Quizzes Tutor
Software Architecture Document (SAD)

Leonardo Monteiro Nº58250
Henrique Vale Nº58168
Gustavo Henriques Nº64361

Table 1: SAD update history

DOCUMENT NUMBER	RELEASE/ REVISION DATE	RELEASE/REVISION
1.0	28/10/2024	**Purpose and Scope creation:** Defined *Purpose and Scope*.
2.0	30/10/2024	**Stakeholder and Viewpoint Definition:** Added *Stakeholder Representation* and initiate the development of *Viewpoint Definitions*.
2.1	31/10/2024	**Viewpoint 1:** Continue to work on the Viewpoint section.
2.2	01/11/2024	**Viewpoint 2:** Continue to work on the Viewpoints section.
2.3	02/11/2024	**Viewpoint 3:** Continue to work on the Viewpoints section.
2.4	03/11/2024	**Viewpoint 4:** Ended and revised the Viewpoints section.
3.0	06/11/2024	**Architecture Background:** Completed *Architecture Background* (*Problem Background*, *Goals and Context*, *Driving Requirements*).
4.0	08/11/2024	**Solution Background:** Began drafting Solution Background (Architectural Approaches and initial Analysis Results).
5.0	09/11/2024	**Analysis Results:** Understanding how to use SonarQube and start to analyze the results given.
5.1	10/11/2024	**Analysis Results 1:** Start to write in this document the analysis made in the SonarQube platform.
5.2	11/11/2024	**Analysis Results 2:** Ended and revised the analysis results section.
6.0	13/11/2024	**Requirements coverage and summary of background:** Completed requirements coverage and summary of background.
7.0	15/11/2024	**Module Views 1:** Documented *Module Views* (*Decomposition*).
7.0.1	16/11/2024	**Module Views 2:** Documented Module Views (Uses and Data Model).
7.1	17/11/2024	**Component-and-Connector Views:** Documented *Component-and-Connector Views* (*Call-Return*, *Repository*).
7.2	21/11/2024	**Allocation Views:** Documented *Allocation Views* (*Execution*).
7.2.1	22/11/2024	**Allocation Views:** Documented *Allocation Views* (*Development*).
7.3	25/11/2024	**Revisions and Relation Among Views:** Reviewed and refined all *Views* (*Module*, *Component-and-Connector*, and *Allocation*) and added *Relations Among Views*.

8.0	26/11/2024	**Relation Among Views, Referenced Materials and Glossary:** Finalized *Relations Among Views* and completed *Referenced Materials* and *Glossary*.
9.0	28/11/2024	**Final Revisions 1:** Conducted a full review of the project, ensuring all sections are aligned and coherent. Addressed outstanding issues and incorporated necessary adjustments.
9.1	29/11/2024	**Final Revision 2:** Finalized the review on all the sections of this project.

Table of Contents

List of Figures

List of Tables

1 Documentation Roadmap

The Documentation Roadmap should be the first place a new reader of the SAD begins. But for new and returning readers, it is intended to describe how the SAD is organized so that a reader with specific interests who does not wish to read the SAD cover-to-cover can find desired information quickly and directly.

Sub-sections of Section 1 include the following.

- **Section 1.1 (“Document Management and Configuration Control Information”)** explains revision history. This tells you if you’re looking at the correct version of the SAD.

- **Section 1.2 (“Purpose and Scope of the SAD”)** explains the purpose and scope of the SAD, and indicates what information is and is not included. This tells you if the information you’re seeking is likely to be in this document.

- **Section 1.3 (“How the SAD Is Organized”)** explains the information that is found in each section of the SAD. This tells you what section(s) in this SAD are most likely to contain the information you seek.

- **Section 1.4 (“Stakeholder Representation”)** explains the stakeholders for which the SAD has been particularly aimed. This tells you how you might use the SAD to do your job.

- **Section 1.5 (“Viewpoint Definitions”)** explains the *viewpoints* (as defined by IEEE Standard 1471-2000) used in this SAD. For each viewpoint defined in Section 1.5, there is a corresponding view defined in Section 3 (“Views”). This tells you how the architectural information has been partitioned, and what views are most likely to contain the information you seek.

- **Section 1.6 (“How a View is Documented”)** explains the standard organization used to document architectural views in this SAD. This tells you what section within a view you should read in order to find the information you seek

- **Section 1.7 ("Relationship to Other SADs")** notes whether this SAD is related to other architecture documents. If no related documents exist, it simply states "Not applicable," clarifying whether additional architecture documents should be consulted.

- **Section 1.8 (“Process for Updating this SAD”)** describes how to report issues or inaccuracies in the SAD, providing contact details and steps for handling feedback to maintain document accuracy.

1.1 Document Management and Configuration Control Information

- Revision Number: 7.0
- Revision Release Date: 28/11/2024
- Purpose of Revision: Final review of the document and implementation of minor necessary adjustments after the project evaluation.
- Scope of Revision: Performed a final assessment of all sections, ensuring readability, accuracy, and alignment with project objectives. Minor corrections and refinements were applied to improve clarity and coherence.

1.2 Purpose and Scope of the SAD

1.2.1 Purpose

The purpose of this Software Architecture Document (SAD) is to comprehensively document the architecture of Quizzes Tutor, an open-source platform for creating, managing, and evaluating educational quizzes. This SAD could be helpful for the system's development, analysis, and maintenance, supporting effective stakeholder communication and ensuring a shared understanding of its design.

1.2.2 Scope

This SAD outlines key architectural aspects of Quizzes Tutor, including:

Stakeholders and Their Interests: Identification of main stakeholders (teachers, students, development team, system administrators and project manager), highlighting their specific needs and concerns to guide architectural decisions.

Quality Requirements: Definition of key quality attributes such as performance, security, scalability, and maintainability, supported by scenarios illustrating expected system behavior and performance benchmarks.

Architectural Views:

- **Module View:**
  - *Decomposition View*: Describes the hierarchy of modules and submodules, specifying their distinct responsibilities.

    - *Uses View*: Highlights functional dependencies between modules, showing key interactions.
    - *Data Model View*: Illustrates relationships between data entities to ensure consistency in data storage.
- **Component-and-Connector View:**
    - *Call-Return*: Focuses on control flow and function interactions between components.
    - *Repository*: Describes shared data storage and access mechanisms used by the system.
- **Allocation View:**
    - *Deployment View*: Specifies how the software components are mapped onto the underlying hardware or virtual infrastructure.

1.3 How the SAD Is Organized

This SAD is organized into the following sections:

- **Section 1 ("Documentation Roadmap") provides information about this document and its intended audience**. It provides the roadmap and document overview. Every reader who wishes to find information relevant to the software architecture described in this document should begin by reading Section 1, which describes how the document is organized, which stakeholder viewpoints are represented, how stakeholders are expected to use it, and where information may be found. Section 1 also provides information about the views that are used by this SAD to communicate the software architecture.
- **Section 2 ("Architecture Background") explains why the architecture is what it is.** It provides a system overview, establishing the context and goals for the development. It describes the background and rationale for the software architecture. It explains the constraints and influences that led to the current architecture, and it describes the major architectural approaches that have been utilized in the architecture. It includes information about evaluation or validation performed on the architecture to provide assurance it meets its goals.
- **Section 3 (Views") and Section 4 ("Relations Among Views") specify the software architecture**. Views specify elements of software and the relationships between them. A view corresponds to a viewpoint (see Section 1.5), and is a representation of one or more structures present in the software (see Section 1.2).
- **Sections 5 ("Referenced Materials") and 6 ("Director") provide reference information for the reader.** Section 5 provides look-up information for documents that are cited elsewhere in this SAD. Section 6 is a *directory*, which is an index of architectural elements and relations telling where each one is defined and used in this SAD. The section also includes a glossary and acronym list.

1.4 Stakeholder Representation

This section provides an overview of the primary stakeholders involved in the development and use of the Quizzes Tutor system, along with their main concerns regarding the system's architecture.

1. **Teachers**
    - **Role**: Creators and managers of quizzes.
    - **Main Concerns**:
        - Usability: An intuitive interface for creating and managing quizzes.
        - Reliability: System stability during usage to ensure smooth educational activities.
        - Performance Reporting: Access to reports tracking student progress and results.
2. **Students**
    - **Role**: End users who take quizzes.
    - **Main Concerns**:
        - Availability: Reliable access to the system at all times.
        - User Experience: A simple, user-friendly interface for completing quizzes.
        - Responsiveness: Fast response time, especially when submitting answers.
3. **Development Team**
    - **Role**: Engineers and developers responsible for system maintenance and expansion.
    - **Main Concerns**:
        - Modularity: An architecture that supports easy maintenance and future feature additions.
        - Scalability: The ability to handle an increase in users if necessary.
        - Documentation: Clear documentation to support ongoing development.
4. **System Administrators**
    - **Role**: Responsible for system deployment, stability, and monitoring.
    - **Main Concerns**:
        - Stability and Monitoring: Capability to monitor and resolve issues quickly.
        - Security: Protecting against unauthorized access and ensuring data integrity.
        - Resource Efficiency: Efficient management of server and storage resources.
5. **Project Manager**
    - **Role**: Supervisor ensuring project goals, timeline, and budget.
    - **Main Concerns**:

- Schedule and Budget: Ensuring that the system is developed on time and within budget constraints.
- Team Coordination: Facilitating collaboration among developers, designers, and stakeholders.
- Architectural Flexibility: Ensuring that the architecture can adapt to changes or new requirements.

1.5 Viewpoint Definitions

The SAD employs a stakeholder-focused, multiple view approach to architecture documentation, as required by ANSI/IEEE 1471-2000, the recommended best practice for documenting the architecture of software-intensive systems [IEEE 1471].

As described in Section 1.2, a software architecture comprises more than one software structure, each of which provides an engineering handle on different system qualities. A *view* is the specification of one or more of these structures, and documenting a software architecture, then, is a matter of documenting the relevant views and then documenting information that applies to more than one view [Clements 2002].

ANSI/IEEE 1471-2000 provides guidance for choosing the best set of views to document, by bringing stakeholder interests to bear. It prescribes defining a set of viewpoints to satisfy the stakeholder community. A viewpoint identifies the set of concerns to be addressed, and identifies the modeling techniques, evaluation techniques, consistency checking techniques, etc., used by any conforming view. A view, then, is a viewpoint applied to a system. It is a representation of a set of software elements, their properties, and the relationships among them that conform to a defining viewpoint. Together, the chosen set of views show the entire architecture and all of its relevant properties. A SAD contains the viewpoints, relevant views, and information that applies to more than one view to give a holistic description of the system.

For the Quizzes Tutor system, we have identified three primary viewpoints to capture essential system qualities and address the specific needs of stakeholders. Within each viewpoint, we have also defined key subtypes that we consider critical for understanding the architecture.

- **Module Viewpoint**: Focuses on the static structure, dividing the system into modules, to provide a clear understanding of its organization.

  Decomposition View: Displays the hierarchy of modules and submodules, showing how each part of the system has a specific responsibility.
  Uses View: Highlights functional dependencies between modules, illustrating essential interactions for system functionality.

Data Model View: Describes relationships between data entities, ensuring a clear and consistent structure for information storage.

- **Component-and-Connector Viewpoint**: Captures the runtime interactions between components, showcasing data flow and communication during execution.

  Call-Return: Components receive control and data from others, executing a function and then returning control to the invoker.
  Repository: Large stores of persistent data managed centrally or across several databases, enabling shared access to data.

- **Allocation Viewpoint**: Maps software elements to their physical or virtual infrastructure necessary for development and operation.

  Deployment View: Describes how the software components are mapped to physical or virtual nodes in the infrastructure, detailing the deployment of the system across servers, cloud platforms, or on-premises environments.

The following table summarizes the stakeholders in the Quizzes Tutor project and the viewpoints selected to address their specific concerns:

Table 2: Stakeholders and Relevant Viewpoints

Stakeholder	Viewpoint(s) that apply to that class of stakeholder's concerns
Teachers	Module Viewpoint: Decomposition View
Students	Component-and-Connector Viewpoint: Call-Return View
Development Team	Module Viewpoint: Decomposition View, Uses View, Data Model View Component-and-Connector Viewpoint: Call-Return View, Repository View Allocation View: Deployment View

Stakeholder	Viewpoint(s) that apply to that class of stakeholder's concerns
System Administrators	Component-and-Connector Viewpoint: Repository Module Viewpoint: Data Model View Allocation View: Deployment View
Project Manager	Module Viewpoint: Decomposition View

1.5.1 Module Viewpoint Definition

1.5.1.1 Decomposition Viewpoint Definition

1.5.1.1.1 Abstract

The Decomposition View displays the system's hierarchy of modules and submodules, dividing it into distinct parts, each with specific responsibilities, to organize and structure the system.

1.5.1.1.2 Stakeholders and Their Concerns Addressed

Project Managers: Need a clear view of the modules to plan tasks and organize the team's work.
Development Team: Benefits from a clear modular structure to facilitate incremental development and maintenance.
Teachers: Require an understanding of the main modular functionalities to support their use of the system.

