HadesCI: Towards a Scalable Continuous Integration System for Programming Exercise Environments

Guided Research - Technical University Munich

Supervisor: Prof. Stephan Krusche **Advisor:** Matthias Linhuber

Robert Jandow
Submission Date: 30.09.2024

Abstract

As programming exercise assignments grow in size and complexity, there is a critical need for effective continuous integration (CI) systems to ensure code quality and facilitate early error detection. HadesCI is an innovative project that aims to redefine continuous integration within programming exercise environments. It seeks to address the pressing challenges of scalability, simplicity, and security, which are crucial for both educational purposes and large-scale software development environments.

1 Introduction

The evolution of software development practices has highlighted the pivotal role of continuous integration (CI) systems in preserving code integrity and consistency [Ela+22]. However, existing CI solutions frequently encounter challenges related to complexity, scalability, and security, particularly in environments characterized by a high volume of parallel build jobs [SAZ17].

The requirements and functionalities of CI systems vary significantly across different domains [SB14]. In enterprise environments, CI systems are tasked with managing large, intricate codebases while integrating with various enterprise tools and adhering to stringent security protocols. In contrast, within the domain of mobile app development, the focus of CI systems is on minimizing build times and ensuring seamless integration with mobile testing frameworks. For open-source projects, the priority lies in accommodating diverse contributors and maintaining broad compatibility across various platforms and environments. This diversity in requirements necessitates that CI systems possess a high degree of adaptability and customizability to meet the specific demands of their respective domains.

In programming exercise environments, the requirements for CI systems diverge significantly from those in traditional software development contexts. These environments are typically employed in educational settings to facilitate the teaching of programming concepts and to assess students' coding competencies. They generally consist of a series of programming exercises, each defined by a specific set of requirements and constraints. Students submit their solutions to these exercises, which are then evaluated by an automated grading system. This system assesses the correctness of the solutions, enforces coding standards, and provides feedback to students. The primary objective of this grading system is to offer students immediate feedback on their code, thereby aiding in the enhancement of their programming skills. Given the high volume of submissions in educational contexts, the grading system must be highly scalable, efficient, and reliable [Sta+15].

The Artemis[1] learning platform [KS18] encounters specific challenges with existing CI solutions like Bamboo[2] and Jenkins[3]. These systems require the creation and maintenance of distinct entities, including build plans and projects, for each individual student. This approach significantly increases administrative overhead and complexity.

Typically, each programming exercise necessitates multiple configurations — one for the template repository and another one for the solution repository. This arrangement demands continuous synchronization between both repositories, leading to redundant configuration storage and fragmented management across two separate storage locations.

Moreover, the existing setup supports only static build configurations, which cannot be dynamically adjusted. Instead, configurations must be duplicated and subsequently modified for each participant. As a result, every student in a programming exercise requires a separate configuration solely to specify their individual repository URL.

To address the identified challenges, we propose a scalable CI system specifically tailored for programming exercise environments. The system aims to streamline the setup and management of build configurations for programming exercises, thereby reducing administrative overhead and complexity. By incorporating principles of scalability, statelessness, and efficiency into its design, the proposed system will offer a reliable and efficient execution environment for build jobs, ensuring prompt feedback for students and ease of maintenance for instructors. Designed for seamless integration with existing educational platforms, the system will offer a cohesive user experience while maintaining a high degree of adaptability and configurability through its API.

The remainder of this report is structured as follows: Chapter 2 provides an overview of related work in the domain of scalable CI systems and programming exercise environments. Chapter 3 presents the design and architecture of the system, detailing its components and interactions. This integration of this architecture in educational platforms is then outlined in Chapter 4. Afterwards, Chapter 5 discusses potential future directions and improvements for the system. Finally, Chapter 6 summarizes the key findings and contributions of this research.

2 Related Work

In the domain of scalable continuous integration (CI) systems, significant contributions have been made by researchers such as Hamed Esfahani et al. [Esf+16] and Kaiyuan Wang et al. [Wan+20], who have developed innovative approaches to managing large-scale build services within major technology companies like *Microsoft* and *Google*. Hamed Esfahani et al. [Esf+16] introduced CloudBuild, a distributed and caching build service engineered to support Microsoft's extensive development teams. CloudBuild distinguishes itself by enabling the integration of diverse codebases with minimal modifications to existing build specifications and tools. The system emphasizes reliability and efficiency through the use of distributed caching and sandboxing techniques to manage under-specified dependencies and non-deterministic build tools. CloudBuild processes approximately 20,000 builds daily for over 4,000 developers, achieving significant build speed enhancements via a caching mechanism that reuses previously generated build outputs.

