each object and action. To do this, a YAML file with the deployment information needs to be supplied to the class and specifically to the `"apply_yaml()"` method. This file contains the object type, which will be used to choose the correct "create" action for that specific object. In turn, the create action checks if an already defined object is available and either creates a new object or replaces an old one. Furthermore, all exception handling and response de-serialization are handled. With all that in place, the "KubernetesApi" can easily deploy any supplied Kubernetes deployment file without the caller classes knowing about the logic behind it. The detailed class structure for the implementation of this workflow can be found in Figure [4.8](#). This enables repeated deployments of the same Kubernetes object type in a simplified form for the most common use cases.

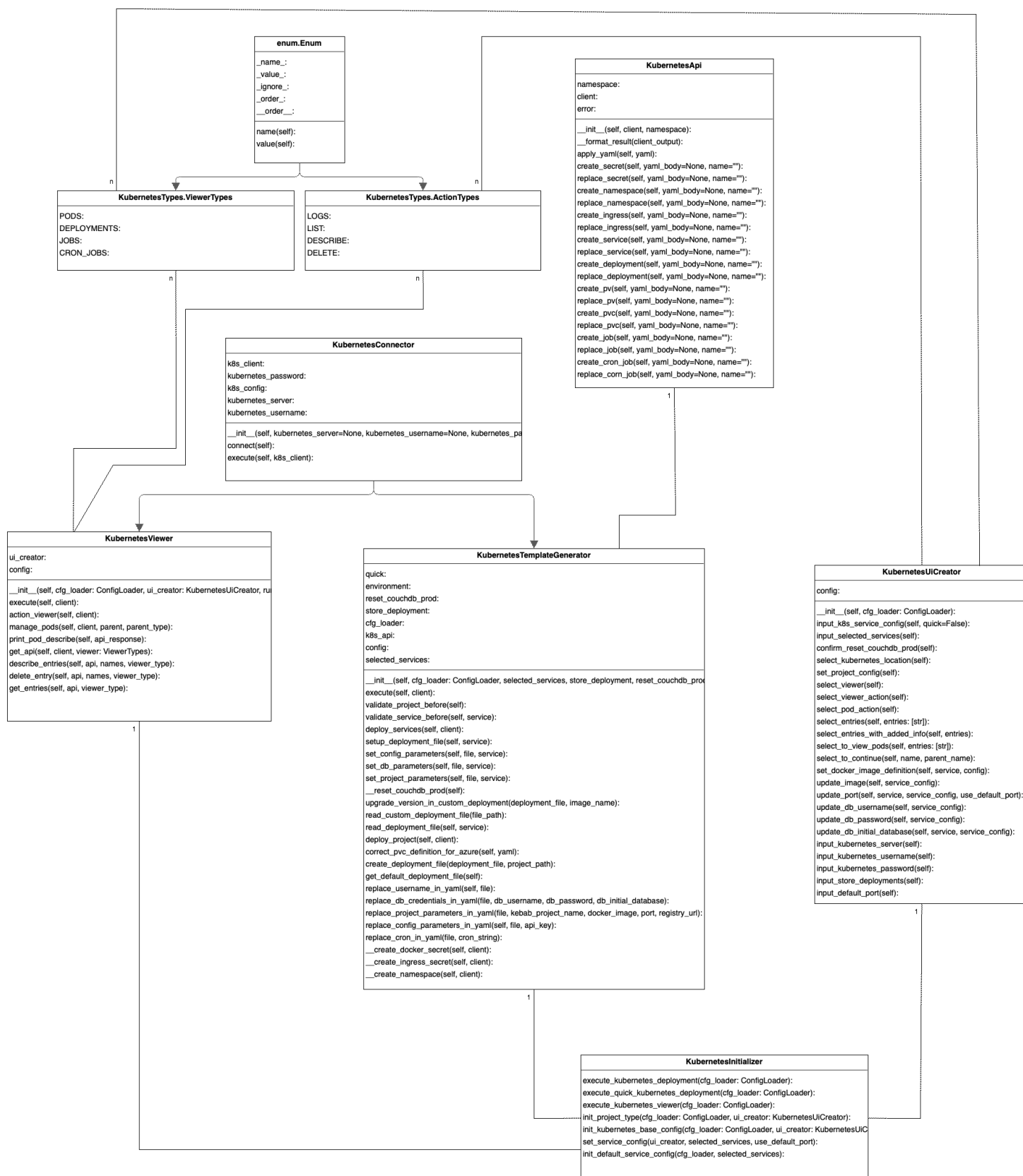


Figure 4.8.: Classes available in the Kubernetes module.

These deployment files have to conform to a specific structure before they can be used for a software deployment. These structures can vary widely depending on what kind of software and what functionality is deployed. They all have in common being based on the available Kubernetes objects. To establish a common base, predefined structures for the most common use cases and software related to the FactCheck services have been defined. This includes patterns for Services/Jobs/CronJobs, MySQL/CouchDB/MongoDB databases and the FactCheck services. To customize these structures the previously mentioned templating approach is used, which replaces strings and integers in YAML files that are annotated with the '«' prefix and the '»' postfix. The following list shows the set of replaced patterns:

- «username» - is the username used for accessing the Kubernetes server, also is used for distincting all Kubernetes objects by adding it as a prefix to all names. Furthermore, it is used to define the namespace for all these object.
- «service-name» - is simply the combination of the Kubernetes username and the postfix "-service".
- «db-username» - only used when databases are deployed and sets the corresponding database username.
- «db-password» - only used when databases are deployed and sets the corresponding database password.
- «db-initial-database» - only used when databases are deployed and sets the corresponding database that will be created on first instantiating.
- «project-name» - is the kebab case name of the currently loaded project.
- «config-server» - is the URL to the factcheck-config-service.
- «api-key» - is the API-Key generated by the factcheck-config-service for downloading the corresponding production configuration.
- «image-name» - is the Docker Image tag of the to-be-deployed software.
- «cron-string» - is the cron string for timing the execution of a software, only used when deploying a 'CronJob' object.
- «port» - is the port on which the software should be accessible inside the Kubernetes instance.

The basic structure for each type of deployment is defined by the following list of deployed software products and their recommended Kubernetes objects:

For user defined projects and FactCheck services:

- Type: Services - is responsible for running a continuous software product
 - Ingress - allows access to the software from an external request (e.g., a browser).

4. Implementation

- Service - allows for communication between pods inside of Kubernetes.
- Deployment - is responsible for creating pods and executing the associated software.
- Type: Job - already contains necessities to create and execute a pod.
- Type: CronJob - already contains necessities to create and execute a pod, but only in a specified time interval.

For all supported databases:

- PersistentVolume - defines an available storage space for pods.
- PersistentVolumeClaim - defines which pods are allowed to access a specific storage space.
- Service - allows for communication between pods inside of Kubernetes.
- Deployment - is responsible for creating pods and executing the associated software, in this case a database.

It is important to note that the Ingress object had to be supplied with special parameters to allow access from external resources while still providing namespace separation inside Kubernetes. A more detailed explanation will be given in Section [4.8.2](#) on why this is necessary and what disadvantages it has. For now it is important that all requests from outside will be routed to the service and rewritten so that each request will be correctly interpreted by the service without knowing about any prefixes. In most cases prefixes are defined as follows: `"/<username>/<service-name>"`. This means that requests coming from an URL, e.g., `"/<username>/factcheck-factserver"`, will be rewritten to just `"/"` for the service. This allows services to be accessible from the outside, whereby every developer can access their frontend via the URL with a prefix while the service itself can be implemented as it would be running on the base URL, e.g., `"/"`.

In addition, there are several deployment files for FactCheck services with the 'default' prefix that use pre-configured deployment files. This configuration includes the image tag of a pre-built FactCheck service, the URL to the factcheck-config-service and the API-key to its default production configuration. These default deployment files are only used if there got no custom Docker Images of the FactCheck services built by the developer. To access all of these images, a Docker Registry secret file is also deployed, which contains the developer's credentials to the Docker Registry. This is necessary, as only verified developers should be able to access their Docker Images and Kubernetes also needs to access them when deploying software artifacts.

With this in place, the overall deployment workflow can be examined. When selected by the user, the first step will be to deploy the currently loaded project. In this case a default deployment file based on the project type (Service/Job/CronJob) will be loaded into memory and a kebab-case name of the project name will be generated. With these two parameters available the templating mechanism will be activated, which will replace

all necessary patterns in the loaded deployment file with their actual values. The next step, storing the deployment file inside the project is optional and will be covered later in this section due to its implementation challenges. However, in the final step, the Docker Registry Secret and the current project file will be deployed via the "KubernetesApi" by supplying the loaded deployment YAML files.

A similar approach to the deployment of the currently loaded project will be applied when deploying additional services (fact-checking services, databases, etc.). The main difference here is that, before any deployment can take place, the configuration of these services will be loaded via the 'Config' class and used to configure the deployment. Furthermore, an additional templating step for databases will take place that configures the credentials and the initial database. Following these additional steps, the "KubernetesApi" will initiate the common deployment process. As previously mentioned, the dedicated deployment file for each service will be loaded and supplied to this class.

The storage of deployment files inside a project was a non-trivial implementation. Normally, the creation, loading and storing of a YAML file is a simple task; in this case, there were some major issues regarding the handling of the file content and its storage. For these kinds of tasks, the "ruamel" library was used which specializes in handling interactions with YAML files. While the "ruamel" library handled the storage of the previously configured templating file perfectly, the resulting file was barely readable, making it unusable for the intended use case. Which is, in this case, that a developer can adapt their deployment file later to their own software. To make this process useful, several additional steps must be taken. First of all, the configured templating file, which is in an internal ruamel data structure, had to be transformed into a JSON-formatted string. This was necessary as ruamel adds class annotations when writing the internal data structure to the disk. These annotations were removed when this conversion step was applied. In the next step, this JSON-formatted string was then converted into an actual JSON object. This step was chosen as ruamel can convert JSON objects directly to YAML without adding additional class annotations when writing it to the file system. After this process, additional features such as a special 'flow-style' and the 'preserve quotes' option were disabled as they made unintended changes to the format of the stored file. In the final step, the now correctly formatted deployment YAML file is stored in the "deployment.yaml" file in the project's root folder. This file will then be used in future deployments to allow the aforementioned adaptation by the developer.

4.2.9. Kubernetes Viewer

Based on all currently examined features, the process concentrated on resource creation and deployment. While this is necessary for the whole framework to be useful, the developer currently has no way to inspect if their actual deployed software artifact is running correctly or at least starting up. To enable this, the Kubernetes Viewer was introduced.

Implementing this feature meant that the previous MVC pattern, which was used in

4. Implementation

all other features, was no longer suitable. The process of loading and displaying data relies on network requests, which alter the display of UI elements related to Kubernetes objects, their metadata and their content. Therefore, the UI library, in this case the class responsible for UI elements 'UiCreator', had to be included in the execution logic.

With this architectural change in place, the display of Kubernetes objects, actions, navigation and metadata was implemented. The connection to Kubernetes was done in the same way as in the "Kubernetes Deployment" section, where the current class inherits the "KubernetesConnector" class and implements the "execute()" method. In this method, the complete logic for displaying data and executing actions will take place.

The creation of the UI was inspired by the Kubernetes object structure, with the intention of familiarizing developers with Kubernetes. This means that the developer first displays the UI regarding the top layer of all Kubernetes objects. This layer consists of a selection for the service type which, in this case, includes "Deployments", "Jobs" and "CronJobs". When a developer selects one of these options, a second question comes up that asks for the action that should be performed on these object entries. Currently this includes "List", which, as the name already states, gives a list of all entries of this type. Then there is also "Describe", which displays a JSON structure with a detailed description of the current state and events that have happened to a selected entry. At last, there is "Delete", which deletes a selected entry. When an action is selected, a list of all entries of that specific Kubernetes object type will be displayed and the developer can then select the entry on which the action should be performed.

The "List" action also allows the selection of an entry to perform another action; in this context, the actions "Logs", "Describe" and "Delete" are available. The "Describe" and "Delete" actions have the same functionality as for the previously stated object. The "Logs" action allows developers to view the current log output of their software. After selecting the action to be executed, a list of "Pods", which are the smallest accessible units in Kubernetes, is displayed. These pods contain the running containers, which in turn run the developer's software artifacts. Once a pod is selected the previously selected option will be executed.

Simply ending the FactDeploy program after all selections are made would result in having the developer open the program several times when intending to view several different objects or apply several different actions. Therefore, a simple navigation feature was introduced, which easily allows the user to repeat the last action on the same layer or to return the user to an upper layer. Now the developer can navigate through all Kubernetes objects and apply actions without restarting the program.

4.2.10. Quick Deploy

Implementing a "Quick Deploy" feature allows for a fast and reliable release process, while reducing the deployment barrier for projects to the minimum required by FactDeploy. To do this, several features were combined in a single process with adapted workflows.

Additionally, several assumptions were made for each of the services on what a potential developer must input in this process and what can be provided by the system to give a better user experience. This section will investigate these assumptions and their implementation. Furthermore, use-case examples will be provided and the limitations will be disclosed.

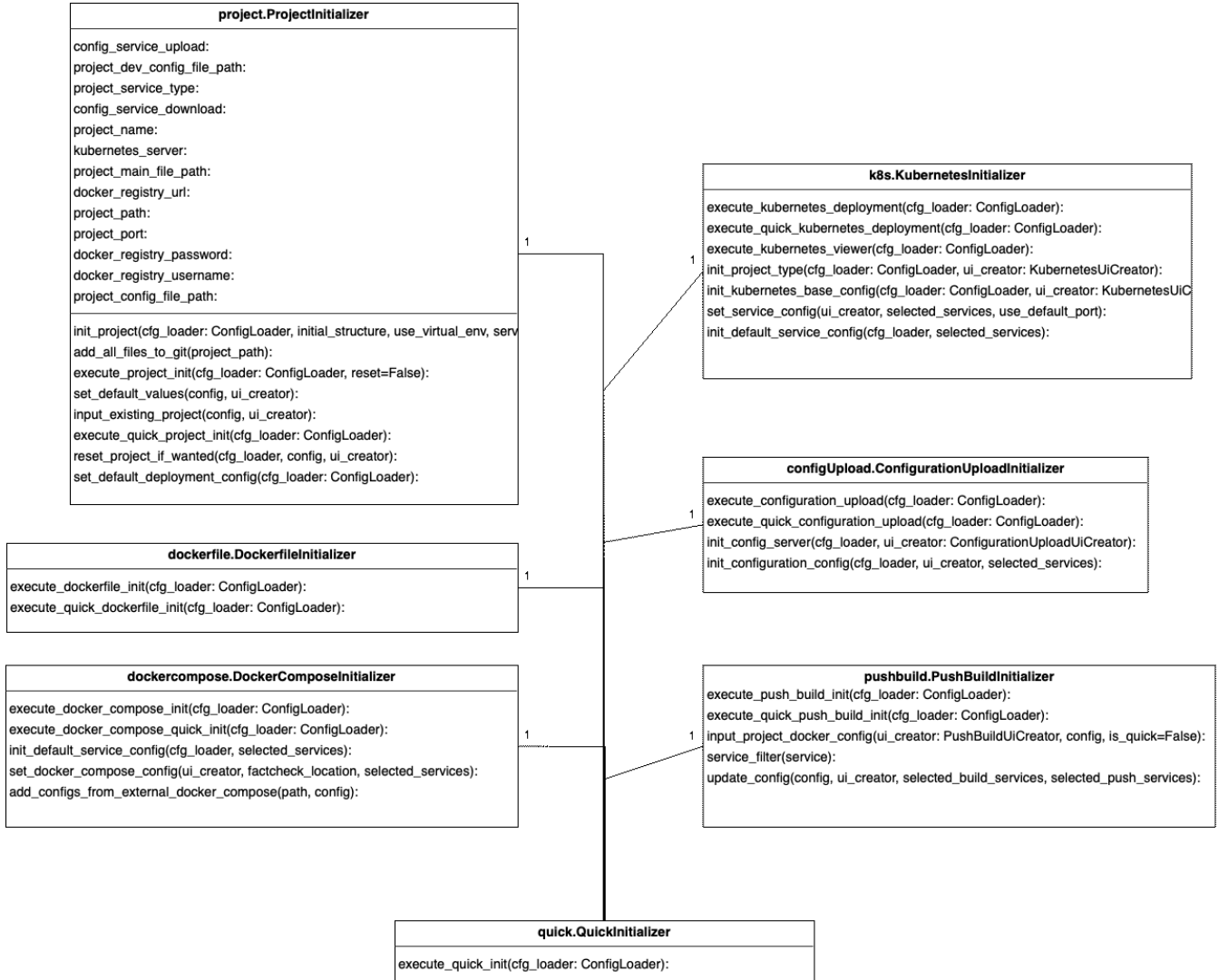


Figure 4.9.: Classes used by the Quick Deploy feature.

Initially, this process checks, if any previously configured external services such as factcheck-config-service, Docker Registry, and Kubernetes are available. If these services are not available or are only partially available, the system will ask the developer if it is

4. Implementation

planned to use the default values. When this question is positively answered, all necessary configurations for these external services will be pre-configured.

The next phase includes skippable features that are not necessary for every deployment process. These features include the project setup feature, the Dockerfile creator and the Docker Compose file generator. All these features have in common that they are very likely to stay the same throughout multiple deployments and therefore need to be executed only once. If an update to these generated artifacts is required the developer always has the option to only execute the desired feature in FactDeploy or update these files manually.

The next phase will include features that are primarily focused on the deployment process and cannot be skipped. These include the configuration upload mechanism, the Docker Image push and build workflow and the Kubernetes deployments. While each of these must be executed on each execution of "Quick Deploy", their necessary inputs can be reduced to a minimum.

These considerations given, the main use case and the assumptions made about this feature can be discussed. Firstly, the most common use case for FactDeploy was evaluated by asking members of the FactCheck team about the most common software type they developed that worked closely with the FactCheck services. The conclusion was that the most common software type was a Python-based project that accesses the FactCheck services, is accessible via REST APIs or directly accesses the related databases (CouchDB and MySQL) running continuously as a service. The implementation of this "Quick Deploy" feature now concentrates on fulfilling these assumptions. This implementation can be examined in detail in the class diagram found in Figure [4.9](#).

The first instance of applying these assumptions occurs during the project generation process. During this process, the developer must input multiple parameters to execute the dedicated feature. To allow this process to happen more quickly, the generation of a project structure, the usage of virtual environments and the service type will be set to be a continuously running "Service". Furthermore, to allow for external access via REST APIs a Flask dependency will be added. For the user the only inputs left are to define the project name and its location. This step, as already mentioned, will be skipped entirely after the initial execution.

For the second step in the process, the Dockerfile will be generated. To accommodate the assumptions made, the parameters for the name of the "requirements" file will be set to "requirements.txt" and the open port for this project will be set to 5000 (which is the default port of the Flask dependency). Additionally, a custom Dockerfile will be used, which is similar to the ones for WSGI applications but includes the use of a predefined startup script. With this startup script, the default behavior is to download the latest configuration from the factcheck-config-service before the startup operations can happen. After the first execution, this step can also be skipped fully.

The last of the skippable features is the Docker Compose file generator. The setup for

a "quick" build contains all necessary components for executing all FactCheck services, as well as the development container. This includes the pre-built FactCheck Backend/-Factbase/Factserver services, the MySQL and CouchDB databases and the development container. With this selection made, the Docker Compose generator does not need to ask for additional information from the developer and will use all default values from the 'AvailableServices' class. This includes default credentials, initial databases, ports and Docker Images. With all these parameters provided, the Docker Compose generation is performed, which will add the docker-compose.yaml file to the generated project.