1.5.1.1.3 Elements, Relations, Properties and Constraints

Elements: Modules, submodules, and their specific functionalities.
Relations: "Is-part-of" relation representing the hierarchy and modular structure of the system.
Properties: Each module's name, main function, and software interface.
Constraints: Each module should be responsible for a distinct function and not overlap responsibilities with other modules.

1.5.1.1.4 Language(s) to Model/Represent Conforming Views

UML Class Diagrams or **Component Diagrams** to illustrate hierarchical structure and relations of each module.

1.5.1.1.5 Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria

Consistency: Each element should have only one parent module.
Completeness: The functionality of all modules combined should cover the system requirements.
Evaluation Techniques: Scenario-based analysis to ensure modularity allows changes with minimal impact.

1.5.1.1.6 Viewpoint Source

Clements, P., et al. *Documenting Software Architectures: Views and Beyond* (2002).

1.5.1.2 Uses Viewpoint Definition

1.5.1.2.1 Abstract

The *Uses View* highlights the functional dependencies between modules, illustrating how the functionality of one module depends on another, helping identify key system interactions.

1.5.1.2.2 Stakeholders and Their Concerns Addressed

Development Team: Needs to understand module dependencies to avoid integration issues.

1.5.1.2.3 Elements, Relations, Properties and Constraints

Elements: Modules with specific functions and functional dependencies.
Relations: “Uses” relations defining execution dependencies between modules.
Properties: Each module’s name, function, and associated dependencies.
Constraints: Dependent modules must be compatible in terms of interface and communication.

1.5.1.2.4 Language(s) to Model/Represent Conforming Views

UML Dependency Diagrams or **Sequence Diagrams** to show interactions and functional dependencies.

1.5.1.2.5 Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria

Consistency: All listed dependencies should be valid and documented.
Completeness: All critical interactions for functionality should be represented.
Evaluation Techniques: Review dependencies to identify potential coupling issues.

1.5.1.2.6 Viewpoint Source

Clements, P., et al. *Documenting Software Architectures: Views and Beyond* (2002).

1.5.1.3 Data Viewpoint Definition

1.5.1.3.1 Abstract

The *Data Model View* shows relationships between the system's data entities, providing a clear and consistent structure for storing and managing information.

1.5.1.3.2 Stakeholders and Their Concerns Addressed

Development Team: Requires a clear data structure to correctly implement and integrate storage and retrieval operations.
System Administrators: Concerned with data integrity and security to ensure proper system functioning.

1.5.1.3.3 Elements, Relations, Properties and Constraints

Elements: Data entities, such as tables and columns in the database.
Relations: "Association" and "dependency" relations between entities (e.g., foreign key).
Properties: Each entity's attributes, including data types and constraints.
Constraints: All relationships must maintain referential integrity and data consistency.

1.5.1.3.4 Language(s) to Model/Represent Conforming Views

Entity-Relationship Diagrams (ERD) or **UML Class Diagrams** to represent the data structure and relationships between entities.

1.5.1.3.5 Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria

Consistency: All data relationships should be consistent and follow integrity constraints.

Completeness: All necessary entities and relationships for data storage should be represented.
Evaluation Techniques: Referential integrity check and normalization analysis to avoid data redundancy.

1.5.1.3.6 Viewpoint Source

Clements, P., et al. *Documenting Software Architectures: Views and Beyond* (2002).

1.5.2 Component-and-Connector Viewpoint Definition

1.5.2.1 Call-Return Viewpoint Definition

1.5.2.1.1 Abstract

The *Call-Return View* models interactions in which components receive control and data from others, execute a function, and then return control to the invoker.

1.5.2.1.2 Stakeholders and Their Concerns Addressed

Development Team: Requires a clear understanding of component interactions and control flow to develop, debug, and optimize system performance.
Students: Interested in understanding how system components work together, which enhances their user experience during runtime.

1.5.2.1.3 Elements, Relations, Properties and Constraints

Elements: Components that act as invokers or receivers in control flows, such as clients and servers.
Relations: “Calls” relation, where an invoker component makes a call to another, transferring control and possibly data.
Properties: Each component’s role (invoker or receiver), and data involved in the call.
Constraints: Each call must follow a compatible interface for control and data exchange, and components should maintain state consistency.

1.5.2.1.4 Language(s) to Model/Represent Conforming Views

Sequence/Activity/Interaction Diagrams to show the sequence and flow of control and data exchange among components.

1.5.2.1.5 Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria

Consistency: All control flows must be compatible with the designated interfaces.

Completeness: All necessary calls to achieve full functionality must be represented.

Evaluation Techniques: Performance analysis of control flows to ensure low-latency interaction; scenario-based testing to validate control dependencies.

1.5.2.1.6 Viewpoint Source

Clements, P., et al. *Documenting Software Architectures: Views and Beyond* (2002).

1.5.2.2 Repository Viewpoint Definition

1.5.2.2.1 Abstract

The *Repository View* focuses on the system's large stores of persistent data, managed centrally or across several databases, enabling shared access and data integrity. This view is crucial for understanding how data is stored, accessed, and maintained.

1.5.2.2.2 Stakeholders and Their Concerns Addressed

System Administrators: Concerned with the data repository's stability, integrity, and security to maintain overall system health.

Development Team: Developers need to ensure efficient data access, maintain consistency and integrity, and enable scalability and seamless integration with the repository

1.5.2.2.3 Elements, Relations, Properties and Constraints

Elements: Central data repositories (e.g., databases), data access components, and storage structures.

Relations: "Stores" and "retrieves" relations showing interactions between data storage elements and components accessing them.

Properties: Each data repository's storage type, access protocols, and security measures.

Constraints: All data access should ensure data integrity, security, and support for concurrent data requests without conflicts.

1.5.2.2.4 Language(s) to Model/Represent Conforming Views

Entity-Relationship Diagrams (ERD) or **UML Component Diagrams** to represent data storage structures and data flow between components and repositories.

1.5.2.2.5 Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria

Consistency: All data access methods should comply with data repository constraints.
Completeness: The repository should include all essential data entities for system functionality.
Evaluation Techniques: Integrity checks, performance analysis to handle data load, and security analysis to prevent unauthorized access.

1.5.2.2.6 Viewpoint Source

Clements, P., et al. *Documenting Software Architectures: Views and Beyond* (2002).

1.5.3 Allocation Viewpoint Definition

1.5.3.1 Deployment Viewpoint Definition

1.5.3.1.1 Abstract

The Deployment View details how the software components are deployed on physical servers, virtual machines, or cloud infrastructure, ensuring that the system is ready for production and its infrastructure is properly scaled to support real-time operations.

1.5.3.1.2 Stakeholders and Their Concerns Addressed

System Administrators: Need a clear view of how the system will be hosted, with appropriate deployment strategies and resource configuration.

Development Team: Need to ensure that the deployment is efficient, scalable, and that the system is correctly implemented, meeting technical requirements and quality standards.

1.5.3.1.3 Elements, Relations, Properties and Constraints~

Elements: Application Servers, Virtual Machines, Frontend Components, Databases.
Relations: “Is-deployed-on”: Relates the software components to the execution nodes (servers or virtual machine instances).
“Communicates-with”: Shows the communication between components on different servers or instances.
Properties: Latency requirements, memory and CPU usage, system scalability.
Constraints: The deployment infrastructure must ensure high availability and resilience to support the system reliably.

The deployment should be automated and easily replicable, using tools like Docker or Kubernetes.

1.5.3.1.4 Language(s) to Model/Represent Conforming Views

UML Deployment Diagrams: To map the deployment infrastructure and how software components are distributed and deployed.

Docker Compose, Kubernetes YAML: To represent specific deployment configurations and container orchestration.

1.5.3.1.5 Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria

Consistency: All software components must be correctly allocated within the deployment infrastructure.

Completeness: The deployment view should cover all dependencies and resources needed to ensure the system's functionality.

Evaluation: Load testing, scalability tests, and resilience analysis to ensure the infrastructure supports system demand.

1.5.3.1.6 Viewpoint Source

Clements, P., et al. *Documenting Software Architectures: Views and Beyond* (2002).

1.6 How a View is Documented

Section 3 of this SAD contains one view for each viewpoint listed in Section 1.5. Each view is documented as a set of view packets. A view packet is the smallest bundle of architectural documentation that might be given to an individual stakeholder.

Each view is documented as follows, where the letter *i* stands for the number of the view: 1, 2, etc.:

- Section 3.i: Name of view.
- Section 3.i.1: View description. This section describes the purpose and contents of the view. It should refer to (and match) the viewpoint description in Section 1.5 to which this view conforms.
- Section 3.i.2: Primary presentation. This section presents the elements and the relations among them that populate this view, using an appropriate language, languages, notation, or tool-based representation.

- Section 3.i.2.1: Element catalog. Whereas the primary presentation shows the important elements and relations of the view, this section provides additional information needed to complete the architectural picture. It consists of the following subsections:
    - Section 3.i.2.1.1: Elements. This section describes each element shown in the primary presentation, details its responsibilities of each element, and specifies values of the elements' relevant *properties*, which are defined in the viewpoint to which this view conforms.
  - Section 3.i. 2.1.2: Relations. This section describes any additional relations among elements shown in the primary presentation, or specializations or restrictions on the relations shown in the primary presentation.
  - Section 3.i. 2.1.3: Interfaces. This section specifies the software interfaces to any elements shown in the primary presentation that must be visible to other elements.
  - Section 3.i. 2.1.4: Behaviour: This section specifies any significant behavior of elements or groups of interacting elements shown in the primary presentation.
  - Section 3.i. 2.1.5: Constraints: This section lists any constraints on elements or relations not otherwise described.
- Section 3.i.3: Architecture background. Whereas the architecture background of Section 2 pertains to those constraints and decisions whose scope is the entire architecture, this section provides any architecture background (including significant driving requirements, design approaches, patterns, analysis results, and requirements coverage) that applies to this view.
- Section 3.i.4: Variability mechanisms. This section describes any architectural variability mechanisms (e.g., adaptation data, compile-time parameters, variable replication, and so forth) described by this view, including a description of how and when those mechanisms may be exercised and any constraints on their use.

This organizational structure in Section 1.6 serves as a template for Section 3, ensuring consistency and clarity across all views in the SAD.

1.7 Relationship to Other SADs

Not applicable.

1.8 Process for Updating this SAD

Not applicable.

2 Architecture Background

2.1 Problem Background

The Quizzes Tutor system was developed to address specific needs in the educational field, particularly in simplifying quiz creation and management. This demand emerged from a gap in traditional assessment methods, which can be time-consuming and difficult to tailor for each class and subject. Quizzes Tutor aims to fulfill these needs by providing a practical, accessible, and secure digital platform.

2.1.1 System Overview

The Quizzes Tutor project considers the following key constraints and influences on its architecture:

- **Flexibility and Ease of Use:** The system must allow teachers to create quizzes with various question types (e.g., multiple-choice, true/false) and provide an intuitive interface for both teachers and students.
- **Data Security:** Protecting user data and quiz results is a primary concern. The architecture must support data encryption and user authentication.
- **Performance and Scalability:** Given the potential for multiple simultaneous accesses, particularly during peak assessment periods, the architecture must support horizontal scalability and high availability.
- **Chosen Technologies:** The use of Vue.js for the frontend, Spring Boot for the backend, and PostgreSQL for the database influences architectural decisions, as each of these tools provides specific capabilities in modularity, performance, and security.

These combined factors form the background for the architectural choices made for Quizzes Tutor, ensuring the system effectively meets pedagogical and technical requirements.

2.1.2 Goals and Context

This section outlines the main goals and contextual factors shaping the architecture of Quizzes Tutor. The system's primary goal is to facilitate teaching and learning through interactive quizzes, ensuring that teachers can easily create and manage quizzes, while students have a seamless experience accessing and completing them. In the educational context, the system needs to be robust, focusing on usability, scalability, and security. The architecture of Quizzes Tutor plays a central role in the software lifecycle, aligning with system engineering artifacts and meeting both pedagogical and technical needs.

2.1.3 Significant Driving Requirements

This section highlights the behavioral and quality requirements that drive the architecture of Quizzes Tutor. Performance requirements include the ability to handle high volumes of simultaneous requests, particularly during peak testing periods. Security requirements ensure that only authorized users have access to sensitive information, for example, a student is not allowed to have access to the grades of another student. Other quality attributes, such as availability and maintainability, were evaluated using methods like ATAM, ensuring the system can support future adaptations and enhancements without compromising data integrity or user experience.

2.2 Solution Background

This section outlines the rationale behind the architecture chosen for Quizzes Tutor and explains how it meets the defined behavioral and quality objectives.

Architecture Goal: Designed to be a robust, scalable, and user-friendly platform for creating and managing educational quizzes, supporting both teachers and students. The system must be intuitive, reliable, and maintain consistent performance.

