Technical University of Munich

School of Computation, Information and Technology

-- Informatics --

Bachelor’s Thesis in Informatics

Load Testing and Performance Evaluation of the Theia Online IDE

Tobias Wen Klingenberg

Technical University of Munich

School of Computation, Information and Technology
-- Informatics --

Bachelor's Thesis in Informatics

Load Testing and Performance Evaluation of the Theia Online IDE

Belastungstests und Leistungsbewertung der Theia Online IDE

Author:	Tobias Wen Klingenberg
Supervisor:	Prof. Dr. Stephan Krusche
Advisors:	Matthias Linhuber, M.Sc.
Start Date:	03.07.2025
Submission Date:	03.11.2025

I confirm that this Bachelor's thesis is my own work and I have documented all sources and material used.

[illegible]

Munich, 03.11.2025 Tobias Wen Klingenberg

Transparency in the use of AI tools

In preparing this thesis, I utilized Grammarly[1] for grammar and style correction throughout the thesis, ensuring clarity and coherence in my writing. I used DeepL[2] to enhance language quality and translate parts of the Literature Review. I used ChatGPT[3] to generate initial drafts and expand on ideas, providing valuable suggestions and examples. Additionally, I used Claude Sonnet[4] and ChatGPT to generate code snippets for the developed functionality. I have carefully checked all texts created with these tools to ensure they are correct and make sense.

[1]https://app.grammarly.com
[2]https://deepl.com
[3]https://chatgpt.com
[4]https://anthropic.com/

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor, Matthias Linhuber, for his invaluable guidance, support, and continuous encouragement throughout the duration of this thesis. His expertise, insights, and thoughtful suggestions have been instrumental in shaping this work and ensuring its success.

Furthermore, I would like to thank my supervisor, Prof. Dr. Stephan Krusche, for making this work possible and for his support and guidance throughout this thesis.

I would also like to thank the colleagues from Eclipse Source, who helped me with technical questions and provided valuable insights, as well as everyone involved in the Scalable Cloud Tools project at AET for providing the infrastructure and support for this work.

Last but not least, I would like to thank my family and friends for their support and encouragement throughout the duration of this thesis.

Abstract

Theia is a browser-based Integrated Development Environment (IDE) increasingly adopted in educational environments, including the Artemis[5] system at the Technical University of Munich. As web-based learning platforms become more interactive and student-centered, the reliability and scalability of tools like Theia are important, particularly during critical scenarios such as programming exams, where hundreds of students may simultaneously rely on the IDE. However, current testing approaches fail to validate Theia's behavior under such conditions, especially when considering real-world interactive usage patterns.

This thesis addresses the dual challenge of ensuring both functional correctness and performance scalability of Theia in an educational context. The first part of the work focuses on developing a maintainable and realistic end-to-end (E2E) test suite using Playwright. These tests simulate key user interactions to identify functional regressions.

The second part of the thesis investigates Theia's scalability and load tolerance when exposed to simultaneous student usage. Theia's dynamic instance model and high reliance on real-time browser rendering introduce unique challenges for automated load testing. To address this, the thesis explores orchestrating Playwright-driven browser sessions to simulate concurrent usage.

[5]https://artemis.tum.de

Zusammenfassung

Theia ist eine browserbasierte integrierte Entwicklungsumgebung (IDE), die zunehmend in Bildungsumgebungen eingesetzt wird, darunter auch das Artemis-System[6] an der Technischen Universität München. Da webbasierte Lernplattformen immer interaktiver und auf Studierenden fokussiert werden, sind die Zuverlässigkeit und Skalierbarkeit von Werkzeugen wie Theia wichtig, insbesondere bei kritischen Szenarien wie Programmierprüfungen, bei denen Hunderte von Studierenden gleichzeitig auf die IDE angewiesen sein können. Aktuelle Testansätze sind jedoch unzureichend, um das Verhalten von Theia unter solchen Bedingungen zu testen, insbesondere wenn man reale interaktive Nutzungsmuster berücksichtigt.

Diese Arbeit befasst sich mit den Herausforderungen, sowohl die funktionale Korrektheit als auch die Skalierbarkeit der Leistung von Theia in einem Bildungskontext sicherzustellen. Der erste Teil der Arbeit konzentriert sich auf die Entwicklung einer wartbaren und realistischen End-to-End (E2E) Testsuite mit Playwright. Diese Tests simulieren wichtige Benutzerinteraktionen, um funktionale Regressionen zu identifizieren.

Der zweite Teil der Arbeit untersucht die Skalierbarkeit und Lasttoleranz von Theia bei gleichzeitiger Nutzung durch Studenten. Das dynamische Instanzmodell von Theia und die hohe Abhängigkeit von Echtzeit-Browser-Rendering stellen besondere Herausforderungen für automatisierte Lasttests dar. Um dies anzugehen, untersucht die Arbeit die Orchestrierung von Playwright-gesteuerten Browsersitzungen, um eine parallele Nutzung zu simulieren.

[6]https://artemis.tum.de

Contents

1 Introduction

Cloud-based Integrated Development Environments (IDEs) have transformed how we develop, teach, and deliver software. Starting with traditional desktop IDEs such as Eclipse[7] or IntelliJ IDEA[8], the field has evolved toward fully browser-based environments that eliminate the need for local setup. Advances in web technologies, containerization, and cloud computing, which make it possible to provide standardized and instantly accessible development environments at scale, have driven this evolution. Platforms such as Gitpod, VS Code for the Web, and Theia[9] exemplify this trend. They are particularly relevant in education, where reducing technical barriers for students and ensuring consistency across heterogeneous hardware and operating systems is of central importance.

Theia Cloud, an open-source extensible IDE framework, is deployed at the Technical University of Munich (TUM) in conjunction with Artemis[10], a learning platform that supports programming exercises with automatic assessment [KS18]. Within this setup, Theia Cloud is connected to Artemis through a dedicated OpenVSX extension [Jan24] and launched via a custom integration, which provisions a dedicated Theia instance for each student [Sch24]. This design enables a seamless workflow where students can directly solve programming tasks, test their code, and submit their solutions within the same environment. From a pedagogical perspective, such integration reduces friction, shortens setup times, and allows instructors to focus on learning objectives rather than technical support.

However, the integration also introduces significant technical challenges. Unlike traditional single-user IDEs, Theia Cloud must reliably serve hundreds of concurrent users in scenarios such as programming exams, where time pressure and robustness are crucial. Performance degradations, high latency, or failures in launching IDE instances can disrupt exams and negatively affect the fairness and reliability of assessments. Beyond exams, scalability and stability are equally

[7]https://eclipseide.org/
[8]https://www.jetbrains.com/idea/
[9]https://theia-cloud.io/
[10]https://artemis.tum.de/

critical for large courses in self-study and homework contexts, where the platform must remain responsive under sustained load.

1.1 Problem

Despite its advantages, the current Theia Cloud integration lacks systematic automated testing and performance evaluation. At present, there is no comprehensive end-to-end (E2E) test suite to verify that core functionalities, such as file editing, terminal usage, code execution, and version control, work reliably from the perspective of a student or instructor. This gap increases the risk that regressions, UI inconsistencies, or broken features remain undetected until they cause disruptions during real usage scenarios [Pau01]. Manual testing is insufficient to guarantee stability, as Theia Cloud is under active development and frequent updates can unintentionally introduce new issues.

In addition to functional reliability, scalability represents another unresolved challenge. Theia Cloud has not undergone systematic testing under realistic high-load conditions. Hundreds of students may access the system concurrently during programming courses and exams, performing tasks such as compiling, running, and submitting code. Although research regarding creating a more resource-light and easy-to-set-up environment for Theia Cloud has been conducted [Jan25], without structured load testing, the system may exhibit degraded performance, including slow response times, unresponsive interfaces, or even complete service failures. Such problems directly impact students, who rely on the IDE to complete assignments or exams within strict time limits, and may lead to unnecessary stress and frustration [SB10], as well as instructors, who must deal with interruptions and technical failures.

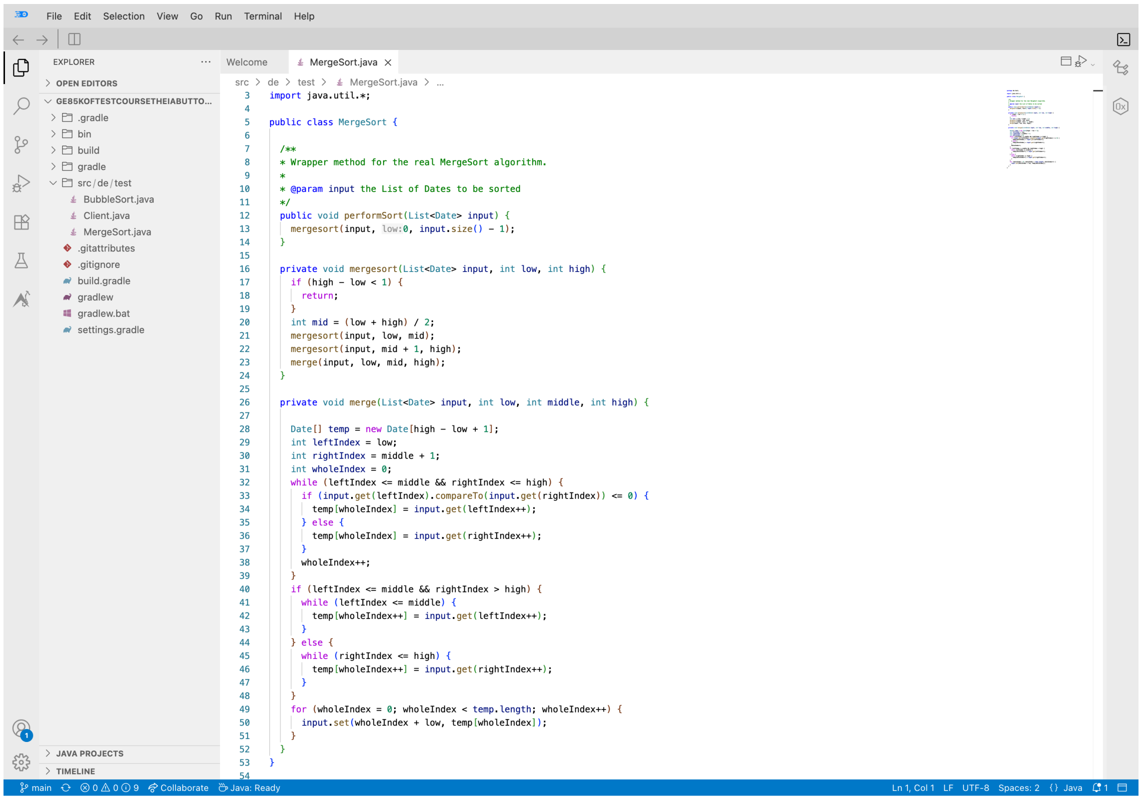
Although monitoring tools such as Grafana[11] provide valuable runtime insights into metrics like CPU, memory, and request throughput, they are insufficient for identifying where and when the system breaks under stress [Luc71]. Monitoring shows symptoms, but does not reproduce the dynamic and unpredictable behavior

[11]https://grafana.com/

of actual users. As a result, critical bottlenecks and scalability limitations often only become visible during live usage, when it is too late to mitigate them effectively. This lack of systematic, automated testing, both at the functional and performance levels, represents a significant obstacle to ensuring the reliability, usability, and scalability of Theia Cloud in educational settings.