There are also non-skippable features that are executed during each "Quick Deploy" execution. These include the configuration upload feature, the push/build feature and the Kubernetes deployment feature. The configuration upload feature will work the same as already stated in Section 4.3, but will only upload the configuration for the currently loaded project. The push/build feature operates on the same principle, solely focusing on building and pushing the currently loaded project, without any specific impact on its workflow. On the other hand, there is the Kubernetes deployment feature, which has several specializations for this use case. In this special case all necessary FactCheck services will be deployed, including the FactCheck Backend/Factbase/Factserver services, the MySQL database and the CouchDB database. For all of these services the default values from the 'AvailableServices' class will be used to prepare and configure these deployments. Finally, in the deployment process itself, the FactCheck services, the databases and the currently loaded project will all be deployed at the same time. Although this action may appear to cause a performance issue, any subsequent deployments will primarily focus on deploying any modified deployment files, thereby updating only the currently loaded project in most cases.

Overall, this "Quick Deploy" feature can be used in combination with the other features. This allows developers to change their deployment workflow at any time as needed. For example, the initial usage of the "Quick Deploy" feature creates everything necessary for a deployment and when a developer wants to change the configuration of a specific feature, they can simply execute that feature and update it. The subsequent "Quick Deploy" execution will utilize this configuration.

4.3. FactCheck Config Service

Externalized configuration is one of the main concerns in this thesis, as storing sensitive data and the portability of applications are vital. Furthermore, it allows for easier configuration modifications without the need to update or redeploy software. This, in turn, encourages a more modular and maintainable system. Allowing software to take advantage of these possibilities requires two critical components.

4. Implementation

4.3.1. REST Endpoints

The first component is on the server side, which hosts all configuration files. These configuration files are uploaded or downloaded by a developer or FactDeploy via REST calls. The basis of this application is the Flask library, which allows for fast implementation of these REST endpoints. These REST calls are available via the following requests:

For uploading a new configuration file to the factcheck-config-service:

- HTTPS Request
 - Type: POST
 - URI: `https://kubclient3.cs.univie.ac.at/config/add`
 - Content-Type: `text/plain`
 - Body: content of the configuration file in plain text.
- HTTP Response
 - Content-Type: `application/json`
 - Body: a JSON structure with a key named 'api-key' and a value containing the api-key as a string.

For downloading a new configuration file to the factcheck-config-service:

- HTTPS Request
 - Type: GET
 - URI: `https://kubclient3.cs.univie.ac.at/config`
 - Header: Key:x-api-key, Value:<api-key from upload>
- HTTP Response
 - Content-Type: `text/plain`
 - Body: content of the configuration file in plain text.

The expected workflow for these services consists in the uploading of a configuration file (in plain text) to the service the stated POST request, which in turn generates an API-key that can be used to access the configuration content again when submitting it via the "x-api-key" header in the stated GET request.

4.3.2. Data Management

To allow developers to securely store their configuration files, the process to generate their API-key had to be implemented with this requirement in mind. Such a key is based upon two separate keys, each of which is generated by the Fernet algorithm provided by the "cryptography" library. One of these keys is used as an identifier for the stored

configuration file. Therefore, accessing the configuration file content can only be done via the first key. The other key is then used to encrypt the configuration file content. The encrypted configuration file is then stored in a SQLite database with the identifier key as the primary key. After inserting this data into the database, the returned API- key is a combination of both keys, where the first part is the identifier key joined by a ":" character with the encryption key. It is important to note that the encryption key will not be stored and, when lost, will not be recoverable. The configuration data can be accessed only if the combined key is provided in the "x-api-key" header of the GET request. This means that even with access to the entire database, the configuration file content cannot be accessed without the combined key. Since only the initial configuration uploader possesses the complete key, they are the only ones who can access their configuration file. However, it is also their responsibility to ensure that this key is securely stored.

Storing of the configuration file is done in an SQLite database that has a single table with the following columns:

Tablename: applications

- Column Name: apiKey
 - Type: TEXT
 - Constraints: Unique, Not Null
- Column Name: configContent
 - Type: TEXT
 - Constraints: Not Null

Access to this table is done via the "DatabaseConnection" class which is responsible for executing DDL, insert, and query statements. During application startup this class will create this table. On every GET request to the "/config" endpoint, the "x-api-key" header's identifier key will be checked against the database to see if any content exists for this identifier. If there is no content the system will return a 401 HTTP code. If there is any content it initiates the specified decryption process. If this process is successful it will return the actual content of the configuration file. These checks can be applied to multiple endpoints, as these checks are implemented as a decorator called "@require_key". Furthermore, to avoid uploading files that could impact the availability of the factcheck-config-service, the maximum size for uploads is set at 5 megabytes.

4.3.3. Deployment and External Access

Only the index.html and the POST endpoint will be publicly available. The first one can be accessed by the name of the Kubernetes instance, which in the context of this thesis is "kubclient3.cs.univie.ac.at". It specifies the factcheck-config-service's available REST endpoints and HTTP methods. The POST endpoint, as already stated, will allow any user able to access the Kubernetes instance to upload configuration files. To allow for

4. Implementation

other developers to access to this service, the factcheck-config-service had to be deployed to Kubernetes with a specific Ingress configuration.

This configuration for a dedicated Ingress allows external services to access the factcheck-config-service via the "/" path, which corresponds to the hostname "kubclient3.cs.univie.ac.at." Every request made via this path will be forwarded to this service. Additionally, as this service is running on port 8080, external requests from port 80 will be redirected to the actually used port via the service definition in the deployment file.

When further evaluating the service structure, the default deployment files for databases can be reused, although they are adapted to fit the SQLite database location. This is due to the requirement that all configuration files be stored across service restarts. With this in mind any restart of the service does not affect the availability of other services to access their configuration files.

As these services are intended to be managed by administrators, their deployment has to be done manually. This includes specifying the correct version and tag. To ease this process, both a build.sh script and a push.sh have been added, which build a Docker image and push it to the FactCheck-Docker-Registry. To update the deployed application the pod of the factcheck-config-service has to be deleted, which in turn automatically updates the application.

4.4. FactCheck Services

For the FactCheck services, several changes had to be made so as to be compatible with the Kubernetes environment. While these changes affected each service, their functionalities and technology stacks remained the same. Therefore, this chapter will solely focus on the changes made to these services, and readers should refer to the relevant papers, such as the one from Kalchgruber et al. [Kal18], for further detailed information on these services. Nevertheless, a short introduction to some functionalities will be given to make the impact of these changes understandable.

4.4.1. Changes made to all FactCheck services

As changes were made to all supported FactCheck projects, a new multi-stage Dockerfile and a dedicated startup script for containers were included. This enables the inclusion of the API-key and the factcheck-config-service URL, which in turn automatically loads the service configuration at startup.

Also, a configuration for Kubernetes deployments of the default version and one with a connection to the CouchDB production database have been added. These versions can be built and pushed to a Docker registry by executing the "build_and_push.sh" bash script.

4.4.2. FactCheck Backend

The changes made to the FactCheck backend project also concern the frontend component. When these services are deployed on Kubernetes they are only accessible via subpaths of the hostname and not directly on the hostname, due to the multi-user support. In this case, the project is based on the Python framework "Django" which does server-side rendering of the HTML files to inject data. As subpathing is not automatically activated some customization had to be done.

First of all, the subpath has to be injected at startup. This was done by providing the correct path for an environmental variable in the deployment process. To allow Django to correctly work on subpaths, the "FORCE_SCRIPT_NAME" variable has to be set to the subpath, and in the startup process, the "PATH_INFO" and "wsgi.url_scheme" have to be set accordingly. This tells Django that all dynamically loaded static files and URLs must be accessed via this subpath. As most of the URLs were hardcoded in the HTML templates, they had to be replaced with dynamically created template patterns, which corrected these routing issues.

4.4.3. FactCheck Factserver

The changes made to the FactCheck Factserver project also concern mainly the frontend component. When this service is deployed on Kubernetes it is only accessible via subpaths of the hostname and not directly on the hostname, due to the multi-user support. This means that all links and static resources in the HTML files have to be dynamically created. To do this, all links for relative routing are converted via JavaScript by appending the correct subpath and hostname. With this change applied both the original and the subpathed versions work correctly.

4.4.4. FactCheck FactBase

Changes made to the FactCheck FactBase project are minimal, as there is no direct web application involved. Important to mention, was the split of the CouchDB instance and the FactBase service into dedicated services in the docker-compose file to comply with the single responsibility principle.

4.5. User Generator

Introducing a custom user generation project was necessary as it eases the introduction of users onto the Kubernetes instance for administrators. Furthermore, it provides all necessary roles and permissions for the newly created user, which enhances overall security.

4. Implementation

4.5.1. Usage

User creation is done via a singular bash script called "user_create.sh" which, when executed, prompts for a username for the new user. To work properly, this script has to be run by an administrator. By executing and by providing a unique username with the option "-u", a new user will be created. After this process, the administrator will be prompted the specified username and its automatically generated password. These credentials should then be provided to the new user, who should be informed that they needed to update their password. When the option "-f" is specified, the FactCheck production data is copied to the users folder, allowing them to use this data when deploying a dedicated CouchDB instance.

User deletion is done via a similar process; here, a bash script called "user_remove.sh" needs to be executed, and only the username of the user that should be deleted needs to be supplied. When executing this script, the administrator has to be careful, as all data related to this user, including Kubernetes resources, databases and credentials will be deleted.

4.5.2. User Creation

This workflow creates a user on the Ubuntu server on which the Kubernetes instance is running. For this workflow, the supplied username and a generated password will be used. The password generation is based on the `secrets.token_urlsafe()` method, which will generate a base64 password. With these prerequisites available, the user has an additional `"/home/<username>"` directory and a default shell, which is `"/bin/bash"`. Additionally, a directory called `"/opt/<username>"` is created on which the user has reading writing and executing permissions, which will be used for persistently storing data (e.g. databases).

4.5.3. Kubernetes User Setup

Before a user is able to access the Kubernetes instance several prerequisites have to be met. The user creation script will automatically create all these prerequisites with the minimal necessary permissions in mind. As this process is based on RBAC (role-based access control) for the user, and the user has supplied their SSL certificates and keys to the Kubernetes instance, several files have to be generated to fulfill these requirements.

In the first step after user creation, the necessary SSL certificate, certificate signing request, and key are generated. This is done by generating an RSA key with 2048 bits, and based on that key, a certificate signing request is generated. Now the default Kubernetes certificate and key will be used in combination with this certificate signing request to generate a signed certificate for the new user. Now, to allow users to access the Kubernetes cluster, both the RSA key and the certificate are deposited inside the Kubernetes instance. This allows Kubernetes to verify any logged-in user and to grant them access to its resources. In the final step, all these files will be copied to the `"/home/<username>/certs"` folder,

and their permissions will be updated accordingly. Additionally, a "config" file, which is necessary for accessing Kubernetes, will be generated and stored in the ".kube" folder. Now the user is able to authenticate itself with its Ubuntu user account, and they can access and manage their deployments.

As users should be able to access each other's deployments and resources, the creation of a namespace and the generation of an RBAC configuration YAML file are necessary. A namespace is simply defined by the current user's username. Only the current user can manage the associated resources in their specific namespace. To declare accessibility and available resources for that user, the RBAC configuration YAML file must be supplied. Currently, all users have the same role and permission structure, which grants them access to the following resources:

- API Groups: networking.k8s.io, extensions, apps, batch
- All resources in these groups (pods, deployments, etc.)
- All verbs in these groups (get, watch, list, update, etc.)

For each user, a dedicated role will be created, which is associated with the user via a so-called RoleBinding. When applying this RoleBinding, the user is now finally allowed to manage their resources.

Additionally, for that specific user a ClusterRole and a ClusterRoleBinding are created, allowing them to access the following resources:

- API Groups: none
- Resources: persistentvolumes
- Verbs: create, update

This is necessary as the creation of databases and their persistent storage have to be managed by all Kubernetes cluster users. This means that all users have access to the creation and update processes for persistent volumes. All verbs related to retrieval and deletion of these persistent volumes were explicitly disabled, as this would allow users to influence each other's access to their data. In the event that too many persistent volumes are created an administrator would be responsible for deleting them accordingly. Applying both the namespace creation and RBAC rights now grants the user access to all fundamental functionalities.

4.5.4. User Deletion

As users can permanently leave the Kubernetes instance a user deletion process was additionally implemented. This script is accessible via the "user_remove.sh" script and can be executed when supplying the username of the to-be-removed user. When executing the script all data will be removed from the system. This includes their Ubuntu username, their persistent volumes, all of their roles and permissions and their namespace.

4. Implementation

4.5.5. User Certificate Renewal

As Kubernetes only allows its self-signed certificates to be valid for one year since their creation, a dedicated script to update them has to be provided. It reuses the certificate generation process from the user creation process. After applying this script users can correctly access the Kubernetes instance again.

4.6. Development Container

Containers, in general, are entities that simply run a Docker image on an associated engine. Furthermore, these containers have the functionality to be accessed from outside and also execute commands from outside. However, this also means that a command-line process can be started. With this feature and the ability to mount external data into a container, the development workflow can be completely moved inside the container. This carries the advantage that, when all developers work in the same development container, the software runs in the same environment and is therefore independent of the user's operating system and its peculiarities. Furthermore, when the development container's image is similar to or equal to the production image, deployments become more consistent.

This container is based on the base Docker image for Ubuntu 24.04 and contains several useful development dependencies. This includes the GCC compiler, Python 3.12 development dependencies, a MySQL client, and an OpenSSL dependency. Additionally, a selection of useful software packages was included, being the following: git, curl, wget, unzip, tree, htop, npm, and sudo. These packages were chosen based on common usage throughout the fact-checking services. As this set does not include every possible package available or necessary for the development of a software product, a developer is free to choose and install additional packages and dependencies. As a result, when a developer decides to make such modifications, it may be necessary to adapt the production Docker image file accordingly.

Another important dependency to mention is Yelp's "dumb-init" package [Kue16], which is also installed. It is a simple process supervisor and init system designed to run as PID 1 inside Docker containers. It was created to address issues with signal handling and zombie processes that can occur when running applications inside containers. Since these issues can lead to the creation of numerous zombie processes during software development, the inclusion of this dependency was deemed necessary.

In addition, a user called "developer" is created, which also functions as an administrator and will be used by the developer in their work. This is due to the fact that external dependencies can be compromised, and with root user access, this can not only happen to the Docker Image but also to the host system. Therefore, this security measure was taken. Additionally, some deployment platforms, such as "Openshift", only allow for non-root images to be installed, which means that the developer needs to adapt to this circumstance already while developing their software.

To make this development container usable, the command `"tail -f /dev/null"` will be executed at startup. Because this command does not finish execution, the container effectively runs indefinitely and only stops when it is shut down by the developer. This means that the container can be used similarly to a virtual machine but with less overhead. This also implies a reduced feature set, which is still enough for developing common software. Accessing this container can then be done by using the command `"docker exec -ti bash"` which allows a terminal on the host system to directly relays all commands inside the development container. Now the developer has all the necessary capabilities to develop their software in a dedicated development container.

4.7. Private Docker Registry

A dedicated private Docker registry was introduced, as Docker Images created by student developers should not be uploaded to public repositories and registries for reasons of privacy and data protection. With private repositories and private registries such issues can be mitigated.

Two essential steps were necessary to create this repository and make it available inside the university network. Creating and running the repository itself requires a Docker instance, to host it. To install the repository, a `docker-compose.yaml` file is created. It holds the necessary image definition and the environment variable to configure the registry. In this case, the image is the `"registry:latest"` image, which is directly provided by Docker and provides all necessary functionality for managing Docker Images.

Accessing the Docker Registry from outside the instance requires that both an exposed port and an authentication method be provided. In this case the port is set to 5000 and is accessible from the outside by binding to the host computer. This port was chosen because it is the standard port used by the registry's Docker image. For authenticating users, several authentication methods, which include HTTP, OAuth, LDAP and Active Directory are available. In this case, basic HTTP authentication via `htpasswd` is used to minimize setup effort and simplify user creation. This authentication method is set up using the following environment variables in `docker-compose.yaml`.

- `REGISTRY_AUTH`: `htpasswd`
- `REGISTRY_AUTH_HTPASSWD_PATH`: `/auth/htpasswd`
- `REGISTRY_AUTH_HTPASSWD_REALM`: `Registry Realm`

After this setup process is done, an administrator can add a username and a password to the `"/auth/htpasswd"` file to let a user upload their Docker Images. In its current state this setup would only allow uploading images from the host system on which the registry is running, and only on port 5000. To allow external users from outside the host system to upload images to the registry an additional Nginx reverse proxy had to be set up.

Configuring this reverse proxy was done by creating/editing the Nginx configuration

4. Implementation

files called "registry.conf" and "nginx.conf." In the "registry.conf" file, the rewriting and routing of incoming requests take place. Every incoming request that has "docker" as a user agent on port 80 or 443 will be redirected to port 5000, where the registry is running. Additionally, some Docker-required headers are forwarded, which include "Host", "X-Real-IP", "X-Forwarded-For", "X-Forwarded-Proto" and a timeout of 900s. In the "nginx.conf" file the property `client_max_body_size` is configured, which sets the maximum size of the incoming request and is currently set to 4 gigabytes. This reduces the possibility of overloading the registry with too large requests. With this configuration in place, each developer can now upload and distribute their Docker Images by prefixing their image tag with "dockerregistry-lab.cs.univie.ac.at".

4.8. Kubernetes

4.8.1. Setup

While installing a Kubernetes on-premise solution is a straight-forward procedure on a server, some obstacles still had to be resolved or circumvented. This chapter explores these issues and their solutions.

The first hurdle was selecting the container runtime, as the default runtime "Dockershim" has been removed with Kubernetes version 1.24. Therefore, the selection of compatible runtimes had to be made. In the selection process, the following runtimes are directly supported by Kubernetes:

- containerd
- CRI-O
- Docker Engine
- Mirantis Container Runtime - only commercially available

The runtime choice was based on compatibility with the current technology stack, best practices, and cost. The "Mirantis Container Runtime" was not chosen due to the cost factor. The "containerd" runtime was not chosen due to the setup complexity. For similar reasons the Docker Engine was not chosen, as it requires Docker to be installed and reduces image compatibility to only Docker Images. In the end, CRI-O was chosen as it implements all the basic functionalities of a container runtime, is completely compliant with the Kubernetes CRI specification, and is an open-source project. This means that not only Docker Images but also all other OCI-compliant images are compatible.

After Kubernetes and container runtime resources have been installed, Docker Images should be able to be deployed. However, when attempting to deploy any resource, it does not appear to be scheduled. This is due to the expected Kubernetes setup style, which requires multiple worker instances and a single control plane instance for security reasons. Because only one server with limited resources is available for this thesis, only one

instance of Kubernetes can be deployed. In order for this instance to schedule resources, the Kubernetes instance had to be "untainted." After executing this process, this single Kubernetes instance is also allowed to schedule resources.

With scheduled resources available communication between them can be attempted. To allow this interoperability Kubernetes requires that a supported networking add-on be installed. Due to the sheer number of add-ons available for this functionality, a compatible, configurable, and popular container networking interface (CNI) was chosen. In this regard, Calico was selected, because it offers the required compatibility, allows for network policy configuration, and is among the most favored choices in this domain. Flannel would have been the next popular choice, but it would not have provided the necessary configuration options for networking policies.