Behavioral Goals:

User Experience: The Vue.js frontend offers an interactive and responsive interface, crucial for a positive user experience in educational environments, where usability is key.

Reliability: Ensures consistent execution of core functionalities (e.g., quiz creation, submission, and grading), minimizing disruptions.

Quality Attributes:

Performance: The frontend-backend separation optimizes each component's performance, keeping the interface responsive while the backend manages business logic and data processing efficiently.

Scalability: Using PostgreSQL and a modular implementation supports the system's growth for increasing users, while preserving integrity and performance.

Maintainability: The modular architecture facilitates feature additions and updates without impacting the entire application, enabling continuous improvement.

2.2.1 Architectural Approaches

This section discusses the key design decisions underlying Quizzes Tutor's architecture, including adopted architectural styles, patterns, and considered alternatives.

Architectural Styles and Patterns:

Separate Frontend-Backend Architecture: Vue.js is used for the frontend, separate from the Spring Boot (in java) backend, creating a responsive and scalable interface independent of business logic and data processing.

Client-Server Model: Sensitive operations and data are securely processed on the server (backend), while the client (frontend) offers an interactive user experience.

Data Persistence with PostgreSQL: Chosen for its reliability, scalability, and compatibility with complex transactions, PostgreSQL is ideal for managing quiz and assessment data.

Design Rationale:

Vue.js for Frontend: Chosen for its quick development cycle and easy learning curve, enabling the team to build an intuitive interface.

Spring Boot for Backend: Selected for robustness and modularity, managing business logic and integrating well with PostgreSQL.

PostgreSQL as Database: Ensures data integrity and consistency, essential for educational systems.

COTS Issues:

Use of Open-Source Solutions: The architecture leverages open-source solutions, like PostgreSQL, to meet quality requirements without significant added cost.

IST Authentication Integration: Integrated with the Instituto Superior Técnico's authentication system to ensure secure and straightforward access for users.

2.2.2 Analysis Results

SonarQube is a powerful tool for static code analysis, designed to assess and improve the quality of software projects. It provides insights into various aspects of code quality, including security, reliability, maintainability, test coverage, and code duplication. By identifying potential vulnerabilities, technical debt, and areas for optimization, SonarQube supports developers in building robust and high-quality software systems.

This section focuses on the analysis performed using SonarQube, detailing the project's strengths and weaknesses in key quality metrics. Each subsection will provide an in-depth examination of specific aspects, including recommendations for improvement, supported by relevant visualizations and examples from the analysis results.

2.2.2.1 Security Analysis

The security analysis performed by SonarQube identified several areas of concern categorized by review priority: High, Medium, and Low. These findings highlight potential vulnerabilities and areas that may require further inspection to ensure the application's resilience against threats.

High-Priority Issues:

Cross-Site Request Forgery (CSRF):

- Occurrences: 2
- Description: CSRF vulnerabilities allow attackers to trick authenticated users into performing unintended actions on a web application.
- Impact: May compromise user accounts and application functionality.

Cross-Site Scripting (XSS):

- Occurrences: 19

- Description: XSS vulnerabilities can allow attackers to inject malicious scripts into web pages viewed by other users.
- Impact: Can lead to theft of user data, session hijacking, or website defacement.

Medium-Priority Issues

Denial of Service (DoS):

- Occurrences: 4
- Description: DoS vulnerabilities can lead to resource exhaustion, rendering the application unresponsive.
- Impact: May disrupt service availability for legitimate users.

Weak Cryptography:

- Occurrences: 2
- Description: Use of weak or outdated cryptographic algorithms.
- Impact: Could compromise the integrity and confidentiality of sensitive data.

Low-Priority Issues

Encryption of Sensitive Data:

- Occurrences: 8
- Description: Potential areas where sensitive data encryption may not be properly enforced.
- Impact: Could expose user information if intercepted.

Insecure Configuration:

- Occurrences: 4
- Description: Misconfigurations that may introduce security risks.

- Impact: Could be exploited to gain unauthorized access or expose sensitive functionality.

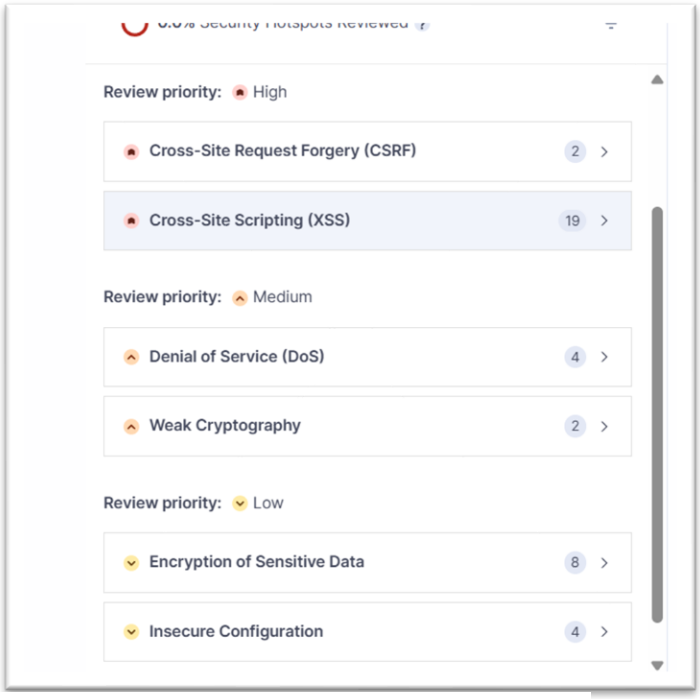


Figure 1 - Security Analysis of Sonarqube

2.2.2.2 Reliability Analysis

The reliability analysis identified 230 open issues, distributed across various levels of severity. Below is a detailed breakdown:

Severity distribution:

- Critical: 4 issues
- Major: 223 issues
- Minor: 3 issues

Examples of issues:

Critical Issues:

Improper Stream Handling:

- Certain code sections do not properly close streams or ensure they are managed correctly.
- Example: stream() pipelines left open, leading to potential memory leaks or resource exhaustion.

Null Pointer Dereference:

- Code paths that dereference objects without null-checks, resulting in possible runtime exceptions.
- Example: A method assumes a variable is non-null without proper validation, which could crash during edge cases.

Figure 2 - Example of critical reliability issue

Major Issues:

Redundant Resource Management:

- Several instances where resources are not released in a timely manner, though not as critical as open streams.

Inconsistent Exception Handling:

- Areas where exceptions are either swallowed silently or handled inconsistently across the codebase.

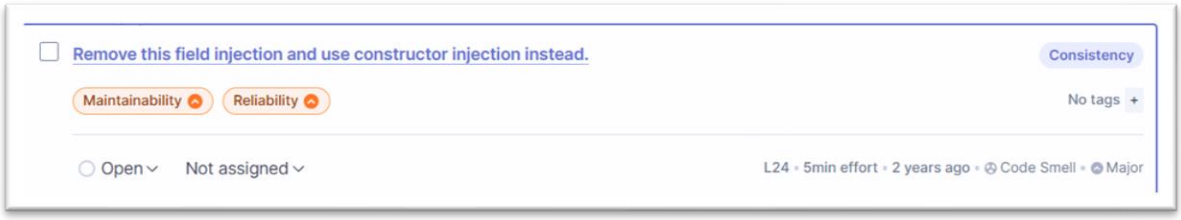


Figure 3 - Example of major reliability issue

Minor Issues

Code Formatting and Readability:

- Minor concerns with inconsistent formatting or redundant code snippets.
- Example: Excessively long methods with inline logic that could benefit from modularization for better clarity.

Figure 4 - Example of minor reliability issue

2.2.2.3 Maintainability Analysis

The maintainability analysis highlights 484 open issues, which are distributed across different severities as follows:

Severity distribution:

- **Critical**: 28 issues
- **Major**: 270 issues
- **Minor**: 186 issues

Examples:

Critical Issues:

Improper Use of Serialization:

- Classes such as CodeFillInAnswer, CodeOrderAnswer, and MultipleChoiceAnswer are flagged for not being properly marked as transient or serializable.

- **Example**: In CodeFillInStatementAnswerDetailsDto.java, serialization warnings indicate potential data leakage or issues in object state persistence.

Hard-Coded Literals:

- Repeated occurrences of hard-coded values (e.g., email addresses) without being defined as constants.
- **Example**: The literal "rito.silva@tecnico.ulisboa.pt" is duplicated three times in AnswerService.java.

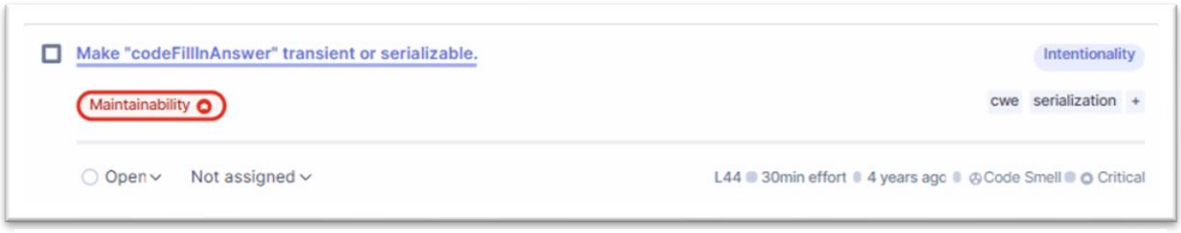


Figure 5 - Example of critical maintainability issue

Major Issues:

Field Injection Over Constructor Injection:

- Instances where fields are injected directly, rather than using constructor-based dependency injection.
- **Example**: Classes such as TutorApplication and AdminController are flagged for this practice, which reduces testability and increases coupling.

Complex Conditionals:

- Overly complex logic in conditionals, making the code harder to read and maintain.

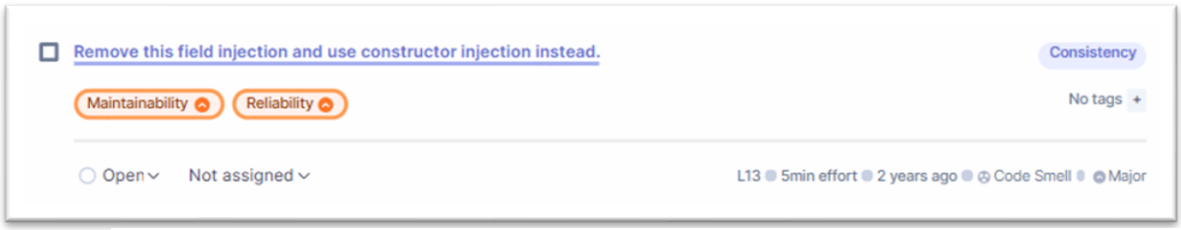


Figure 6 - Example of major maintainability issue

Minor Issues:

Inconsistent Formatting:

- Code formatting is not standardized, leading to inconsistent indentation or spacing.

- Example: Several files have redundant blank lines or irregular alignment of blocks.

Redundant Code:

- Duplicate code snippets that could be refactored into shared methods or utility functions.
- Example: Repeated logic for handling similar operations across different modules.

Unused Variables and Imports:

- Variables and imports that are declared but never used, adding unnecessary clutter to the codebase.

Figure 7 - Example of minor maintainability issue

2.2.2.4 Test Coverage Analysis

The analysis of the test coverage for the project reveals an overall code coverage of **56.2%**, indicating that slightly more than half of the codebase is being executed during testing. Detailed statistics are as follows:

- **Total Lines to Cover:** 9,835
- **Uncovered Lines:** 4,056
- **Line Coverage:** 58.8%
- **Conditions to Cover:** 2,015
- **Uncovered Conditions:** 1,130
- **Condition Coverage:** 43.9%

The scatter plot below highlights the distribution of coverage against technical debt. While there are several files with high coverage, some critical areas remain under-tested, representing potential risks and opportunities for improvement.