1.2 Motivation

The reliability and scalability of browser-based IDEs are critical to modern programming education. Students rely on seamless access to development tools as seen in Figure 1, especially during assessments, where interruptions can lead to frustration, lost time, or unfair grading outcomes. A responsive and stable IDE directly influences learning success and overall student satisfaction [HOC17]. In high-stakes settings such as exams, even short outages or delays can undermine trust in the system and compromise the validity of the evaluation process.



```java
import java.util.*;

public class MergeSort {

  /**
  * Wrapper method for the real MergeSort algorithm.
  *
  * @param input the List of Dates to be sorted
  */
  public void performSort(List<Date> input) {
    mergesort(input, low:0, input.size() - 1);
  }

  private void mergesort(List<Date> input, int low, int high) {
    if (high - low < 1) {
      return;
    }
    int mid = (low + high) / 2;
    mergesort(input, low, mid);
    mergesort(input, mid + 1, high);
    merge(input, low, mid, high);
  }

  private void merge(List<Date> input, int low, int middle, int high) {

    Date[] temp = new Date[high - low + 1];
    int leftIndex = low;
    int rightIndex = middle + 1;
    int wholeIndex = 0;
    while (leftIndex <= middle && rightIndex <= high) {
      if (input.get(leftIndex).compareTo(input.get(rightIndex)) <= 0) {
        temp[wholeIndex] = input.get(leftIndex++);
      } else {
        temp[wholeIndex] = input.get(rightIndex++);
      }
      wholeIndex++;
    }
    if (leftIndex <= middle && rightIndex > high) {
      while (leftIndex <= middle) {
        temp[wholeIndex++] = input.get(leftIndex++);
      }
    } else {
      while (rightIndex <= high) {
        temp[wholeIndex++] = input.get(rightIndex++);
      }
    }
    for (wholeIndex = 0; wholeIndex < temp.length; wholeIndex++) {
      input.set(wholeIndex + low, temp[wholeIndex]);
    }
  }
}
```

Figure 1: Example View for a student during a programming exam using Theia Cloud

Ensuring such reliability requires more than just stable infrastructure; it depends on a deep understanding of how students and instructors interact with the IDE in real educational contexts. Examining their respective needs and expectations makes it clear why system robustness and scalability are not merely technical goals but essential prerequisites for effective teaching and learning. The following will examine how different target groups may benefit from a stable infrastructure and a reliable testing system.

1.2.1 Instructors And Students

For instructors and educational institutions, a robust IDE infrastructure enables the creation of richer and more complex exercises without being limited by technical uncertainties. When instructors can assume a stable baseline for the environment, they can design tasks that emphasize problem-solving, algorithmic thinking, and creativity rather than troubleshooting setup issues. This shift allows more focus on pedagogy while reducing the support burden during courses and exams.

1.2.2 Operational Perspective

From an operational perspective, universities must simultaneously serve large cohorts of students, often under strict time constraints. Without systematic scalability testing, institutions risk performance bottlenecks and failures that can jeopardize entire examinations. Therefore, automated verification and realistic load simulations are essential to quality assurance in digital learning environments.

1.2.3 Scientific Perspective

From a scientific standpoint, testing interactive web applications such as cloud-based IDEs remains a challenging and underexplored problem [Mar+11]. End-to-end testing approaches must account for asynchronous user interactions, real-time feedback, and integration with external services. Furthermore, scalability testing in cloud-native education infrastructure introduces unique challenges: hundreds of parallel user sessions, varying network conditions, and the need to reproduce realistic student workflows. Existing research on Software-as-a-Service (SaaS) testing and performance engineering [THS11] provides valuable foundations, but the educational use case introduces distinct requirements regarding fairness, reliability, and usability.

By addressing both functional correctness and load behavior in a unified framework, this thesis contributes to ongoing research in end-to-end testing, performance engineering, and educational technology. The results enhance Theia Cloud's reliability within Artemis and provide transferable insights for designing and testing other cloud-based educational systems.

1.3 Objectives

This thesis aims to ensure the reliability, usability, and scalability of Theia Cloud within Artemis by developing a structured, automated testing framework. This involves both E2E UI and scalability testing, enabling the identification of performance bottlenecks under real-world conditions [Pau01]. The key objectives of this thesis are:

Develop an automated Theia E2E testing suite

To ensure the correctness and stability of Theia Cloud's interactive features, this thesis develops an automated E2E test suite. The basic structure of the test suite, as seen in Figure 2, covers essential functionalities, including code editing, file management, terminal interactions, and version control integration.

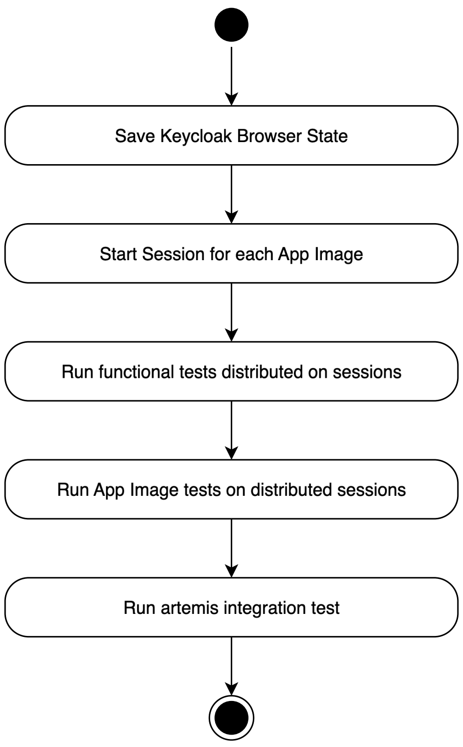

Save Keycloak Browser State
Start Session for each App Image
Run functional tests distributed on sessions
Run App Image tests on distributed sessions
Run artemis integration test

Figure 2: Basic Activity Diagram of a functional test workflow

Implement A Scalable Load Testing Framework

To evaluate Theia Cloud's performance under high demand, this thesis implements a scalable load testing framework using Artillery[12]. The tests simulate realistic user behavior, including concurrent file operations, terminal interactions, and version control operations.

Analyze System Performance And Identify Bottlenecks

This thesis analyzes the performance of Theia Cloud under load, identifying bottlenecks in the system's response time and resource utilization. The goal is to comprehensively evaluate Theia Cloud's scalability limits, helping system administrators and developers make informed decisions about optimizations and infrastructure scaling.

[12]https://artillery.io/

Create Randomized And Personalized Tests Using LLMs
The scalable load testing framework should be able to generate randomized and personalized tests using LLMs to ensure the test suite is comprehensive and practical. It uses the MCP (Model Context Protocol)[13] to create diverse test cases covering various programming languages, IDE features, and user behaviors. Therefore, we can evaluate Theia Cloud's capabilities and performance under different conditions more comprehensively.

1.4 Outline

This thesis is organized into seven chapters as follows:

Chapter 1 Introduction presents the research problem and introduces the existing system, its architecture, and the motivation for building an automated E2E and load testing framework.

Chapter 2 Related Work reviews existing approaches and research in automated testing, performance evaluation, and educational IDEs.

Chapter 3 Requirements Analysis

identifies and analyzes the functional and non-functional requirements for testing Theia Cloud.

Chapter 4 System Design

describes the architecture and implementation of the proposed testing framework, including E2E and load testing strategies.

Chapter 5 Test Results

discusses the results, challenges and limitations of the testing suite.

Chapter 6 Summary

summarizes the findings and outlines directions for future research.

[13]https://modelcontextprotocol.io/

2 Related Work

E2E testing has become an integral part of modern software development practices, enabling developers to validate entire workflows' functionality from the end user's perspective [Pau01]. In recent years, a variety of frameworks have emerged for E2E testing, with Playwright[14], developed by Microsoft, standing out due to its cross-browser support, reliable automation capabilities, and ease of integration into CI/CD pipelines.

2.1 E2E Testing With Playwright

Various projects have adopted Playwright, from web applications to complex interactive platforms. Its strengths lie in its ability to automate user interactions across multiple browsers (Chromium, Firefox, and WebKit), while offering features such as network interception, precise control over timeouts, and built-in support for parallel execution. Studies and case reports have demonstrated that Playwright is well-suited for verifying user flows that involve complex asynchronous operations, such as form submissions, authentication procedures, or real-time updates [HN25].

Compared to earlier tools such as Selenium and Cypress, Playwright's architectural design reduces test flakiness and improves reproducibility, critical in large-scale systems [HN25]. Several research efforts and industrial reports highlight the importance of reproducible E2E testing environments when simulating real-world usage under varying conditions [Pau01].

2.1.1 Use Case Scenario

Playwright provides several benefits to the scenario addressed in this thesis. First, it enables the automation of student workflows in the IDE through realistic browser-driven interactions, such as editing code, running programs, and using version control. Second, its reliability in handling dynamic content and asynchronous operations is essential, since online IDEs often involve real-time updates and collaborative features. Third, Playwright integrates well with modern development environments. It can be easily extended with load testing tools or monitoring

[14]https://playwright.dev/

systems, making it a suitable foundation for correctness testing and performance evaluation. Finally, its open-source nature and active community support ensure long-term maintainability and adaptability to evolving requirements.

2.2 E2E Testing Of Online IDEs

The testing of Online IDEs presents unique challenges due to their highly interactive and resource-intensive nature. Unlike traditional web applications, Online IDEs must support continuous user interaction in file editing, syntax highlighting, version control integration, and terminal execution. These event-driven interactions often require real-time feedback, making E2E testing a non-trivial task.

Although research on automated testing of Online IDEs is still limited, some initiatives have begun to explore automated validation of cloud-based programming environments. A notable example is a bachelor's thesis at Brno University of Technology, which developed a UI testing module for Eclipse Che compatible with the VS Code Extension Tester. The thesis highlights several core challenges: testers cannot reuse system libraries from traditional local IDE testing in a cloud context, resource constraints reduce test stability, and developers must introduce component abstractions to enable cross-IDE compatibility. The module builds on Selenium WebDriver, which forms the foundation of the VS Code Extension Tester. Selenium enabled simulation of user interactions with the graphical interface, but also introduced complexities such as managing multiple browser drivers and handling asynchronous waits. The author notes that while Selenium's explicit waits are essential, its implicit waits cannot be combined without causing unstable behavior, complicating reliable test execution in CI environments [Lor21].

In practice, most Online IDE test strategies focus on the correctness of basic workflows (e.g., file creation, compilation, terminal commands) rather than large-scale stress testing. Thus, a research gap exists concerning how developers can systematically apply E2E testing frameworks to Online IDEs in educational or exam settings.