4.8.2. External access to services

As communication between resources inside the Kubernetes instance is handled, communication and interaction with resources outside of the instance have to be accomplished. This communication method can be achieved using external or internal load balancers and/or ingresses. The most common solution for this is to supply an external load balancer to Kubernetes and route all requests from the load balancer to the services via an ingress. While this could be accomplished in the context of this thesis, no external load balancer was available, alternatives had to be evaluated. The first one was an internal load balancer called "MetalLB", also known as an on-premise load balancer. While this was a promising solution, this load balancer is best used with an IP address per service approach. Although there is a workaround that enables IP address sharing between several services, these services however have to be predefined. In the environment where this Kubernetes instance is running, this is not possible due to the dynamic creation of services, which allows their creators to access new services from outside the instance without explicit administrator intervention. To circumvent this limitation the load-balancer approach is abandoned and replaced by the direct usage of an ingress as an endpoint accessible from the outside. While this approach does have some downsides in terms of availability and load-balancing capabilities, it allows for dynamically adding services, which is of greater importance in an ever-changing multi-user environment.

Such an approach requires a slight customization of the installed ingress. Therefore, the current deployment of the used ingress had to be changed to allow direct connection via the host network, which in turn means that the ingress now has a connection type of "NodePort". To make such a modification, the deployment yaml is modified to add the value "hostNetwork: true" in its "spec" definition. Once that definition in place ingresses for services can be defined, enabling external access to those services.

To configure the ingress in a way that it fits the current environment and use-cases, the "NGINX Ingress Controller" which is directly developed by NGINX, had to be installed. Although the most commonly used ingress, which is called "Ingress-nginx Controller" and was directly developed by Kubernetes and was initially used, it lacked

4. Implementation

an important feature called "ingress merging". The importance of this feature can be explained by the initially intended process for externally available services. In the first implementation, it was intended for every user to have a single service publicly available via a single endpoint. In this case, the endpoint would be defined by this pattern: "https://<hostname>/<username>" and this endpoint would be created at the same time when an admin creates a user. Such an approach would yield several consequences. First of all, the user must create a service with the exact service name as defined in the ingress definition, which would restrict users and require them to change the deployment definition each time they want another service to be publicly available. Second of all, in general, an ingress can only be defined in one namespace, which is the "default" namespace. Because only admins have access to this namespace, the user would have no option to change the ingress configuration if they wanted to allow, for example, custom headers. All of these issues can be avoided with "ingress merging", which has production-ready support for on-premise solutions provided by the "NGINX Ingress Controller."

Ingresses can be merged by defining "minion" and "master" annotations in the ingress configuration. This annotation would be placed in the "metadata.annotations" section and would have the following definition: "nginx.org/mergeable-ingress-type: "master"" and correspondingly "minion". The "master" annotation is commonly placed in the "default" namespace and waits for "minion" ingresses to be created. This "master" ingress has access to all namespaces, so it collects newly created "minion" namespaces and adds them to its own configuration. This implies that each user can now generate externally accessible ingresses. As this would possibly allow users to have overlapping endpoints, each endpoint is now defined by "https://<hostname>/<username>/<service-name>". Now, the user can create as many services as they need and have them all publicly available. A better endpoint definition would have been to use the username as a subdomain, like "https://<username>.<service-name>.<hostname>/" to avoid routing issues for services. This approach was based on the certificate used in the later-described TLS termination section, which was not possible. While these ingresses allow for external access, for example, through a browser the risk of accidentally or intentionally accessing somebody else's services is now given.

As a security mechanism is already provided by allowing access to the Kubernetes instance exclusively through an authenticated university account, a user identification mechanism still has to be implemented. This allows the system to identify the correct users having access to a service. This identification system is implemented using the already-available "Basic Authentication" method provided by the ingress, which requires the input of a simple username and password. For this to work, a "Secret" Kubernetes object has to be provided to the "nginx.org/basic-auth-secret" annotation in the ingress. This secret is provided on each deployment and is based on the username and password employing to access the Kubernetes instance. Before deployment the password is hashed via the "httpasswd" dependency, and the secret is set to the type "nginx.org/httpasswd" to ensure secure handling of these credentials by the ingress. After the service has been successfully deployed, each external access requires the username and password to be provided via the

"Basic Authentication" protocol. Therefore, when accessing a service through a browser, the user will need to provide these credentials on first access for identification.

As modern browsers only trust communication with a CORS-enabled server, the ingress had to be supplied with the following annotations with the prefix "nginx.ingress.kubernetes.io/".

- enable-cors: "true" - enables Cross-Origin Resource Sharing (CORS) functionality on the server.
- cors-allow-methods: "PUT, GET, POST, PATCH, OPTIONS" - specifies the HTTP methods allowed for CORS requests.
- cors-allow-headers: "X-Forwarded-For, x-api-key, Authorization, Content-Type, Accept, X-Requested-With, X-CSRF-Token" - defines the HTTP headers allowed in CORS requests.
- cors-expose-headers: "*" - exposes all response headers to the requesting client in CORS requests.
- cors-max-age: "600" - sets the maximum time, in seconds, that a preflight request (OPTIONS) can be cached by the client.
- cors-allow-credentials: "false" - indicates whether the server allows requests with credentials (such as cookies or HTTP authentication).

TLS termination is another important configuration in the "master" ingress. To allow for this feature to be available to all users, a so-called "Secret" has to be defined in Kubernetes, which contains both the certificate and the private key for the hostnames that should be secured. In this case, it would be "kubclient3.cs.univie.ac.at", which has no wildcard definitions for subdomains. Additionally, the annotation "nginx.ingress.kubernetes.io/ssl-redirect: "true"" must be set to allow for incoming requests on port 80 to be redirected to port 443 to force a secure communication with the Kubernetes instance. When this secret and this annotation is supplied to the ingress, and one specifies the hostnames the ingress should use this secret on, all communication with the outside environment will be secure and occur on port 443 (HTTPS).

4.9. Production Data Replication

Allowing developers to access production data enables them to better debug and see the results of their work directly. To do this, the production data has to be replicated on the Kubernetes instance. The replication service's implementation is based on the replication mechanism of the CouchDB database used by the FactCheck project. As a result, it was named fact-syncer because it replicates only data-related facts that are stored in the production database.

4. Implementation

4.9.1. Configuring Replication Processes

The configuration of a replication process consists of several options regarding selected databases, data structure creation, and replication options. In this context, the configuration is done via YAML files and structured as follows:

- `continuous_sync` - defines if a replication process should stop if all changes are replicated or if it should continue running.
- `max_sync_percentage` - defines how much data from the source database should be synced to the target database.
- `create_db_at_target` - defines if a database should be created at the target database if none is existent before.
- `execute_on` - defines on which CouchDB instance (source or target) the replication process should be executed on.
- `databases_to_sync` - list of databases in the source CouchDB instances to be synced to the target CouchDB instance.
- `source_database` - definition of the source CouchDB instance identified by an URL, username and password.
- `target_database` - definition of the target CouchDB instance identified by an URL, username and password.

With these configuration options in place, they can be loaded by the 'ruamel.yaml' library and used in the workflow.

4.9.2. Adapting to Environmental Constraints

Replicating data via the CouchDB replication mechanism is a highly parallelized but resource-intensive task. In this thesis, the Kubernetes cluster is configured with 2 CPUs and 4 GB of RAM, which meets the minimal requirements for a Kubernetes instance. In several tests, when starting the replication process in the Kubernetes environment, both CPUs were fully used until the processing was stopped by Kubernetes with the condition "OOMKilled". This condition also means that the allocated memory limit was reached and the pod was terminated.

Fortunately, CouchDB has several settings that can be adapted to comply with these memory and CPU limitations, which can all be set via the provided REST API. According to the documentation of the replication functionality, the following three parameters are the most effective for reducing resource usage:

- `http_connections` - number of HTTP connections per replication process with the default value of 20. By lowering the amount, the CPU usage will be reduced and is therefore set to 2.

- `worker_batch_size` - number of entries in a batch of documents with a default value of 500. By lowering the amount, the total amount of RAM used will be reduced and is therefore set to 50.
- `worker_processes` - number of workers used for reading/writing to the disk and network in parallel. By lowering the amount, the CPU usage will be reduced and is therefore set to 1.

The numbers chosen for each of these values are based on several trials of different values to keep the replication process from being terminated. These numbers can vary from system to system depending on the hardware used and the workload involved. Therefore, these numbers should be seen as an example, and not a recommendation, for running a replication workflow on a relatively slow system.

4.9.3. Replication Workflow

Let's now delve into the replication workflow and its execution pattern. After the configuration is loaded, the initial step is to reduce the resource usage of the CouchDB instance. As previously described, CouchDB replication is a highly resource-intensive process; therefore the three settings: `http_connections`, `worker_batch_size` and `worker_processes` are reduced to viable values via the REST API. To avoid interfering with other activities and processes, these values are first read and cached, then set to their respective values to allow for a continuous replication process.

After reducing the resource usage, the replication process can start. For all databases defined in the configuration the replication process will be started. To do this, the CouchDB Python dependency is used, and the source and target databases are set. Additionally, the parameters `"continuous"` and `"create_target"` are set to their respective values from the configuration. After this process is started, the progress of this process has to be determined.

Progress is tracked by sending a request for the state of each replication process every 15 seconds. In this request, the document count between the source and the target database is compared, and when the previously mentioned threshold from the configuration value for the `"max_sync_percentage"` key is met, the replication will be stopped. Because these progress requests only occur during the specified time interval, small databases can be completely replicated thereby circumventing this threshold. This is not an issue in the context of this thesis, as an increased amount of data is better for debugging. Furthermore, the approach of replicating based on a percentage is not problematic in this context, as each data entry is a dedicated JSON object and not directly related to other data. In contrast, the replication of a relational database would be more complicated due to constraints such as foreign keys.

Due to the previously mentioned performance issues, and in addition to determining progress, an additional check was implemented. This check determines if there was a change in the number of synchronized documents between two checks. If this is not the

4. Implementation

case, the replication process will be stopped, a timer will wait for 120 seconds for the databases to regain their availability state, and another attempt at replication will be initiated.

Finally, after the process has successfully replicated enough documents and the replication has stopped, the parameters are reset. This reset mechanism uses the cached original values, and if none are available, the default values for these parameters. After this step the fact-syncer is completed and has to be restarted to locate newly added documents.

4.9.4. Deployment

The fact-syncer project is first deployed for the main CouchDB instance, then for each user who wants to replicate the FactCheck data. To reduce strain on the Kubernetes instance, the administrator will be able, on user creation, to supply the parameter '-f', which will automatically copy the current production database to the user's CouchDB instance.

In both deployments the fact-syncer will be directly deployed to Kubernetes and run as a so-called "CronJob." This type of deployment will execute the fact-syncer repeatedly at a specified point in time. For the default deployment of the CouchDB instance that replicates directly from the production database, the fact-syncer will be executed each day at 4 a.m. The fact-syncer used for syncing inside the Kubernetes cluster will be executed at 5 a.m. each day to give the main database enough time to fully replicate from the production database.

4.10. Microsoft Azure

As one of the main goals of this thesis is to connect to Azure services, the possible access points have to be explored. Two options possible, either directly deploying a Kubernetes cluster to the Azure cloud or making the Azure cloud environment accessible to the private on-premise environment. Both of these options will be explored in this section.

4.10.1. Creating a development environment in Azure

The setup of a development environment, based on this thesis, software requires a Kubernetes cluster as the basis for all deployments. Azure provides an API and a Web UI to create this cluster by supplying a set of parameters. Important for this thesis is only one parameter and that is the definition of a virtual network to which this cluster belongs, which is simply the resource name of this network. If this virtual network is established in the same network in which the FactCheck projects are currently running, the connection to the services used by the FactCheck project will also be possible. Therefore the created Kubernetes cluster running in the Azure environment has been setup by the FactCheck team with the correct network definition and it can therefore access all necessary services.

To make this Kubernetes cluster compatible with the deployments done via the FactDeploy application, a few adaptations had to be made. First of all an Ingress had to be installed and configured which is in this case the nginx-ingress. To allow external communication via a valid FQDN, the following settings have to be set on installation via helm:

- `controller.service.annotations.`
`"service\.beta\.kubernetes\.io\/azure-load-balancer-internal"=\\"true\\"`
- `controller.service.annotations.`
`"service\.beta\.kubernetes\.io\/azure-dns-label-name"="factcheck-dev"`

This allows for accessing the Kubernetes cluster via the following FQDN: **factcheck-dev.westeurope.cloudapp.azure.com**

This FQDN has then to be used as the host in the definitions of all further Ingress objects in this clusters.

4.10.2. Adaptation to FactDeploy

Now, to enable the operation of FactCheck services on this cluster, establishing a connection to the FactCheck Docker registry is necessary. This can be done by setting a Docker secret in the Kubernetes cluster. While this step is the same as on-premise deployment, the management of the registry is different. The registry itself is managed by the FactCheck team and is only accessible via dedicated credentials and a specific namespace. While this does not affect the functionality of FactDeploy, several scripts had to be updated to support this namespace for Docker image building and pushing.

Changes to the creation of storage volumes had to be made, because they are different in an on-premise environment. This means that the creation of the PersistentVolume is done by Azure, and it does not need to be created manually. This then affects the selection of the storage class and the selector in the PersistentVolumeClaim. The storage class has to be set to one provided by Azure, which in the case of this thesis is "managed-csi". Furthermore, the "selector" property is not allowed to be defined for the PersistentVolumeClaim to work.

To connect to Kubernetes, the access mechanisms had to be adapted, as the Kubernetes cluster can be directly managed through the developer's workstation and not via an SSH connection. As the Kubernetes Python API was used in the development of the FactDeploy software, the connection method simply had to be switched to use Kubernetes as it was running locally.

Another small adaptation had to be made, which was the selection of default values. Now, when Azure is selected as the deployment environment, the default values provide the corresponding values.

Other than that and a few textual adaptations, the adaptation of FactDeploy to support Azure deployments required less time than anticipated and shows the portability

4. Implementation

advantages of FactDeploy and Kubernetes.

4.10.3. Connecting on-premise infrastructure to Azure

The main focus of this section is on the setup and adaptations required for use of the FactDeploy software and other software in a hybrid cloud environment. As the main goal is the interaction of on-premise software with cloud services, access from on-premise Kubernetes nodes to Azure cloud services is the primary prerequisite.

To achieve this, a functioning VPN gateway is required with either a Point-to-Site connection or a Site-to-Site connection. Within this thesis, the possibility of establishing a Site-to-Site connection was evaluated. However, this necessitated the access to on-premise infrastructure for configuration. Since the university's infrastructure is restricted to access by dedicated administrators, this setup was done by the FactCheck team.

The primary use case for this thesis will be a Site-to-Site connection. This allows a specific IP address range in the on-premise environment to be accessible in the Azure environment, and vice versa. For connecting a single server to the Azure environment a Point-to-Site connection can be used, if necessary.

The Site-to-Site connection allows for a Kubernetes node to interact with the Azure environment. While this would be enough to provide access from on-premise environments to Azure services, the goal of managing the infrastructure would not be achieved. Setting up the Azure Arc was necessary to accomplish this on both the Azure site and the on-premise site.

The Point-to-Site connection allows for single clients to access the Azure environment via a VPN client installed on-premise. As the on-premise infrastructure consists of Linux servers, the choices for VPNs are limited to either OpenVPN or IKEv2, both of which have their disadvantages. IKEv2 requires ports 500 and 4500 to be opened, which is not possible as the current university infrastructure only allows the HTTPS ports (80 and 443). To circumvent this, OpenVPN must be used, as it only uses port 443. This, in turn, means that no other software can access this port, which can also be circumvented by allowing port-sharing in OpenVPN or installing a tool like SSLH. These options were not explored in the context of this thesis, as the Point-to-Site connection is not the primary option for establishing a hybrid cloud connection.

In both environments, establishing an OpenVPN connection between the on-premise server and the Azure environment required setup steps. On the Azure site, a VPN gateway and a virtual network must be created. It is important to mention that the Kubernetes cluster has to be on the same virtual network as the VPN gateway.

After the VPN gateway is deployed, a connection with the client must be established. To do this, a self-signed root and client certificate has to be generated and the OpenVPN client installed. The root certificate is then supplied to the VPN gateway, where it is used to authenticate the client. Azure provides generated configuration files which can be

downloaded after supplying the root certificate. Starting the OpenVPN client on the client side with this configuration, the client certificate, and the client key allows for connecting to the Azure environment and therefore to all services available in this network.

The CoreDNS configuration must now be adapted to allow access from Azure to the client via a hostname. By adapting the "hosts" setting, the configuration for that specific hostname has to be overwritten. The client's IP and hostname must be defined here. Also, to not hinder the internal DNS resolution by CoreDNS, the keyword "fallthrough" has to be provided in a dedicated line. This now allows for all services in the Kubernetes instance to access the client directly through the hostname.

With these connections in place and Azure Arc enabled, the FactCheck development environment can now be managed and deployed onto Azure. Also, a communication channel between the on-premise infrastructure was established, thus the main requirements of this chapter have been met.

5. Evaluating Deployments

5.1. Background and Goals

The deployment of software for development, testing, or production purposes is different for every software stack and production stack used. To determine if the approach used is suitable for each of these environments, each developer must determine whether their software is ready for the desired domain. To achieve that the so-called "deployability" has to be evaluated [BD19]. Due to the wide spectrum of software deployment practices, technologies and use cases a common metric definition for all technologies is hardly determinable. Therefore, this thesis aims to define deployability for a specific sub-spectrum of technologies. Furthermore, this metric aims to serve as a starting point for common use cases and be extensible for current and future technologies while still being comparable.

5.2. Defining Deployability

As already stated, the surrounding technologies used when deploying software are very volatile. To find a metric that can work in these surroundings, well-defined structures, requirements, scopes and rules have to be established.

5.2.1. Requirements

To define this deployability metric, several requirements have to be fulfilled. As a result, key requirements have been identified that are relevant to the overall objective and application of this metric.

- **Measurability:** Metrics should be quantifiable and measurable [SK23]. This requires a clear and objective way to collect data and assess the capabilities based on the defined metric.
- **Comparability:** Contextualizing the metrics results by comparing them between applications, enables understanding whether the capabilities are meeting expectations [SK23].
- **Adaptability:** Allowing to adopt for new circumstances and adapting the metric accordingly allows for more use-cases and longevity [CG10].

5. Evaluating Deployments

- Consistency: It ensures that the metric produces reliable results over time and across different contexts [SK23]. Furthermore, consistent metrics allow for valid comparisons.
- Validity: The metric should measure what is intended to be measured [SK23]. It should be a valid representation of the concept or capability aspects it is designed to assess.
- Applicability: The practicality and feasibility of collecting data for the metric, and evaluating it, ensure that the usage of the metric does not outweigh its usefulness [Kum22].

5.2.2. Scope

The metric needs to be scoped to a specific environment in order to capture the essential elements relevant to this thesis and to specific use cases. This includes the FactCheck project, Kubernetes, and Docker Images as the main relevant actors.

In regards to the FactCheck project, the scope is well defined. All backend services are currently implemented in a Python version, while frontend apps are implemented via different JavaScript frameworks. Therefore, the metric will focus on these two technologies, grading execution environments and project structures, while being agnostic of the frameworks used in these projects.

For deploying projects the scope has to be set to only include the direct creation of Docker Images, which are then deployed on Kubernetes. This means that the metric will include only relevant deployment files and workflows. Extensions on this basis will not be covered, such as Gitlab CI/CD workflows or deployments on OpenShift (a derivative of Kubernetes).

As one of the main goals of this metric is to be a starting point for further developments, the focus lies on being agnostic and relying only on commonly used best practices regarding the technologies employed.

5.3. Metric Definitions

Fulfilling the previously stated requirements can be done in several ways. For this thesis, the metric will be defined using predefined structures, verification methods and predefined concepts as a basis.