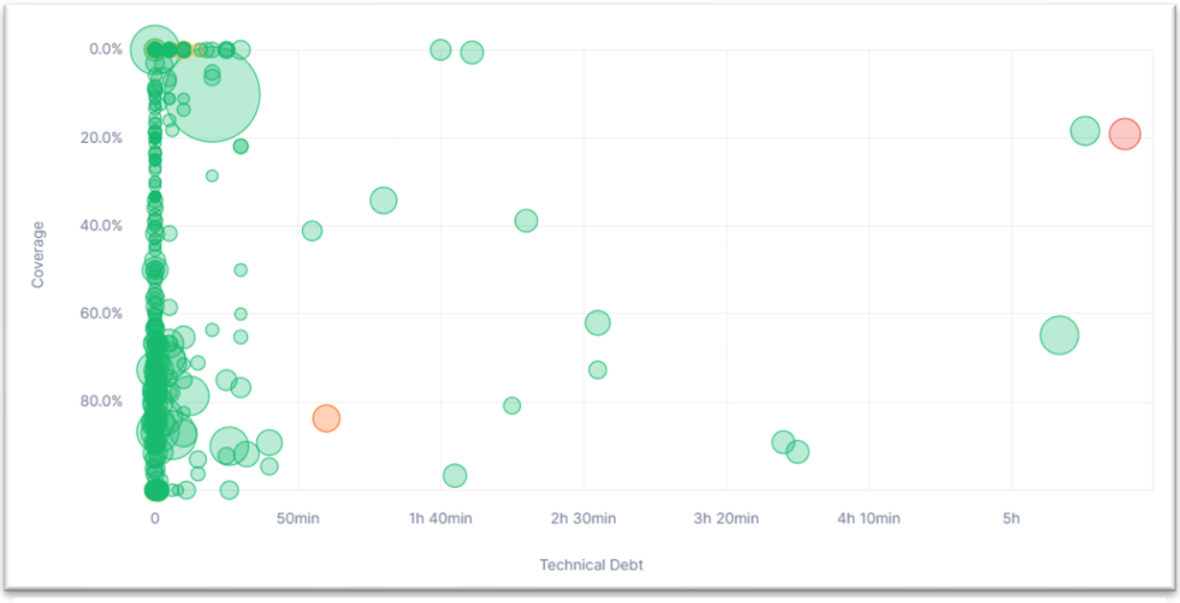


Figure 8 - Plot of test coverage distribution

2.2.2.5 Duplication Analysis

The duplication analysis indicates a **duplication rate of 1.6%** across the codebase, calculated on **41,000 lines of code**. This duplication level is quite low, which is a positive indicator of code quality and adherence to software engineering best practices. However, it's essential to identify and assess duplicated blocks to ensure maintainability and reduce the risk of inconsistencies in future updates.

Density: 1.6%

- This indicates that 1.6% of the codebase is duplicated.
- A low duplication density like this suggests that the project is well-structured with minimal redundancy.

Duplicated Lines: 672

- A total of 672 lines across the project are duplicated.
- While this is not an excessive amount, reducing it further can improve maintainability and readability.

Duplicated Blocks: 117

- These represent blocks of code that are repeated across different files or modules.
- Duplication at the block level often results from repeated logic or functionality that could be abstracted into reusable components.

Duplicated Files: 15

- 15 files contain duplicated code, suggesting specific areas of the project that may benefit from refactoring.
- Focusing on these files can have a significant impact on reducing overall redundancy.

While the overall duplication density of 1.6% is relatively low, there are some outliers like CoursesView.vue, as we can see in the plot below, suggesting opportunities for optimization. By addressing these high-duplication files, the project can enhance maintainability, reduce technical debt, and promote cleaner code.

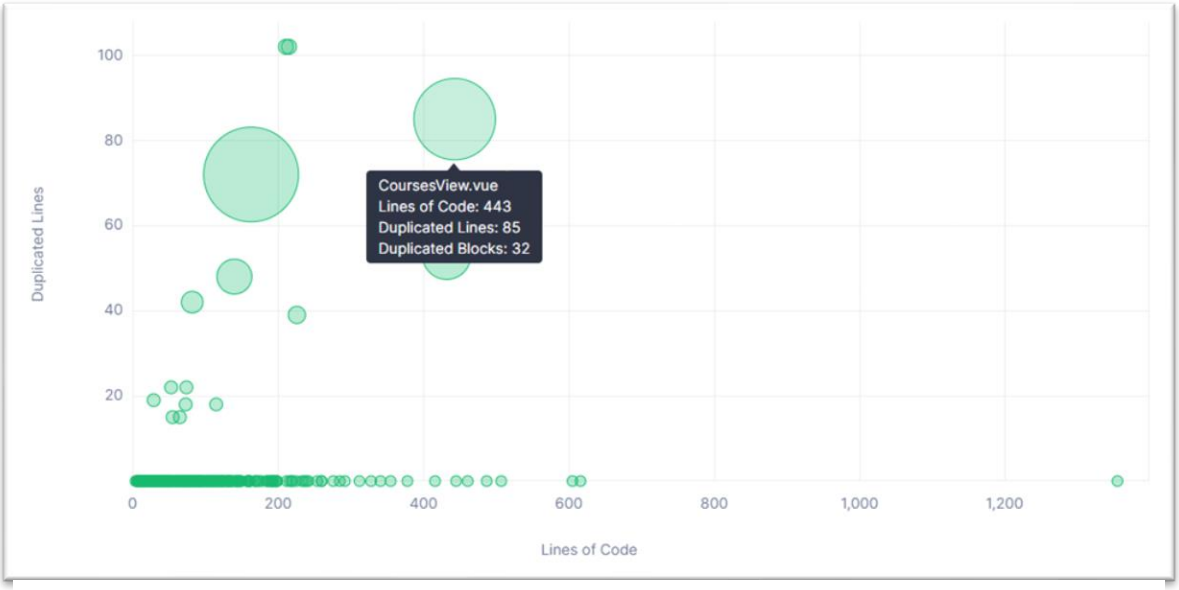


Figure 9 - Plot of duplication density

2.2.3 Requirements Coverage

This section summarizes how the Quizzes Tutor architecture meets both original and derived functional and quality requirements established for the project.

Functional Requirements (Original):

<u>Quiz Creation and Management</u>: The modular architecture allows the backend to handle quiz creation and management independently, fulfilling the original requirement to support the educational process.

User Interaction and Experience: The Vue.js frontend ensures that the system is accessible and responsive, meeting the original requirement to provide an intuitive experience for students and teachers.

Quality Requirements (Original and Derived):

Security (Original): The architecture ensures data protection and controlled access through the integration of robust authentication mechanisms, meeting the original security requirement.

Performance (Derived): The separation of frontend and backend, along with the choice of PostgreSQL, guarantees fast response times, addressing the derived requirement for performance even under heavy load.

Scalability and Flexibility (Derived): The system's modularity and the ability to add backend instances support user growth without significant reengineering, meeting the derived requirements for scalability and flexibility.

2.2.4 Summary of Background Changes Reflected in Current Version

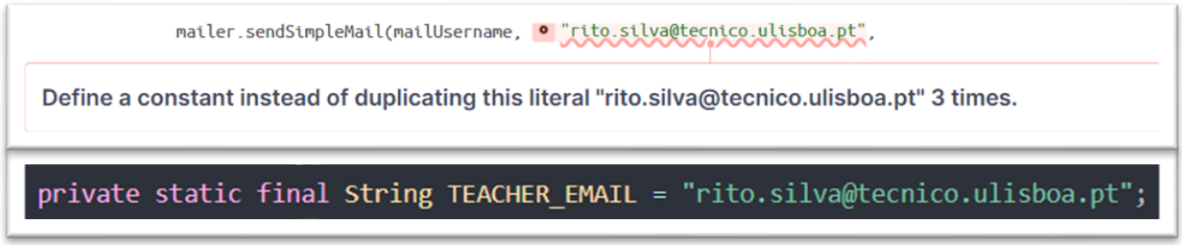
During the analysis and refactoring of the code, several critical points affecting the **reliability** of the system were identified. The implemented changes significantly improved the stability of the code. Below are some of the main corrections made and the impact they had on the overall behavior of the application:

Removal of Duplicate Literals

- **Correction**: Various literals that were duplicated in the code, such as email addresses and specific identifiers, were replaced with constants.
- **Impact**: This change avoids repetition of strings in the code, improving both readability and maintainability. It also reduces the risk of errors due to inconsistent values. Using constants makes future modifications easier without needing to update multiple locations in the code.

- **Example:**

Figure 10 - Removal of Code Literals



Changing use of field injection to constructor injection

In this change, instead of using field injection, constructor injection was implemented. In field injection, the dependencies were directly injected into the fields of the class, which can sometimes lead to issues such as difficulty in testing or maintaining the code. By switching to constructor injection, the dependencies are now passed through the constructor, which is generally considered a better practice.

This change improves the code in several ways:

Immutability: Dependencies are now passed as parameters and can be marked as final, ensuring that they cannot be modified after object construction, improving immutability.

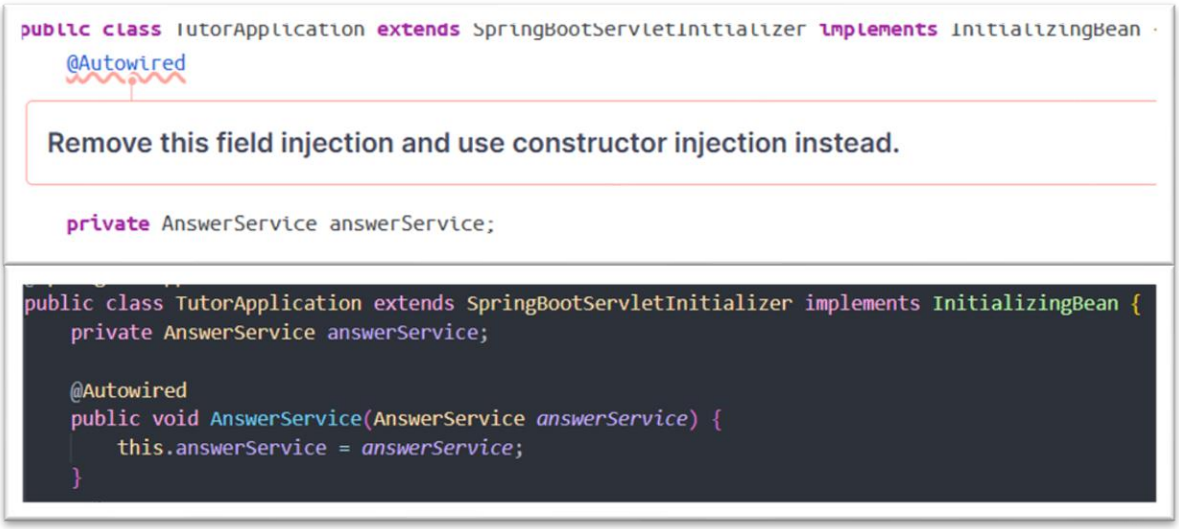
Testability: With constructor injection, it becomes easier to mock dependencies in unit tests, leading to better isolation of components during testing. This reduces the potential for errors during the testing phase.

Clarity: Constructor injection makes the dependencies of the class explicit, as they are provided through the constructor. This improves readability and understanding of the class's requirements, making the codebase easier to maintain.

Decoupling: Constructor injection helps decouple classes from their dependencies, making the code more modular and flexible for future changes.

- **Example**

Figure 11 - Constructor injection instead of field injection



In addition to the corrections mentioned above, several other improvements were made to the codebase, further enhancing the overall quality. As a result, the number of reliability issues was reduced from 230 to 179, as it is possible to see in the image down below, representing a significant improvement. Furthermore, the maintainability of the project was also positively impacted. These changes contribute not only to the immediate reduction of issues but also to the long-term sustainability of the code.

Figure 12 - Code refactored

2.3 Product Line Reuse Considerations

Not applicable.

3 Views

This section presents the views of the *Quizzes Tutor* architecture according to the viewpoints defined in Section 1.5. Each view offers a representation of the system from the perspective of a specific set of stakeholder concerns, as defined by [IEEE 1471]. The views illustrate architectural elements, their properties, and the relationships between them according to a specific viewpoint.

Overview of Architectural Views

The views are organized into three main categories, based on the nature of the elements they represent:

1. **Module Views**: These focus on the static structure of the system, dividing it into modules or implementation units. These views help answer questions such as:
   What is the primary functional responsibility assigned to each module?
   What other software elements is a module allowed to use?
   Which modules are related to others by generalization or specialization?

The subtypes included in this category are:

Decomposition View: Displays the hierarchy of modules and submodules, showing how each part of the system has a specific responsibility.
Uses View: Highlights functional dependencies between modules, illustrating essential interactions for system functionality.
Data Model View: Shows relationships between data entities, ensuring a clear and consistent structure for information storage.

2. **Component-and-Connector Views**: These represent the runtime components and the connectors that facilitate communication between them. This view emphasizes the system's behavior during execution and helps answer questions such as:
   What are the major executing components, and how do they interact?
   Which parts of the system can operate in parallel, and how does data flow through the system?

The subtypes included in this category are:

Call-Return View: Models interactions where components receive control and data from others, execute a function, and return control to the invoker.
Repository View: Represents large stores of persistent data managed centrally or distributed, allowing shared access to data.

3. **Allocation Views**: These show how software elements relate to the physical or virtual infrastructure, including hardware and runtime environments. These views address questions about resource distribution and allocation, such as:
   Where is each software element deployed in terms of physical or virtual infrastructure?
   How are resources assigned to development and operations teams?

The subtypes included in this category are:

Deployment View: Specifies how the software components are mapped onto the underlying hardware or virtual infrastructure.

Structure of Architectural Views

Each view documented in this SAD addresses a specific set of stakeholder concerns, using a clear separation to focus on the main areas of architectural decision-making:

- **System Code Structure** (via the **Module View**),
- **Organization of Runtime Components and Interactions** (via the **Component-and-Connector View**),
- **Allocation of Software Elements to Physical or Virtual Infrastructure** (via the **Allocation View**).