2.3 E2E Testing Within Artemis

Artemis, the learning platform developed at the Technical University of Munich, has integrated automated testing approaches to ensure the reliability of its features, including exercise management, grading pipelines, and interactive programming assignments. Within Artemis, the system utilizes E2E testing to validate workflows that students and instructors rely on, such as creating exercises, submitting solutions, and receiving feedback.

Playwright has recently been introduced into the Artemis testing ecosystem to replace or complement Cypress-based setups, providing more robust and maintainable test suites [Tal24]. Early efforts focus on automating the workflows around programming exercises, such as importing exercises, managing repositories, and simulating student submissions.

However, E2E testing in Artemis faces challenges similar to those of Online IDEs. Programming exercises rely on external tools and environments, such as IRIS and many different extensions. Current work in Artemis has primarily concentrated on verifying core platform workflows. Systematic large-scale E2E testing, especially load testing, using E2E frameworks, is still an open area of research.

3 Requirements

This chapter outlines the requirements for the automated testing framework for Theia Cloud, based on the requirements analysis by Bruegge et al. [BD04]. It begins with an overview of the envisioned solution in Section 3.1, followed by a presentation of several dynamic models in Section 3.2. We will continue with a structured presentation of the proposed system in Section 3.3, which derives the functional and non-functional requirements from the system models in Section 3.3.1 and Section 3.3.2. These requirements are the foundation for the subsequent design, implementation, and evaluation phases.

3.1 Overview

The primary objective of this thesis is to define, design, and evaluate a system capable of performing automated E2E testing and load testing for an Online Integrated Development Environment. The system aims to provide a reliable and repeatable method for assessing both the functional correctness and performance of the IDE under realistic conditions. Simulating user interactions at scale enables the identification of potential bottlenecks, stability issues, and areas for optimization before they impact real users. Furthermore, the system is intended to support continuous evaluation, allowing developers and administrators to maintain high-quality, scalable IDE deployments over time. Overall, the goal is to create a comprehensive testing framework that enhances confidence in the IDE's reliability, responsiveness, and suitability for large-scale educational use.

3.2 Dynamic Models

To better illustrate the requirements and interactions of the proposed system, we present two dynamic models.

3.2.1 Activity Diagram

The activity diagram shown in Figure 3 showcases a typical student workflow for an IDE. It starts with the student opening the IDE, potentially changing preferences in the UI and the behavior of the IDE, installing project-specific extensions, cloning a repository, editing the code, running the code, and committing and pushing the changes to the remote repository.

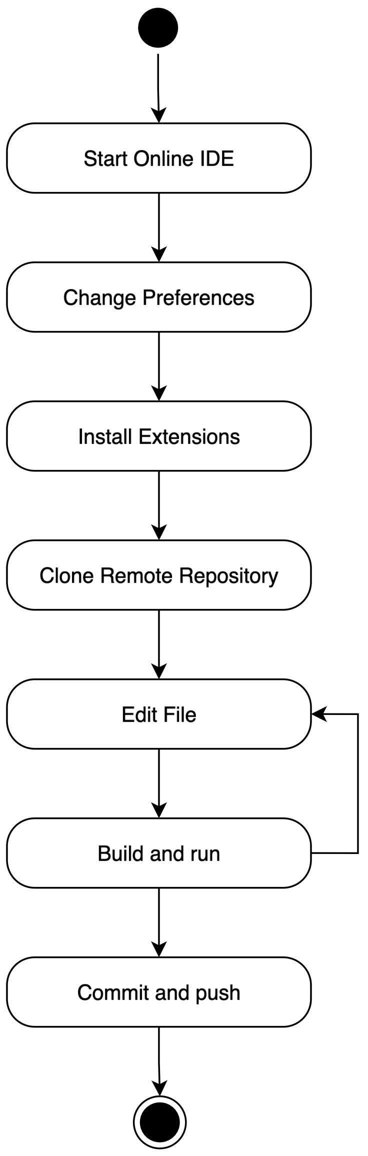


Figure 3: Activity Diagram showcasing a typical student workflow for the IDEs

3.2.2 Use Case Model

Focusing on integrating the IDE with Artemis, we can derive the following use case models, as shown in Figure 4.

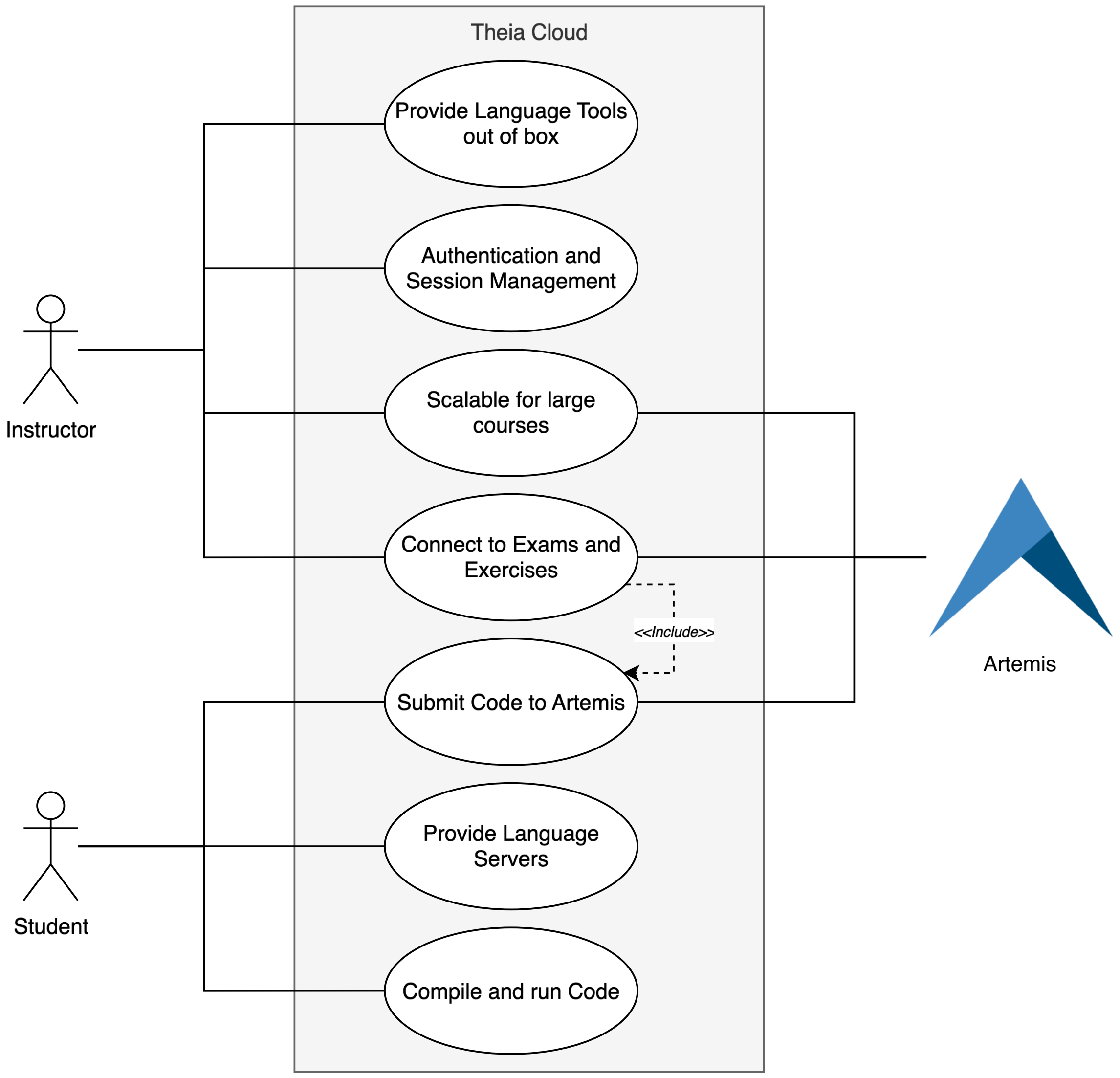

Theia Cloud
Provide Language Tools out of box
Authentication and Session Management
Scalable for large courses
Connect to Exams and Exercises
<<Include>>
Submit Code to Artemis
Provide Language Servers
Compile and run Code
Instructor
Student
Artemis

Figure 4: Use Case for Online IDEs integrated with Artemis

Students primarily interact with the IDE to perform programming tasks. Their core use cases include compiling and running code, accessing language server features such as code completion or error highlighting, and submitting code to Artemis for assignments or exams. These use cases reflect typical workflows in computer science courses and emphasize the importance of an IDE that behaves reliably under load, especially in exam settings where hundreds of students may be active simultaneously.

Instructors, on the other hand, rely on the IDE and its integration with Artemis to support teaching activities. Their use cases include ensuring that language tools are available out of the box, without requiring manual setup for different programming languages. They also require secure authentication and session management, both to protect student data and to enforce academic integrity. Furthermore, instructors expect the system to be scalable, capable of supporting many concurrent users without performance degradation, and to be seamlessly connected with exams and exercises hosted on Artemis. These requirements ensure that instructors can confidently use the IDE for both regular coursework and high-stakes assessments.

Together, these use cases highlight the dual perspective of students and instructors and underscore the system's role as a bridge between interactive coding environments and the educational infrastructure provided by Artemis. By analyzing these use cases, we can identify the requirements for the system under test and conclude the functional requirements for the proposed functional testing system.

3.3 Proposed System

We propose a new system because there is currently no system that can perform automated E2E testing and load testing for an Online Integrated Development Environment IDE.

3.3.1 Functional Requirements

By assessing the previous dynamic models, we can derive the following functional requirements and quality attributes, according to the requirements analysis by Bruegge et al. [BD04]:

- **FR1. Simulate User Interactions**:

The system must simulate realistic user interactions within the IDE, including opening, editing, and saving files, using the terminal to compile and run programs, managing projects or exercises, and performing version control operations such as committing, pushing, and pulling changes. It must support the execution of sequences of actions that accurately mimic real student workflows.

- **FR2. Automated End-to-End Testing**:

The system must provide automated E2E testing capabilities to verify the correctness of IDE features. It should allow test scenarios to be defined, reused, and parameterized to cover different workflows and use cases, and it must detect errors or failures in core functionality.

- **FR3. Load and Performance Testing**:

The system must also support load and performance testing by simulating multiple concurrent users. It should measure response times, throughput, and error rates under varying levels of load, and it must be capable of reproducing peak usage scenarios, such as those observed during exams or large-scale course sessions.

- **FR4. Monitoring and Reporting**:

During testing, the system must collect relevant performance and behavioral metrics. It should generate detailed reports summarizing test results, errors, and bottlenecks, and provide visualizations or summaries easily interpretable by developers and system administrators.

- **FR5. Scenario Management**:

The system must allow test scenarios to be defined, configured, and managed efficiently, including the ability to configure user behavior profiles and test parameters, and to enable repeatable execution of scenarios for regression testing and validation of system stability over time.

- **FR6. Integration with Intelligent Agents**:

The system should support integration of intelligent agents based on the Model Context Protocol. These agents can simulate adaptive and realistic student behavior, and their behavior should be customizable to represent different skill levels or interaction patterns.