Conversely, Kaiyuan Wang et al. [Wan+20] describe Google's scalable build service system, which focuses on optimizing build performance through smart scheduling. This system employs an occupancy model and a workspace selection algorithm to enhance resource allocation and

[1] https://github.com/ls1intum/Artemis/releases/tag/6.9.6
[2] https://www.atlassian.com/software/bamboo
[3] https://www.jenkins.io/

reduce build execution times, achieving an impressive throughput of over 15 million builds daily, serving more than 50,000 developers. Unlike CloudBuild's coarse-grained project builds, Google's approach utilizes fine-grained target builds, resulting in higher cache hit rates due to its monolithic codebase structure [Wan+20]. Both CloudBuild and Google's system underscore the critical roles of caching and scheduling in CI systems, although their methodologies diverge to meet their respective organizational needs and codebase architectures.

Notably, these systems are optimized for environments where a large number of developers collaborate on relatively few, extensive projects. This contrasts with our use case, which involves managing a diverse array of smaller projects with potentially fewer users per project - a scenario common in the educational setting. Recognizing these unique requirements, it becomes essential to explore how CI/CD practices can be effectively integrated into educational environments.

In this context, recent studies on the introduction of Continuous Integration (CI) and Continuous Delivery (CD) into educational environments, researchers have identified several challenges, particularly regarding the complexity of tools and the limited time available for teaching these practices. A pilot study demonstrated that utilizing a CI/CD pipeline in undergraduate software engineering courses significantly enhanced students' comprehension by providing visual feedback and automated testing environments, tailored to educational needs [Edd+17].

Thus, in a similar vein to HadesCI, Laurenz Fabian Blumentritt [Blu24] implemented a container-based CI system directly integrated into Artemis to streamline the build process for programming exercises. By tightly coupling the CI system with Artemis, the solution leverages existing infrastructure and user management capabilities, simplifying both the setup and operation of the CI system. This integration utilizes the shared database and data structure of Artemis to store build job information and results, ensuring consistency and reliability across the systems. Additionally, by executing build jobs within Docker containers, the system enhances security and isolation, preventing interference between build jobs. However, this tightly coupled nature presents potential challenges in scaling and adapting to environments beyond Artemis, limiting the system's flexibility and portability.

To address the issue of static build configuration within Artemis, Andreas Resch [Res24] introduced a domain-specific language (DSL) named Aeolus, designed to define build configurations independent of the platform. This DSL eliminates the need for individual configurations for each participant, enabling dynamic configuration adjustments. However, while Aeolus mitigates some configuration challenges, it does not fully address the issues of scalability and complexity, as it remains dependent on an external CI system for build job execution.

3 Architecture

Having outlined the problem and the motivation behind HadesCI, this section delves into the architecture of the system. The architecture of HadesCI is designed to embody its core principles while meeting the demands of a scalable and efficient continuous integration (CI) system tailored for programming exercise environments.

3.1 Design Goals

The architecture of HadesCI is built around several key design goals, which are as follows:

API Configuration The system should be configurable through a RESTful API, facilitating seamless integration with other systems. The API must be versioned to ensure backward compatibility and provide a consistent user experience. Furthermore, the API should incorporate robust authentication and authorization mechanisms to safeguard the system's resources from unauthorized access.

System Scalability HadesCI must efficiently handle a large volume of parallel build jobs. The system is designed to scale horizontally, distributing the workload across multiple nodes to accommodate increasing demand without sacrificing performance. Additionally, it should scale vertically, efficiently utilizing the resources of individual nodes to maximize throughput.

Stateless Architecture The system should operate with a stateless architecture, meaning it does not retain build job information between requests. This design choice ensures that the system can quickly recover from failures and manage a high number of concurrent requests without encountering issues related to state management.

System Adaptability The architecture should be adaptable to various environments and requirements. This flexibility is achieved by avoiding assumptions about the nature of the build jobs, instead providing an architecture that supports a wide range of build job configurations and requirements.