This separation allows each view to maintain its focus on a specific area of architectural interest, avoiding confusion and ensuring a clear and organized presentation of the system architecture.

The views presented in this SAD are the following:

Table 3: Views of this sad

Name of view	**Viewtype that defines this view**	**Types of elements and relations shown**		**Is this a module view?**	**Is this a component-and-connector view?**	**Is this an allocation view?**
Decomposition View	Modules and submodules hierarchy	Modules and Submodules	“Is-part-of”	Yes	No	No
Uses View	Functional dependencies between modules	Modules	“Uses”	Yes	No	No
Data Model View	Data entities and relationships	Data Entities	Associations relations	Yes	No	No
Call-Return View	Control flow between components	Components	“Calls” relation	No	Yes	No

Repository View	Data storage components and access connectors	Data Repositories and Access Components	“Stores” and “Retrieves” relations	No	Yes	No
Deployment View	Deployment of software components	Software Components, Nodes, Servers, Virtual Machines	“Is-deployed-on” and “Communicates-with”	No	No	Yes

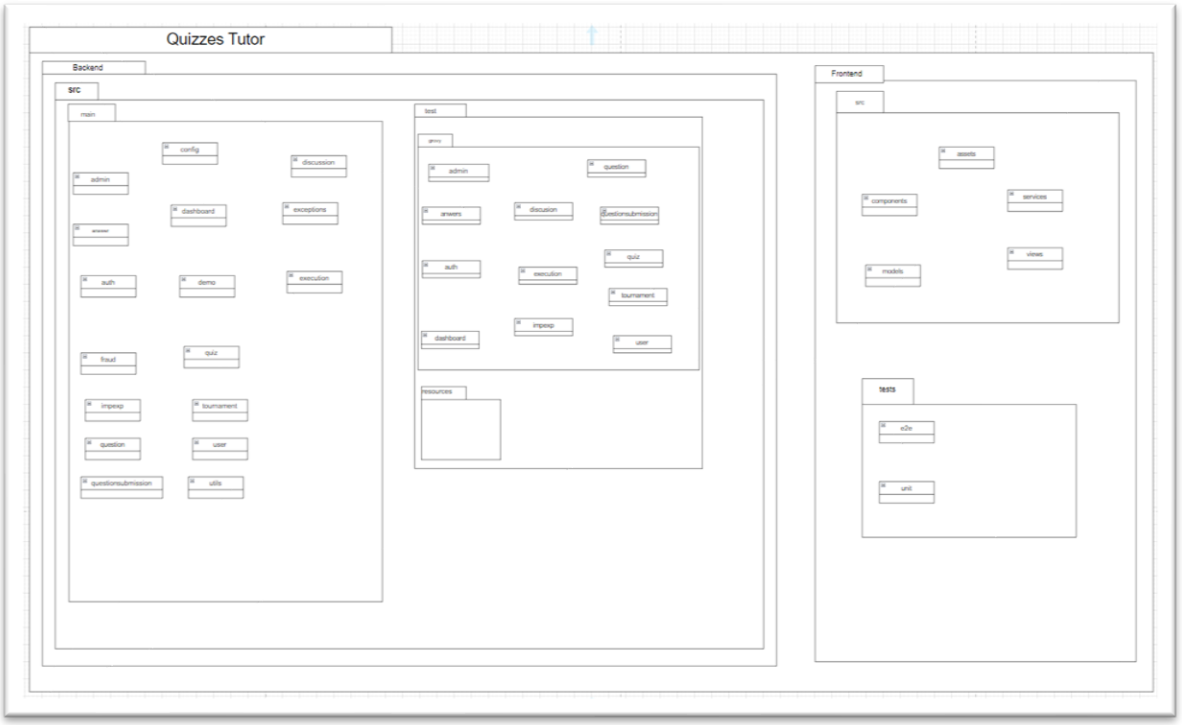
3.1 Decomposition View

3.1.1 View Description

The Decomposition View presents the hierarchical structure of the system, organizing it into modules and submodules to reflect its functional composition. This view is essential for understanding the distribution of responsibilities across the system, clearly identifying the functions assigned to each module and how they interact with one another. In addition to providing a clear picture of the system’s division, it helps identify dependencies and relationships between different components, which is crucial for ensuring modularity and maintainability of the architecture. The Decomposition View also allows for the visualization of the layers or functional groupings that make up the system, offering a detailed and organized perspective that supports both development and long-term maintenance.

3.1.2 Primary Presentation

Figure 13 - Decomposition VIew



3.1.2.1 Element Catalog

3.1.2.1.1 Elements

Frontend:

- **src**: Contains the source code for the frontend application.
    - **assets**: Contains static files (e.g., images, fonts) used by the frontend.
    - **components**: Contains main UI components such as buttons, forms, and other Vue.js widgets.
    - **models**: Defines data models that are used to interact with the UI.
    - **services**: Contains modules that make API calls to the backend to perform CRUD operations or fetch data.
    - **views**: Contains pages for the Vue.js frontend that are rendered to the user (e.g., home page, quiz pages, etc.).
- **tests**: Contains tests for the frontend.
    - **e2e**: End-to-end tests covering full user interactions in the frontend.
    - **unit**: Unit tests for ensuring individual components and functions work as expected.

Backend:

- **src**: Contains the source code for the backend application.
    - **config**: Contains configuration files for the system, such as security settings, authentication, etc.
    - **admin**: Logic related to system administration (e.g., user management, permissions).
    - **dashboard**: Logic and components for managing and displaying dashboard data in the backend.
    - **discussion**: Logic for handling discussions and messages (e.g., forums or chat systems).
    - **exceptions**: Handles and defines exceptions or errors in the system.
    - **answer**: Module for managing answers in the system (e.g., quiz responses).
    - **auth**: Handles user authentication and authorization (login, tokens, etc.).
    - **demo**: Handles demo-related functionality or features for system demonstration.
    - **execution**: Manages the execution of tasks in the backend (e.g., running quizzes).
    - **fraud**: Detects and prevents fraud in the system.
    - **quizz**: Manages quizzes, questions, answers, and user interactions related to quizzes.
    - **impexp**: Handles the import and export of data (e.g., CSV, JSON files).
    - **tournament**: Manages tournaments (e.g., rankings, competitions).
    - **question**: Handles the questions in quizzes.
    - **user**: Manages user information and actions (e.g., user profile, settings).
    - **questionsubmission**: Manages user-submitted questions.
    - **utils**: Helper functions and utilities used across the backend.
- **test/groovy**: Contains tests for the backend, with submodules corresponding to the logic of each backend area, such as **admin**, **question**, **answers**, **discussion**, etc.
- **test/resources:** CSV files that are used for testing the handling of user imports and data validation, particularly with incorrect formats or bad data.

3.1.2.1.2 Relations

The system's modules and submodules are organized with **is-part-of** relationships, meaning that each submodule is part of a larger module. These relationships define the structure of the system, where each module is composed of smaller, specialized submodules that contribute to the overall functionality.

3.1.2.1.3 Interfaces

The frontend interacts with the backend through **RESTful API interfaces**. These interfaces allow the frontend to send requests for data (e.g., quizzes, questions, user information) and perform CRUD operations. The communication is done via HTTP requests, ensuring data flow between the client-side and server-side components.

3.1.2.1.4 Behaviour

- The assets, components, models, services, views, and utils share a relation of is-part-of with the src submodule in the frontend.
- The tests submodule contains the e2e and unit directories, both responsible for testing the frontend components and their functionalities.
- In the backend, the config, admin, dashboard, discussion, exceptions, answer, auth, demo, execution, fraud, quizz, impexp, tournament, question, user, questionsubmission, and utils submodules are part of the src module and manage different backend processes and logic.
- The test/groovy submodule corresponds to the backend areas and has separate folders to test logic related to each area, such as admin, question, answers, etc.
- The test/resources contains CSV files that are used in testing, particularly for validating data imports and edge cases (e.g., wrong formats, missing values).

3.1.2.1.5 Constraints

- Each module should be responsible for a specific function and should not overlap responsibilities with other modules.
- No circular dependencies are allowed between modules.
- A module can only have one parent, ensuring a clear hierarchical structure.
- All backend modules should interact with the src submodule, and frontend components should interact with the src folder in the frontend module.
- Any change to a module should not affect the functionality of other modules unless explicitly intended.

3.1.3 Architecture Background

The system consists of two main modules: the frontend and the backend. The user interacts directly with the frontend, which handles the UI and user experience. When the user performs actions that require backend processing (e.g., creating quizzes or managing users), the frontend sends requests to the backend via REST API endpoints.

For instance, the authentication process is initiated through the frontend, allowing users to log in and gain access to various functionalities. Once authenticated, users can interact with documents in real-time. To support this, the system uses REST API connections to transmit data instantly between the frontend and the backend. This real-time communication ensures that all users working on the same document are synchronized.

REST API was selected for its straightforward implementation and ease of use, providing a reliable method for handling CRUD operations and other user requests.

3.1.4 Variability Mechanisms

This decomposition allows for flexibility in enhancing or scaling specific modules, such as expanding backend capabilities independently of the frontend.

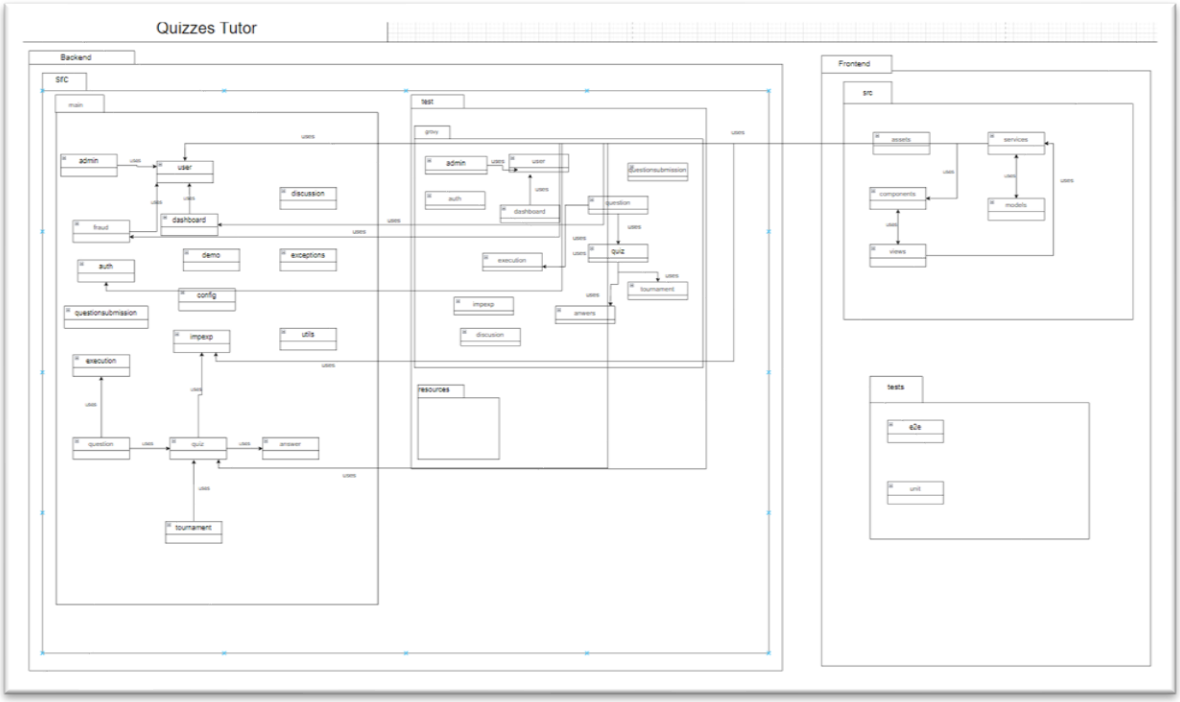
3.2 Uses View

3.2.1 View Description

The Uses View emphasizes the functional dependencies between modules, demonstrating how various system components work together to provide the desired functionality. It plays a key role in understanding the interconnections and relationships between modules, revealing which components depend on one another to ensure the system operates smoothly. By mapping these dependencies, it becomes easier to pinpoint essential relationships that must remain intact for the system to function correctly. This approach also facilitates the evaluation of the potential impact of changes or updates to specific modules. Overall, this view is crucial for maintaining the integrity of the system and ensuring that all parts interact harmoniously.

3.2.2 Primary Presentation

Figure 14 - Uses View



3.2.2.1 Element Catalog

3.2.2.1.1 Elements

This view uses the same elements as the decomposition view above.