3.3.2 Quality Attributes & Constraints

This section lists the requirements not directly related to the system's functionality. We categorize them using the FURPS+ model (Functionality, Usability, Reliability, Performance, Supportability, and Constraints) described in [BD04]:

- **NFR1. Concurrent Test Execution (Performance)**:

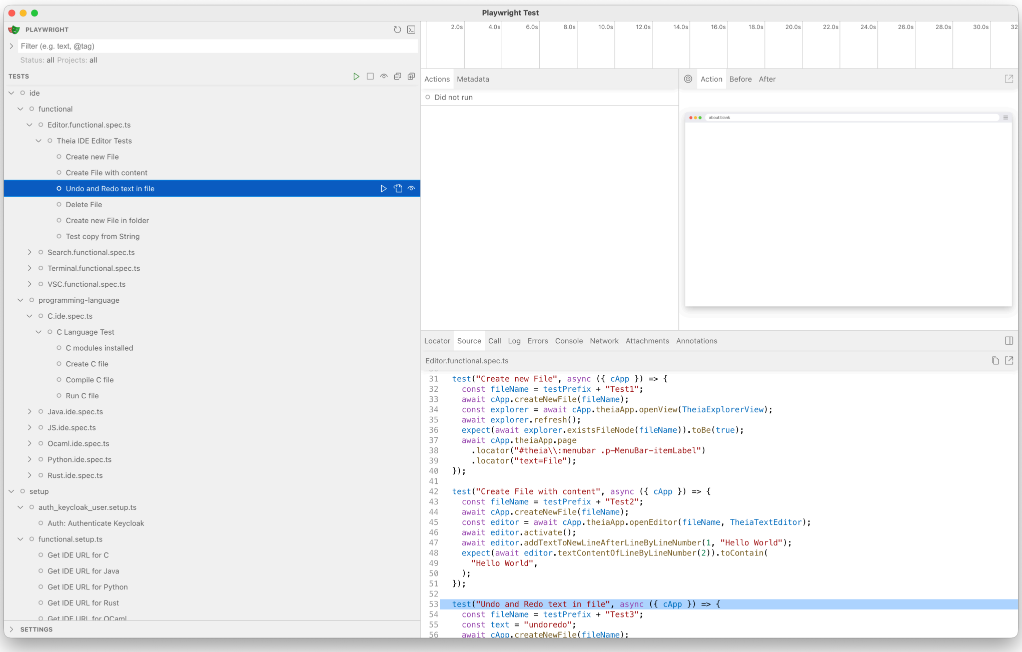
The system must handle high-performance requirements, such as the ability to simulate complex workflows and measure response times, throughput, and error rates under varying levels of load.

- **NFR2. Reproducible Test Results (Reliability)**:

The system must handle high reliability requirements, such as the ability to reproduce test results and detect errors or failures in core functionality.

- **NFR3. User Behavior Profiles (Usability)**:

The system must handle high usability requirements, such as the ability to configure user behavior profiles and test parameters, and to enable repeatable execution of scenarios for regression testing and validation of system stability over time, as seen in Figure 5.



```
test("Create new File", async ({ cApp }) => {
  const fileName = testPrefix + "Test1";
  await cApp.createNewFile(fileName);
  const explorer = await cApp.theiaApp.openView(TheiaExplorerView);
  await explorer.refresh();
  expect(await explorer.existsFileNode(fileName)).toBe(true);
  await cApp.theiaApp.page
    .locator("#theia\\:menubar .p-MenuBar-itemLabel")
    .locator("text=File");
});

test("Create File with content", async ({ cApp }) => {
  const fileName = testPrefix + "Test2";
  await cApp.createNewFile(fileName);
  const editor = await cApp.theiaApp.openEditor(fileName, TheiaTextEditor);
  await editor.activate();
  await editor.addTextToNewLineAfterLineByLineNumber(1, "Hello World");
  expect(await editor.textContentOfLineByLineNumber(2)).toContain(
    "Hello World",
  );
});

test("Undo and Redo text in file", async ({ cApp }) => {
  const fileName = testPrefix + "Test3";
  const text = "undoredo";
  await cApp.createNewFile(fileName);
```

Figure 5: Playwright UI for repeatable test execution and evaluation

- **NFR4. Scalable Test Execution (Performance / Reliability)**:

The system must be able to handle scalability requirements, such as the ability to simulate multiple concurrent users and the ability to measure response times, throughput, and error rates under varying levels of load.

- **NFR5. Cross-Platform Compatibility (Supportability)**:

The system must be able to handle portability requirements, such as the ability to run on different platforms and the ability to integrate with different systems.

- **NFR6. Data Security (Constraint)**:

The system should ensure that simulated interactions do not compromise IDE user data or access controls.

4 System Design

To present the system design, we will follow the System Design Template presented by Bruegge et al. [BD04]. After an Overview of the complete system architecture, we will present the design goals derived from the non-functional requirements in Section 4.1. We will continue presenting the different parts of the test suite, divided into the functional testing architecture in Section 4.3 and the load testing architecture in Section 4.4. Finally, we will present the MCP testing architecture in Section 4.5.

4.1 Overview

The Deployment Diagram in Figure 6 presents the overall architecture of the testing suite. We divided the test suite into three main components: the functional tests, the load tests, and the MCP tests. The functional tests are responsible for testing the system's functionality, the load tests are responsible for testing the system's performance, and the MCP tests are a proof of concept for using LLMs in the test suite. As we run the tests using the Playwright Test Runner, we can execute the test environment in a headless browser context. Furthermore, the tests are executed in parallel, which is possible due to the usage of the Page Object Model and fixtures, as described in Section 4.3.2. The Test Suite interacts with an LLM using the MCP Interface, as described in Section 4.5. All tests access the Online IDE using the Theia Landing Page and each Theia Client, which connects to the Theia Operator and each Session running in the K8s Cluster.

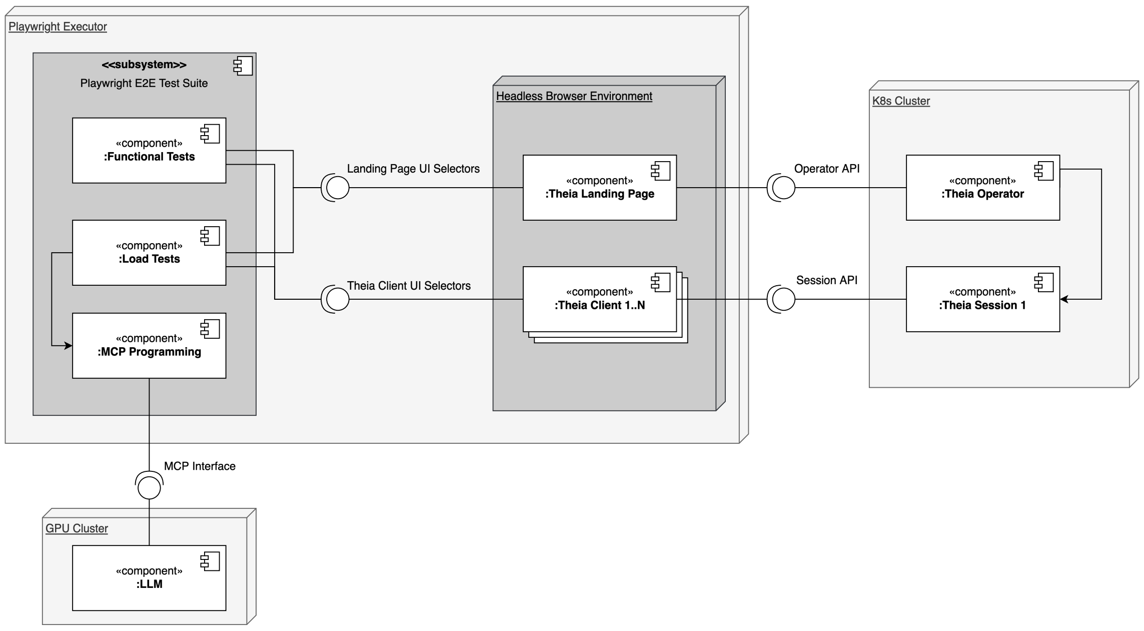


Figure 6: Proposed Deployment Diagram for testing suite

4.2 Design Goals

We derive the design goals from the non-functional requirements discussed in Section 3.3.2, ordered by their importance and the rationale behind the prioritization.

1. Scalability: As one of the most important non-functional requirements, scalability is a key design goal. The test suite should be able to handle many tests and users concurrently (NFR4).

2. Usability: Usability is a key design goal. The test suite should be easy to use and understand, which is important for the usability of the test suite and the reliability of the test results. Furthermore, adding more test cases should be easy and not require much effort if the developers add new features to the system (NFR3).

3. Reliability: Reliability is a key design goal to showcase reasonable and reproducible test results. The test suite should be able to reproduce the test results and should be able to handle a large number of tests and users concurrently (NFR2). This is important for the test suite's performance and the test results' reliability.

4. Performance: As many tests require a complex workflow, and load tests can reach parallel execution of up to 1000 users, performance is a key design goal. The test suite should be able to handle many tests and users concurrently (NFR1).

5. Portability: As the test suite should be able to run in CI/CD pipelines, portability is a key design goal. The test suite should be able to run on different platforms and integrate with different systems (NFR5).

6. Security: The test suite should be able to handle data security requirements, as we use the test suite in a CI/CD pipeline (NFR6).

4.3 Functional Testing

The functional tests are responsible for testing the functionality of the system. In the following, we will present the workflow of the functional tests, as well as the Page Object Model used in the test suite.

4.3.1 Activity Diagram

In the following, we will present the workflow of the functional tests, as seen in Figure 7. First, the test suite logs in to the Landing Page of Theia Cloud and saves the browser state to a file. This is helpful as each test uses an isolated browser context, which we do not share with other tests. After saving the browser state, the test suite starts a new Theia Instance for all programming languages, as we not only test the functionality of each App Image, but we also want to reuse the same instance for multiple tests. Creating a new Theia Instance for each test would be too time-consuming and would stress the system, which is not the purpose of the functional tests. Therefore, the link of each instance is stored in the runtime and can be accessed by the fixture as seen in Figure 8.

After the instances finish loading, the test suite starts the functional tests with the given number of workers in parallel. As seen in Table 2, we divide the functional tests into three categories: Editor Tests, Search Tests, Terminal Tests, and Version Control Tests. Several separate workers execute each category. Furthermore, the tests regarding the programming languages execute sequentially, as they require the same instance and depend on each other.

After all workers have finished, the test suite logs out of the Landing Page and deletes the browser state file.