3.2 Build State Management

To realize these design goals, the architecture of HadesCI is composed of several components, each with a distinct role in the system.

The general states of a job within the HadesCI system are depicted in Diagram 1.

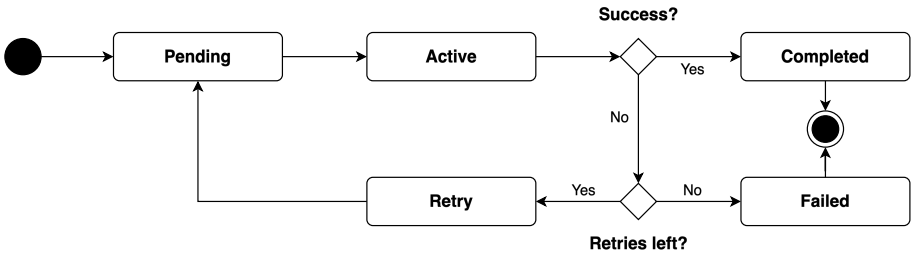

Diagram 1: Job States (State Diagram): This diagram illustrates the different states a job can transition through in the HadesCI system. The states include "Pending" (awaiting execution), "Running" (currently being executed by a Build Agent), "Completed" (successfully executed), "Retry" (job failed and is rescheduled), and "Failed" (retry attempts exhausted and job terminated).

The job states in the HadesCI system are defined as follows:

Pending	The job is awaiting execution, having been scheduled by the Queue but not yet assigned to a Build Agent.
Running	The job is currently in execution by a Build Agent. The Scheduler has assigned the job, and the Executor is actively running it.
Completed	The job has been successfully executed. The Executor has completed the job without errors, and it has transitioned to a successful state.
Retry	The job has failed during execution. The Executor encountered an error, resulting in a failure. If retry attempts remain, the job will be rescheduled by the Scheduler and re-executed.
Failed	The job has failed after all retry attempts have been exhausted. The job will not be rescheduled and will terminate in the failed state.

3.3 Subsystem Decomposition

The high-level component diagram of the HadesCI architecture is shown below in Diagram 2.

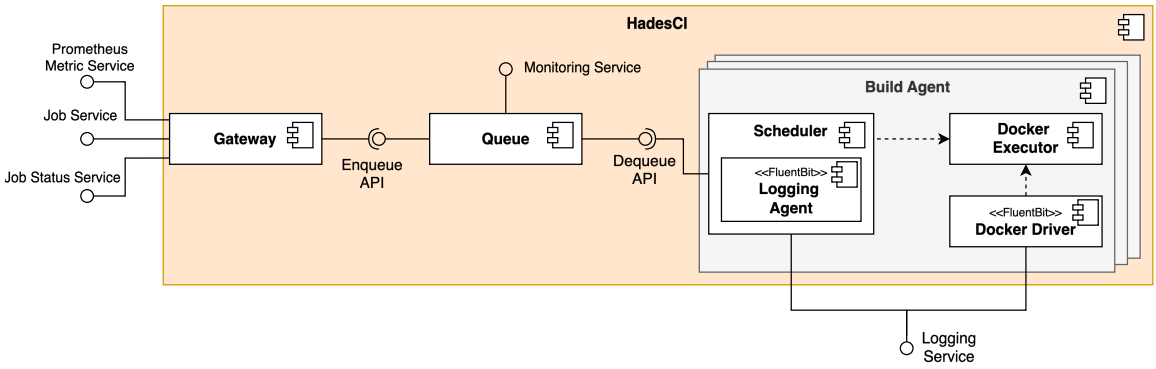

Diagram 2: HadesCI Architecture (Component Diagram): This diagram provides an overview of the main components in the HadesCI system, including the Gateway, Queue, Build Agent, Scheduler, and Executor, and shows how they interact to process and execute build jobs efficiently in a scalable environment.

The execution process overview is illustrated in the sequence diagram below, Diagram 3. The process begins with the Gateway receiving a request from the user, validating it, and then forwarding it to the Queue for processing. The Queue schedules the build job and delegates it to the Build Agent for execution. The Scheduler orchestrates the execution on the Build Agent, while the Executor manages the creation and operation of entities necessary for job execution.