3.2.2.1.2 Relations

This view establishes “use” relationships between the elements. A **use relationship** establishes a dependency where one element relies on another to perform its function or achieve its purpose. Now we will point out every use dependency we found and give a brief explanation about each one:

Backend

- auth -» user: Auth uses user component to authenticate users
- quiz -» question: quiz uses questions to define the content of the quiz

- quiz -» answer: quiz uses answers to associate responses the corresponding questions

- execution -» quiz: execution uses quiz to manage and track the quizzes done in a certain course

- dashboard -» user: dashboard relies on user to display information about that user like weekly scores.

- admin -» user: admin uses user to anonymize the user

- fraud -» user: fraud uses user to monitor and detect suspicious activities associated with users

Frontend

- services -» models: services uses models to define data structures and enforce integrity

- components -» views: components interact with the views to render interfaces and enable user interaction.

- models -» services: Models rely on services to fetch, store, or process data in alignment with the defined structure

- views -» components: Views use components to organize the visual elements of the user interface.

- views -» services: Views depend on Services to retrieve data or execute app logic needed for display.

Frontend/Backend

- services (Frontend) -» auth (Backend): Frontend services use the auth backend to manage authentication.

- services (Frontend) -» quiz (Backend): Frontend services rely on the quiz backend for quiz-related data, including questions and answers.

- services (Frontend) -» user (Backend): Frontend services use the user backend to retrieve or update user-specific data.

- services (Frontend) -» dashboard (Backend): Frontend services rely on the dashboard backend to access results

- services (Frontend) -» impexp (Backend): Frontend services interact with the impexp backend to manage data import/export functionalities.

- services (Frontend) -» fraud (Backend): Frontend services use the fraud backend to access fraud detection or monitoring results.

3.2.2.1.3 Interfaces

This view has the same interfaces as the decomposition view above.

3.2.2.1.4 Behaviour

- **Backend relationships:** Behavior generally involves retrieving, processing, or storing information needed for their tasks.

- **Frontend relationships:** Frontend components like services, models, and views work together to ensure seamless user experience, Services handle data processing and API communication, models define data structures, and views and components manage the UI and interactions.

- **Frontend/Backend relationships:** Frontend services interact with backend modules (auth, quiz, dashboard, etc.) to fetch or send data, ensuring the application operates as intended.

3.2.2.1.5 Constraints

- **Backend constraints:** Ensure modularity: backend components should provide well-defined APIs, maintain data integrity: shared resources like user, question, and answer must enforce strict validation and consistency, Security: sensitive components like auth and fraud must implement strong encryption and secure access.

- **Frontend constraints:** Scalability, Separation of concerns: keep services, models, views, and components modular and maintain a clear distinction between data handling, UI logic, and presentation.

- **Frontend/Backend constraints:** API reliability: Backend services must return consistent, accurate, and timely responses to frontend requests.

3.2.3 Architecture Background

This project uses **use dependencies** to establish clear relationships between components, ensuring that each element relies on others only when necessary to fulfil its function. These dependencies are central to achieving modularity, scalability, and maintainability within the system. For example, backend modules like auth depend on user for authentication, while frontend components like views rely on services to retrieve and display data dynamically.

By leveraging these use dependencies, the project ensures a well-structured architecture where components remain reusable and independent. This approach supports seamless interaction between frontend and backend through APIs and enforces clear boundaries between functionalities.

3.2.4 Variability Mechanisms

Use dependencies help in **variability mechanisms** by defining clear relationships between components, which allows the system to be flexible and adaptable. These dependencies enable configuration management, modularity, and feature toggles, allowing the system to be easily customized, extended, or scaled without affecting other parts.

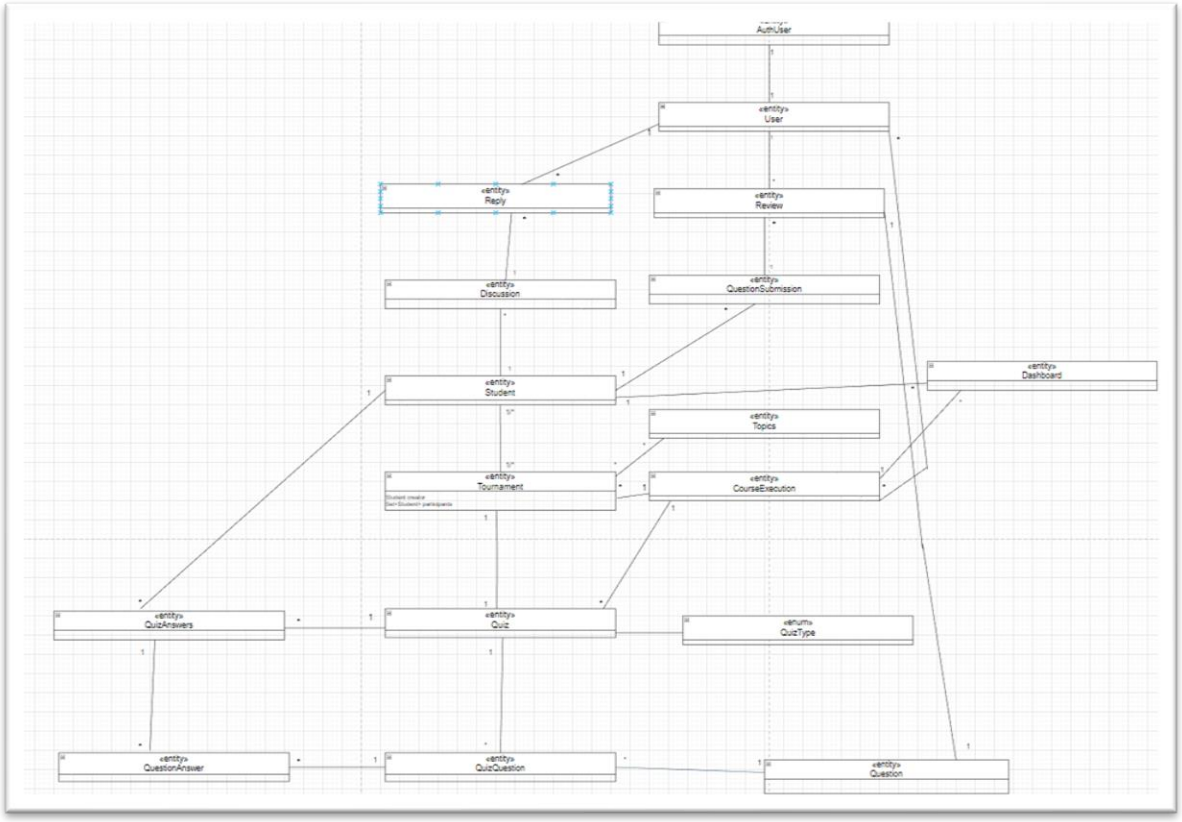
3.3 Data Model View

3.3.1 View Description

The Data Model View illustrates the relationships between data entities in the system, ensuring consistent data storage and access. It helps define how entities like user profiles and quiz results interact, supporting efficient data retrieval. By mapping these relationships, it ensures data integrity and guides database design decisions. This view is crucial for maintaining consistency and scalability as the system grows. It also supports backend development by providing a clear structure for how data flows within the system.

3.3.2 Primary Presentation

Figure 15 - Data Model View



3.3.2.1 Element Catalog

3.3.2.1.1 Elements

- **Quiz:** Represents a Quiz with a specific QuizType with multiple QuizQuestion and QuizAnswer.
- **QuizQuestion:** Represents a Question of a quiz. Has a list of answers to that question.
- **QuizAnswer:** Represents if a Quiz state, if its completed or not or if its fraudulent. It has a list of QuestionAnswer that represent an answer to a question.
- **Question:** Represents a question of a Topic, that as a Discussion and is part of one/many QuizQuestion. This question has an image, title and is part of a Course.
- **QuizType:** Represents the type of Quiz. The quiz can have one of these types EXAM, TEST, GENERATED, PROPOSED, IN_CLASS, TOURNAMENT.
- **Tournaments:** Represents tournament created by a Student.
- **Topics:** Represents a topic of a Question.

- **CourseExecution:** Represents an execution of a Course. This execution has a set of Quizzes and set of Users and set of Discussions.
- **Student:** Represents a Student that has his QuestionSubmissions, Dashboards ,Tournaments and QuizAnswers.
- **Discussion:** Represents a discussion to a QuestionAnswer to a Question given by a Student.
- **Dashboard:** Represents a dashboard that shows information.
- **QuestionSubmission:** Represents a submission of a Question and its Status given by a Student. This submission has Reviews.
- **Reply:** A response to a Discussion post. A Reply allows users to engage in conversations and provide feedback or answers to questions raised in discussions.
- **Review:** Represents a review of a QuestionSubmission.
- **User:** Represents a user that can be a Student, Teacher , Admin.
- **AuthUser:** Represents the authentication credentials of a user.

3.3.2.1.2 Relations

- **Quiz ↔ QuizQuestion:** A **Quiz** contains one or more **QuizQuestions**, a **QuizQuestion** can only belong to one **Quiz**.
- **QuizQuestion ↔ Question:** A **Question** can be associated with multiple **QuizQuestions**, as it might appear in different quizzes.
- **Quiz ↔ QuizAnswer:** Each **QuizAnswer** is tied to a specific **QuizQuestion** within the **Quiz**.
- **Quiz ↔ QuizType:** A **Quiz** is associated with a specific **QuizType**.
- **Tournaments ↔ Quiz:**A **Tournament** involves one or more **Quizzes**, a **Quiz** may be part of multiple **Tournaments**.
- **Student ↔ Quiz:** A **Student** can take multiple **Quizzes** as part of their learning process. A **Quiz** may have many **Students** who attempt it.
- **Student ↔ Tournament:** A **Student** can participate in multiple **Tournaments**, a **Tournament** can involve many **Students**.
- **Topics ↔ QuizQuestion: Topic** can contain multiple **QuizQuestions**, a **QuizQuestion** can belong to multiple **Topics** based on the subject or category.
- **CourseExecution ↔ Student:** A **CourseExecution** represents a specific instance of a course and is associated with multiple **Students** who are enrolled in it.
- **CourseExecution ↔ Quiz:** A **CourseExecution** can have multiple **Quizzes** as part of the course. A **Quiz** can be linked to multiple **CourseExecutions** if the same quiz is used across different instances of the course.
- **CourseExecution ↔ Topics:** A **CourseExecution** can include multiple **Topics** to structure the course content. A **Topic** can be associated with multiple **CourseExecutions**.

- **Discussion ↔ User:** A **Discussion** can be created by a **User**. A **User** can participate in multiple **Discussions**.
- **Reply ↔ Discussion:** A **Reply** is linked to a specific **Discussion** and provides responses to posts in the discussion. Multiple **Replies** can belong to one **Discussion**.
- **Reply ↔ User:** A **Reply** is written by a **User**, who is responding to a discussion post.
- **Review ↔ User:** A **Review** is provided by a **User**. A **User** can leave multiple **Reviews**.
- **Review ↔ Quiz:** A **Review** is given for a **Quiz**, providing feedback on the quiz content or difficulty.
- **User ↔ AuthUser:** Each **User** has one corresponding **AuthUser** that handles their login credentials and UserType.
- **User ↔ Dashboard:** A **User** has one **Dashboard**, which provides an overview of their progress and activity within the system.

3.3.2.1.3 Interfaces

Quiz ↔ QuizQuestion:

- A **Quiz** contains one or more **QuizQuestions**. The system allows adding or retrieving questions from a quiz via API.

Quiz ↔ QuizAnswer:

- Each **QuizAnswer** is linked to a **QuizQuestion**. After completing the quiz, answers are submitted via the **QuizAnswer** API.

Quiz ↔ QuizType:

- A **Quiz** is associated with a specific **QuizType** (e.g., EXAM, TEST). The type is defined via API when creating the quiz.

Tournaments ↔ Quiz:

- A **Tournament** can contain multiple **Quizzes**. Quizzes are linked to tournaments via API.

Student ↔ Quiz:

- A **Student** can take multiple **Quizzes**. Student responses and progress are updated through the **QuizAnswer** API.

Student ↔ Tournament:

- A **Student** can participate in multiple **Tournaments**. Enrolment and progress tracking are handled via API.

CourseExecution ↔ Student:

- **CourseExecution** manages enrolled students. Enrolment and student progress are handled through API calls.

CourseExecution ↔ Quiz:

- **CourseExecution** can have multiple **Quizzes**. Quizzes are assigned to courses via API.

CourseExecution ↔ Topics:

- **CourseExecution** includes several **Topics**. Topics are linked to courses via API.

Discussion ↔ User:

- **Discussions** are created by **Users**. Creation and participation in discussions are managed via API.

Reply ↔ Discussion:

- **Reply** is linked to a specific **Discussion**. Replies are posted and retrieved via API.

Reply ↔ User:

- **Reply** is created by a **User**. Each reply is associated with the user who posted it via API.

Review ↔ User:

- **Review** is submitted by a **User** for a **Quiz**. Reviews are submitted via API.

Review ↔ Quiz:

- **Review** provides feedback on a **Quiz**. Reviews are stored via API.

