

**Grupo 4 – Faculdade de Ciências   
Quizzes Tutor   
Software Architecture Document (SAD)**

**CONTENT OWNER: Leonardo Monteiro Nº58250;Henrique Vale Nª58168; Gustavo Henriques Nº64361**

**Amarelo – É o que já foi colocado no documento**

**Roxo – É o que ainda está para ser posto**

|  |  |  |
| --- | --- | --- |
| **DOCUMENT NUMBER** | **RELEASE/REVISION DATE** | **RELEASE/REVISION** |
| 1.0 | 28/10/2024 | **Initial Release:** Created the *Documentation Roadmap*, defined *Purpose and Scope*. |
| 1.1 | 02/11/2024 | **First Update:** Added *Stakeholder Representation* and *Viewpoint Definitions*. |
| 1.2 | 06/11/2024 | **Updates:** Completed *Architecture Background* (*Problem Background*, *Goals and Context*, *Driving Requirements*). |
| 1.3 | 09/11/2024 | **Updates:** Began drafting *Solution Background* (*Architectural Approaches*, *Analysis Results*, *Requirements Coverage*). |
| 1.4 | 12/11/2024 | **More Updates:** Completed *Solution Background* (including *Summary of Background Changes Reflected*) and initiated *Views*. |
| 1.5 | 15/11/2024 | **More Updates:** Documented *Module Views* (*Decomposition*, *Uses*, *Data Model*). |
| 1.6 | 17/11/2024 | **Last Updates:** Documented *Component-and-Connector Views* (*Call-Return*, *Repository*). |
| 1.7 | 21/11/2024 | **Updates:** Finalized *Allocation Views* (*Execution*, *Development*). |
| 1.8 | 23/11/2024 | **Revisions:** Reviewed and refined all *Views* (*Module*, *Component-and-Connector*, and *Allocation*) and added *Relations Among Views*. |
| 1.9 | 26/11/2024 | **Enhancements:** Finalized *Relations Among Views* and completed *Referenced Materials* and *Glossary*. |
| 2.0 | 28/11/2024 | **Final Release:** Completed *Architecture Background* (*Problem Background*, *Goals and Context*), ensuring alignment with all sections. |

# 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.

## *Document Management and Configuration Control Information*

***Revision Number:***

***Revision Release Date:***

***Purpose of Revision:***

**Scope of Revision:**.

## Purpose and Scope of the SAD

### 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 serves as a central reference for the system’s development, analysis, and maintenance, supporting effective stakeholder communication and ensuring a shared understanding of its design.

### Scope

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

**Stakeholders and Their Interests**: Identification of main stakeholders (teachers, students, development team, and system administrators), 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:**
  + *Execution View*: Maps software components to physical or virtual infrastructure, ensuring scalability and reliability.
  + *Development View*: Outlines the organization of development artifacts, such as source code and dependencies, to facilitate collaboration and project management.

This document focuses on high-level architectural decisions and component interactions, leaving detailed technical specifications and implementation guidelines to supplementary documentation. This approach provides a top-down view, making the system’s architecture accessible without delving into low-level code details.

## 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 (“Directory”) 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

## 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:

* **Module Viewpoint**: Focuses on the static structure, dividing the system into modules.

**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.

* **Component-and-Connector Viewpoint**: Highlights runtime interactions, showing data flow and communication.

**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 deployment infrastructure.

**Execution View**: Maps software components to the execution infrastructure, such as physical servers or cloud environments, to support real-time operation.

**Development View**: Shows the organization of development artifacts, like source code, to facilitate team coordination and management.

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

*Table 1:* *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** |
| Development Team | Module Viewpoint: **Decomposition View**, **Uses View**, **Data Model View**  Component-and-Connector Viewpoint: **Call-Return**  Allocation Viewpoint: **Development View** |
| System Administrators | |  | | --- | |  | | Component-and-Connector Viewpoint: **Repository**  Allocation Viewpoint: **Execution View**  Module Viewpoint: **Data Model View** | |
| Project Manager | Module Viewpoint: **Decomposition View** |

### 1.5.1 Module Viewpoint Definition

### Decomposition Viewpoint Definition

1. **Abstract**
2. 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.
3. **Stakeholders and Their Concerns Addressed**
   1. **Project Managers**: Need a clear view of the modules to plan tasks and organize the team's work.
   2. **Development Team**: Benefits from a clear modular structure to facilitate incremental development and maintenance.
   3. **Teachers**: Require an understanding of the main modular functionalities to support their use of the system.
4. **Elements, Relations, Properties, and Constraints**
   1. **Elements**: Modules, submodules, and their specific functionalities.
   2. **Relations**: “Is-part-of” relation representing the hierarchy and modular structure of the system.
   3. **Properties**: Each module’s name, main function, and software interface.
   4. **Constraints**: Each module should be responsible for a distinct function and not overlap responsibilities with other modules.
5. **Language(s) to Model/Represent Conforming Views**
   1. **UML Class Diagrams** or **Component Diagrams** to illustrate hierarchical structure and relations of each module.
6. **Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria**
   1. **Consistency**: Each element should have only one parent module.
   2. **Completeness**: The functionality of all modules combined should cover the system requirements.
   3. **Evaluation Techniques**: Scenario-based analysis to ensure modularity allows changes with minimal impact.
7. **Viewpoint Source**
   1. Clements, P., et al. *Documenting Software Architectures: Views and Beyond* (2002).