5.3.1. Structure

As the applicability of this metric is a main concern human readability and a simple evaluation method lie in focus when defining structures. Structures such as to-do lists would be easily understandable, but they do not suffice to exemplify the decision-making

process and do not allow for clear annotations. Therefore, the use of a classical spreadsheet was deemed necessary, as the contained data can be structured, relationships can be easily identified, and readability can be ensured. An example of this structure can be seen in Table 5.1

The structure of that spreadsheet is defined by the following elements:

- First-Level: Actor - defines a group of definitions
- Second-Level: Feature Groups - defines a group of features, their requirements and their impacts
- Third-Level:
 - Feature: defines a group of requirements to fulfill that feature
 - Requirement: defines a single or multiple steps to fulfill that requirement
 - Impact: defines the importance of that step/requirement/definition/actor of the specific level
- Fourth-Level: defines a step to fulfill the requirement. This step can also contain sub-steps with a recommended maximum of three layers to ensure readability and reduce unnecessary granularity.

Supporting Actors - Actor			
Software Security - Feature Group			
Feature	Requirement		Impact
Dependency Management - Feature	Choosing a dependency - Requirement		100 - Impact
	Steps	Select dependencies based on usefulness, community support, stability, licensing and time since last update - Step	50 - Impact
		Check for possible vulnerabilities in public databases - Step	50 - Impact
		Check and evaluate CVE Score - Sub-Step	25 - Impact
	Steps	Check and evaluate CVSS Score - Sub-Step	25 - Impact
	Upgrade existing dependencies to latest version - Requirement		100 - Impact
	Compare SHA-256 checksum with the downloaded library - Requirement		100 - Impact

Table 5.1.: Illustration of the layer structure in the deployability metric.

5.3.2. Symbols

The introduction of symbols is necessary to provide additional context to the annotated steps and aid the metric's usability. Their main purpose is to add an additional but optional importance factor to each step.

5. Evaluating Deployments

The current definition of these symbols is the following:

- + Recommended for optimal execution of the chosen step
- Not Recommended, but can be useful in some cases
- # Impact numbers that are styled in a bold format, describe that these values are at the highest level - only used for easier distinction

5.3.3. Phrases

Phrases are required to allow for complex definitions of steps, such as specific sequences and choices, or for only a subset to be necessary to complete a higher level. These are oriented toward common programming phrases to allow for easier understanding, consistency, and extensibility throughout the metric.

- steps - a specific sequence of steps to be executed to allow for completion of the higher level step.
- if - a specific precondition for lower steps to be executed. The impact value is empty as it represents a choice and has therefore no importance for the completion of the overall step.
- then - a list of steps that will be executed if the preconditions are fulfilled
- else if - alternative when a previous precondition does not apply. The impact value is empty as it represents a choice and has therefore no importance for the completion of the overall step.
- else - describes an alternative without preconditions, impact value is defined as steps can be directly executed
- choice - a list of mutual exclusive steps

5.3.4. Impact

The introduction of the "Impact" definition means that every step of the metric has a value assigned which corresponds to the importance of that step to the higher-level step. If all sub-steps are completed, the higher-level step is fully completed; if not, it is only partially completed. Partial completion does not mean that these higher-level steps cannot work; however, according to best practices, the deployability of the feature cannot be guaranteed.

Calculating the impact value is always constrained by the group in which a step occurs. Typically, each impact value will be the higher-level value divided by the number of steps in their corresponding group. Although it may seem counterintuitive, each sub-step of the same group, up until the requirement section, has the same value. This choice was made because the impact value does not describe whether a software's functionality is

fulfilled, but whether it is adhering to the best practices described to ensure deployability. Therefore, these impact values are completely independent and equally valuable within their respective groups. This also means that impact values in different groups, although they may be on the same level (step, sub-step, etc.), should not be compared. As these values only describe their part of the parent step, unintended relationships could be formed which would not result in a usable comparison.

As already stated, the completion of all sub-steps of the higher-level step results in the higher-level step being completed. This concept applies to the entire metric, all the way up to the highest level. This, in turn, means that completing all steps in a group of steps will result in an impact value of 100. A single sub-step can therefore be defined as follows:

$$S_s = trunc(\frac{100}{k}, 2) \quad (5.1)$$

With the variables defined as follows:

- S_s is the impact value of the current sub-step
- k is the number of sub-steps in the parent step
- $trunc(x,y)$ truncates a number's decimal places to a specified count, in this case, two.

The truncation of decimal places is necessary because divisions can result in large or even infinite decimal places which are difficult to display in a readable format in the metric.

To calculate the impact value of a requirement all steps are summed up and rounded up to the nearest whole number, which should be at most 100. It is important to mention that the following equation has to be applied recursively when there are larger nested structures.

$$S = \begin{cases} if k > 0 & \lceil \sum_{n=0}^k S_s[n] \rceil \\ else & 100 \end{cases} \quad (5.2)$$

With the variables defined as follows:

- S is the impact value of the current step (or requirement)
- k is the number of contained sub-steps
- $S_s[n]$ is the impact value of a single sub-step

When evaluating the completion of a feature, all the impact values of each requirement have to be summed up and divided by the number of steps contained in this feature. This verifies that the total completion of a feature also results in an impact value of a maximum of 100.

5. Evaluating Deployments

$$F = \frac{\sum_{i=0}^u S[i]}{u} \quad (5.3)$$

With the variables defined as follows:

- F is the impact value of the feature
- u is the number of requirements contained by a feature

Calculating the total amount of impact achieved can then be simply done by summing up all the impact values of each feature:

$$I = \sum_{i=0}^x F[i] \quad (5.4)$$

With the variables defined as follows:

- I is the total impact value of the metric
- F is the impact value of a single feature

With these constraints in mind, the conditions for fulfilling the measurability requirement are met. Measurability is fulfilled as the addition of extra steps or sub-steps never changes the maximum value or previous values and result in a concrete value. This is based on the statement that only fulfillment of all sub-steps makes a step complete, and therefore the highest-level step completion is consistent even after extensions.

5.4. Actors

In this thesis an actor is defined as an entity that defines the importance, necessity, and grouping of features. Furthermore, it serves as the basis for selecting the environment in which software should run.

5.4.1. Main Actors

This kind of actor is fundamental for a deployment. Without fulfilling at least one of these features, contained by this actor, software may not be deployed in the development or production environment.

5.4.2. Supporting Actors

This actor category's features are primarily for improving deployments and are highly recommended for good maintainability and better software quality.

5.4.3. Supplementary Actors

The features in this actor category primarily aim at improving the quality of deployments, much like those in the supporting actor category. The main difference is that these features are of lesser importance in the overall system and may only bring improvements in a production environment, e.g., monitoring capabilities.

5.5. Environment Constraints

When the highest levels of features and actors are combined, the environment in which software can be deployed can be evaluated. As the decision for the proper environment cannot be made by a single set of requirements as they are different for every software product, a minimal and recommended requirements set was defined.

The minimal requirements are:

- Development Environment
 - One feature from the main actor group
 - None from the supporting or supplementary actor groups
- Production Environment
 - One feature from the main actors group
 - One or more from the supporting actors group
 - None from the supplementary actor groups

The recommended requirements are:

- Development Environment
 - One or more features from the main actor group
 - ~25% from the supporting actors group
 - None or more from the supplementary actor groups
- Production Environment
 - One or more features from the main actors group
 - ~75% from the supporting actors group
 - ~50% from the the supplementary actor groups

When all requirements for a specific environment are met, the software has fulfilled the prerequisites for deploying in the related environment. Furthermore, they also fulfill the previously defined comparability and consistency requirements, as two systems can now be compared by which state (minimal, recommended, development or production) they

5. *Evaluating Deployments*

are in. For a more detailed comparison the impact value shall be used, which preferably compares systems that are in the same state.

5.6. The Deployability Metric

In this section, the table used to calculate the overall metric is given, and it can be used via the previously defined rule set.

5.6.1. Main Actors

All requirements and features in this section belong to the so-called "Main Actors." These describe a set of features and requirements that are essential for a deployment on Kubernetes and are therefore not circumventable.

Main Actors					
Software Deployment Definitions					
Feature	Requirement		Impact		
Dockerfile Definition	Defining a base Dockerfile		100		
	If	Single-Stage build wanted			
	Then	Define single image	100		
	Choices	Latest Ubuntu LTS Version (+)		100	
		Python 3.x - slim			
		Python 3.x			
		Python Alpine (-)			
	Else If	Multi-Stage build wanted			
	Then	Define a two stage build file		100	
	Steps	Define “Builder” Image		50	
		Choices	Python 3.x		50
			Latest Ubuntu LTS (+)		
		Define “Runner” Image		50	
		Choices	Python 3.x		50
			Python 3.x - slim		
			Python Alpine (-)		
			Latest Ubuntu LTS (+)		
	Disable user prompts (+)		100		
	Creating non-root user (+)		100		
	Install dependencies		100		
	Choices	Install each dependency independently		100	
		Installing via pip and requirements.txt (+)			
		Installing via Conda			
		Installing via Poetry			

5. Evaluating Deployments

Using a virtual environment (+)			100
Setting the environment variable "PYTHONUNBUFFERED" (+)			100
Using EXPOSE Port			100
Copying Project files to container and update permissions			100
If	Software is a web application		
then	Use WSGI with Python		100
Steps	If	Django, Flask or other WSGI Library is used	
	Then	Use gunicorn as a production web server	100
	Else If	WSGI with gunicorn not wanted/compatible	
	Then	Try uWSGI or similar	100
else	Use standard python execution		100

Table 5.2.: Software Deployment Definitions - Dockerfile Definition

5.6.1.1. Software Deployment Definitions

Dockerfile Definition When it comes to deployments, as can be seen in Table 5.2, the first step is to define the Dockerfile (or similar) within the specified context. This requirement was chosen to be part of the metric as Dockerfiles in general are highly popular for generating OCI images and are already used throughout the FactCheck projects. Being the basis for every deployment in this context the placement in the "Main Actor" category is valid.

For the base of the Dockerfile, two different approaches are valid. The first one is based on a single base image, which can be extended to fit the needs of the application and execution¹. The second option expands on this concept by utilizing multiple base images, each of which can be tailored to meet specific requirements and provide cleaner images². This is because the workflow uses the results of each step and discards other dependencies. In the metric several examples are given, such as technology choices for base images and the "builder and runner image" pattern for multi-stage images³.

User prompts had to be disabled as a requirement, due to the fact that user inputs to an automated environment cannot be processed⁴. Therefore these choices have to be disabled. To circumvent this issue, parameters should be added, if the software supports

¹<https://docs.docker.com/build/building/base-images/>

²<https://docs.docker.com/build/building/multi-stage/>

³<https://medium.com/@kaperys/create-lean-docker-images-using-the-builder-pattern-37fe2b5d97d4>

⁴<https://medium.com/vantageai/how-to-make-your-python-docker-images-secure-fast-small-b3a6870373a0>

it, or an always 'yes' or 'no' input should be defined.

The creation of a "non-root" user is a useful and important security measure, as an infected container can be exploited via privilege escalation techniques to access the host system⁴.

Installation of dependencies via one of the recommended systems results in a more efficient workflow, better automation and easier management⁵.

The use of a virtual environment is also recommended as depending on the base image that is used, several already installed packages might interfere with the application's dependency⁶.

Setting the PYTHONUNBUFFERED environmental variable to "1" is recommended since the application's output will not be buffered. This will enable better debugging because important information is not discarded when an application ends unexpectedly⁴.

Copying the project files inside the container and updating their permissions to be executable is necessary. Without this step, the software would not be executable within the runtime⁴.

On the condition that the application to be executed is a web application, dedicated production WSGI servers are recommended for better performance and increased throughput⁴.

⁵https://www.data-mining.co.nz/docker-for-data-scientists/best_practices/

⁶<https://pythonspeed.com/articles/activate-virtualenv-dockerfile/>

5. Evaluating Deployments

Main Actors				
Software Deployment Definitions				
Kubernetes Resources Definition	Usage of API level of at least: v1 or latest.		100	
	Definition of the associated namespace		100	
	Steps	Set value “apiVersion” to “v1”	25	
		Set value “kind” to “Namespace”	25	
		Set metadata value “name” to a specified name	25	
		Set metadata value “labels” to a single/multiple key value pairs for grouping objects (+)	25	
	Defining networking between services		100	
	Steps	Set value “apiVersion” to “v1”	14,28	
		Set value “kind” to “Service”	14,28	
		Set metadata value “name” to a specified name	14,28	
		Set metadata value “namespace” to an available namespace	14,28	
		Set spec value “selector” to a label defined on a pod or deployment	14,28	
		Set “ports” value to a list of objects containing a port definition	14,28	
		Set the ports definition which contains the “port”, the “protocol” and the “targetPort”	14,28	
	Defining an application		100	
	if	Running a service just 1 to n times		
	then	Define a “Job”	100	
	Steps	Set value “apiVersion” to “batch/v1”	20	
		Set value “kind” to “Job”	20	
		Set metadata value “name” to a specified name	20	
		Set metadata value “namespace” to an a vailable namespace	20	
		Define the value for the “spec” definition for the key “template” with a list of containers	20	
		Steps	Define a name for the container	2,22
			Define an image for the container	2,22
			Define a “restartPolicy”	2,22
			Define resource requests and limits	2,22
			Define liveness, readiness and startup probes (+)	2,22

5.6. The Deployability Metric

Kubernetes Resources Definition			Define a “completions” number (+)	2,22			
			Define a “parallelism” number	2,22			
			Define a list of values for the “imagePullSecrets” (+)	2,22			
			Define additional key/value pairs according to the documentation if needed	2,22			
	Else if	Running a service on a chronical basis					
	then	Define a “CronJob”			100		
	Steps		Set value “apiVersion” to “batch/v1”			16,66	
			Set value “kind” to “CronJob”			16,66	
			Set metadata value “name” to a specified name			16,66	
			Set metadata value “namespace” to an available namespace			16,66	
			Define the value for the “spec” definition for the key “schedule” with a cron expression			16,66	
			Define the value for the “spec” definition for the key “jobTemplate.spec” with a list of containers			16,66	
		Steps		Define a name for the container			1,85
				Define an image for the container			1,85
				Define a “restartPolicy” (+)			1,85
				Define a “completions” number (+)			1,85
				Define resource requests and limits			1,85
				Define liveness, readiness and startup probes			1,85
				Define a “parallelism” number (+)			1,85
				Define a list of values for the “imagePullSecrets” (+)			1,85
			Define additional key/value pairs according to the documentation if needed			1,85	
	Else if	Running a service indefinitely without internal state					
	then	Define a “Deployment”			100		
	Steps		Set value “apiVersion” to “apps/v1”			12,5	
			Set value “kind” to “Deployment”			12,5	
			Set metadata value “name” to a specified name			12,5	
			Set metadata value “namespace” to an available namespace			12,5	

5. Evaluating Deployments

Kubernetes Resources Definition	Steps	Define a spec key and add the “replicas” key with the specified number of replicas		12,5
		Set metadata value “labels” to a single/multiple key value pairs		12,5
		Define for the “spec.template.selector.matchLabels” key the label with the same label from the “container”		12,5
		Define the value for the “spec” definition for the key “template” with the following content:		12,5
		Steps	Define a “metadata” key with a “labels” key containing at least one label contained in the “spec.selector.matchLabels” key	6,25
			Define a "containers" key with a the container definition	6,25
		Steps	Define a name for the container	1,04
			Define an image for the container	1,04
			Define resource requests and limits	1,04
			Define liveness, readiness and startup probes (+)	1,04
			Define a list of values for the “imagePullSecrets” (+)	1,04
			Define additional key/value pairs according to the documentation if needed	1,04
	Else if	Running a service with internal state		
	then	Define a “StatefulSet”		100
	Steps	Create a “PersistentVolume”		7,69
		Set value “apiVersion” to “apps/v1”		7,69
		Set value “kind” to “StatefulSet”		7,69
		Set metadata key “name” to a specified name		7,69
		Set metadata key “namespace” to a specified name		7,69
		Define a spec key and add the “selector.matchLabels” key with a label which has to match “spec.template.metadata.labels”		7,69

5.6. The Deployability Metric

Kubernetes Resources Definition	Steps	Define a spec key and add the “serviceName” key with a specified name	7,69
		Define a spec key and add the “replicas” key with the specified number of replicas	7,69
		Define a spec key and add the “minReadySeconds” key with the specified number of seconds until every container is ready	7,69
		Define a spec key and add the “template.metadata.labels” key with labels that match “spec.selector.matchLabels”	7,69
		Define a spec key and add the “template.spec.terminationGracePeriodSeconds” key with a number	7,69
		Define a spec key and add the “template.spec.containers” key with a container definition	7,69
		Define a name a container	0,96
		Define an image a container	0,96
		Define resource requests and limits	0,96
		Define liveness, readiness and startup probes (+)	0,96
		Define a list of values for the “imagePullSecrets” (+)	0,96
		The “ports” definition contains a key “containerPort”	0,96
		The “volumeMounts” definition contains a “name” and a “mountPath”	0,96
		Define additional key/value pairs according to the documentation if needed	0,96
		Define a “volumeClaimTemplates” with a list of volumeClaim definitions, if no dedicated “PersistentVolumeClaim” exists	7,69
		Define a “metadata.name” which matches “template.spec.containers.volumeMounts.name”	1,92
		Define a value for “spec.accessModes”	1,92

5. Evaluating Deployments

		Define a value for "spec.storageClassName"	1,92
		Define a value for "spec.resources.requests.storage"	1,92

Table 5.3.: Software Deployment Definitions - Kubernetes Definition

Kubernetes Resource Definition As the main prospect in this thesis is to deploy software upon a Kubernetes instance, the definition of Kubernetes resources is essential and therefore has a valid placement in the "Main Actor" category. While there are several more Kubernetes resources available, in the metric they encompass all the objects used for deploying FactCheck services (as seen in Table 5.3) and essentials for deploying software in general. Server administration objects (e.g., Ingress, Secrets, Persistent Volumes, etc.) were especially excluded, because they are commonly defined and created by administrators rather than developers.

For all resources the first line defines which API should be used by Kubernetes when creating this resource. Which fields are allowed to be used when the developer defines them? It is always recommended to use the latest version for the longest compatibility⁷

Defining a namespace is, although highly probable and only needed once, an essential part of a deployment, as the used Kubernetes resources can only exist in a namespace. As a result, it is incorporated into deployments to ensure consistency⁸

Services, which describe Kubernetes resources, are meant to define how internal networking with one or multiple resources should take place⁹. This includes port forwarding as well as the protocols used.

For defining an application several different options are available in the metric which are: Jobs, CronJobs, Deployments, and StatefulSets¹⁰. There are several more available, but these were chosen as they are most commonly used for deploying software, as Kubernetes defines them as "workload" and are also the necessary resources for deploying FactCheck services.

Defining a "Job" resource¹¹, which is useful for applications that run once (or even multiple times back-to-back), requires several parameters to be defined in order to run it. Overall, global parameters like apiVersion, kind and metadata have to be set. Most importantly containers must be defined with a name and an image as minimum requirements. If necessary, additional parameters, such as environmental variables and several other parameters should be added. Also required is a "restartPolicy" parameter

⁷<https://kubernetes.io/docs/reference/using-api/>

⁸<https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>

⁹<https://kubernetes.io/docs/concepts/services-networking/service/>

¹⁰<https://kubernetes.io/docs/concepts/workloads/>

¹¹<https://kubernetes.io/docs/concepts/workloads/controllers/job/>

which can be set to "Never" or "OnFailure" so that the behavior changes on an unexpected completion. It is also recommended to define "completions" as how often the job should be executed and "parallelism" as how many jobs should be executed at the same time. It is recommended to use an "imagePullSecret" which is a secret that contains all the necessary credentials for accessing a Docker Registry, depending on the availability of the image.

The definition of "CronJob" is an extension of the "Job" resource¹². Here, the job definition is included via the "jobTemplate" parameter. Additionally, a schedule must be defined, which is done by setting a "cron expression" and by specifying the interval in which the job should be executed.