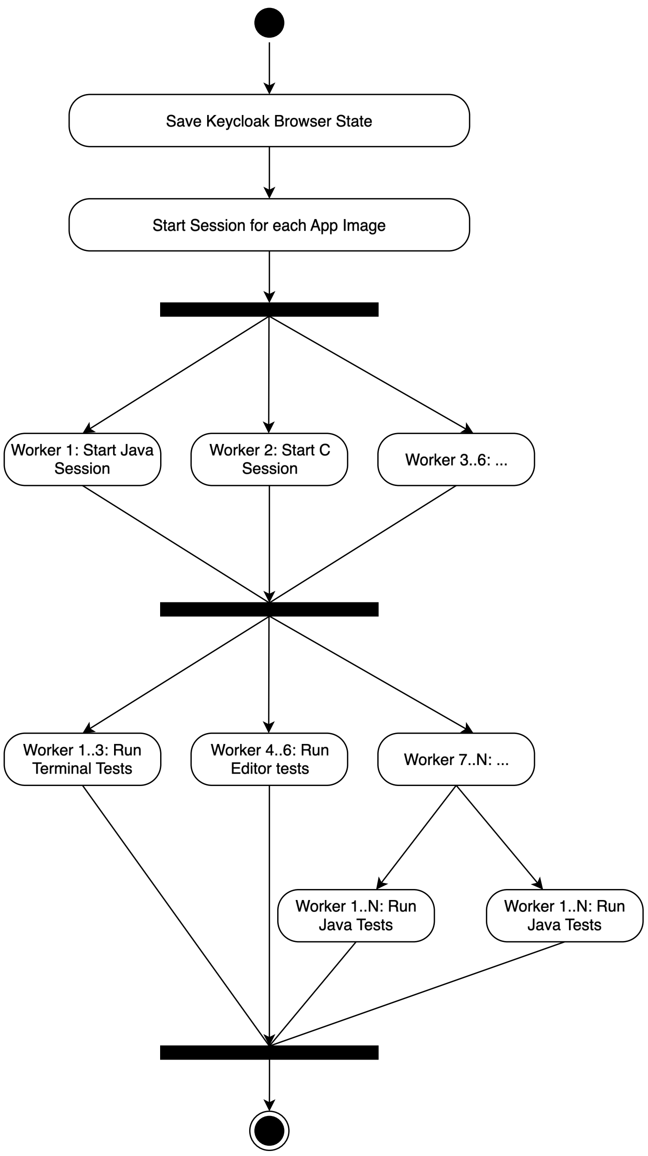

Save Keycloak Browser State
Start Session for each App Image
Worker 1: Start Java Session
Worker 2: Start C Session
Worker 3..6: ...
Worker 1..3: Run Terminal Tests
Worker 4..6: Run Editor tests
Worker 7..N: ...
Worker 1..N: Run Java Tests
Worker 1..N: Run Java Tests

Figure 7: Activity Diagram of functional tests workflow

4.3.2 Page Object Model

Sophisticated Playwright Testing often uses the Page Object Model (POM) to structure the test code [KK23]. It is a design pattern that separates the representation of the page from the test logic, making the code more maintainable and reusable. A Page Object Model is a class that represents a page in the application.

It contains the locators for the elements on the page and the methods to interact with them. An example of a method is the `launchLanguage` method, which starts a new Theia Instance for a specific programming language from the Landing Page:

```
async launchLanguage(language: string) {
    const languageButton = await this.page
.getByRole("button", { name: `Launch ${language}` })
.first();
    await languageButton.click();
}
```

To illustrate the Page Object Model used in the test suite, we present the class diagram in Figure 8.

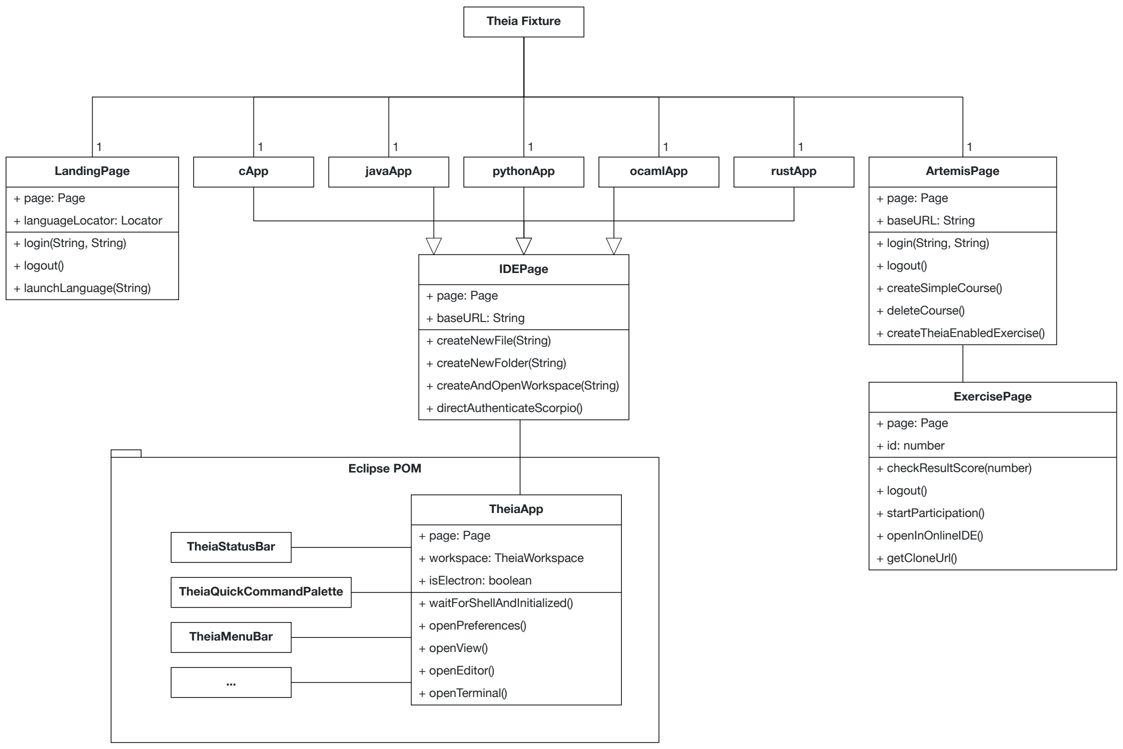


Figure 8: Class Diagram for Playwright Page Object Model

Another design pattern used in Playwright Testing is the usage of fixtures to set up the test environment. A fixture is a function that returns a value that we can use in the test. An example of a fixture is the `cApp` fixture, which accesses the Theia Instance that we started in the setup part of the test workflow. As we previously saw in Figure 7, the test structure starts a Theia Instance for a specific programming language and accesses it later through one of the given fixture methods. Each test can then access the instance as a parameter in the test function:

```
test("Compile C file", async ({ cApp }) => {
  const terminal = await cApp.theiaApp.openTerminal(TheiaTerminal);
  await terminal.submit(`gcc -o ${fileName}.out ${fileName}`);
  const explorer = await cApp.theiaApp.openView(TheiaExplorerView);
  await explorer.waitForVisible();
  await expect(
    await explorer.existsFileNode(`${fileName}.out`)
 ).toBeTruthy();
});
```

4.3.3 Sequence Diagram

As we also want to test the integration with Artemis, we present the sequence diagram in Figure 9. This is a separate test project, not part of the functional tests. We divide this test into several test steps, which execute in the given order:

1. Using the POM from the Artemis Fixture, the Test Runner creates a new course and exercise using the Artemis API running on a Test Server [Tal24].
2. After creating the exercise, the Test Runner redirects the user to the Exercise Page and starts the exercise in the Online IDE using the provided Button. The Runner gets redirected to the Theia Landing Page, logs in, and automatically starts a new instance.
3. Because the instance receives specific parameters, it automatically clones the exercise, allowing the user to start solving it.
4. The test runner opens and edits some of the code templates.
5. The test runner commits and pushes the changes to the remote repository.
6. After the user pushes the changes and Artemis completes the build, the test runner redirects to the Exercise Page and checks the results.

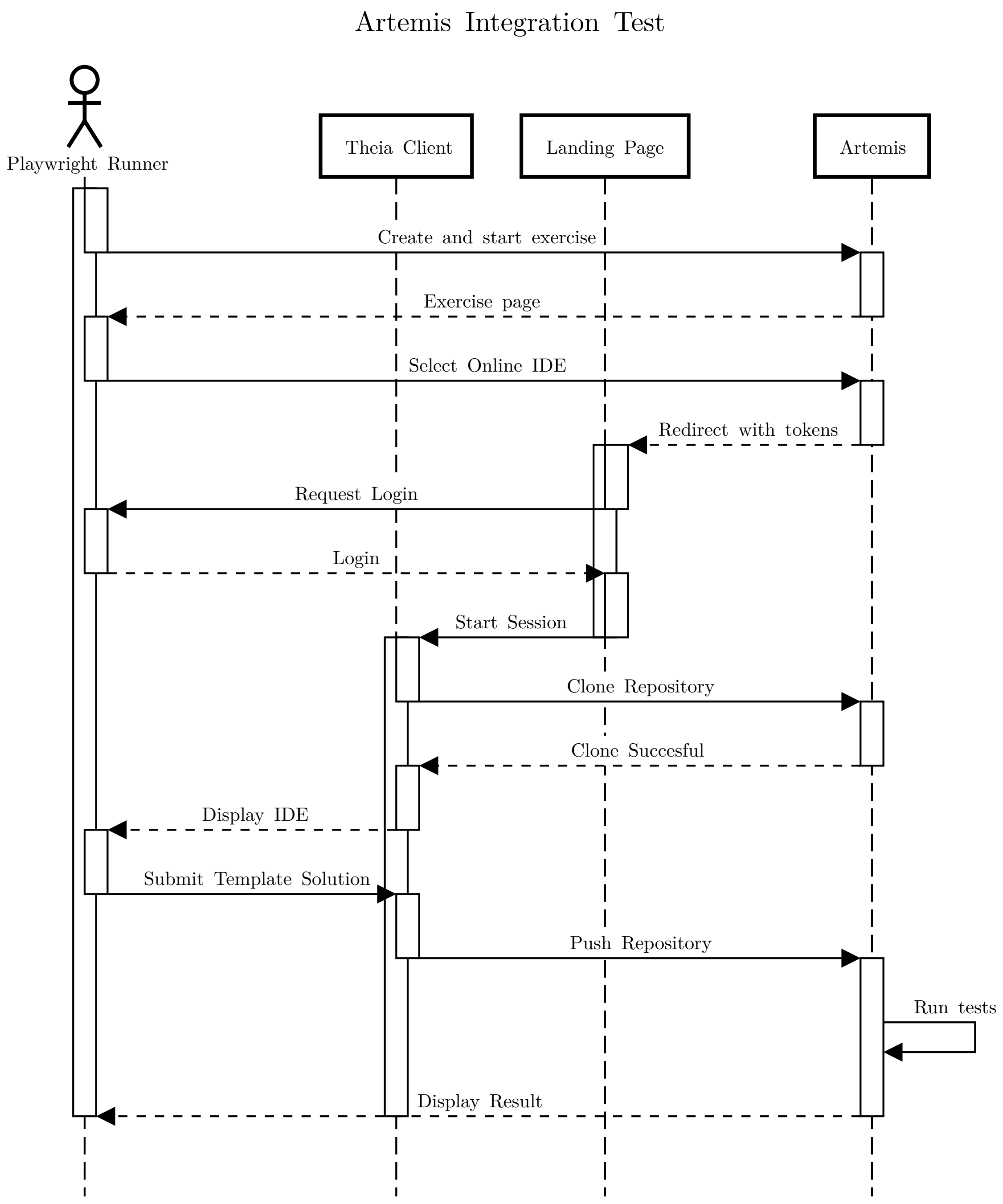


Figure 9: Sequence Diagram of Artemis Integration Test

4.4 Load Testing

The load tests are responsible for testing the system's performance under load. In the following, we will present the architecture of the load tests and the Deployment Diagram used in the test suite. As load tests are typically not part of E2E testing, but rather part of testing APIs under load, we will present a custom load testing architecture native for Playwright and one that utilizes the framework `Artillery.io`[12].