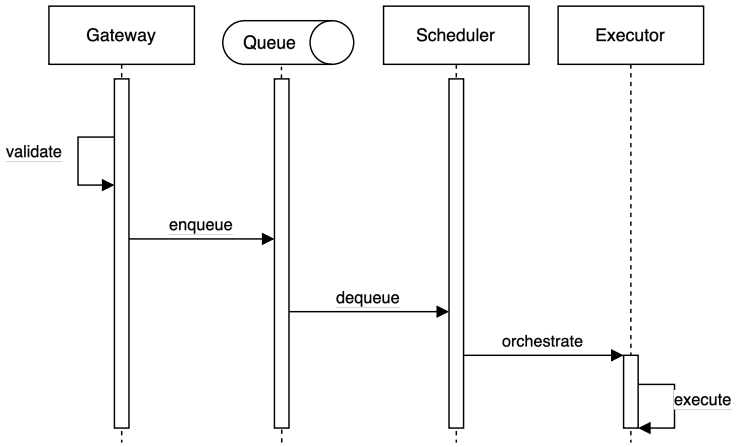

Diagram 3: HadesCI Execution Process (Sequence Diagram): This diagram shows the sequence of events that occur during the execution of a build job in HadesCI. The process begins with a user request through the Gateway, followed by job validation, scheduling via the Queue, and execution managed by the Build Agent and Scheduler.

As scalability and performance are crucial aspects of the system, we have written the components in Golang[4], a statically typed, compiled language known for its efficiency and performance. The choice of Golang aligns with the system's requirements for handling a high volume of concurrent requests and executing build jobs efficiently.

3.3.1 Gateway

The Gateway serves as the primary entry point for all requests to the HadesCI system. It is responsible for processing incoming requests, validating them, and routing them to the appropriate components for execution. Designed for robustness and user accessibility, the Gateway provides a seamless interface for users to interact with the system. It is fully configurable via a RESTful API, enabling programmatic interactions and integration with external systems.

[4]https://github.com/golang/go

An example of a request submitted to the Gateway is illustrated below:

```
{
  "name": "",
  "metadata": {},
  "timestamp": "2021-01-01T00:00:00.000Z",
  "priority": 1, // optional, default 3
  "steps" : [
    {
      "id" : 1, // mandatory to declare the order of execution
      "name" : "Clone",
      "image" : "git:alpine", // mandatory
      "metadata" : {
          "GIT_USERNAME[_*]" : "user",
          "GIT_PASSWORD[_*]" : "pass",
          "GIT_URL_*": "fake.fake"
      }
    },
    {
      "id" : 2,
      "name" : "Execute",
      "image" : "java:alpine",
      "script" : "maven run", // optional
      "metadata" : {}
      "cpu_limit": 1,
      "memory_limit": "1G"
    }
  ]
}
```

Listing 1: Example request to the Gateway, scheduling a build job with two build steps (Cloning a Git repository and executing a Maven build)

The request payload includes the following fields:

- **name**: Identifies the build job, aiding in debugging and tracking.
- **metadata**: A dictionary for additional metadata related to the build job. The parent-level metadata pertains to general job information, while job-specific metadata is specified within the job level.
- **priority**: Specifies the job's priority. This optional field defaults to 3 if not provided and is used by the Queue to prioritize and allocate resources accordingly.
- **steps**: An array of steps defining the build job, each containing:
  - **id**: Mandatory, defines the order of execution for the steps.
  - **name**: Name of the step.
  - **image**: Mandatory, specifies the Docker image to be used for the step.
  - **script**: Optional, contains the commands to be executed within the Docker container.
  - **metadata**: Optional, provides additional information specific to the step.
  - **cpu_limit**: Optional, specifies the CPU limit for the Docker container, allowing for artificial limitation of CPU usage.
  - **memory_limit**: Optional, specifies the memory limit for the step, similar to the **cpu_limit**.

To ensure an efficient and lightweight implementation of the Gateway, we decided to use the Gin framework[5], a high-performance HTTP web framework for Golang. Gin provides a robust set of features for building RESTful APIs, making it an ideal choice for implementing the

[5]https://github.com/gin-gonic/gin

Gateway. By using Gin, we can ensure that the Gateway is fast, reliable, and scalable, providing a seamless experience for external systems interacting with HadesCI.