### Uses Viewpoint Definition

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. **Stakeholders and Their Concerns Addressed**
   1. **Development Team**: Needs to understand module dependencies to avoid integration issues.
2. **Elements, Relations, Properties, and Constraints**
   1. **Elements**: Modules with specific functions and functional dependencies.
   2. **Relations**: “Depends-on” relations defining execution dependencies between modules.
   3. **Properties**: Each module’s name, function, and associated dependencies.
   4. **Constraints**: Dependent modules must be compatible in terms of interface and communication.
3. **Language(s) to Model/Represent Conforming Views**
   1. **UML Dependency Diagrams** or **Sequence Diagrams** to show interactions and functional dependencies.
4. **Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria**
   1. **Consistency**: All listed dependencies should be valid and documented.
   2. **Completeness**: All critical interactions for functionality should be represented.
   3. **Evaluation Techniques**: Review dependencies to identify potential coupling issues.
5. **Viewpoint Source**
   1. Clements, P., et al. *Documenting Software Architectures: Views and Beyond* (2002).

### Data Model Viewpoint Definition

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. **Stakeholders and Their Concerns Addressed**
   1. **Development Team**: Requires a clear data structure to correctly implement and integrate storage and retrieval operations.
   2. **System Administrators**: Concerned with data integrity and security to ensure proper system functioning.
2. **Elements, Relations, Properties, and Constraints**
   1. **Elements**: Data entities, such as tables and columns in the database.
   2. **Relations**: “Association” and “dependency” relations between entities (e.g., foreign key).
   3. **Properties**: Each entity’s attributes, including data types and constraints.
   4. **Constraints**: All relationships must maintain referential integrity and data consistency.
3. **Language(s) to Model/Represent Conforming Views**
   1. **Entity-Relationship Diagrams (ERD)** or **UML Class Diagrams** to represent the data structure and relationships between entities.
4. **Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria**
   1. **Consistency**: All data relationships should be consistent and follow integrity constraints.
   2. **Completeness**: All necessary entities and relationships for data storage should be represented.
   3. **Evaluation Techniques**: Referential integrity check and normalization analysis to avoid data redundancy.
5. **Viewpoint Source**
   1. Clements, P., et al. *Documenting Software Architectures: Views and Beyond* (2002).

### Call-Return Viewpoint Definition

1. **Abstract**
2. 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. This style is typical in client-server or peer-to-peer architectures, where components rely on specific calls to perform tasks.
3. **Stakeholders and Their Concerns Addressed**
   1. **Development Team**: Requires a clear understanding of component interactions and control flow to develop, debug, and optimize system performance.
   2. **Students**: Interested in understanding how system components work together, which enhances their user experience during runtime.
4. **Elements, Relations, Properties, and Constraints**
   1. **Elements**: Components that act as invokers or receivers in control flows, such as clients and servers.
   2. **Relations**: “Calls” relation, where an invoker component makes a call to another, transferring control and possibly data.
   3. **Properties**: Each component’s role (invoker or receiver), and data involved in the call.
   4. **Constraints**: Each call must follow a compatible interface for control and data exchange, and components should maintain state consistency.
5. **Language(s) to Model/Represent Conforming Views**
   1. **Sequence Diagrams** or **Activity Diagrams** to show the sequence and flow of control and data exchange among components.
6. **Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria**
   1. **Consistency**: All control flows must be compatible with the designated interfaces.
   2. **Completeness**: All necessary calls to achieve full functionality must be represented.
   3. **Evaluation Techniques**: Performance analysis of control flows to ensure low-latency interaction; scenario-based testing to validate control dependencies.
7. **Viewpoint Source**
   1. Based on standard client-server and peer-to-peer interaction models as described in *Documenting Software Architectures: Views and Beyond* by Clements et al.