4.4.1 Load Testing Architecture

One approach to running load tests using Playwright is using the `Artillery.io` framework[12]. This framework is a tool for running load tests using a simple YAML configuration file or a JavaScript file. The deployment diagram in Figure 10 shows the architecture of the load tests using the `Artillery.io` framework. The `Artillery.io` framework utilizes Playwright as the engine to run a function on each runner. Each runner uses a separate instance of the Theia Session running in the K8s Cluster. The `Artillery.io` framework is responsible for scaling the number of runners to the given number of instances. Running tests are divided into phases, each with two specific parameters: the duration of the phase and the arrival rate. These parameters let us control load testing by defining how frequently the system creates new runners and how long they run.

Every runner runs the predefined function `virtualStudent`, which defines the workflow of the virtual student. After setting up the IDE environment, the runner randomly selects a scenario from the predefined list in Table 6 and executes the workflow until the phase finishes.

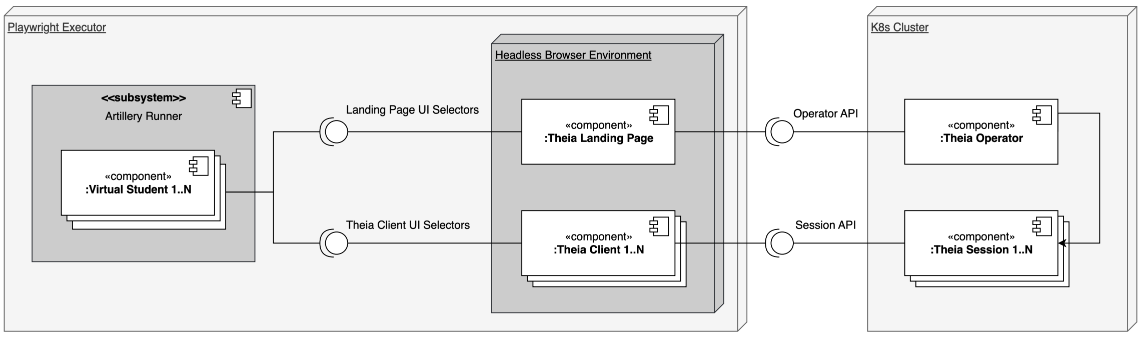


Figure 10: Deployment Diagram of Load Testing Architecture using Artillery and N instances of a Theia Session

As seen in Figure 10, the Artillery Runner runs in the Playwright Executor, and spins up 1..N new runners, which are then each accessing the single Theia Landing Page and one of the N Theia Instance Clients running in the Headless Browser

Environment. These components access the Theia Operator and Theia Session API running in the K8s Cluster, similar to the proposed architecture in Figure 6.

4.5 MCP Testing

Model Context Protocol is a tool for interacting with LLMs with context. By leveraging the MCP, this thesis also explores the possibility of generating tests that adapt to individual user profiles, preferences, and learning styles. This approach aims to enhance the realism and relevance of the testing scenarios, making them more representative of actual user interactions. As MCP is already an established protocol, we divided the MCP testing into two parts: the MCP Client running locally on one of the supported IDEs and a custom MCP Client that could run in the CI/CD pipeline.

4.5.1 Local MCP Client

The most common way to set up a Model Context Protocol (MCP) environment is to run the server component either inside a Docker container or as a separate process launched by the client application. Modern integrated development environments such as Theia, Visual Studio Code, and Cursor already include native support for this protocol. The IDE can establish a connection between the server and a large language model by providing the prebuilt Playwright MCP server[15] to the client.

As seen in Figure 11, the IDE acts as the MCP client and coordinates the Playwright MCP server and the LLM agent. Depending on the IDE's setup, the client connects to a remotely hosted LLM, enabling bidirectional communication between the two components. The client uses the MCP interface to give the server controlled access to the LLM's capabilities.

Whenever the language model needs to interact with the MCP server, it sends a message through the client, which forwards it accordingly using the Chrome DevTools Protocol (CDP)[16]. The server can execute Playwright commands based

[15]https://github.com/microsoft/playwright-mcp
[16]https://chromedevtools.github.io/devtools-protocol/

on the received instructions, such as navigating to a page, clicking on elements, or performing input actions. Meanwhile, the IDE forwards user inputs to the client, which converts them into context for the LLM, allowing it to generate appropriate prompts for testing and interacting with the system.

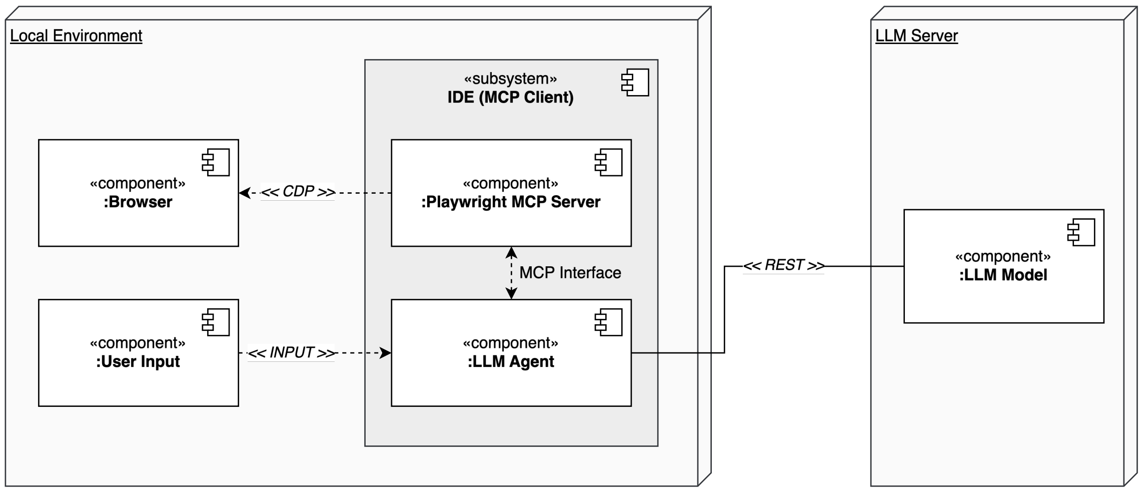


Figure 11: Hardware Software Mapping of MCP running locally

4.5.2 Remote MCP Client

As we potentially do not want to depend on a specific IDE, we also explore the possibility of running a custom MCP Client, which we can use in the CI/CD pipeline. As seen in Figure 12, we first manually start the MCP Server and then start a custom MCP Client, which connects to the MCP Server using the MCP Interface and the LLM Server using the appropriate API. The custom MCP Client then automatically sends the predefined prompt to the LLM and waits for the response. The system then passes the response to the MCP Server, which executes the Playwright Commands, such as navigating to a specific page or clicking on a specific element.

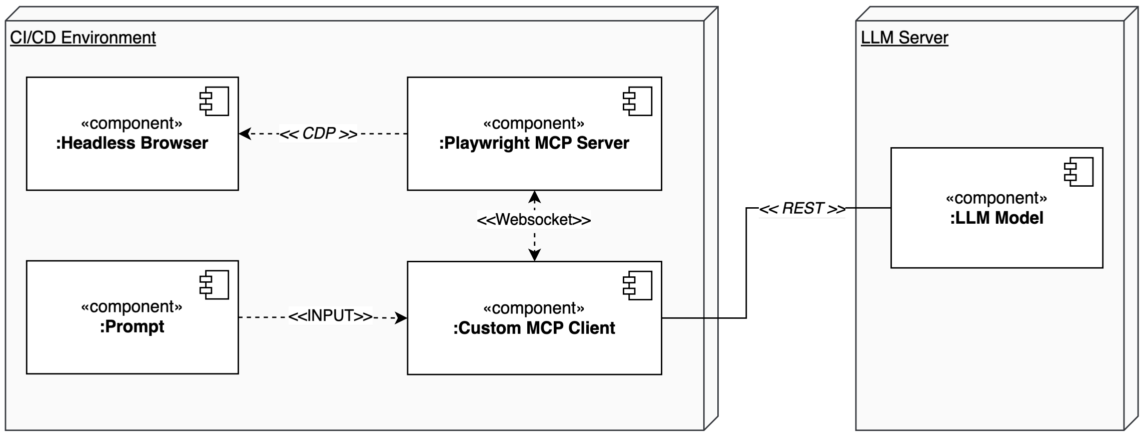


Figure 12: Hardware Software Mapping of MCP running in CI/CD

4.5.2.1 Activity Diagram

To further illustrate the workflow of the MCP Client, we present the activity diagram in Figure 13.

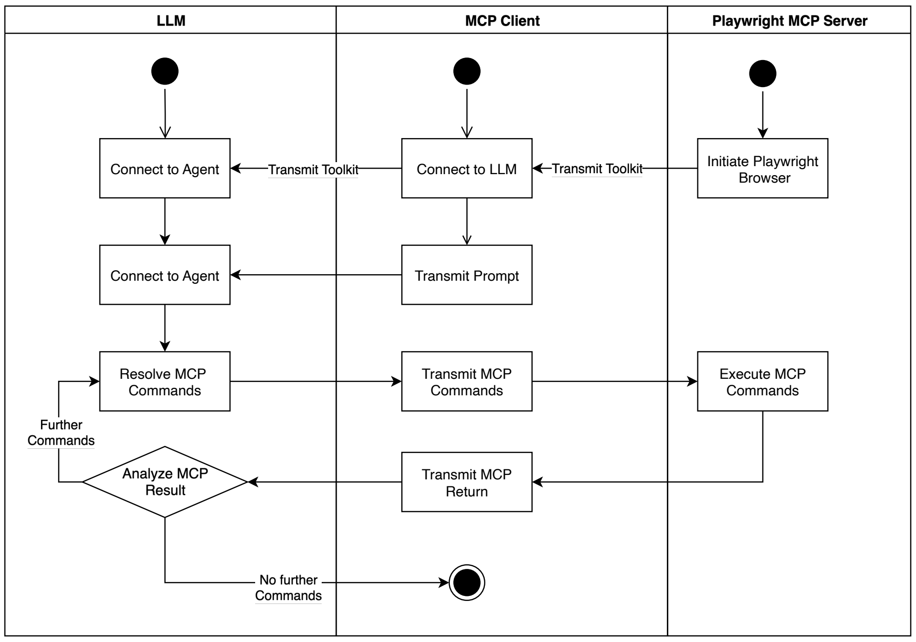


Figure 13: Activity Diagram of standalone MCP Client Connection

As seen in Figure 13, we individually start the LLM, the MCP Client, and the Playwright MCP Server. The MCP Server initiates the Playwright Browser and waits for the MCP Client to connect. The MCP Client then connects to the MCP Server and sends the Toolkit provided by the MCP Server and the Prompt to the LLM. The LLM then resolves the MCP Commands and sends a message to the

MCP Client, which then sends the message to the MCP Server. The MCP Server then executes the Playwright Commands, such as navigating to a specific page or clicking on a specific element. The MCP Server returns the result to the MCP Client, which then returns the result to the LLM. The LLM then analyzes the result and either returns the result to the MCP Client or further commands to the MCP Server.