3.3.2 Queue

The Queue is a pivotal component of the HadesCI architecture, responsible for the efficient scheduling of build jobs. It ensures timely execution and optimal utilization of system resources. Designed to manage a high volume of parallel build jobs, the Queue is essential for maintaining system throughput and scalability. It prioritizes and allocates resources based on the requirements and priorities of individual jobs, ensuring efficient operation. Given its critical role, the Queue is implemented using Redis, a well-established and reliable platform. Redis's versatility allows it to be used for caching, messaging, and real-time analytics, offering potential for future feature enhancements. For interaction with Redis, we selected the *asynq* framework[6]. *asynq* provides a simple, efficient interface for managing Redis Queues in Golang. It is lightweight and user-friendly, making it well-suited for the Queue implementation in HadesCI. The framework also supports task prioritization, retries, and scheduling, ensuring that the Queue can handle a wide array of build job requirements. Additionally, *asynq* offers a small web interface for monitoring the queue and its tasks. The payload for the queue mirrors the Gateway payload but omits the priority field, as it is only relevant during job enqueuing and not during job execution.

3.3.3 Build Agent

The Build Agent is a subsystem responsible for executing build jobs. It comprises three primary components: the Scheduler, the Executor, and the Log Aggregator. While the initial version of HadesCI supports Docker as the execution environment, the system is designed with future extensibility in mind, allowing for the integration of other environments such as Kubernetes or virtual machines. With the choice of Golang as the implementation language, we can leverage the native Docker client library to interact with Docker, ensuring efficient and reliable execution of build jobs. For the future addition of other execution environments, the wide adoption of Golang and its extensive ecosystem of libraries will facilitate seamless integration.

To facilitate future extensions, the Scheduler logic has been abstracted into a simple interface, enabling easy implementation across different execution environments. This modular design ensures that additional environments can be supported without significant architectural changes. The interface is detailed in Listing 2.

```
type JobScheduler interface {
  ScheduleJob(ctx context.Context, job payload.QueuePayload) error
}
```

Listing 2: Interface for the Scheduler

Scheduler The Scheduler orchestrates the execution of build jobs on the Build Agent. It manages resource allocation, job prioritization, and scheduling to ensure efficient execution. The Scheduler is designed for adaptability, capable of handling diverse build job requirements and priorities. By employing a pull model, the Scheduler retrieves jobs from the Queue in order of priority, ensuring optimal resource utilization and timely execution.

Docker Executor The Executor serves as an abstraction layer responsible for executing build jobs within isolated environments. It is designed to support different execution environments and technologies, with a focus on Docker for the initial version of HadesCI. The Executor manages Docker containers, ensuring secure and consistent execution of each job. By using

[6]https://github.com/hibiken/asynq

Docker, the system provides a high level of isolation, preventing jobs from interfering with one another and ensuring each job is executed in a controlled environment. To enhance isolation, shared volumes are used to share build artifacts between different containers within a job, while maintaining the isolation of the containers themselves.

Logging Aggregator Recognizing the critical role of logging in the build process, HadesCI integrates a logging aggregator into its architecture. The aggregator collects logs generated during build job execution and stores them for analysis. Centralized log aggregation provides a unified view of the build process, simplifying issue identification and troubleshooting. Integrated with both the Build Agent and the Gateway, the aggregator ensures consistent log collection across the system. Logs can be accessed through the aggregator, enabling users to review the build process and identify potential issues. For the logging aggregator, *FluentBit*[7], an open-source log processor and forwarder, is employed. *FluentBit* offers a lightweight and efficient solution for log management and analysis. Its various output plugins allow logs to be stored in multiple destinations, facilitating future integration with other log management solutions and supporting a variety of technologies.

4 Integration

To evaluate the practical application of HadesCI in a real-world context, we envisioned its integration with a learning platform that supports programming exercises. This platform facilitates the creation and execution of programming exercises for educational purposes, providing a web-based interface for instructors to define exercises, manage student submissions, and assess results. Students access the platform to complete exercises, submit their solutions, and receive feedback.