User ↔ AuthUser:

- **User** is associated with an **AuthUser** for authentication. Authentication is handled via API.

User ↔ Dashboard:

- **User** has a **Dashboard** to track their progress. The dashboard is updated in real time via API.

3.3.2.1.4 Behaviour

- **User Interaction**: Users, including Students, Professors, and Administrators, interact with the system through a variety of interfaces (e.g., quizzes, tournaments, discussions, etc.). Each user performs actions based on their role (e.g., a Student takes quizzes, a Professor manages courses and quizzes).
- **Course Execution**: A **CourseExecution** represents a running instance of a course. It coordinates the flow of quizzes, topics, and students. Students participate in quizzes, and their progress is tracked in real time.
- **Quiz Flow**: When a quiz is started, the system retrieves the relevant **QuizQuestions** and displays them to the user. As the student answers questions, the system evaluates and stores the responses in **QuizAnswer**. The quiz can be part of a **Tournament** or directly linked to a **CourseExecution**.

- **Discussion and Feedback**: Students can engage in discussions related to quiz questions or course topics. Responses to discussions are captured as **Replies**, and these are linked to the original **Discussion**. Feedback is also provided through **Reviews**, where students can leave comments about quizzes.

3.3.2.1.5 Constraints

Mandatory Entity Relations (Quiz, Question, Answer)

- Constraint: Each Quiz must have at least one QuizQuestion associated with it. Every QuizQuestion must have multiple QuizAnswers, ensuring that quizzes are properly structured and provide multiple choices for answers.

Unique Identification (User, Quiz, Tournament)

- **Constraint**: Each User, Quiz, and Tournament must have a unique identifier (ID). This ensures that no two users, quizzes, or tournaments share the same identifier, maintaining the integrity of the database and allowing for reliable access and referencing.

Role-Based Access Control:

- The system enforces role-based access control, restricting actions based on the user's role (Student, Teacher, Admin). For example, only Professors can create and manage quizzes, while Students are limited to taking quizzes and viewing their results.

3.3.3 Architecture Background

The data model provides the foundation for managing the various components of the quiz system. It includes entities such as **User**, **Quiz**, **Question**, **QuizAnswers**, **Tournaments**, and **CourseExecution**, each serving a specific role in the system. These entities are connected by relationships, with **User** interacting with **QuizAnswers**, **Discussion**, and **Dashboard**, and **Quiz** connected to **Question** and **QuizAnswers**.

The model supports complex workflows, like handling user submissions, tracking tournament results, and managing course execution data. The architecture allows for flexible interactions, such as retrieving quiz data, storing answers, and managing feedback. It also supports scalability by accommodating future additions, like new quiz types or expanded user roles. Constraints are applied to maintain data integrity and ensure consistent access control, providing a solid base for system growth and future enhancements.

3.3.4 Variability Mechanisms

The data model incorporates variability mechanisms to support flexible and scalable functionality. Key mechanisms include:

QuizType: Allows different quiz formats (e.g., multiple choice, short answer) to be easily added.

CourseExecution: Associates quizzes with specific courses, supporting variability in course contexts.

Tournaments: Supports different tournament formats and quiz structures.

Topics: Enables categorization of quizzes and questions, offering flexibility in content organization.

User Roles (AuthUser): Manages different user roles (students, admins), allowing customizable permissions.

Dynamic Question Management: Enables easy modification of quiz questions and answers to meet evolving needs.

These mechanisms ensure the system can adapt to future changes and scale effectively while maintaining flexible user and quiz configurations.

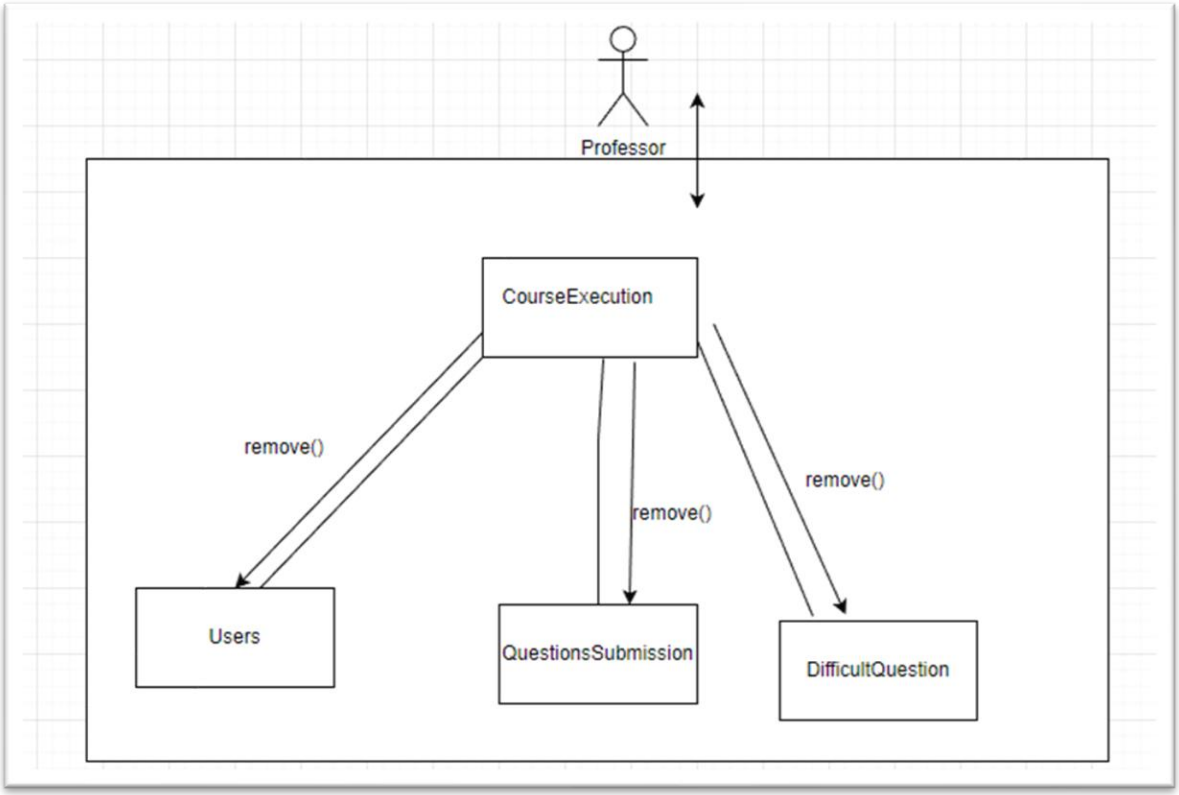
3.4 Call Return View

3.4.1 View Description

The Call-Return View models the interactions between components where control and data flow from one component to another, with control being returned once the task is completed. This view captures the sequence of method calls and responses, illustrating the dynamic flow of execution within the system. It is essential for understanding how different components collaborate in a coordinated manner, ensuring that tasks are executed in the correct order and dependencies are respected. By mapping these interactions, this view helps identify potential bottlenecks, delays, or areas where optimization is needed to improve system performance.

3.4.2 Primary Presentation

Figure 16 - Call Return View



3.4.2.1 Element Catalog

3.4.2.1.1 Elements

- **Professor (User)**: Represents the professor user who interacts with the system.
- **CourseExecution**: Manages the execution of courses, coordinating related activities.
- **Users**: Handles the management of system users (e.g., students, professors).
- **QuestionSubmission**: Manages the submission and storage of questions.
- **DifficultQuestion**: Handles the identification and management of difficult questions.

3.4.2.1.2 Relations

- **Professor ↔ CourseExecution**: Bidirectional relationship indicating that the professor interacts with the CourseExecution, both sending and receiving information or control.

- **CourseExecution → Users**: "Calls" relationship where CourseExecution interacts with the Users component, to manage user data (e.g., removing a user from a course).

- **CourseExecution → QuestionSubmission**: "Calls" relationship where CourseExecution interacts with QuestionSubmission, for adding or removing questions during course execution.

- **CourseExecution → DifficultQuestion**: "Calls" relationship indicating that CourseExecution can interact with the DifficultQuestion component to handle or remove difficult questions during the course execution.

3.4.2.1.3 Interfaces

Professor ↔ CourseExecution:

- The professor interacts with CourseExecution through a front-end interface. This allows the professor to start, pause, or complete the course execution, as well as view or modify the course execution status.

CourseExecution ↔ Users:

- CourseExecution communicates with the Users component via REST API endpoints to manage user data, such as adding, removing, or updating users (e.g., enrolling or unenrolling a user from a course).

CourseExecution ↔ QuestionSubmission:

- CourseExecution interacts with QuestionSubmission via REST API calls, allowing it to send or receive questions. This can involve adding new questions or removing them during course execution.

CourseExecution ↔ DifficultQuestion:

- The interaction between CourseExecution and DifficultQuestion occurs through REST API calls, enabling CourseExecution to mark, remove, or adjust difficult questions during the course execution.

3.4.2.1.4 Behaviour

Professor ↔ CourseExecution:

- When the professor interacts with CourseExecution, the professor can initiate the execution of a course (start, pause, resume, or end the course). This may involve querying the status of the course, viewing current user progress, or modifying the flow (e.g., skipping questions or adjusting the course content).
- Example: When the professor sends a request to start the course execution, the system triggers a sequence of actions, such as loading users, retrieving questions, and preparing the course environment.

CourseExecution ↔ Users:

- CourseExecution manages the enrolment and removal of users through the Users component. When a user is added or removed, CourseExecution may update the user status in the context of the course (e.g., enrolled, completed, or removed from the course).
- Example: If a user is removed from the course, CourseExecution triggers a "remove" action, which updates the system's user data and ensures the user is no longer part of the course.

CourseExecution ↔ QuestionSubmission:

- CourseExecution interacts with QuestionSubmission to fetch, display, or remove questions. When a question is submitted, CourseExecution evaluates it for inclusion in the course or quiz. If a question is marked for removal, CourseExecution triggers an action to delete it from the active set.
- Example: When a question is marked for removal, CourseExecution ensures the question is deleted from the course flow, preventing it from being displayed to users.

CourseExecution ↔ DifficultQuestion:

- When interacting with DifficultQuestion, CourseExecution identifies and handles difficult questions. If a question is flagged as difficult, CourseExecution may adjust how it is presented to users (e.g., presenting it as part of a challenge or skipping it for specific users).
- Example: If a question is deemed difficult, CourseExecution could decide to remove it temporarily or mark it for review by the professor before it is presented to students again.

3.4.2.1.5 Constraints

Access Control:

Only authorized users (e.g., professors) can perform certain actions like removing users or questions from the course. The CourseExecution component must enforce

access control to ensure that only users with the appropriate role have permission to modify data.

Uniqueness of User Management:

A **Users** component must ensure that each user is unique (e.g., no duplicate user IDs or email addresses) when interacting with **CourseExecution**. This ensures proper tracking and identification of users within courses.

3.4.3 Architecture Background

The architecture of the system is based on a modular design, where each component has a distinct responsibility and communicates with other components via REST APIs. This design allows for a clear separation of concerns, which simplifies development, maintenance, and scalability.

The **Call-Return View** models the interactions between the components as a series of method calls and responses. In this model, components communicate in a controlled sequence, with the control being returned to the caller once the task is completed. This approach is well-suited for this system, where tasks such as course execution, user management, and question handling need to occur in a defined order to ensure the system operates as expected.

The use of **REST APIs** facilitates communication between the frontend and backend, ensuring that data flows efficiently between components while maintaining a scalable and flexible architecture. The API-based interactions also support real-time updates, such as the removal of users or questions from a course, which is crucial for maintaining data consistency and ensuring that the system reflects the most up-to-date state.

3.4.4 Variability Mechanisms

The system allows for variability in user roles, enabling different interactions with components based on the role of the user (e.g., professor, student, administrator). This is controlled through a user management system that adjusts access rights and privileges dynamically based on the user's role.

Example: The professor could remove users or questions from a course, while students have limited access, allowing them only to view content and submit answers.

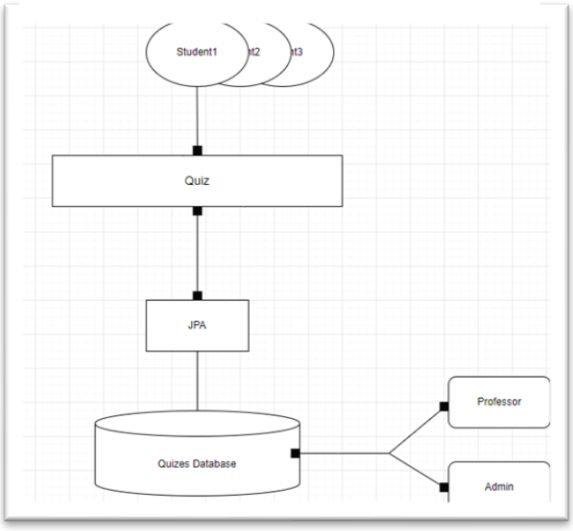
3.5 Repository View

3.5.1 View Description

The Repository View focuses on the central repository where persistent data is stored, managed, and retrieved. It outlines how different components interact with the repository to access and update data, ensuring consistency across the system. This view is key to understanding how data is shared and organized. It highlights the relationship between the repository and other system components, ensuring efficient data flow. By centralizing storage, it ensures that all components can access the required information consistently.