For software that is meant to run indefinitely, a "Deployment" is the most likely workload to be defined¹³ and particularly useful for software that is "stateless" and therefore not reliant on historical information. Global parameters like apiVersion, kind and metadata have to be set. Optionally a value for the number of replicas for this "Deployment" can be set to allow for scaling. It is crucial to note that the labels specified in these definitions must share at least one common key/value pair. This is due to the fact that the Deployment definition abstracts the creation of ReplicaSets and Pods, but needs to find the corresponding entities in Kubernetes based on these labels. When defining a pod with the "template" definition, a list of containers has to be defined, including their name, their image, and additional parameters if necessary.

For stateful applications the definition of a "StatefulSet" is necessary¹⁴. A prerequisite for this definition is an already-created "PersistentVolume" which defines a storage location. Initially, global parameters like apiVersion, kind, and metadata have to be set. As with Deployments, matching label definitions have to be added for Pods, ReplicaSets and the StatefulSet to find each other on the Kubernetes instance. The definitions of the "minReadySeconds" and "terminationGracePeriodSeconds" keys are optional but crucial, as they enable uninterrupted rolling updates and graceful termination, respectively. When defining a Pod with the "template" definition a list of containers has to be defined, including their name, their image, and additional parameters if necessary. Additionally, a "volumeMount" key has to be defined, which describes the path inside the container on which the storage location will be mapped, to be accessible by the software. To enable this, a "volumeClaimTemplates" key must be defined, which binds the "PersistentVolume" to the application and allows access to the storage location. In addition, parameters can be defined to allow for detailed access control.

¹²<https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/>

¹³<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

¹⁴<https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>

5. Evaluating Deployments

5.6.2. Supporting Actors

All requirements and features described in this section belong to the so-called "Supporting Actors." The main objective is to either directly support the deployment process or provide better integration with the deployment environment.

Supporting Actors			
Software Deployment			
Feature	Requirement		Impact
Helm Structure Definition	Create a root folder for all following files and folders		100
	Create a folder called "charts" and add "subcharts" on which the application depends		100
	Create a folder called "templates"		100
	Steps	Create a folder called "tests" and add chart files for testing if the deployment is working as expected	50
		Create for each used Kubernetes resource a dedicated YAML file and replace values via the templating pattern used by Helm	50
	Create a Chart.yaml file		100
	Steps	Set a value for the "name" key	20
		Set a value for the "description" key	20
		if Chart is deploying an application	
		Then Set "type" to "application"	20
		Else if Chart is used for utilities and functions for other chart developers	
		Then Set "type" to "library"	20
		Set or update the value of the "version" key on any change to the chart (+)	20
		Set or update the value of the "appVersion" key on any change to the software (+)	20
	Create a values.yaml file		100
	Steps	Create for each template pattern used in the resource definition YAML file a dedicated structure with all necessary values.	100

Table 5.4.: Software Deployment - Helm Structure Definition

5.6.2.1. Software Deployment

Helm Structure Definition This feature is placed in the "Supporting Actors" group for two reasons. The first reason is that it is optional to define, and it makes defining repeated information easier, which improves convenience. The second reason is that deployments through helm, which is also often described as a package manager for Kubernetes, require a dedicated repository, which must be set up separately. The most commonly used structures and components can be found in Table 5.4.

Commonly, all helm-related files are stored in a root folder. The definition of these files results in a valid Kubernetes deployment¹⁵.

Other charts (e.g., archive files with chart definitions) can be included in the "charts" folder. These are charts on which the main chart depends, and will be installed before the main chart itself¹⁵.

All Kubernetes resources are included in the template folder as well as a tests folder with all chart tests. Kubernetes resources are defined by providing each resource with a dedicated file containing the resource's YAML definition and the necessary values replaced via the helm templating language. In the tests folder, a developer can define test YAML files that check if the deployment worked as expected¹⁵.

The Charts.yaml file contains the chart's metadata, including its name, description, version and appVersion¹⁵. The "version" value defines the version of the chart and the "appVersion" number defines the version of the software to be installed. Additionally, a key called "type" can be defined. When the value "application" is used, the chart is dedicated to standalone applications. When the value "library" is used, the chart is should contain utilities and functions for the chart builder, they can however not be installed on their own.

The values.yaml file contains, as the name suggests, values for replacing all template patterns defined in the available resource definitions¹⁶.

¹⁵ https://helm.sh/docs/chart_template_guide/getting_started/

¹⁶ https://helm.sh/docs/chart_template_guide/values_files/

5. Evaluating Deployments

Supporting Actors						
Software Deployment						
Docker Compose Definition	Define the Docker Compose Version			100		
	Define networks			100		
	Steps	Define a list of networks		20		
		Define a name for each network		20		
		Define the driver		20		
		Choices	bridge (+)		20	
			host			
			overlay			
			ipvlan			
			macvlan			
			none			
	Define other attributes if needed according to the documentation		20			
	Define a list of available volumes			100		
	Define a list of secrets			100		
	Steps	Define a “file” secret and the path to the file in which the secret is stored		50		
		Define a “environment” secret where a secret is created from an environmental variable		50		
	Define a list of services			100		
	If	Service represents the currently worked on project				
	then	if	Development is happening			
		Then	Use a container for development		100	
		Steps	Choices	Use custom development container		33,33
				Use an OS (e.g. Ubuntu) base image		
				Use a Programming Language base image (e.g. Python)		
			Install dependencies		33,33	
		Mount the project folder to the container		33,33		
		Else if	Testing of the software for running inside of a container			
		then	Use the build mechanism		100	
			Create “build” definition		16,66	
	Define a “context” which is the path to the project		16,66			

Steps

		Define a “dockerfile” which is the path to the Dockerfile that should be build	16,66
		Define “no_cache” and set to True or False (+)	16,66
		Define “args”, by defining key-value pairs for “ARG” values inside the Dockerfile	16,66
		Define other configuration if needed according to the documentation	16,66
	Else If	Service is an external/existing software image	
	then	Define a “image” and assign it the “tag” of the external software	100
	Add additional container configuration to a service if necessary		100
	Steps	Define the mapping of ports accessible from the host system to the internal software	10
		Define necessary environmental variables	10
		Define a hostname (+)	10
		Define a container name	10
		Define the network on which the software is communicating with	10
		Define the volumes on which a software can store/access data on the host system or from a named volume	10
		Define via “depends_on”, which services should be healthy before the current one is started	10
		Define a health check if the current software is running and healthy (+)	10
		Define secrets available to the container	10
		Define additional configurations according to the documentation if needed	10

Table 5.5.: Software Deployment - Docker Compose Definition

Docker Compose Definition The use of Docker Compose in a project not only helps when developing software, as development environments can be set up locally, but it can also be used for production environments. For these reasons, and because it is optional for the deployment process, it was placed in the "Supporting Actors" group and is available in Table [5.5](#).

Although optional, to guarantee that the syntax used in the docker-compose file is valid

5. Evaluating Deployments

and compatible, the version key has to be defined. For current installations, at least version 3 is recommended¹⁷.

It is also optional to define one or multiple networks, as a default network is provided¹⁸. If communication between containers has to be regulated, which is a highly recommended step, and should include the definition of names, drivers, and additional configuration.

Named volumes, used to store persistent data, should be defined at the top of a Docker Compose file and allow for sharing volumes between multiple services and advanced configuration¹⁹.

When using confidential information in a docker-compose file it is recommended to use objects called "Secrets". With these objects in place a mechanism can be used that enables directly mapping a secret file or an environmental variable into a container (similar to a volume) for secure access²⁰.

To run software inside a container several options are available. Development can be done in a dedicated development container to create a consistent environment within a team. These include necessary dependencies and base operating systems²¹.

Directly running the currently developed software can be done by using the included build mechanism and a provided Dockerfile. Furthermore, this build mechanism can be adapted to fit the build's current context²².

The last option is to execute existing images by providing the image's tag which can be loaded from a publicly available or private Docker registry²².

In addition to defining the service, several attributes can be defined that specify the interaction with the environment and configuration information²². Although most are optional, they are highly recommended.

To allow access from the host system, port-mapping of container ports to host ports is the primary mechanism. Also, for communication between containers, defining a hostname and network is highly recommended²³. Important to mention is also defining volumes, which can either be named volumes defined in the compose file, or mapping of files on the host system into the container and vice versa¹⁹.

¹⁷<https://docs.docker.com/compose/compose-file/04-version-and-name/>
¹⁸<https://docs.docker.com/compose/compose-file/06-networks/>
¹⁹<https://docs.docker.com/compose/compose-file/07-volumes/>
²⁰<https://docs.docker.com/compose/compose-file/09-secrets/>
²¹<https://containers.dev/overview>
²²<https://docs.docker.com/compose/compose-file/build/>
²³<https://docs.docker.com/compose/networking/>

Supporting Actors						
Software Structure						
Python Software Structure	Managing python versions			100		
	Choices	Use pyenv (+)		100		
		Use docker				
	Managing dependencies			100		
	If	Working on multiple python projects with different conflicting dependency versions				
	then	Use virtualenv/venv for locally stored dependencies (+)		100		
	Else	Install dependencies system globally (-)		100		
	if	Developing an application with external dependencies				
	Then	Choices	pip (+)	100		
			poetry			
			Conda			
	Else if	Developing a library or a package				
	then	Use setuptools		100		
	Definition of a folder structure			100		
	Steps	Define “src” folder containing source files			20	
		Steps	Define a folder named after the project		10	
			Steps	Add subfolders to group modules		5
				Add python and other source files		5
					Define a “data” folder for binary data files	
		Define a folder “config” containing configuration files			20	
		Define a “venv” folder and generate the virtual environment files via virtualenv / venv			20	
		Define a folder containing documentation			20	
		Define a “tests” folder in project root folder			20	
			Define for each methodology a different folder containing tests		10	
			Define a separate configuration file if necessary (+)		10	
		Provide a “.gitignore” file in the project root folder			100	

5. Evaluating Deployments

	Provide a requirements.txt file in the project root folder	100
	Provide a LICENSE file in the project root folder	100
	Provide a README.md file in the project root folder	100
	Provide a pyproject.toml file in the project root folder	100
	Provide a CONTRIBUTING.md file in the project root folder	100
	Provide a CHANGELOG.md file in the project root folder	100

Table 5.6.: Software Deployment - Python Software Structure

5.6.2.2. Software Structure

Python Software Structure As software can be structured in any way inside a container, as long as a valid entry point is given, applying best practices regarding structure and functionalities results in easier and more consistent deployments. The recommended elements can be found in Table 5.6. Therefore, this section is placed inside the "Supporting Actors" group.

If several different Python versions are used for different projects, clear separation and management of these versions is necessary and can be done either by pyenv²⁴ or dedicated docker containers²⁵. This is recommended to avoid executing the wrong version, resulting in unexpected errors.

When various projects are developed, that have different or even conflicting dependency versions, the use of virtual environments is recommended. Otherwise, all dependencies will be installed globally, using only the most recent version of each dependency, potentially introducing unwanted bugs into projects²⁶.

There are several options for application dependency management, each with pros and cons. Here a selection of the most popular options is given. When developing a standalone library for other Python projects, the "setuptools" library is recommended²⁷.

The recommended folder structure is based upon the "Python Packages" book but is extended with a definition for configuration file placement. The .gitignore, requirements.txt, and pyproject.toml files provide meta- and build-information on the project. The LICENSE, README.md, CONTRIBUTING.md, and CHANGELOG.md files are important documentation tools for future maintainers and users of the software²⁸.

²⁴<https://realpython.com/intro-to-pyenv/>

²⁵<https://kevinquinn.fun/blog/manage-multiple-python-versions-with-docker-an-alternative-to-pyenv/>

²⁶<https://docs.python.org/3/library/venv.html>

²⁷<https://packaging.python.org/en/latest/guides/tool-recommendations/>

²⁸<https://py-pkgs.org/04-package-structure#including-non-code-files-in-a-package>

Supporting Actors			
Software Availability Features			
Database Access Check	Check if database resources are available		100
	Steps	A connection to the database is possible	25
		The database itself and if data is available	25
		The database structure is correct (+)	25
		The data itself is in the correct structure	25
Configuration Check	Check if configuration resources are available		100
	Steps	The configuration file is available	33,33
		The configuration file itself contains valid data	33,33
		The configuration file is for the correct environment	33,33
API Access Check	Check if API resources are available		100
	Steps	All necessary external APIs are accessible and return data	25
		The correct API version is used when accessing it	25
		Disable any available caching when executing checks	25
		API response has the expected structure	25
Environment Check	Check if environmental resources are available		100
	Steps	Check if a valid Python version is installed (+)	33,33
		Check for enough disk space	33,33
		Check for network connectivity	33,33
Health Probes	Create API endpoints for the states of the software for external state checkers		100
	Steps	Create a dedicated endpoint for the liveliness of the software	33,33
		Create a dedicated endpoint for the readiness of the software	33,33
		Create a dedicated endpoint if the software is currently starting up	33,33

Table 5.7.: Software Availability Features

5.6.2.3. Software Availability Features

All checks were placed in the "Supporting Actors" group, as they provide only additional information for the deployment environment and are therefore not required. These checks can be found in Table 5.7. They especially help when updating and releasing new software versions, as new versions will be accessible when all health probes succeed.

5. Evaluating Deployments

Database Access Check Provides database availability and data consistency checks helps to ensure reliable data storage mechanisms and data consistency²⁹.

Configuration Check These checks provide the necessary assurance that, in the event of a wrongly used or invalid configuration file, the software does not access unwanted data or does not function correctly³⁰.

API Access Check To allow for correct and complete functionality, external APIs should be checked for their availability and correctness so that unwanted errors are prevented³¹.

Environment Check Such checks, which should happen before the software is installed, provide the necessary assurance that the software will start up in the current execution environment³².

Health Probes These health probes provide information on REST endpoints that the deployment environment can access. In the Kubernetes environment, older versions will not be terminated until all probes of the new version succeed, when configured accordingly³³.

²⁹<https://www.billyokeyo.com/posts/health-checks-dotnet/>

³⁰<https://tech.preferred.jp/en/blog/working-with-configuration-in-python/>

³¹<https://realpython.com/python-api/>

³²<https://docs.python.org/3/library/platform.html>

³³<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>

Supporting Actors			
Software Upgradability			
Database Upgrades	Upgrading the structure / data of a database		100
	If	Using the Django Framework	
	Then	Use provided migration functionality	100
	Else if	Using SQLAlchemy dependency	
	Then	Use the migration tool Alembic	100
	Else if	Using Oracle or ODBC (or PostgreSQL, MySQL and SQLite)	
	Then	Use the Yoyo Migration Tool	100
	Else if	Using MySQL (incl. Commercial), MariaDB, PostgreSQL, DuckDB	
	Then	Use the Pyway Migration tool	100
	Else	Write your own migration scripts	100
Version- Management	Change source files and determine release type		100
	Choices	Patch – for bug and maintenance fixes	100
		Minor – for new features, upgrades and optimizations	
		Major – for breaking changes	
	Document changes via a changelog (+)		100
	Increase version number according to the release type		100
	Choices	Automatic version upgrades via tool	100
		Manual version upgrade	
	Execute any pre-release scripts (tests, documentation builds, etc.)		100
	Commit changes to version control system		100
	Create a tag on the version control system with the new version number (+)		100
API Management	Providing an API for external access		100
	Steps	Create Versions for breaking changes in the API	50
		Create metainformation endpoint with the following information	50
		Steps	API Version
			Endpoints and Operations
			Authentication and Authorization
			Rate Limits and Quotas
			Response Formats
			Error Handling

5. Evaluating Deployments

		Documentation Links	5
		Change Log	5
		Contact Information	5
		Terms of Service	5

Table 5.8.: Software Upgradability

5.6.2.4. Software Upgradability

Upgrading software is a common part of a software's lifecycle and, therefore, also influences deployments, as software stability and availability have to be accounted for. This entry, like all other entries in the "Supporting Actors" group, is optional but recommended, and can be found in Table [5.8](#).

Database Upgrades A database upgrade, which includes changes to data structures and data definitions, is a highly volatile operation, as in the worst case, data loss can happen. Therefore, several migration tools, depending on the database and the frameworks used, are recommended [34](#)[35](#)[36](#)

Version Management Upgrading versions frequently results in new deployments, so it plays a critical role in the deployment mechanism. Here, the version should be changed based on the release type, and depending on the kind of changes, a different value of the version number should be increased. Also, describing important and especially breaking changes results in better traceability when errors happen after a deployment. To increase a version number, the option between a manual and an automatic version upgrade is available. When automatic version upgrades and automatically run pre-release scripts are chosen, the deployment process can be accelerated. Documenting the release in a version control system like Git allows for easier traceability and rollbacks to previous versions [37](#).

API Management When software provides APIs for external software, it is recommended that API versioning and meta-information endpoints be introduced. API versioning provides the external software with the most basic information on which version of the API they are connected to. Meta-information endpoints then provide additional information for external developers to consider in terms of compatibility with the current API and its changes [38](#).

³⁴<https://docs.djangoproject.com/en/5.0/topics/migrations/>

³⁵<https://medium.com/@johnidouglasmarangon/using-migrations-in-python-sqlalchemy-with-a-lemibic-docker-solution-bd79b219d6a>

³⁶<https://github.com/jasondcamp/pyway>

³⁷<https://py-pkgs.org/07-releasing-versioning.html>

³⁸<https://www.moesif.com/blog/technical/api-development/essential-REST-API-best-practices/>

Supporting Actors				
Software Configuration Structure				
Development and Production Configuration	Define a configuration file		100	
	if	Separate files for each environment		
	then	Environment check on startup	50	
		Loading of specified configuration file	50	
	Else if	Single YAML file for all environments		
	Then	use the “three dashes” operator to internally separate configuration between environments and only use the environment relevant configuration	100	
	Else	use profiles, which activate part only the environment relevant configuration	100	
Configuration Injection	Providing production configuration files		100	
	Choices	Build Time Injection		100
		if	Separate configuration files for production and development environment are available	
		then	Creating build time arguments for identifying and locating the production configuration	50
			Loading production configuration while building and replace development configuration	50
		Else if	Single configuration file for both production and development environment are available	
		then	Activate/Deactivate respective configuration via profiles set via environment variables on build time	100
		Startup Injection		100
		Steps	Upload or make the production configuration file available for the service in a predefined location	33,33

5. Evaluating Deployments

			Set the location to the file via build time arguments	33,33	
			Load and apply the configuration file once at startup	33,33	
		Automatic Injection			100
		Steps	Upload or make the production configuration file available for the service in a predefined location	25	
			Set the location to the file via build time arguments	25	
			Continuously check the configuration server if an updated version is available	25	
			Load and apply updated configuration	25	
		Configuration File Format	Choosing a file format for configuration data		
Choices	ini - File Format		100		
	YAML - File Format (+)				
	JSON - File Format				
	Python - File Format (-)				
	Other custom formats				

Table 5.9.: Software Configuration Structure

5.6.2.5. Software Configuration Structure

Not all software requires dedicated configuration files, but for those which do, a workflow for handling these files should be established. Therefore, this entry was placed in the "Supporting Actors" group. These recommended workflows can be found in Table 5.9.

Development and Production Configuration As software more often than not requires different configurations for different environments, a dedicated configuration file has to be provided³⁹. There are several options described, each of which solves this issue based on the file type or by allowing software to only read the part of the configuration relevant to the current environment⁴⁰.

³⁹<https://learn.microsoft.com/en-us/aspnet/web-forms/overview/older-versions-getting-started/deploying-web-site-projects/common-configuration-differences-between-development-and-production-cs>

⁴⁰<https://dxiaochuan.medium.com/summary-of-python-config-626f2d5f6041>

Configuration Injection Configuration files often contain confidential information, such as passwords and API-keys, especially in a production environment. Therefore, the secure storage and management of these files is necessary. To get these securely stored files into the software three methods for injecting them are described.