The integration between HadesCI and the learning platform is illustrated in Diagram 4. This integration comprises two primary components: the HadesCI service and the platform service. The HadesCI service manages the execution of programming exercises, while the platform service handles exercise configuration and student submissions. Communication between the two services is facilitated by a REST API, which allows the platform service to initiate exercise execution on the HadesCI service and retrieve results upon completion.

An additional component, the *FluentBit* component, is responsible for collecting and forwarding logs generated during exercise execution. This server is integrated with both the HadesCI and platform servers, ensuring that logs are systematically collected and stored for further analysis. Through the platform's web interface, instructors can review the logs to monitor exercise execution and identify potential issues, while students can utilize the logs to debug their solutions.

To facilitate the integration of HadesCI's logging capabilities with the learning platform, an adapter has to be developed to aggregate the build logs generated by *FluentBit* for each specific exercise and forward them to the platform server. This adapter ensures that compatibility is maintained between the log format generated by *FluentBit* and the platform's log storage and retrieval mechanisms.

[7]https://fluentbit.io/

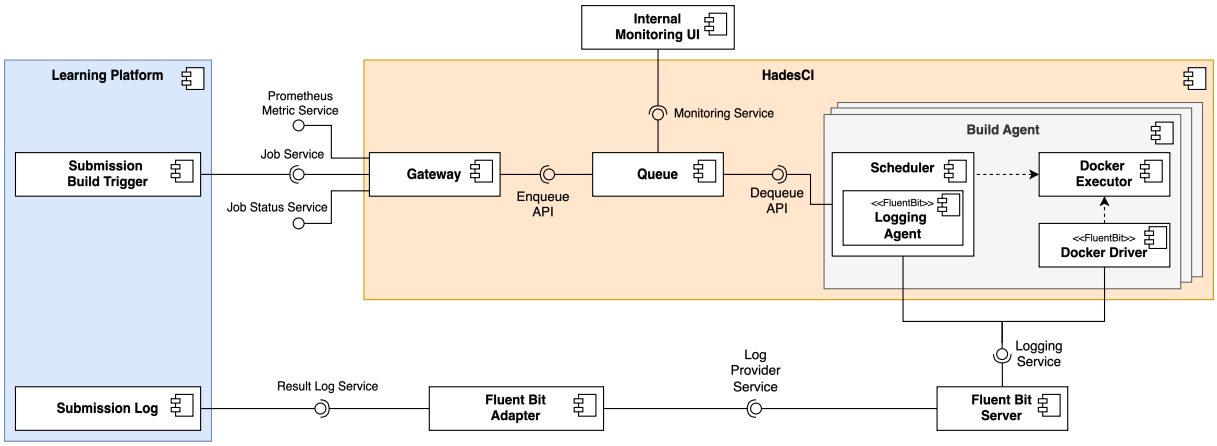

Diagram 4: Integration of HadesCI with Learning Platform (Component Diagram): This diagram demonstrates the integration of HadesCI into a learning platform for programming exercises. It shows the interactions between the HadesCI service and the platform, log aggregation via FluentBit, and how the system manages exercise execution and feedback.

Although the detailed implementation of HadesCI into a learning platform is beyond the scope of this research, the proposed integration offers a high-level overview of the components involved and their interactions. This integration aims to demonstrate the feasibility of deploying HadesCI in a real-world scenario and to provide a foundation for further development and integration efforts.

5 Future Work

HadesCI establishes a foundational framework for a scalable continuous integration system tailored to programming exercise environments. However, several areas present opportunities for further enhancement and expansion.

First, the current implementation of HadesCI is confined to executing programming exercises steps within a single container. While this approach suffices for many exercises, it may not be suitable for those requiring the simulation of distributed systems, which necessitate the parallel execution of multiple containers. To accommodate such exercises, HadesCI could be extended to manage and orchestrate the execution of multiple containers. This enhancement would necessitate architectural modifications to support the coordination and synchronization of multiple containers during execution.

Second, HadesCI currently relies on Docker as the container runtime. Although Docker is widely adopted, its limitations become apparent when considering the needs for scaling and orchestration. To overcome these constraints, HadesCI could be extended to support alternative container runtimes, such as Kubernetes. This extension would enable HadesCI to leverage the advanced scaling and orchestration capabilities of Kubernetes, allowing for more distributed and fault-tolerant execution of exercises. Additionally, Kubernetes would eliminate the need for pre-provisioning resources, as it can dynamically allocate resources based on current demand. Other CI/CD systems, such as Jenkins, already support Kubernetes as a runtime environment, demonstrating the potential benefits of this approach.