5 Test Results

This thesis contributes to the field of automated testing and performance evaluation of browser-based IDEs by developing a testing framework for Theia Cloud and evaluating its performance under load. Developers and administrators of Theia Cloud and Artemis can especially benefit from this work, as it provides a way to test the functionality and performance of Theia Cloud and Artemis under load.

As several research topics regarding the optimization of the performance of Theia Cloud and Artemis are still ongoing, this thesis does not provide a comprehensive evaluation of the performance of Theia Cloud and Artemis. Instead, it provides a framework for evaluating the performance of Theia Cloud and Artemis under load.

5.1 Implementation Challenges

Developing a system for automated end-to-end and load testing in the context of an Online IDE involves several unique challenges that extend beyond conventional software testing. Unlike traditional desktop or standalone applications, the Online IDE is a Software-as-a-Service platform, which imposes restrictions on direct system access, lifecycle control, and integration with external tools. These limitations complicate the design of reliable testing workflows, as the testing framework must operate within the boundaries of a managed service environment [THS11].

Furthermore, the inherent complexity of the system introduces additional difficulties. An Online IDE integrates many tightly coupled components, including code editors, terminals, language servers, version control systems, and backend evaluation pipelines. Each component is highly interactive and stateful, making it harder to automate and verify through testing. Ensuring that tests are stable, reproducible, and representative of real user behavior is non-trivial when asynchronous interactions, long-lived sessions, and concurrent usage patterns are involved.

This section discusses the most significant challenges encountered during the implementation. It highlights how the SaaS delivery model's characteristics and the Online IDE's distributed architecture influence the testing approach, and how we addressed these to enable meaningful evaluation of system functionality and performance.

5.1.1 Scorpio Testing

One of this thesis's goals is to include the Scorpio extension in the workflow of the Artemis integration tests, because the Scorpio extension is a crucial part of the Artemis integration and allows the user to interact with Artemis via a browser-based IDE [Jan24]. Features include the ability to clone repositories, automatically detect already existing repositories, show the exercise description and test result, and submit the solution to Artemis.

As Scorpio utilizes Webviews of the Visual Studio Code Extension API[17] to render custom user interfaces required for the exercise description and test result, it is necessary to capture the webview inside of a `iFrame`[18].

As every webview runs under a custom subdomain in the `*.webview.` domain space[19], we must consider the restrictions of CORS (Cross-Origin Resource Sharing). The `iFrame` has the sandbox attribute set to `allow-same-origin` to allow the `iFrame` to access the parent view's resources. Unfortunately, this makes it impossible to access the `iFrame` from the parent view, as the `iFrame` is not part of the same origin.

```
<iframe sandbox="allow-scripts allow-forms allow-same-origin allow-
downloads">
```

One of the possible solutions was to dynamically remove the sandbox attribute from the `iFrame` after the `iFrame` has loaded. Due to the nature of removing attributes at runtime, the actual change is not reflected on already loaded pages.[20]

Removing the CORS restrictions in Playwright's browser settings did not achieve the desired result. Therefore, we made no further attempts to include the

[17]https://code.visualstudio.com/api/extension-guides/webview
[18]https://developer.mozilla.org/de/docs/Web/HTML/Reference/Elements/iframe
[19]https://theia-cloud.io/documentation/addapplication/
[20]https://html.spec.whatwg.org/multipage/iframe-embed-object.html#attr-iframe-sandbox

Scorpio extension in the workflow of the Artemis integration tests, as the priority of this thesis was to develop a testing framework for Theia Cloud.

5.1.2 Testing Local And Production System

To cross-check the functionality of the testing framework, the tests can test against the running system as well as against a local system. The local system only includes a single Theia Session running in a local Docker container.

Because of several differences between the local and production system, we encountered challenges when running the same tests against both systems.

These differences include:

- Handling of workspaces: The local system utilizes workspaces in the file system of the local Docker container. Therefore, tests can create temporary directories for each workspace to separate test workflows. The production system does not utilize workspaces similarly, as the user can only access the pre-made workspace in the container under the `/home/project` directory.
- Landing page: As we only utilize a single Theia Session in the local system, there is no need to log in or rely on the landing page. This differentiation makes it more challenging to distinguish between the test setup of the local system and the production systems' test setup.
- Different CSS locators: During the development of the testing framework, we noticed that the CSS locators of the production system were not equal to the CSS locators of the local system. That is because the production system utilizes a different version of the Theia IDE and has a different DOM structure. Updating the version of the Theia IDE in the production system solved this issue.

5.1.3 Parallel Testing In One Session

The priority of the testing framework is to test a production system running on a cluster with scaling capabilities. As we do not want to start a new instance for each test, we run tests in parallel in one session.

During development, we noticed that the tests running in parallel interfered with each other. The tests that created new files or directories interfered with each other because they often created files simultaneously when another test opened a file using the explorer view, which resulted in the IDE updating the file list. Therefore, the tests could not find the corresponding file nodes.

To solve this issue, we refactored the tests to create a temporary workspace for each test file in the root workspace. Tests in a single test file run sequentially, avoiding interference from the other tests.

5.1.4 Running Tests Using A Single Account

Testing the scalability of the system is a crucial part of this thesis. In the production system, we utilize a single account for all tests. We cannot create new users for each test because sessions are bound to user accounts provided by `Keycloak`[21]. Therefore, we have to use the same test account for all tests. While running the load tests on the cluster, we noticed a session limit for each user account. Therefore, we were not able to run the load tests with more than 10 instances at the same time.

5.2 Test Infrastructure and Execution Strategy Limitations

The developed test suite relies on two separate testing environments: the Artemis Test Server and a dedicated Test Server for Theia Cloud. These environments provide the necessary backend and frontend infrastructure to execute automated end-to-end and load testing scenarios under realistic conditions.

5.2.1 Integration Limitations in the Artemis E2E Pipeline

During implementation, it was not feasible to integrate the Theia-related tests directly into the Artemis end-to-end testing pipeline. The primary reason for this limitation is that the Artemis testing environment operates in an isolated context, without access to external services not deployed within the same containerized setup. Since Theia Cloud is managed and deployed independently, we could not establish direct communication between Artemis and Theia within the pipeline.

[21]https://www.keycloak.org/

An additional constraint arises from deploying an extra Theia instance inside the Artemis CI environment, requiring significant resources and configuration overhead. This setup would complicate the testing workflow and increase the maintenance burden, without substantial benefit for the current development focus. Therefore, this approach was deemed infeasible.

5.2.2 Use of External Test Servers

As a result, all tests requiring integration between Artemis and Theia Cloud run against the respective systems' externally deployed or self-hosted instances. We thus configured the test suite to connect to one of the available Artemis test servers and the designated Theia Cloud test server. This setup mirrors the production environment and ensures that tests validate realistic system behavior, including network latency, service communication, and authentication processes.

5.2.3 Pipeline Execution and Manual Testing

Due to the nature of the load tests, we intentionally decided not to run them automatically in the CI/CD pipeline. Load tests can generate high system utilization and may interfere with ongoing development or active system usage. Instead, developers execute such tests manually when evaluating performance or verifying changes that may impact scalability.

We are not significantly modifying client-facing features or the user interface at the project's current stage. Therefore, performing manual validation combined with targeted redeployments provides a more efficient and controlled testing workflow than running automated tests on every pull request.

5.2.4 MCP Testing Status

Model Context Protocol-based testing integration is still experimental and not yet included in the automated test pipelines [Wan+24]. The current MCP implementation requires connectivity to Anthropic's Claude[22] models and other specialized infrastructure components unavailable in the cloud-based testing environment. Consequently, automated execution and continuous deployment of these tests are

[22]https://www.anthropic.com/

not yet possible. Future work will focus on stabilizing the MCP testing components and integrating them into a dedicated pipeline once the necessary infrastructure becomes available.

6 Summary

This chapter summarizes the development of the testing framework for Theia Cloud. We discuss the system's status and functionality as of this thesis's writing and provide an overview of the thesis's achievements and remaining challenges. Finally, we discuss potential future work to enhance the system's functionality.

6.1 Status

This section provides an overview of the status of the functional requirements as of the writing of this thesis, as presented in Section 3.3.1.

Status	
FR1 Simulate User Interactions	●
FR2 Automated End-to-End Testing	●
FR3 Load and Performance Testing	●
FR4 Monitoring and Reporting	◐
FR5 Scenario Management	●
FR6 Integration with Intelligent Agents	◐

Table 1: Status of the testing framework

● Implemented - ◐ Partially implemented

1. **Design of a Test Architecture for Cloud-Based IDEs**

   We designed a modular test architecture to enable automated end-to-end and load testing of online IDEs integrated into complex systems such as Artemis. The design separates the test logic from the system under test, allowing flexible deployment against self-hosted or external environments.

2. **Implementation of a Realistic End-to-End Test Suite**

   We developed a test suite to simulate authentic student workflows, including file editing, compilation, and terminal interaction. These tests validate the IDE's functional correctness and responsiveness, providing a foundation for future regression testing.

3. **Scalable Load Testing Setup**

   The project introduced a load testing framework that reproduces high-concurrency scenarios to evaluate system performance under stress. By intentionally decoupling load testing from continuous integration pipelines, the framework ensures safe execution without affecting live or shared environments.

4. **Integration Strategy Between Artemis and Theia Cloud**

   We developed a hybrid testing strategy to handle the limited interoperability between Artemis' CI pipelines and external Theia instances. The configuration runs tests against deployed test servers, enabling realistic cross-system validation while preserving a maintainable workflow.

5. **Exploration of Intelligent MCP-Based Testing**

   The thesis explored the integration of Model Context Protocol (MCP) agents as a future-oriented approach to simulate adaptive and autonomous user behavior. Although not yet deployable in automated pipelines, this represents an innovative direction for creating more dynamic and human-like test interactions.

6.2 Conclusion

This thesis sets out to develop and evaluate a systematic approach to testing the functionality and performance of an Online Integrated Development Environment used in an educational SaaS context. The main objective was to establish a reliable foundation for automated end-to-end and load testing that can realistically reproduce user behavior and reveal potential scalability limitations.