Build-time injection will provide the production configuration for each software build process. Implementation is straightforward, yet every modification to the configuration file necessitates a fresh build⁴¹.

Startup injection has the advantage that the software automatically retrieves its own configuration at startup and is fairly easy to implement. Its main disadvantage is that if any changes to the configuration file occur, the software must be restarted⁴¹.

Automatic injection enables the continuous modification of software configuration by automatically detecting and applying changes. Its main disadvantage is the implementation effort, which requires that the software adapt itself to a new configuration⁴¹.

Configuration File Format In this section, several options and recommendations for configuration file formats are displayed. Depending on the use case and the frameworks used, this might affect the choice. Highly recommended is the YAML file format, due to its readability and extensibility⁴².

⁴¹<https://reflectoring.io/externalize-configuration/>

⁴²<https://martin-thoma.com/configuration-files-in-python/>

5. Evaluating Deployments

Supporting Actors			
Software Security			
Secure Configuration	Handle critical information in configuration files		100
	Choices	Remove all critical information from configuration files and use alternative secure storage options	100
		Remove all configuration from project and instead load on demand from secure area	
		Load critical information from environmental variable	
		Encrypt critical information	
	Monitor and log access to critical information		100
	Rotate credentials in critical information regularly		100
External Communication	Communication in the same environment with other services		100
	if	Communication with other services via a (public) network	
	then	Use secure messaging protocols	100
	If	Two processes communicate on same device	
	then	Use inter-process communication when possible	100
	If	Communication with software with support for messaging queues	
	then	Use secure messaging queues	100
Dependency Management	Choosing a dependency		100
	Steps	Select dependencies based on usefulness, community support, stability, licensing and time since last update	50
		Check for possible vulnerabilities in public databases	50
		Steps	
		Check and evaluate CVE Score	25
		Check and evaluate CVSS Score	25
	Upgrade existing dependencies to latest version		100
	Compare SHA-256 checksum with the downloaded library		100

Table 5.10.: Software Security

5.6.2.6. Software Security

Securing software is also an important subject in the context of deployments. Interactions with external software, dependencies, data, and communication methods are entry points for software exploitation. The recommended handling for these situations can be found in

Table 5.10. It is placed in the "Supporting Actors" group as it is not directly responsible for deployments.

Secure Configuration For handling confidential information in configuration files several options are available. Removal of this information and the use of alternative secure access and storage mechanisms are highly recommended, as non-existent data is more secure than encrypted data⁴³.

Also, logging and monitoring access to critical information should be implemented, as possible security breaches could be detected if unexpected or too many accesses happen. To prevent even more attacks, stored credentials should be regularly rotated so that even when a security breach happens, the time for exploitation can be reduced⁴⁴.

External Communication Communication with external services should be done in a secure manner and must be available in the deployed environment. This includes secure messaging protocols (TLS, SSL, etc.)⁴⁵ inter-process communication (named pipes, etc.), and secure messaging queues (Kafka, Pub/Sub, etc.)⁴⁶. All of these must be configured correctly to enable secure communication.

Dependency Management Supply-chain attacks, vulnerabilities, and counterfeit dependencies can be possible attack vectors when wanting to compromise a system. These attack vectors can be introduced into the software during the deployment process at build time via outdated or compromised dependencies. Therefore, routinely upgrading dependencies and validating them against their SHA-256 checksum provides a baseline for security. Nevertheless, dependency choice is important because outdated and non-supported dependencies may be compromised in the future, resulting in complicated porting to alternatives⁴⁷.

⁴³<https://www.kamilgrzybek.com/blog/posts/how-to-store-sensitive-configuration-data>

⁴⁴<https://www.akeyless.io/secrets-management-glossary/secret-rotation/>

⁴⁵<https://www.infosecinstitute.com/resources/network-security-101/a-deep-dive-into-network-security-protocols-safeguarding-digital-infrastructure-2024/>

⁴⁶<https://dev.to/nikl/choosing-the-right-message-queue-technology-for-your-notification-system-2mji>

⁴⁷<https://learn.microsoft.com/en-us/nuget/concepts/security-best-practices>

5. Evaluating Deployments

5.6.3. Supplementary Actors

In the category "Supplementary Actors", features were chosen that only indirectly support deployments. Furthermore, these are primarily useful for production deployments and to increase software availability.

Supplementary Actors			
Metrics Aggregation			
Metrics Collector Integration	Choose a metric collection software		100
	Choices	Prometheus (+)	100
		SigNoz	
		Graphite	
		Others	
	Use metric collectors for collecting metadata based on observability strategies		100
	If	Hardware resource availability should be tracked	
	then	Apply the “USE” method	50
	Steps	Collect Utilization metrics	16,66
		Collect Saturation metrics	16,66
		Collect Error metrics	16,66
	If	User Experience should be tracked	
	Then	Apply the “RED” method	50
	Steps	Collect Rate metrics	16,66
		Collect Error metrics	16,66
		Collect Duration metrics	16,66
Metric Visualization	Choose a metric visualization software		100
	Choices	Grafana (+)	100
		OpenSearch	
		SigNoz	
		Datawrapper	
		Redash	
		Others	
	Connect metric collector to a compatible visualization provider		100
Create dashboards for collected metrics		100	

Table 5.11.: Metrics Aggregation

5.6.3.1. Metrics Aggregation

The aggregation of metrics was placed in the "Supplementary Actors" category, as these features provide mainly support in maintaining a valid software state by collecting and visualizing data. The recommended steps can be found in Table 5.11.

Metrics Collector Integration The most prominent options for metrics collectors are displayed. "Prometheus" was chosen as the recommended option because it is the most popular, open-source, and widely supported option⁴⁸.

Depending on which metric should be tracked, a different set of methods can be chosen to collect data. For hardware resources, the "USE" method can be used to check if the environment has enough resources to support all deployments. For user experience tracking, the "RED" method can be applied. In this case, user experience tracking focuses mainly on throughput and error rate, which can indicate scaling, deployment, and software issues⁴⁹.

Metrics Visualization To make the collected data understandable to humans, visualization tools are necessary. A selection of popular visualization tools is given, with the recommended choice being Grafana, as it is highly popular, open-source, and widely used. After connecting it to the metrics collector of choice, the collected data can be compiled into dashboards. These can display a variety of information in different graphs, tables, and gauges⁵⁰.

⁴⁸<https://www.techtarget.com/searchitoperations/feature/Compare-8-tools-for-IT-monitoring>

⁴⁹https://lantern.splunk.com/Observability/Product_Tips/Observability_Cloud/Adopting_monitoring_frameworks_-_RED_and_USE

⁵⁰<https://www.metricfire.com/blog/top-8-open-source-dashboards/>

5. Evaluating Deployments

Supplementary Actors			
Software Availability			
Performance Management	Provide a load balancer in front of the application		100
	if	Software is deployed on-premises and extremely high load is expected	
	Then	Use hardware load balancers	100
	Else if	Software is deployed on-premises and a small to medium traffic is expected	
	Then	Use software load balancers	100
	Else if	Software is deployed in the cloud or has access to cloud resources	
	Then	Use cloud providers load balancer	100
	Use caching mechanisms		100
	Steps	In-Memory Caching	16,66
		Reverse Proxy Cache	16,66
		Database Query Result Cache	16,66
		Application-Level Caching	16,66
		Content Delivery Networks	16,66
		Browser Caching	16,66
Data Management	Create a scalable database setup		100
	Steps	Create a dedicated database for each service	25
		Provide replicas for each database	25
		Continuously replicate data between databases of the same service	25
		Provide a cache in front of the databases	25
	Create an automatic backup and disaster recovery mechanism		100
	Implement security measures to protect data against unauthorized access		100
	Implement safety measures against unintended data destruction		100

Table 5.12.: Software Availability

5.6.3.2. Software Availability

The software availability category was placed in the "Supplementary Actors" category as the provided features only increase the availability of software and help to keep the

deployment stable. The recommended steps can be found in Table 5.12

Performance Management Given that software availability depends on its ability to manage resources and process requests efficiently, it is advisable to place a load balancer in front of the application. This software handles the distribution of requests among several instances of the same application, which in turn increases throughput and effective resource usage⁵¹.

Providing several caching layers is also recommended, because it enables faster access to data but there is a risk of outdated data being provided. Additionally, it reduces resource usage, which results in faster response times for non-cached content⁵².

Data Management The distribution of data among several databases also increases the software's scalability and availability. Since all databases cannot handle all storage requests at once, the replication of data to other deployments of the same database allows for "eventual consistency"⁵³.

Implementing automatic backups and data recovery mechanisms also results in increased availability, even when disasters strike. To prevent disasters in the first place, security and safety measures should be implemented. These measures help prevent data theft and destruction, which in turn increases software availability⁵⁴.

⁵¹<https://www.databasejournal.com/features/5-advantages-of-load-balancing-for-it-companies/>

⁵²<https://medium.com/@aggarwalapurva89/different-caching-techniques-660ad8d97a8a>

⁵³<https://www.fivetran.com/blog/distributed-databases>

⁵⁴<https://www.linkedin.com/advice/3/how-can-you-ensure-your-database-project-fault-tolerant-w8j6c>

5. Evaluating Deployments

Supplementary Actors			
Software Management			
Software Management	Design software to allow replication		100
	Steps	Apply a microservice architecture	25
		Apply state management patterns	25
		if Software must depend on an internal state	
		Then Use stateful replication patterns	
		Choices Active – Passive replication	25
		if Software does not rely on internal state	
		Then Apply stateless design throughout the software	25
		Support externalized configuration	25
		Support health monitoring	25
	Use automatic failover mechanisms		100
	Use rolling update and rollback mechanisms		100
	Use monitoring tools		100
	Use geographic redundancy mechanisms		100
	Use high availability hardware and horizontal scaling		100

Table 5.13.: Software Management

Software Management Software, as it grows, has to be adapted to a growing user base, resource usage and new functionality. To support these circumstances the Table 5.13 shows the necessary steps. Multiple deployments of the same software, also known as replicas, are included in deployment mechanisms. To support software replication the software itself must be supported. This replication process is easily feasible for stateless applications, as they do not depend on previously supplied data and can be started as often as wanted⁵⁵. On the other hand, stateful applications have to handle their data consistency through data replication to achieve their own replication⁵⁶. Externalized configuration is another helpful mechanism, as replicas should all have the same configuration to ensure consistency⁴¹. Furthermore, with replicated software the importance of health monitoring mechanisms increases, as an increase in number also leads to an increase in data and errors⁵⁷.

Kubernetes also supports a wide variety of tools to support new deployments, upgrades, failovers, retries, and monitoring mechanisms. Furthermore, geographic redundancy,

⁵⁵<https://blog.dreamfactory.com/stateful-vs-stateless-web-app-design#2>

⁵⁶<https://semaphoreci.com/blog/stateful-applications-kubernetes>

⁵⁷<https://de.catchpoint.com/api-monitoring-tools/microservices-monitoring>

high hardware availability, and horizontal scaling are supported, which in turn increases software availability⁵⁸.

5.7. Usage

This metric is intended to serve as a quick comparison and evaluation tool for deployments. Therefore, if such a comparison or evaluation should take place it is recommended to use a top-down approach. With this approach the first step is to evaluate the environment. Here, the expected environment has to be selected, so either development or production environments are possible choices. Furthermore, the minimal or recommended requirement set must be selected. This process influences the result of the evaluation, as the set of necessarily fulfilled requirements changes depending on this choice.

Once this decision is made, one can start evaluating the tables, selecting the actors and features that need to be fulfilled based on the chosen environment. As there are several requirements for each feature, the fulfillment of these requirements means progress in the fulfillment of the overall feature. It is important to mention that not every requirement in a feature has to be fulfilled to be functional or even to be seen as completed. This is solely dependent on the user's expectations and requirements.

Because only a subset of all requirements in a feature needs to be fulfilled, the comparison aspect between two deployments must be preserved. Here, the impact value comes into play. Whereas a direct 1:1 comparison of fulfilled requirements could be seen as a comparison option, the impact value allows for a comparison of partial fulfillment of the overall feature. For the calculation of the impact value on a partially fulfilled feature the steps and sub-steps of a requirement can be used for detailed comparisons and a better understanding of the overall state.

Finally, by calculating the impacts of the used features in the metric and applying the environmental constraints, the software can be evaluated to see if it is ready to be deployed in the selected environment. Furthermore, two deployments with a similar software stack can be sufficiently compared to evaluate which one is further in its deployment process.

⁵⁸<https://www.tigera.io/learn/guides/kubernetes-security/microservices-kubernetes/>

6. Methodology

This chapter aims to transparently show how choices were made and which methods were used for the individual topics.

6.1. Prototypical Implementation

6.1.1. Requirement Choices

Before developing any (prototypical) software implementation, the software requirements must be defined. To do this, a top-down approach was used, first defining the overarching topics which, in this case, are the research questions, and then narrowing down until specific features and their functionality could be defined. The requirements in the context of this thesis mainly focus on project generation, software deployments, and student access. To further narrow down these overarching topics they were split up and transformed into requirements, features and last but not least the implementation itself.

6.1.2. Development Methods

This top-down approach allowed for an incremental development cycle in which a subset of all features was developed while implementation-specific details of other requirements and topics were specified later. The length and content of each specific cycle were defined together with the supervisors of this thesis. This resulted in a period of approximately one month between status meetings and the evaluation of current progress. Each of these definitions and evaluations was documented in an internal OneNote notebook and reviewed in the next cycle. Overall, this approach was similar to a simplified variant of the SCRUM process model.

6.1.3. Architectural and Technological Choices

In this thesis on prototypical implementation, architectural patterns are used based on established design patterns. This includes communication, user interaction patterns and coding patterns. The following patterns were used:

- REST-APIs Pattern - used for communicating with the factcheck-config-service.
- YAML-Configuration Files Pattern - used for direct mapping between configuration files and classes.

6. Methodology

- Model-View-Controller Pattern - is used for user interactions in FactDeploy.
- Facade Pattern - used for abstracting deployments to Kubernetes in FactDeploy.
- Template Pattern - used for abstracting the access method to Kubernetes in FactDeploy.

The placement of these patterns in the implementation process was determined based on previous experience with software structuring and architecture.

The selection of technologies and dependencies was influenced by the following factors:

- Open-Source
- Well-maintained
- Popularity
- Ease of usage

All projects in this thesis were developed using Python, making them platform-independent. Furthermore, all platform-dependent functionality was abstracted to achieve compatibility with Linux-based and Windows systems.

6.2. Deployability Metric

The structure and content of the deployability metric had to be carefully chosen, as there are many different technologies and possibilities for defining features.

6.2.1. Methods for choosing features and best practices

The choice for including a certain feature was based upon several considerations, while the main impact was the work done by Bolscher et al. [BD19], which described beneficial software architecture characteristics for software deployments. This list served as the foundation for the definition of "feature groups" and was then adapted and extended to fit the deployment environment of this thesis. For instance, Agility/Modifiability were separated into "Software Upgradeability" and "Software Configuration Structure."

On this basis the necessary "features" in these "feature groups" had to be chosen. To choose the required features, the following steps were taken:

- Directly used by the environments found in this thesis, e.g., Kubernetes, Docker, and Python. For these features, the primary literature for their configuration was chosen as a basis. Therefore, the main focus was on official documentation.
- Directly referenced features by scientific literature as necessary for fulfilling a feature group.

These features were then further broken down into requirements. Requirements directly represent best practices that were either mentioned in the previously described resources for feature selection or recommended by publicly available resources created by experts

in the DevOps and software development/architecture fields. They are referenced via footnotes with links to the related articles or software documentations.

The steps, sub-steps, and choices describe a detailed view of a requirement and are therefore also based upon the aforementioned resources. Since the symbols used (+ and -) on this level refer to the step's recommendation, their usage is more subjective. Therefore, they correspond to the recommendation based on the author's experiences while writing this thesis and may be disregarded at the reader's discretion.

6.2.2. Writing Tools

The text in this thesis was primarily written using the online LaTeX editor called "Overleaf"^[1]. For the tables used in this thesis a combination of a Microsoft Word document and the online table to LaTeX-Converter called "Latex Table Editor"^[2] was used. For proofreading and paraphrasing an online AI tool called "Quillbot"^[3] was used. All UML diagrams displayed in this thesis were created via "StarUML"^[4] or "Lucidchart"^[5].

¹<https://de.overleaf.com>
²<https://latex-tables.com>
³<https://quillbot.com>
⁴<https://staruml.io>
⁵<https://www.lucidchart.com>

7. Conclusion

The chapter aims to evaluate the work done in this thesis, highlight the accomplishments, and critically reflect on the choices made.

7.1. Results

In this thesis, the creation of a framework for a development environment based on Kubernetes in the context of the FactCheck project was explored. The main contributions of this thesis include the development of tools that enable one-click deployments of a development environment and a deployability metric that facilitates comparison between deployments. Moreover, the integration of Azure hybrid cloud functionalities allows for direct usage of cloud services from an on-premise environment.

Several tools were required to enable the one-click deployment process. The main contribution was the creation of FactDeploy, which allows for the creation and deployment of Python projects upon an on-premise Kubernetes cluster without prior knowledge of Kubernetes deployment practices. This approach proved to be effective even for deployments in Kubernetes instances hosted in Azure, as the same architecture can be directly transferred.

Tools such as the factcheck-config-service and the factcheck-sync-service are also worth mentioning. The factcheck-config-service, which holds configuration files for FactCheck and other deployed services, shows how to securely encrypt, store, and retrieve configuration files without any access by third parties or even administrators. As production data is necessary for better testing, the development of the factcheck-sync-service showed the possibilities for managing data in a development environment but also the limitations for transporting these large amounts of data to it.

While these tools provide functionality in the deployment environment the factcheck-dev-container represents the local environment. This project enables users to develop in the same environment as their software, which is a running Docker image.

The usage of Kubernetes as a development environment was shown as highly usable for separating environments for users, and also proved to be a viable solution for providing access to services even in constraint environments such as the university network infrastructure. Access to the Azure cloud environment and services was also successful by providing a S2S connection to an on-premise Kubernetes cluster and a proof of concept

7. Conclusion

for a P2S connection. This allowed the on-premise services to access the Azure cloud environment and its features.

The definition of the deployability metric, which unites software architecture characteristics and best practices in its scope, is the continuation of the definition for measuring the deployability of software. This metric not only advances our understanding of the impact of software architecture on deployability, but also provides practical solutions for evaluating the progress and comparing the current deployability state of software. Furthermore, it allows student developers to evaluate their projects with a common denominator to identification their current progress towards a production deployment.

With these achieved results in place, the research questions and their derived objectives are seen as adequately fulfilled, as all defined requirements have been completed.

7.2. Discussion and Limitations

While the previous section shows the completed objectives and the overall results, there is still room to discuss possible improvements and minor limitations.

7.2.1. Evaluation of FactDeploy

The approach for using the custom software FactDeploy in conjunction with the objectives of this thesis proved to be effective, although time-consuming. While the project creation part is fully functional, it could get extended to include the selection of frameworks and additional examples so as to introduce and describe functionalities to the user. Furthermore, alternatives such as the CookieCutter project could have been used to create a more robust setup, and buildpacks could have been used to maintain alternatives to the Docker Images built by FactDeploy.

Instead of relying on SSH connectivity, a more efficient way to access and deploy Kubernetes could have been used. Although this would have required a complete restructure of the authentication against the Kubernetes instance, the speed of deployments would have increased substantially.

7.2.2. Evaluation of Auxiliary Services

Further research could have been conducted to find an alternative to the factcheck-config-service, but the current alternatives require more adaptations to existing projects' protocols. As a result, the factcheck-config-service implementation was deemed necessary.