3.5.2 Primary Presentation

Figure 17 - Repository View



3.5.2.1 Element Catalog

3.5.2.1.1 Elements

- **Students:** Users who interact with quizzes to participate and submit answers.
- **Quiz:** Main component that manages the logic for interaction with quizzes and users.
- **JPA:** Persistence layer that translates Java objects into relational database operations.
- **Quizzes Database:** Database that stores persistent data such as quizzes, users, answers, and results.
- **Professor:** User who manages quizzes, creates questions, and monitors students' performance.

- **Admin:** User responsible for managing the overall system administration, including users and permissions.

3.5.2.1.2 Relations

The Repository View describes the "stored" and "retrieved" relations between the system and the data repository. The persistence layer (JPA) manages all read and write operations to the database, ensuring that data is accessed in a consistent and efficient manner. This centralized structure enables system operations to be carried out in an organized and controlled way, maintaining data integrity and consistency over time.

3.5.2.1.3 Interfaces

Students interact with the Quiz through HTTP requests, participating in quizzes and submitting answers.

The Quiz uses JPA to access the Quizzes Database, where quizzes, questions, student responses, and results are stored.

JPA manages read and write operations between the backend and the database.

Professors and Admins manage quizzes and users, with permissions to create and monitor questions and results, also interacting with the system via HTTP requests.

3.5.2.1.4 Constraints

- **Data Consistency:** Ensure that read and write operations on the database are consistent, without data corruption or duplication.
- **Permission Security:** Only Professors and Admins can create, edit, or delete quizzes and questions.
- **Performance and Efficiency:** Operations must be efficient, even under high user traffic.
- **Referential Integrity**: Ensure related data remains consistent, and deleting data does not cause errors.
- **Data Access Limitation:** Restrict data operations based on user roles, such as limiting Students to viewing quizzes and results only.

3.5.3 Architecture Background

The architecture of Quizz Tutor in the Repository View centralizes data management using the JPA persistence layer to interact with the Quizzes Database. Communication between users and the system happens through HTTP requests, triggering RESTful APIs to access and modify data.

3.5.4 Variability Mechanisms

The Repository View allows flexibility through modular components, enabling updates without disruption. The JPA persistence layer supports easy database changes, while the separation of data from the frontend ensures independent evolution of backend capabilities, promoting scalability and future growth.

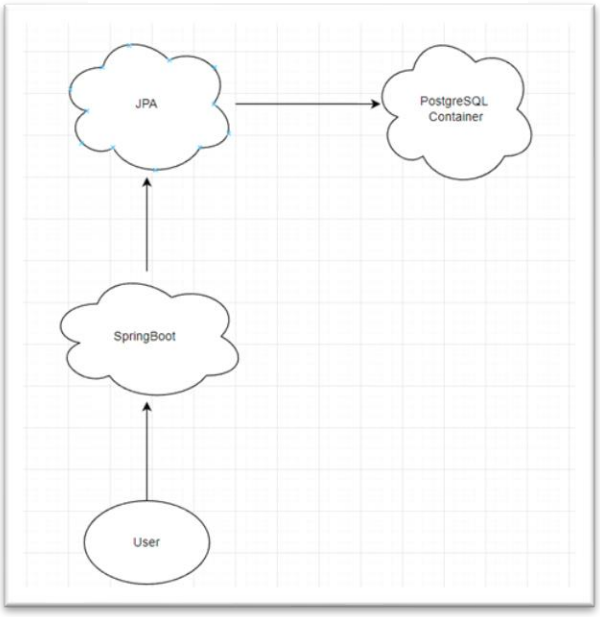
3.6 Deployment View

3.6.1 View Description

The Deployment View focuses on how the system's components are physically distributed across various infrastructure elements such as servers, networks, and databases. It illustrates how these components interact and communicate within the environment, ensuring efficient data flow and scalability. This view is essential for understanding the system's operational setup, including configuration across development, testing, and production environments

3.6.2 Primary Presentation

Figure 18 - Deployment View



3.6.2.1 Element Catalog

3.6.2.1.1 Elements

- **User:** The end user interacting with the system, accessing it through the SpringBoot application.
- **SpringBoot:** A framework that serves as the backend server, running the application logic and providing access to other components.
- **JPA:** The Java Persistence API that facilitates interaction between the SpringBoot application and the PostgreSQL container.
- **PostgreSQL Container:** A containerized database system (PostgreSQL) that stores all the application's data persistently, and interacts with JPA for data operations.

3.6.2.1.2 Relations

The system's deployment is structured around key components with specific is-deployed-on and communicates-with relationships, where SpringBoot is deployed on a server, communicating with JPA and the PostgreSQL Container, while Users interact with SpringBoot via HTTP requests.

3.6.2.1.3 Interfaces

SpringBoot exposes RESTful API endpoints that allow Users to interact with the system via HTTP requests, enabling actions like submitting quizzes, retrieving results, etc.

SpringBoot communicates with JPA through Java-based interfaces to manage data interactions, ensuring that data is properly retrieved and stored in the PostgreSQL Container.

JPA interfaces with the PostgreSQL Container using SQL-based queries to perform CRUD operations on the database.

3.6.2.1.4 Constraints

- **Scalability:** The system should allow the addition of more servers or instances to support higher load without affecting performance.
- **Resilience:** Components must be resilient to failures, ensuring the system continues to function even if one component fails.
- **Security:** Communication between components must be secure, using encryption and authentication.
- **Isolation and Data Consistency:** PostgreSQL and SpringBoot should be isolated to prevent interference, ensuring efficient data exchange without loss.
- **Maintainability:** The system should be modular, allowing updates and maintenance without impacting other components.

3.6.3 Architecture Background

The Quizz Tutor architecture in the Deployment View is centered around SpringBoot deployed on a server, which handles HTTP requests and communicates with the PostgreSQL database container. This setup ensures efficient data management, scalability, and flexibility, with isolated components that allow smooth interaction between the backend and users.

3.6.4 Variability Mechanisms

The Deployment View supports flexibility by isolating components like SpringBoot and PostgreSQL, allowing independent scaling and updates. This modular design ensures the system can grow and adapt to future needs without disrupting performance.

4 Relations Among Views

Each of the views specified in Section 3 provides a different perspective and design handle on a system, and each is valid and useful in its own right. Although the views give different system perspectives, they are not independent. Elements of one view will be related to elements of other views, and we need to reason about these relations. For example, a module in a decomposition view may be manifested as one, part of one, or several components in one of the component-and-connector views, reflecting its runtime alter-ego. In general, mappings between views are many to many. Section 4 describes the relations that exist among the views given in Section 3. As required by ANSI/IEEE 1471-2000, it also describes any known inconsistencies among the views.

4.1 General Relations Among Views

In the **Quizzes Tutor** project, each view offers a unique perspective on the system's architecture. However, these views are not independent, as elements from one view often interact and correspond with elements in other views. Below are the general relationships among the key views of the system:

- **Decomposition View ↔ Uses View**
- **Decomposition View ↔ Data Model View**
- **Uses View ↔ Data Model View**
- **Call-Return View ↔ Repository View**
- **Call-Return View ↔ Data Model View**
- **Deployment View ↔ Decomposition View**
- **Deployment View ↔ Users View**

4.2 View-to-View Relations

Below are the specific relationships between elements of the different views, demonstrating how they interact to form a cohesive and consistent system:

Decomposition View ↔ Uses View:

Relation: Modules in the Decomposition View correspond to functional dependencies in the Uses View.

Example: The "Quiz Management" module depends on the "User Authentication" module to verify users.

Decomposition View ↔ Data Model View:

Relation: Modules interact with data entities described in the Data Model View.

Example: The "Quiz Submissions" module uses entities like "Submissions" and "Users."

Uses View ↔ Data Model View:

Relation: Functional dependencies between modules involve data entities from the Data Model View.

Example: The "Results Calculation" module depends on entities like "Answers" and "Grades."

Call-Return View ↔ Repository View:

Relation: Components in the Call-Return View perform operations on repositories defined in the Repository View.

Example: A component responsible for saving quiz results invokes repository operations to store them.

Call-Return View ↔ Data Model View:

Relation: Components utilize data entities described in the Data Model View during runtime interactions.

Example: A component processing quiz answers uses data entities like "Questions" and "Answers."

Deployment View ↔ Decomposition View:

Relation: The Deployment View allocates the modules described in the Decomposition View to execution nodes (servers, virtual machines, containers).

Example: The quiz management module is deployed on servers or containers.

Deployment View ↔ Uses View

Relation: The infrastructure in the Deployment View supports the functional dependencies between the modules described in the Uses View.

Example: The user authentication module is deployed on servers capable of processing authentications quickly.

Deployment View ↔ Data Model View:

Relation: The Deployment View allocates the databases and data entities described in the Data Model View.

Example: The PostgreSQL database is deployed on database servers or cloud infrastructure.

5 Referenced Materials

Table 4: Reference Materials

Barbacci 2003	Barbacci, M.; Ellison, R.; Lattanze, A.; Stafford, J.; Weinstock, C.; & Wood, W. *Quality Attribute Workshops (QAWs)*, Third Edition (CMU/SEI-2003-TR-016). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr016.html>.
Bass 2003	Bass, Clements, Kazman, *Software Architecture in Practice*, second edition, Addison Wesley Longman, 2003.
Clements 2001	Clements, Kazman, Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison Wesley Longman, 2001.
Clements 2002	Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord, Stafford, *Documenting Software Architectures: Views and Beyond*, Addison Wesley Longman, 2002.
IEEE 1471	ANSI/IEEE-1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, 21 September 2000.
Main references of the project	**GitHub Repository:** Quizzes Tutor. Available at https://github.com/socialsoftware/quizzes-tutor Video Domain Model: Explainer Video of the Architecture of Quizzes Tutor, available at YouTube
Tools of analysis and code metrics	SonarQube Documentation: Code analysis platform used to evaluate the quality of the project's source code. Documentation available at (https://docs.sonarqube.org/latest/.)
References for Examples and Case Studies	DESOSA: Examples of software architecture projects carried out by master's students at the University of Delft: DESOSA 2019, DESOSA 2020, DESOSA 2021, DESOSA 2022.

6 Directory

6.1 Glossary

Table 5: Glossary

Term	Definition
software architecture	The structure or structures of that system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 2003]. "Externally visible" properties refer to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.
view	A representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. A representation of a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A view conforms to a defining viewpoint.
viewpoint	A specification of the conventions for constructing and using a view; a pattern or template from which to develop individual views by establishing the purposes and audience for a view, and the techniques for its creation and analysis [IEEE 1471]. Identifies the set of concerns to be addressed, and identifies the modeling techniques, evaluation techniques, consistency checking techniques, etc., used by any conforming view.
maintainability	The ease with which the system can be modified to fix defects, improve performance, or adapt to a changing environment.

modularity	The degree to which the system is composed of separate and independent components that can be altered without affecting the rest of the system
reliability	Ensures the system consistently performs its core functions without errors.
scalability	The ability of the system to handle an increased workload or user expansion while maintaining functionality and performance.
Frontend	The part of the system that interacts directly with the user, usually responsible for the graphical user interface.
Backend	The part of the system responsible for business logic, data processing, and integration with data storage.
Authentication	Verifies the identity of users before granting access to the system.
Horizontal Scaling	Increases system capacity by adding more machines or server instances.
Database Connection Pooling	A mechanism to reuse database connections for efficient access and performance improvement
Data Integrity	Ensures that stored and transmitted data remains consistent and unaltered.
Encryption	The process of encoding information to ensure security and prevent unauthorized access.
Caching	Temporarily stores frequently accessed data to speed up future requests.
Clustering	Connects multiple servers to work as one, improving availability and fault tolerance.

Load Balancing	Distributes traffic across multiple servers to prevent overload and ensure reliability
Quality Attribute	A property or characteristic of a system that impacts its ability to meet stakeholder needs.

6.2 Acronym List

Table 6: Acronym List

API	Application Programming Interface; Application Program Interface; Application Programmer Interface
ATAM	Architecture Tradeoff Analysis Method
COTS	Commercial-Off-The-Shelf
IEEE	Institute of Electrical and Electronics Engineers
QAW	Quality Attribute Workshop
RUP	Rational Unified Process
SAD	Software Architecture Document
SEI	Software Engineering Institute Systems Engineering & Integration Software End Item
UML	Unified Modeling Language
CI/CD	Continuous Integration/Continuous Deployment
ORM	Object-Relational Mapping
CDN	Content Delivery Network
SQL	Structured Query Language
ERD	Entity-Relationship Diagram
CRUD	Create, Read, Update, Delete