The design and implementation of a dedicated testing framework demonstrated that comprehensive validation of an Online IDE is achievable even with restricted system access and distributed service architectures. The resulting test environment provides a practical means to assess correctness and performance under real-world conditions, bridging the gap between controlled testing and production-like scenarios.

The work further emphasized balancing automation and manual evaluation when operating within a managed infrastructure. By decoupling heavy load tests from continuous integration pipelines, the testing approach ensures both safety and flexibility for developers.

Finally, this thesis laid the groundwork for future advancements in intelligent, autonomous testing by exploring Model Context Protocol (MCP)–based agents. While still experimental, such approaches represent a promising direction toward more adaptive and human-like test behavior in complex online learning environments.

Overall, this work improves cloud-based IDE platforms' reliability, scalability, and maintainability, supporting their continued use in large-scale educational settings.

6.3 Future Work

This section discusses potential future work to enhance the system's functionality, especially the integration of the MCP testing framework into the Artemis E2E pipeline and the integration of the MCP testing framework into the Artemis E2E pipeline.

While this thesis establishes a solid foundation for automated end-to-end and load testing of cloud-based IDEs, several directions remain for future research and development. Each proposed area represents an opportunity for a dedicated follow-up project or thesis to extend and refine the system's capabilities.

6.3.1 Automated Pipeline Execution

A significant next step is integrating the testing framework into a fully automated continuous integration and deployment (CI/CD) pipeline. Such integration would enable the execution of end-to-end and regression tests automatically upon new code submissions or deployments, thereby ensuring that functional and performance regressions are detected early in the development cycle.

Future work could focus on designing a scalable and resource-aware pipeline architecture that allows automated test execution without interfering with production environments. This would involve dynamic environment provisioning (e.g., ephemeral test instances for Theia Cloud and Artemis) and result aggregation for trend analysis over multiple runs. A comparative evaluation across different execution platforms, such as various browsers, operating systems, and network conditions, could reveal platform-specific weaknesses and help optimize the testing setup for educational contexts. Such an effort would significantly increase confidence in each release and provide developers with actionable performance metrics directly integrated into their workflow.

6.3.2 Automated MCP Testing

Integrating Model Context Protocol-based testing into the automated testing pipeline represents another promising direction. MCP introduces the ability to test user interfaces through autonomous agents driven by large language models, allowing the simulation of realistic and adaptive user behavior rather than strictly scripted interactions.

Future research could aim to stabilize and generalize the MCP integration, addressing the current technical limitations, such as unreliable text input, code formatting inconsistencies, and dependency on external model providers (e.g., Anthropic Claude or OpenAI models). Once stabilized, this setup could be connected to a dedicated CI pipeline to automatically validate new IDE versions using LLM-driven agents. A subsequent thesis could evaluate how well LLMs can learn to interact with development environments, how consistent their behavior is across different prompts, and how effectively they can detect UI or workflow regressions without explicit scripting.

6.3.3 LLM Testing for Artemis

Beyond testing the IDE, we could extend the LLM-based testing framework to automatically test the Artemis learning platform. Let LLM agents generate and execute test cases for Artemis features such as exercise creation, submission handling, or grading workflows. This would represent a significant leap toward autonomous system-level testing.

Future work could focus on integrating LLM-based behavior generation with Artemis' REST API to automatically construct diverse, high-coverage test scenarios. In addition to correctness verification, these tests could include load and stress evaluations, providing valuable data on system performance under real-world conditions. Such a project could also explore the potential for AI-assisted security testing, where LLMs attempt to identify potential vulnerabilities by generating adversarial inputs or edge cases, an area with strong research potential in educational platforms.

6.3.4 Further Test Cases

Although the current test suite covers a broad set of IDE functionalities, Online IDEs are continuously evolving systems with growing feature sets. Future work should therefore focus on systematically extending test coverage, especially for features that involve collaborative editing, real-time synchronization, or advanced debugging capabilities.

A potential follow-up thesis could focus on creating a comprehensive test generation framework, leveraging LLMs or model-based testing techniques to derive test cases directly from use-case descriptions or system models. This would improve coverage and reduce the maintenance effort for large test suites. Additionally, comparative studies could be conducted to assess the effectiveness and stability of these automatically generated tests against traditional scripted tests, contributing valuable insights to software testing automation.

List of Figures

List of Tables

Appendix A: Supplementary Material

```yaml
name: Artemis Integration Tests
on:
  push:
    branches: [ main, master ]
  pull_request:
    branches: [ main, master ]
jobs:
  test:
    timeout-minutes: 60
    runs-on: [self-hosted, e2e-test]
    env:
      KEYCLOAK_USER: ${{ secrets.KEYCLOAK_USER }}
      KEYCLOAK_PWD: ${{ secrets.KEYCLOAK_PWD }}
      LANDINGPAGE_URL: ${{ vars.LANDINGPAGE_URL }}
      ARTEMIS_URL: ${{ vars.ARTEMIS_URL }}
      ARTEMIS_USER: ${{ secrets.ARTEMIS_USER }}
      ARTEMIS_PWD: ${{ secrets.ARTEMIS_PWD }}
      NUM_INSTANCES: 10

    steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: lts/*
    - name: Install dependencies
      run: npm ci
    - name: Install Playwright Browsers
      run: npx playwright install --with-deps
    - name: Run Playwright tests
      run: npx playwright test --project=artemis --workers=1
    - uses: actions/upload-artifact@v4
      if: ${{ !cancelled() }}
      with:
        name: playwright-report
        path: playwright-report/
        retention-days: 30
```

Listing 1: Example of a GitHub Actions workflow for Artemis Integration Tests

```
- To run the functional tests, run:
 npx playwright test --project=local

 - To run the load tests, run:
 npx playwright test --project=scale

 - To run the Artemis integration tests, run:
 npx playwright test --project=artemis

 - Set the amount of instances in the ENV file or pass it like this:
 NUM_INSTANCE=100 npx playwright test --project=scale

 - To run the tests using the Artillery.io framework, run:
 npx artillery run tests/ide/scalable/artillery/Artillery.ts
```

Listing 2: Execution of different playwright projects

Editor Tests
Create new File
Create File with content
Undo and Redo text in file
Delete file
Create new File in folder
Test copy from String
Search Tests
Search for text in the editor
Search for text using menu bar
Search for text using sidebar
Search for text using sidebar multiple files
Terminal Tests
Open Terminal
Terminal command: ls
Terminal command: touch
Terminal command: rm
Terminal multiple tabs
Version Control Tests
Commit
Push

Table 2: Newly developed functional tests

C App Image
C modules installed
Create C file
Compile C file
Run C file
Java App Image
Java modules installed
Create Java file
Compile Java file
Run Java file
JavaScript App Image
JavaScript modules installed
Create JavaScript file
Run JavaScript file
Ocaml App Image
Ocaml modules installed
Create Ocaml file
Compile Ocaml file
Run Ocaml file
Python App Image
Python modules installed
Create Python file
Compile Python file
Run Python file
Rust App Image
Rust modules installed
Create Rust file
Run Rust file
Compile Rust file
Run Rust file

Table 3: Newly developed programming tests for the App Images

Landing Page Tests (unauthenticated)
Login button should be visible
Login should redirect to Keycloak
Login via the UI
Landing Page Tests (authenticated)
User should be logged in
Programming language instances should be visible
Launch C instance

Table 4: Newly developed landing page tests

Artemis Integration Tests
Theia IDE loads from Artemis
Creation of course and exercise is possible
Repository is cloned
Student submits code
check result on Artemis

Table 5: Newly developed Artemis integration tests

Load Tests Scenarios
editBubbleSort
editMergeSort
editClient
editContext
editPolicy
editSortStrategy
commit
createNewRandomFile
createAndEditRandomFile
useTerminal
runTests
searchForWords
changePreferences
openAboutPage
reloadPage
buildAndRun

Table 6: All scenarios accessed by the load testing framework

Bibliography

[KS18] S. Krusche and A. Seitz, “Artemis: An Automatic Assessment Management System for Interactive Learning,” in *Proceedings of the 49th Technical Symposium on Computer Science Education (SIGCSE)*, ACM, 2018, pp. 284–289. doi: 10.1145/3159450.3159602.

[Jan24] D. Jandow, “Scorpio: A Visual Studio Code Extension for Interactive Learning Platforms,” *Bachelor Thesis at TUM*, Nov. 2024.

[Sch24] Y. Schmidt, “Inclusive Learning Environments in the Cloud: Scalable Online IDEs for Higher Education,” *Master Thesis at TUM*, Dec. 2024.

[Pau01] R. Paul, “End-to-End Integration Testing,” in *Proceedings Second Asia-Pacific Conference on Quality Software*, Hong Kong, China: IEEE Comput. Soc, 2001, pp. 211–220. doi: 10.1109/APAQS.2001.990022.

[Jan25] R. Jandow, “From Desktop to Web: Development of an Advanced Browser-based IDE for Education,” *Master Thesis at TUM*, Feb. 2025.

[SB10] J. R. Stowell and D. Bennett, “Effects of Online Testing on Student Exam Performance and Test Anxiety,” *Journal of Educational Computing Research*, vol. 42, no. 2, pp. 161–171, Mar. 2010, doi: 10.2190/EC.42.2.b.

[Luc71] H. Lucas, “Performance Evaluation and Monitoring,” *ACM Computing Surveys*, vol. 3, no. 3, pp. 79–91, Sept. 1971, doi: 10.1145/356589.356590.

[HOC17] C. D. Hundhausen, D. M. Olivares, and A. S. Carter, “IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda,” *ACM Transactions on Computing Education*, vol. 17, no. 3, pp. 1–26, Sept. 2017, doi: 10.1145/3105759.

[Mar+11] B. Marin, T. Vos, G. Giachetti, A. Baars, and P. Tonella, “Towards Testing Future Web Applications,” in *2011 FIFTH INTERNATIONAL CONFERENCE ON RESEARCH CHALLENGES IN*

INFORMATION SCIENCE, Gosier, France: IEEE, May 2011, pp. 1–12. doi: 10.1109/RCIS.2011.6006859.

[THS11] W.-T. Tsai, Y. Huang, and Q. Shao, "Testing the Scalability of SaaS Applications," in *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, Irvine, CA, USA: IEEE, Dec. 2011, pp. 1–4. doi: 10.1109/SOCA.2011.6166245.

[HN25] A. Halilaj and V. Norlin, "A Comparative Evaluation of End-to-End Testing Frameworks for Cross-Browser Web Applications: Cypress vs Playwright," 2025.

[Lor21] M. Lorinc, "Extension of the Eclipse Che Editor for UI Testing Module," *Bachelor Thesis at VUT*, 2021.

[Tal24] M. Talibov, "Improving Stability of Web Applications by Optimized E2E Testing," *Master Thesis at TUM*, June 2024.

[BD04] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, 2. ed., internat. ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2004.

[KK23] I. V. Krasnokutska and O. S. Krasnokutskyi, "Implementing E2E Tests with Cypress and Page Object Model: Evolution of Approaches," 2023.

[Wan+24] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing With Large Language Models: Survey, Landscape, and Vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, Apr. 2024, doi: 10.1109/TSE.2024.3368208.