A few compromises were also necessary when implementing the factcheck-sync-service. With stronger hardware, the limitations of using resources and allowing only a subset of the complete data to be synchronized could have been avoided. Although avoidance

would have resulted in faster development time, adaptations proved to be beneficial for less powerful systems.

Regarding the use of development containers, their maintenance could have been externalized using a publicly available project. As some FactCheck projects required specific customizations, dedicated Docker Images had to be created to support them.

7.2.3. Evaluation of the Hybrid Cloud Architecture

Although a hybrid cloud architecture was successfully implemented by providing a S2S connection (and a proof of concept for P2S connections) between the Azure environment and university hardware, yet several limitations have been observed.

Connections via P2S and S2S require several ports to be accessible, which may impact the functionality of deployed software or necessitate additional configuration. In a P2S connection via OpenVPN, port 443 would be permanently taken, as OpenVPN uses this port for establishing and running the VPN connection. Alternatives such as IKEv2 use different ports, which circumvent this limitation but require additional configuration for opening these ports. To even establish a connection with a Kubernetes cluster, access to the network on which the VPN is running must be configured, which requires administrator rights.

Connecting via a S2S VPN necessitated advanced networking configurations and administrative rights, resulting in high bureaucracy and communication efforts. This setup was thankfully handled by the FactCheck team, and therefore only the setup process of Azure Arc had to be done by the author. After setting up Azure Arc, another limitation was found. This meant that integrating an on-premise Kubernetes node into an Azure Kubernetes cluster was not possible, and only management of that on-premise node was allowed. While this does not hinder functionality, the initial idea of extending an existing cluster and spanning it across on-premise and cloud infrastructure could not be fulfilled.

All in all, the process of creating a hybrid cloud infrastructure resulted in a high effort for rather limited functionality regarding direct inclusion in the Azure ecosystem.

7.2.4. Evaluation of the Deployability Metric

Structuring of the deployability metric was limited by tools used in its creation. Here, the limitations stem from readability and granularity issues when displaying the metric. The choice for the maximum recommended sub-steps, to be three layers, stems from these issues. Furthermore, the software for creating these large tables, which in this case was Microsoft Word, Google Docs, and LaTeX, is not optimized, resulting in sluggish performance and issues with table structuring and display.

As for the content of the deployability metric, the scope of the included best practices was intentionally severely limited. This was due to the huge number of possible configurations

7. Conclusion

and technology options. Also, several options had to be combined into "additional configuration according to documentation" steps, as their complexity and overall usefulness for the objective were so limited that the need for readability was a higher priority than the advantages they provided. Furthermore, the usage of the deployability metric by students currently is based on a manual evaluation process which, in practices, would result in a substantial amount of effort. Tool support for the process could not be provided in this thesis as it would extend the scope enormously and would not be feasible in the available time frame.

7.3. Future Work

In this section, potential directions for future research and improvements are discussed that may build upon the work of this thesis. It outlines possible extensions and enhancements to the implementations, methodologies, and definitions laid out in the previous chapters.

One potential area for improvement is FactDeploy's templates, features, and usability. Future work could focus on:

- **JavaScript project support** - While the creation of Python projects is fully supported, JavaScript-related projects are currently not. This is due to the vast selection of frameworks. In theory, deployments to Kubernetes are possible when manually adapting the configuration of FactDeploy, but this was not tested in the course of this thesis.
- **Support for Python Frameworks** - Currently, project creation only supports plain Python projects. This could be extended to support frameworks like Django for the centralized creation of these projects.
- **Extension of Dockerfile Creation** - Dockerfile support is currently based on four main templates. This does not cover all of the possible and even necessary definitions for creating a Dockerfile. Therefore, the creation of a tool to dynamically create these files or advanced templating could make creating Dockerfiles more accessible and dynamic.
- **Further Kubernetes deployment types** - FactDeploy currently supports only a subset of all possible workloads in Kubernetes. Therefore, the extension for more Kubernetes workloads and other objects would benefit the current system.
- **Advanced user instructions** - Currently, every menu has its own set of instructions; this could be extended to every menu point if the necessary features are available in InquirerPy.

Several improvements and extensions could be made to the deployment of the on-premise Kubernetes instance, which could concentrate on:

- **Networking Capabilities** - Currently, the ingress listens directly to incoming

requests. This could be replaced by an on-premise load balancer if the need arises for handling large amounts of requests.

- **Worker node usage** - The on-premise Kubernetes instance is currently running on one node, which combines a control plane and worker node. Additional worker nodes could be added to allow for better scaling when the need arises.
- **Automatic storage provisioning** - All persistent data storage is currently manually created for each user; this could be replaced with an automatic provisioning method of storage infrastructure.
- **Authentication** - Users are currently required to authenticate themselves when accessing the university network, and they use "Basic Authentication" to access the HTTPS resources of the services they deploy on Kubernetes. This could be extended by providing centralized user management and authentication software, like, for example, Keycloak.

Regarding the deployability metric several adaptations would be possible, and the following points could be extended:

- **Automatic evaluation** - While manual use of the deployability metric is possible for humans, automatic evaluation via a dedicated tool is currently not available. Already, a partial automatic evaluation of parts based on structured content could make the metric easier to use and would greatly benefit future users.
- **Cohesive definition of best practices** Best practices are currently defined by a simple text passage and a more detailed explanation. This could be improved by defining a dedicated structure with fields like, e.g., pre- and post-requisites.
- **Further inclusion of technologies** - Currently only selected technologies, such as Python, are supported; this could be extended to several languages, deployment platforms, and other technologies.
- **Further inclusion of best practices** - As best practices change or extend over time, the appropriate adaptations will be necessary.
- **Evaluation on a user basis** - The usage of this metric on a broader group of people and the corresponding evaluation of its usability may bring a better understanding of the necessary structures and overall usability with a view towards improvements to this metric.

This chapter has outlined several potential directions for future work that can build upon the findings and implementations of this thesis. By addressing these areas, future research may enhance the methodologies and implementations and apply the findings to real-world problems, ultimately contributing to the definition of deployability and the technological background.

Bibliography

- [AG20] Ella Albrecht and Jens Grabowski. Sometimes it's just sloppiness - studying students' programming errors and manual onceptions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education, SIGCSE '20*, page 340–345, New York, NY, USA, 2020. Association for Computing Machinery.
- [Ame24] Data Science Council Of America. What makes python the go-to language for data scientists? *Data Science Council Of America*, April 2024. Available at: <https://www.dasca.org/world-of-data-science/article/what-makes-python-the-go-to-language-for-data-scientists> accessed on: 2025-01-12.
- [Aut25a] KEDA Authors. *KEDA Documentation*, 2025. Available at: <https://keda.sh/docs/2.16/> accessed on: 2025-01-12.
- [Aut25b] The Kubernetes Authors. *Kubernetes Documentation - Role Based Access Control Good Practices*, January 2025. Available at: <https://kubernetes.io/docs/concepts/security/rbac-good-practices/> accessed on: 2025-01-12.
- [Bar12] Eugene Bardach. *A Practical Guide for Policy Analysis: The Eightfold Path to More Effective Problem Solving 4E*. A Practical Guide for Policy Analysis: The Eightfold Path to More Effective Problem Solving. SAGE Publications, 2012.
- [BD19] Robin Bolscher and Maya Daneva. Designing software architecture to support continuous delivery and devops: A systematic literature review. In *Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019*, page 27–39, Setubal, PRT, 2019. SCITEPRESS - Science and Technology Publications, Lda.
- [BT22] Tomas Beuzen and Tiffany Timbers. *Python Packages*. CRC Press, April 2022. Available at: <https://py-pkgs.org> accessed on: 2025-01-12.
- [Car22] Carmen Carrión. Kubernetes as a standard container orchestrator - a bibliometric analysis. *Journal of Grid Computing*, 20(4):42, December 2022.
- [CG10] Jitender Chhabra and Varun Gupta. A survey of dynamic software metrics.

Bibliography

- Journal of Computer Science and Technology*, 25:1016–1029, September 2010.
- [Cou25] Apache CouchDB. *Replication - CouchDB Documentation*, 2025. Available at: <https://docs.couchdb.org/en/stable/replication/intro.html> accessed on: 2025-01-12.
- [DB22] Pieter De Bruin. *Manage and deploy Kubernetes in Azure Arc - Azure Architecture Center*, 2022. Available at: <https://learn.microsoft.com/en-us/azure/architecture/hybrid/arc-hybrid-kubernetes> accessed on: 2025-01-12.
- [Doc25a] Inc. Docker. *Compose file reference - Docker Documentation*, 2025. Available at: <https://docs.docker.com/reference/compose-file/> accessed on: 2025-01-12.
- [Doc25b] Inc. Docker. *Registry - Docker Documentation*, 2025. Available at: <https://docs.docker.com/registry/> accessed on: 2025-01-12.
- [Eku25] Martin Ekuan. *Microservices architecture on AKS - Azure Architecture Center*, 2025. Available at: <https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/containers/aks-microservices/aks-microservices> accessed on: 2025-01-12.
- [Fou25] Python Software Foundation. *3.12.4 Documentation*, 2025. Available at: <https://docs.python.org/3> accessed on: 2025-01-12.
- [GB23] João Gonalo and Martins Bonacho. *Agnostic Cloud Services with Kubernetes*. PhD thesis, Instituto Superior de Engenharia de Lisboa, 2023.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.
- [HRJ21] Michael Haider, Michael Riesch, and Christian Jirauschek. Realization of best practices in software engineering and scientific writing through ready-to-use project skeletons. *Optical and Quantum Electronics*, 53(10):568, September 2021.
- [Ini24] Open Container Initiative. *The OpenContainers Image Spec*, April 2024. Available at: <https://specs.opencontainers.org/image-spec/?v=v1.0.1> accessed on: 2025-01-12.
- [Int25] Digital Interaction. The evolution of integrated development environments: A historical perspective. *digitalinteraction*, 2025. Available at: <https://digitalinteraction.co.uk/the-evolution-of-integrated-development-environments-a-historical-perspective> accessed on: 2025-01-12.

- [Kal18] Peter Kalchgruber. *Detecting and resolving conflicting data on the web*. PhD thesis, University of Vienna, Wien, 2018.
- [KFMJ18] Vojdan Kjorveziroski, Sonja Filiposka, Anastas Mishev, and Boro Jakimovski. Designing a paas solution for large scale deployment of student projects. In *2018 26th Telecommunications Forum (TELFOR)*, pages 420–425, 2018.
- [Kue16] Chris Kuehl. Introducing dumb-init, an init system for docker containers. *Yelp Engineering Blog*, January 2016. Available at: <https://engineeringblog.yelp.com/2016/01/dumb-init-an-init-for-docker.html> accessed on: 2025-01-12.
- [Kum22] Sushant Kumar. Software metrics in software engineering - scaler topics. *Scaler Topics*, December 2022. Available at: <https://www.scaler.com/topics/software-engineering/software-metrics-in-software-engineering/> accessed on: 2025-01-12.
- [Lem18] Gillian Lemke. *The software development life cycle and its application*. PhD thesis, Eastern Michigan University Eastern Michigan University, September 2018. Available at: <https://commons.emich.edu/honors/589/> accessed on: 2025-01-12.
- [LFK22] Pierre Le Fevre and Emil Karlsson. *Designing and implementing a private cloud for student and faculty software projects*. PhD thesis, KTH Royal Institute of Technology, 2022.
- [McG23] Cheryl McGuire. *Connect your on-premises network to a VNet: Site-to-Site VPN (classic): Portal - Azure VPN Gateway*, October 2023. Available at: <https://learn.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-howto-site-to-site-classic-portal> accessed on: 2025-01-12.
- [McG24] Cheryl McGuire. *About Azure Point-to-Site VPN connections - Azure VPN Gateway*, June 2024. Available at: <https://learn.microsoft.com/en-us/azure/vpn-gateway/point-to-site-about> accessed on: 2025-01-12.
- [Mer14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.
- [MGvdHW10] Ivan Mistrik, John Grundy, Andre van der Hoek, and Jim Whitehead. *Collaborative Software Engineering*. Springer, Berlin, Heidelberg, January 2010.
- [MNM20] Arthur-Jozsef Molnar, Alexandra NeamȚu, and Simona Motogna. Evaluation of software product quality metrics. In Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek, editors, *Evaluation of Novel Ap-*

Bibliography

- proaches to Software Engineering*, pages 163–187, Cham, 2020. Springer International Publishing.
- [Ove24] Stack Overflow. *2024 Developer Survey*. Stack Overflow, May 2024. Available at: <https://survey.stackoverflow.co/2024/technology/> accessed on: 2025-01-12.
- [PD22] B. Pando and A. Dávila. Software testing in the devops context: A systematic mapping study. *Programming and Computer Software*, 48(8):658–684, December 2022.
- [RSO22] Giovanni Rosa, Simone Scalabrino, and Rocco Oliveto. Fixing dockerfile smells: An empirical study, 2022.
- [SBZ17] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, January 2017.
- [Ser21] Dragos Mihai Serban. *Demystifying NoOps*. PhD thesis, Eindhoven University of Technology, August 2021.
- [Sin22] Gourav Singh. The definitive guide to development environments. *Loft Blog*, September 2022. Available at: <https://loft.sh/blog/the-definitive-guide-to-development-environments/> accessed on: 2025-01-12.
- [SK23] Giancarlo Succi and Artem Kruglov. *Concept and Principles of Measurement*, pages 1–17. Springer International Publishing, Cham, 2023.
- [SMFS⁺16] Olaperi Yeside Sowunmi, Sanjay Misra, Luis Fernandez-Sanz, Broderick Crawford, and Ricardo Soto. An empirical evaluation of software quality assurance practices and challenges in a developing country: a comparison of nigeria and turkey. *SpringerPlus*, 5(1):1921, November 2016.
- [VGP20] Sander Valstar, William G. Griswold, and Leo Porter. Using devcontainers to standardize student development environments: An experience report. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE ’20, page 377–383, New York, NY, USA, 2020. Association for Computing Machinery.

Acronyms

AKS Azure Kubernetes Service.

API Application Programming Interface.

ASCII American Standard Code for Information Interchange.

CD Continuous Deployment.

CI Continuous Integration.

CI/CD Continuous Integration and Continuous Deployment.

CLI Command Line Interface.

CPU Central Processing Unit.

CRI Container Runtime Interface.

CVE Common Vulnerabilities and Exposures.

CVSS Common Vulnerability Scoring System.

DevOps Development and (IT) Operations.

DNS Domain Name System.

FQDN Fully Qualified Domain Name.

GUI Graphical User Interface.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

IDE Integrated Development Environment.

IP Internet Protocol.

JSON JavaScript Object Notation.

Acronyms

JWT JSON Web Token.

K8s Kubernetes.

LDAP Lightweight Directory Access Protocol.

LTS Long Term Support.

MVC Model-View-Controller.

OCI Open Container Initiative.

P2S Point-To-Site Connection.

RED Rate, Error, Duration.

REST Representational State Transfer.

S2S Site-To-Site Connection.

SQL Structured Query Language.

SSH Secure Socket Shell.

TLS Transport Layer Security.

UI User Interface.

UML Unified Modeling Language.

URL Uniform Resource Locator.

USE Utilization, Saturation, Error.

VM Virtual Machine.

VPN Virtual Private Network.

WSGI Web Server Gateway Interface.

YAML YAML Ain't Markup Language.

A. Appendix

A.1. Source Code

The source code for this thesis is available in the following Gitlab Repository: https://git01lab.cs.univie.ac.at/thesis/2023ss/11807161_lukas_greiner. It contains all the necessary projects, documentation, figures, and instructions included in this thesis.

A.2. FactDeploy

A.2.1. Prerequisites

- A running Docker instance
- A VPN Connection or direct access to the network of the University of Vienna
- A valid Python 3.12 and pip installation
- Access to the FactCheck Docker Registry
 - Download permissions of the “factdeploy” namespace (provided by the FactCheck team)
 - Upload/Download permissions for your dedicated namespace = your username (provided by the FactCheck team)
- Access to the Kubernetes Server (fc-k8s.cs.univie.ac.at) with a dedicated user (provided by the FactCheck team)

When deploying to Azure Kubernetes Services (AKS):

- An Azure Account with the rights to create AKS instances

A.2.2. Install

Windows install:

- Install venv with `pip install virtualenv`
- Create venv with `python -m venv venv`
- Activate venv with `venv\Scripts\activate.bat`
- Install dependencies with `pip install -r requirements.txt`

Ubuntu install:

- Install venv with `sudo apt install python3.12-venv`

A. Appendix

- Create venv with `python3.12 -m venv ./venv`
- Activate venv with `source ./venv/bin/activate`
- Install dependencies with `pip install -r requirements.txt`

A.2.3. Quick Start

Run the following command from the root folder (e.g. FactDeploy)

```
python3.12 -m src.factdeploy
```

Select the first option called “Quick Deploy”, this action asks for input on all necessary parameters and directly deploys your generated project to the Kubernetes cluster.

The health endpoint for your application is then available via

`https://fc-k8s.cs.univie.ac.at/{username}/{project-name}/health`

Where the {username} is your username for accessing Kubernetes and {project-name} is the name of your project in camel case e.g. TestProject1 -> test-project1. When accessing this and any other endpoint you are provided to enter your username and password of the Kubernetes instance as “Basic Authentication” is activated.

A.2.4. Description

FactDeploy is a project generation and deployment tool for Python projects in relation to the FactCheck project. It deploys all further mentioned software to an on-premise hosted Kubernetes instance. Currently, the following projects are directly supported:

- FactCheck Backend
- FactCheck Factbase
- FactCheck Factserver

Additionally, the following databases are directly supported:

- CouchDB
- MongoDB
- MySQL

Although the possibility to deploy already available (Docker) Images, which are either publicly available or in an accessible Registry, slight modifications might still be necessary for FactDeploy to achieve all needed functionalities. For further information on highly probable changes information is later provided.

A.2.5. PyCharm Quirk

When running in PyCharm the Run/Debug Configuration has to be adapted due to the UI library used. Simply edit your “Run/Debug Configuration” go to the “Modify Options”

Menu and set the “Emulate terminal in output console”. This will allow to interact with the FactDeploy UI directly from the PyCharm console.

When directly executing it from the terminal or cmd, this is not necessary!

A.3. Main Features

All of the following features can be selected after starting up the FactDeploy application. Most of the parameters are only needed once, and will not be required on the second run. If any changes to the configuration are necessary the configuration can be either reset or edited in the `/config/config.yaml` file.

A.3.1. Quick Deploy

The Quick Deploy feature allows to execute all of the main features at once and can be executed multiple times. This feature allows, that when all necessary information is available to deploy the current project and all FactCheck related dependencies to the Kubernetes cluster in one click.

When executing a quick deployment, the project generation mechanism will create a Project of the type **Service** which means that it is an endlessly running Python application and should be used as a server software.

After that it will upload the content of the generated configuration file to factcheck-config-service. When running the Docker image of this project the `docker-entrypoint.sh` file in the generated project will be executed and will download that configuration file.

In the next step a Dockerfile and after that a Docker Compose file will be generated. This allows your project to be run locally and in the development environment with the same technology stack. With this information a Docker image will be build and pushed to the supplied Docker registry.

Finally with all necessary prerequisites supplied, the project will be deployed upon the Kubernetes cluster. Additionally a `deployment.yaml` file will be created in the project which will be used for future deployments and can be customized to fit the needs of the project.

A.3.2. Project Generation

Initially you will be asked if you want to use an existing project or create a new one. If you select for an already existing project, you will be asked to supply the necessary files for a deployment.