Lastly, a comprehensive evaluation of HadesCI is essential to assess its performance and scalability. This evaluation should encompass a variety of execution scenarios, including high parallel workloads, exercises with varying resource requirements, and exercises with differing execution times. The evaluation should also examine the impact of HadesCI on the performance of the underlying infrastructure and the associated operational costs. Such an evaluation would yield valuable insights into the system's capabilities and limitations, guiding future improvements.

6 Conclusion

HadesCI envisions a future where continuous integration systems can seamlessly scale to meet the evolving demands of both modern software development and educational programming exercises. Through its design principles and proposed architecture, HadesCI lays the groundwork for a CI system that is not only scalable but also straightforward and secure. Although originally conceived with the educational sector in mind, its versatile architecture ensures its applicability across a range of other contexts.

Thus, HadesCI represents not only a theoretical advancement in the field of continuous integration but also a practical solution to the specific challenges faced by educators and students in educational settings.

In conclusion, HadesCI marks a promising step toward realizing a scalable continuous integration system tailored to programming exercise environments. By addressing the shortcomings of existing CI systems and offering a flexible and scalable solution, HadesCI paves the way for new possibilities in educational contexts and beyond. With its emphasis on simplicity, scalability, and security, HadesCI is well-positioned to meet the future needs of both modern software development practices and educational frameworks.

Bibliography

[Ela+22] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, and M.-A. Storey, "Uncovering the Benefits and Challenges of Continuous Integration Practices," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2570–2583, Jul. 2022, doi: 10.1109/TSE.2021.3064953.

[SAZ17] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017, doi: 10.1109/ACCESS.2017.2685629.

[SB14] D. Ståhl and J. Bosch, "Modeling Continuous Integration Practice Differences in Industry Software Development," *Journal of Systems and Software*, vol. 87, pp. 48–59, Jan. 2014, doi: 10.1016/j.jss.2013.08.032.

[Sta+15] T. Staubitz, H. Klement, J. Renz, R. Teusner, and C. Meinel, "Towards Practical Programming Exercises and Automated Assessment in Massive Open Online Courses," in *2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, Zhuhai, China: IEEE, Dec. 2015, pp. 23–30. doi: 10.1109/TALE.2015.7386010.

[KS18] S. Krusche and A. Seitz, "ArTEMiS: An Automatic Assessment Management System for Interactive Learning," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, Baltimore Maryland USA: ACM, Feb. 2018, pp. 284–289. doi: 10.1145/3159450.3159602.

[Esf+16] H. Esfahani *et al.*, "CloudBuild: Microsoft's Distributed and Caching Build Service," in *Proceedings of the 38th International Conference on Software Engineering Companion*, Austin Texas: ACM, May 2016, pp. 11–20. doi: 10.1145/2889160.2889222.

[Wan+20] K. Wang, G. Tener, V. Gullapalli, X. Huang, A. Gad, and D. Rall, "Scalable Build Service System with Smart Scheduling Service," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Event USA: ACM, Jul. 2020, pp. 452–462. doi: 10.1145/3395363.3397371.

[Edd+17] B. P. Eddy *et al.*, "A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses," in *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, Savannah, GA: IEEE, Nov. 2017, pp. 47–56. doi: 10.1109/CSEET.2017.18.

[Blu24] L. F. Blumentritt, "Extending Local Continuous Integration in Artemis," 2024.

[Res24] A. Resch, "A Domain-Specific Language for Streamlining CI Job Configuration for Programming Exercises," 2024.

List of Figures

List of Listings

Transparency in the use of AI tools

In preparing this report, I utilized Grammarly and ChatGPT for grammar and style correction in Chapter 1, 3, 4 and 6, ensuring clarity and coherence in my writing. After using these tools, I carefully reviewed the content and refined the text manually to maintain accuracy and precision.

Additionally, I used Consensus to broaden the landscape of related work in Chapter 2, identifying gaps in the literature and refining the research question. The insights from Consensus were subsequently cross-checked and incorporated into the research after a thorough manual review to ensure relevance and alignment with the study's objectives.