This includes the following parameters:

- Project path - Example: `~/Projects/TestProject1`

A. Appendix

- Main file - Example: `~/Projects/TestProject1/src/test-project1/main.py`
- Production config file - Example: `~/Projects/TestProject1/config/config-prod.yaml`
- Development config file - Example: `~/Projects/TestProject1/config/config.yaml`
- Project type: - Values: Service / Job / CronJob
- Port - Example: 5000
- Use default values for connecting to Kubernetes and Docker Registry
- Create a virtual environment (only for generating a new project)
- Create Initial file structure (only for generating a new project)
- Connection Type to Kubernetes: - Values: On-Premise / Local / Azure

IMPORTANT: When automatically generating a Project, please use the Parent folder when asked for the project path. (e.g. `~/Projects`)

The connection type to Kubernetes selection infos:

- On-Premise means the Kubernetes cluster running on the University infrastructure.
- Local means running directly on the workstation of the current user.
- Azure means running in an Kubernetes cluster setup in the Azure Cloud.

A.3.3. Configuration Upload

As configuration injection on startup for the FactCheck projects and the current project is a supported mechanism, FactDeploy supports the upload of these configuration files. To do this simply select the ‘Upload Configuration File’ option. Here the following parameters will be inquired if not already existing:

- Software for which a configuration should be uploaded (Factcheck Backend/-Factserver/Factbase, Current Project)
- Paths to the projects
- Paths to the production configuration
- Upload Endpoint of the factcheck-config-service
 - e.g., `https://fc-k8s.cs.univie.ac.at/config/add`
- Download Endpoint of the factcheck-config-service beginitemize
 - e.g., `https://fc-k8s.cs.univie.ac.at/config`

A.3.4. Dockerfile Generation

With the Dockerfile generation feature, an adapted Dockerfile from a predefined set can be chosen. The resulting Dockerfile will be stored in the root folder of your project.

This set includes the following base files:

- Simple: Basic Dockerfile with minimal requirements to run a Python project. There aren’t any optimizations and or production optimized dependencies used.
- Default: Optimized Dockerfile which simply executes any provided Python file containing a main function.

- Gunicorn: Optimized Dockerfile which uses a Gunicorn to provide a production web server for a classical web application.
- Custom: Optimized Dockerfile which executes a custom bash script (“docker-entrypoint.sh”) on startup. Necessary when providing custom startup routines or additional parameters.

For these base files the following parameters can be provided:

- Path to the WSGI or main file
- Name of the dependency declaration file (e.g., requirements.txt)
- Port on which the software is accessible
- Additional startup parameters
- Configuration Injection: With this option a bash script will be generated which loads the configuration from the factcheck-config-service and then starts the software.

A.3.5. Docker-Compose Generation

Docker Compose files can also be generated and customized to comply with the current setup.

Services which are useful for the current project can be chosen. These will be included in the docker-compose.yaml file. The choice contains the FactCheck services (FactCheck Backend/Factbase/Factservice) and their dependencies (MySQL/CouchDB databases).

After the selection of services, the choice for either “Prebuild” or “Locally-Build” services has to be made. Prebuild images are Docker images provided by the FactCheck team which are ready to run. Locally build images on the other hand require a copy of the service and the input of their docker-compose.yaml path. These images are then build locally and do not require access to the FactCheck Docker repository. This is useful when these services need to be customized.

Additional inputs include:

- Database credentials (username/password) and initial databases
- Ports exposed to the host system
- Image specification if no default value is available

A.3.6. Docker Build and Push

With this feature, Docker Images can be created and be pushed to a Docker registry. This process involves selecting which Services should be either build, pushed, or both. This includes the current project as well as the FactCheck projects. Additionally the process for increasing the version number by either Patch/Minor/Major version is available. This will increase the version not only in your pyproject.toml file but also in the Docker Image tag, which allows them to be synchronized.

Following inputs and options are available:

A. Appendix

- Selecting the services/project to be build and/or pushed
- Selecting if the Docker Cache should be enabled or not
- Increase Patch/Minor/Major version
- Path to the project and the production Dockerfile
- Credentials and URL to the Docker registry and Kubernetes

A.3.7. Kubernetes Deployment

To deploy any project to the Kubernetes instance, the Kubernetes deployment feature can be used. Which in turn will let you select the services and jobs you want to deploy. This process can be executed as often as one wants. Changes to the deployment will only happen on either version increase (or any change of the Docker image tag) or when changing the deployment file.

As Kubernetes can run in several different environments a simple selection between On-Premise, Azure and Local is possible.

- On-Premise: The deployment will happen on the Kubernetes cluster of the FactCheck team.
- Azure: For deployments on the Kubernetes cluster running on the Azure cloud. Only possible if its connected to the current system.
- Local: For deployments onto a Kubernetes cluster that is installed on the current system (e.g., via Docker, K3s etc)

If there is any missing information you will be asked to supply it. This can include either credentials, Docker image tags or if the deployment file should be stored in the root folder of the current project.

For deployment the following services are available:

- FactCheck Services (Backend/Factbase/Factserver)
- CouchDB / MongoDB / MySQL
- CouchDB Prod - the data from the production environment will be available. This production data will be replicated to this database once a day.
- The current project

A.4. Additional Features

A.4.1. Kubernetes Viewer

The Kubernetes Viewer feature allows for examining and interaction with the Kubernetes environment without prior knowledge of Kubernetes commands. This viewer is build based on layers and all interactions are also based upon existing Kubernetes commands.

1st. Layer - Selecting Kubernetes Objects you want to examine

- Deployments - continuously running software
- Jobs - software running once
- CronJobs - software running in specified intervals

2nd Layer - Selecting an Action

- List - all pods of the selected object will be displayed
- Describe - will show the current state of the selected entity
- Delete - deletes the selected entity and will not be recreated

3rd Layer - Selecting Pod Actions - After selecting the “List” option an action can be performed upon the displayed pods (Kubernetes object which runs the actual software).

- View Logs - will display all currently available log messages of this pod
- Describe - will show the current state of the pod
- Delete - will delete the pod and automatically start a new instance.

After executing a command you can choose to either go a level up, or remain on the current layer.

A.4.2. Configuration Backup, Deletion, and Restore

When a new project is either generated or the configuration is cleared, the backup process is started. This will store a copy of the current configuration file to the root folder of the FactDeploy project and a new, but empty, configuration file will be created.

While performing the backup, you will be asked to either retain existing configuration of the docker-compose generation function and configuration for the Kubernetes deployment functionality. Otherwise these data points will be deleted.

To restore the configuration, just copy the newly created backup file called «project-name»-backup-«current-date»-config.yaml to the config folder of the FactDeploy project. Then delete the current config.yaml file and rename the backup file to config.yaml

A.5. Useful Kubernetes Scripts

These are useful scripts and yaml files to setup a Kubernetes cluster and all necessary plugins.

A.5.1. On-Premise Kubernetes Installations

To setup an on-premise Kubernetes cluster, several pre-requisites are necessary. First of all a valid SSL certificate is recommended as signed SSL certificates are necessary for

A. Appendix

TLS termination, and therefore, to allow secure access via a browser on running resources. Second of all a up-to-date Ubuntu installation, currently this is the Version 24.04.

Installation of a single instance Kubernetes cluster can be done by executing the `kubernetes-setup.sh` script. Here the environmental variables supplied at the top of the file have to be updated occasionally depending on the wanted version of the dependencies and the environment. This script will not only setup Kubernetes but also installs: Helm, Calico, auto-completion of Kubernetes commands for bash, and Kubernetes aliases.

For joining additional worker nodes the `worker-node-setup.sh` script can be executed. **IMPORTANT:** The content of this script is different for any Kubernetes cluster. The content of this script does only work for the fc-k8s cluster currently deployed.

The ingress setup for the on-premise instance can be done via the `kubernetes-ingress-setup.sh` script, which sets up a NGINX Ingress Controller¹ (important to mention not a “Ingress NGINX Controller” - yes there is a huge difference)

To allow access to the cluster an Ingress has to be created, this can be done by applying the `on-prem-ingress.yaml` file. Which requires an already existing TLS secret. How to create this secret can be seen in the Kubernetes documentation². Also important is to update the hostname and the name of the secret according to the current environment.

Allowing Kubernetes to create storage a dedicated storage provider has to be specified. This can be done by applying the `on-prem-storage.yaml` file.

Allowing Kubernetes to access a Docker registry, the Kubernetes documentation³ should be inquired.

IMPORTANT: For any deployments to take place, the factcheck-config-service has to be deployed and read access to the factdeploy namespace of the FactCheck Docker registry has to be given.

A.5.2. Azure Kubernetes Services

The Azure setup and management of a Kubernetes cluster, is called the Azure Kubernetes Services. To setup this for a FactCheck development environment the following steps have to be taken.

The `azure-ingress-setup.sh` script in the `kubernetes-setup` directory has to be executed. This sets up an NGINX Ingress controller and supplies it with the necessary parameters to get an automatic FQDN by Azure. As this FQDN needs to be unique the value of the following parameter has to be changed `azure-dns-label-name="factcheck-dev"`.

¹<https://docs.nginx.com/nginx-ingress-controller/>

²https://kubernetes.io/docs/reference/kubectl/generated/kubectl_create/kubectl_create_secret_tls/

³<https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/>

After this an Ingress definition has to be defined which can be seen in `azure-ingress.yaml`. When applying this file via `kubectl apply -f azure-ingress.yaml` the ingress will be setup.

Now the Azure Kubernetes Cluster will be available via `factcheck-dev.westeurope.cloudapp.azure.com`.

Additionally: In the `azure-k8s-templates` folder, the templates and parameters to setup an Azure Kubernetes Cluster are supplied.

IMPORTANT: For any deployments to take place, the `factcheck-config-service` has to be deployed and read access to the `factdeploy` namespace of the FactCheck Docker registry has to be given.

A.5.3. Azure Arc Kubernetes

To setup a Azure Arc connection to a on-premise Kubernetes cluster, the script

```
./azure-arc-setup.sh
```

has to be executed. To do that first of all, all environmental variables have to be defined in the script according to the values provided by Azure. This script is only for connecting a Kubernetes cluster via a Private-Link-Scope which is connected to the network of the S2S VPN. This file is only usable for administrators as the rights create the necessary resources in the Azure Cloud require administrator permissions.

A.6. Known Issues

A.6.1. Private Link Connection

At the point of writing, Azure Arc does not support interaction and manipulation of workloads (Pods, Deployments, Services etc.) with a private link connection via their web interface. To do this, the Kubernetes cluster would need to be publicly available. Nonetheless, other features, such as monitoring and supplying a publicly available FQDN are available.

A.6.2. Possible DNS Issues

It is possible that pods can not or only sometimes resolve URLs. This can happen as Linux allows only up to 6 values for the “search” parameter in the `/etc/resolv.conf` file.

To solve this issue, the kubelet and core-dns has to be provided a custom `resolv.conf` file. This involves creating a custom `resolv.conf` file via:

```
cp /etc/resolv.conf /etc/custom-resolv.conf
```

A. Appendix

Remove all unnecessary search parameters which are not necessary for pods. In the current custom-resolv.conf configuration the following parameters were removed “his.arc.azure.com guestconfiguration.azure.com dp.kubernetesconfiguration.azure.com”. After these values were updated in the /var/lib/kubelet/config.yaml file, the value for “resolvConf” has to be changed to “/etc/custom-resolv.conf”. Then the kubelet has to be restarted via the following command

```
sudo systemctl restart kubelet
```

This also has to be done for CoreDNS. To do this the yaml deployment file for CoreDNS had to be extracted via

```
kubectl -n kube-system get configmaps coredns -o yaml > coredns.yaml
```

There the line “forward . /etc/resolv.conf” has to be changed to “forward . /etc/custom-resolv.conf”. To apply these changes:

```
kubectl apply -f coredns.yaml
```

After that coredns also has to be restarted via the following command:

```
kubectl rollout restart -n kube-system deployment/coredns
```

In the end all pods who want to access external URLs (e.g., google.com) have to be deleted, which in turn generates a new pod with the correct configuration. This should ultimately resolve this issues.

A.7. User Generator

A.7.1. Installation

- Clone the FactDeploy git repository to Kubernetes Server, ensure that an Ubuntu distribution is installed
- Update the user_config.yaml with the necessary rights/roles (currently apps, extensions, batch, persistantVolume APIs are allowed)
- Install venv with `sudo apt install python3.12-venv`
- Create venv with `python3 -m venv ./venv`
- Activate venv with `source ./venv/bin/activate`
- Install the necessary dependencies with `pip install -r requirements.txt`

A.7.2. Usage

User Creation

To create a user the following script has to be executed on the Ubuntu machine which hosts the Kubernetes cluster.

Execute: `sudo ./user_create.sh -u username -f`, replace username with the related information of the user. Supplying the `-f` parameter is optional and will copy of the synced factcheck production data to that specific users database directory.

User Deletion

When running the user deletion script all information of that user, including Kubernetes objects, their home folder, and all databases will be removed.

Execute: `sudo ./user_remove.sh -u username`, replace username with the related information of the user

Renew User Certificates

As Kubernetes self-signed certificates can expire, the following script should be executed to update to renew them. This will update all certificates for all users on the system.

Execute: `sudo ./user_renew_certs.sh`

A.8. FactCheck Config Service

The factcheck-config-service is the central, secure storage for configuration files used by FactCheck projects. This enables automatic configuration injection on startup by loading the necessary configuration file with the supplied API key.

A.8.1. Installation

- Install venv with `sudo apt install python3.12-venv`
- Create venv with `python3 -m venv ./venv`
- Activate venv with `source ./venv/bin/activate`
- Install dependencies with `pip install -r requirements.txt` in the factcheck-config-service folder
- Update `deployment_k8s.yaml` file if necessary
 - Set the value of Ingress to a valid host. Currently: `host: fc-k8s.cs.univie.ac.at`
 - Set the `hostPath.path` in the PersistentVolume type definition to an available folder on the drive.
 - Verify that an custom storage option called local-storage is available on the Kubernetes cluster.
- Copy the `deployment_k8s.yaml` YAML file to Kubernetes node

A. Appendix

- On Kubernetes node apply the deployment_k8s.yaml YAML file via: `kubectl apply -f deployment_k8s.yaml`
- On Azure Kubernetes apply the deployment_azure.yaml YAML file via: `kubectl apply -f deployment_azure.yaml`

A.8.2. Building and Pushing to a Docker Registry

The building and pushing process can be done by executing the `./build_and_push_k8s.sh` files.

A.8.3. Usage

Uploading a configuration file can be done via a POST request to the REST endpoint available via `https://fc-k8s.cs.univie.ac.at/config/add` and needs the header 'Content-Type: text/plain' and the content of the configuration file in the request body.

Here an example curl call:

```
curl -X POST --location 'https://fc-k8s.cs.univie.ac.at/config/add' --header 'Content-Type: text/plain' -data-binary "@/path/to/your/config.yaml"
```

After making this request the following JSON structure is returned, which contains the API key to access the file again.

```
{
  "api-key": "SGVsbG8gdGhlcm..."
}
```

Downloading a configuration file can be done by a GET request to the REST endpoint available via `https://fc-k8s.cs.univie.ac.at/config` and supplying the API key via the `x-api-key` header.

Here an example curl call: `curl --location 'https://fc-k8s.cs.univie.ac.at/config' --header 'x-api-key: bG9yZW0gaXBzdW0...'`

A.9. FactCheck Development Container

This project contains the development container for all FactCheck related projects. With this container a developer is able to develop directly in a Docker container which is highly similar to the one deployed in the production environment.

Currently the installed dependencies include:

- git, curl, wget, unzip, tree, htop, sudo
- build-essential, software-properties-common

- Yelp-dumb-init
- libcurl4-openssl-dev, libssl-dev, default-libmysqlclient-dev, gcc, libffi-dev
- python3.12, python3.12-dev, python3.12-venv, python3-pip, python3-wheel build-essential
- nvm, node 20.11.0, npm

To build and push this image to a Docker registry execute the `./build_and_push_k8s.sh` script. This image is then available when generating a `docker-compose.yaml` file via the FactDeploy software.

A.9.1. Usage

Prerequisite:

- Ensure that Docker is installed and running
- Generate a Docker compose file via the FactDeploy project

When running this container via the Docker compose file generated by FactDeploy, make sure that all project related files are mounted inside the container by supplying the current project as a volume (Should be automatically done on generation). Running commands in this container the `docker exec -ti dev-environment bash` command has to be executed. Now that you are in the container you have access to the network of the Docker compose file and can access your project and execute it inside of the container. The development of the container should continue outside of the container in your favorite IDE. Only running and debugging of your project should happen inside of the container.

A.10. Docker Registry

A.10.1. Installation

- Install docker and docker-compose
- Update the environment variables in the `docker-compose.yaml` file
 - Generate basic authentication data with `htpasswd`
 - Use certs to enable external availability
- Update volumes configuration in `docker-compose.yaml` file
 - Set path for image storage
 - Set path of the certificates
 - Set path of the `htpasswd` generated authentication file
- Copy the `docker-compose.yaml` file to server with Docker installed
- Execute `docker compose up -d` in the `docker-registry` folder

A. Appendix

A.10.2. Usage

- Execute `docker compose up -d` in the folder with the `docker-compose.yaml` file

Further information available here: <https://docs.docker.com/registry/deploying/>

A.10.3. Recommended:

Deploying Docker Registry behind an NGINX Reverse Proxy: <https://earthly.dev/blog/private-docker-registry/>

A.11. FactCheck Sync Service

This service syncs parts of the production fact database to a local database. It uses the default replication mechanisms of CouchDB but reduces the necessary computation resources needed.

In a Kubernetes cluster this service runs two times. First of all, it runs in the default namespace (for admins) which replicates directly from the production fact database. The other instance is running in the user namespace which replicates its data from the default namespace to its local namespace. This further reduces resource usage and allows for easier debugging.

A.11.1. Prerequisites

- Credentials to the production CouchDB database that allow replicating databases
- Local CouchDB instance
- Docker installed

A.11.2. Local Setup

- Install venv with `sudo apt install python3.12-venv`
- Create venv with `python3 -m venv ./venv`
- Activate venv with `source ./venv/bin/activate`
- Install dependencies with `pip install -r requirements.txt` in the `factcheck-sync-service` folder
- Run `docker compose up -d`

A.11.3. Kubernetes Setup

- Build and Push the `factcheck-sync-service` for the default namespace via `./build_and_push_k8s.sh`

- Build and Push the factcheck-sync-service for the user namespace for in-cluster syncing via `./build_and_push_in_cluster_k8s.sh`
- Deploying the factcheck-sync-service for the default namespace can be done by copying the `deployment_k8s.yaml` file on the Kubernetes cluster by running `kubectl apply -f deployment_k8s.yaml`
- Running the in cluster sync version, use the FactDeploy software and choose this service in the Kubernetes deployment section.

A.12. Factcheck VPN Relay

The project was created as access to the infrastructure of the University of Vienna is, without access to networking settings, only possible via a VPN. This project could be used as a sidecar for a Kubernetes deployment. As only a single connection per user is possible the usage of this project is rather limited and not recommended. For a better VPN connection between the Azure cloud and a university server, please use a Site-to-Site VPN or at least a Point-to-Site connection.

A.12.1. Building a Docker Image of the VPN Relay

Download the University of Vienna VPN Client for Linux⁴ and add them to to this project. Check if the image name and repository is correct in the `build_and_push.sh` script. Then execute this script via: `./build_and_push.sh`

A.12.2. Running the VPN

For testing this image the best way is to build the docker compose image via: `docker compose build --no-cache` then update the environmental variables with your credentials. After that simply execute:

```
docker compose up -d
```

A.12.3. Check if its working

Enter into the container via: `docker exec -ti factcheck-vpn-relay bash` and enter the following command: `f5fpc -i`

If the VPN is running correctly, the connection will show up as established and an overview will be shown.

⁴<https://vpn.univie.ac.at>