### Repository Viewpoint Definition

1. **Abstract**
2. 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.
3. **Stakeholders and Their Concerns Addressed**
   1. **System Administrators**: Concerned with the data repository's stability, integrity, and security to maintain overall system health.
4. **Elements, Relations, Properties, and Constraints**
   1. **Elements**: Central data repositories (e.g., databases), data access components, and storage structures.
   2. **Relations**: “Stores” and “retrieves” relations showing interactions between data storage elements and components accessing them.
   3. **Properties**: Each data repository’s storage type, access protocols, and security measures.
   4. **Constraints**: All data access should ensure data integrity, security, and support for concurrent data requests without conflicts.
5. **Language(s) to Model/Represent Conforming Views**
   1. **Entity-Relationship Diagrams (ERD)** or **UML Component Diagrams** to represent data storage structures and data flow between components and repositories.
6. **Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria**
   1. **Consistency**: All data access methods should comply with data repository constraints.
   2. **Completeness**: The repository should include all essential data entities for system functionality.
   3. **Evaluation Techniques**: Integrity checks, performance analysis to handle data load, and security analysis to prevent unauthorized access.
7. **Viewpoint Source**
   1. Inspired by repository-centric designs for shared-data environments as documented in *Documenting Software Architectures: Views and Beyond* by Clements et al.

### Execution Viewpoint

**Abstract**

The Execution View maps the system's software components to the underlying infrastructure, such as web servers, application servers, and databases, to ensure real-time operation, scalability, and reliability. This viewpoint ensures that the system can operate efficiently across distributed environments, leveraging cloud-based or on-premises infrastructure.

**Stakeholders and Their Concerns Addressed**

* **System Administrators**: Need a clear understanding of deployment strategies, server configurations, and resource allocation to maintain the system's stability.

**Elements, Relations, Properties, and Constraints**

* **Elements**: Application servers, database servers, front-end components, and communication protocols.
* **Relations**: "Is-deployed-on" relationship mapping software components to execution nodes and "communicates-with" relationships showing data flow between components.
* **Properties**: Latency requirements, resource usage (e.g., CPU, memory), and database query performance.
* **Constraints**: Infrastructure must meet high availability requirements to ensure quiz access and must support load balancing during peak times.

**Language(s) to Model/Represent Conforming Views**

* UML Deployment Diagrams for mapping software to execution environments.
* Infrastructure-as-Code (e.g., Docker Compose or Kubernetes YAML files) to represent cloud-based deployment configurations.

**Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria**

* **Consistency**: All components must be associated with an infrastructure node.
* **Completeness**: Execution view should represent all components necessary for system functionality.
* **Evaluation**: Load testing, scalability tests, and failover scenario simulations to ensure system stability and reliability.

**Viewpoint Source**

Derived from Quizzes Tutor deployment documentation and cloud architecture practices.

### Development Viewpoint

**Abstract**

The Development View organizes development artifacts, such as source code, modules, libraries, and build tools, to support team collaboration, modularity, and maintainability. This viewpoint facilitates efficient implementation, testing, and future system enhancements.

**Stakeholders and Their Concerns Addressed**

* **Development Team**: Requires a well-structured codebase to manage dependencies and collaborate effectively.

**Elements, Relations, Properties, and Constraints**

* **Elements**: Front-end and back-end modules, APIs, database migration scripts, and test suites.
* **Relations**: "Depends-on" relations between modules, libraries, and build tools.
* **Properties**: Ownership of modules, versioning of artifacts, and coverage of test cases.
* **Constraints**: Codebase must adhere to defined coding standards and avoid circular dependencies.

**Language(s) to Model/Represent Conforming Views**

* UML Component Diagrams to represent dependencies and modular architecture.
* Repository structure diagrams to model folder organization and version control branching.

**Applicable Evaluation/Analysis Techniques and Consistency/Completeness Criteria**

* **Consistency**: Ensure no conflicting dependencies exist and version control is maintained.
* **Completeness**: Ensure all required artifacts are included for implementation and testing.
* **Evaluation**: Dependency analysis, CI/CD pipeline validation, and automated testing reports.

**Viewpoint Source**

Based on Quizzes Tutor development practices and project repository analysis.

### 

## 1.6 How a View is Documented

This section provides the structure and organization for documenting each view in Section 3 of the Software Architecture Document (SAD). If the format or layout in Section 3 is altered, it is essential to update this description accordingly to ensure consistency. This section acts as a standardized guide, which allows all views to follow a uniform documentation process, thus enhancing readability and comprehension across all stakeholder groups.

### Single-Presentation View Documentation

If a view’s documentation is presented in a single section without subdividing into view packets, the structure will be as follows:

* **Section 3.i: Name of view** – Specifies the view (e.g., *Module View*, *Component-and-Connector View*, etc.).
* **Section 3.i.1: View description** – Outlines the purpose and content of the view.
* **Section 3.i.2: Primary presentation** – Displays the main elements and their relationships using appropriate tools or notation.
* **Section 3.i.3: Element catalog** – Complements the primary presentation with further details on elements, relations, interfaces, behavior, and constraints.
* **Section 3.i.4: Context diagram** – Provides a diagram illustrating the context of the view within the system, marking its scope and interactions.
* **Section 3.i.5: Variability mechanisms** – Details any configuration or adaptation options within this view.
* **Section 3.i.6: Architecture background** – Explains rationale for major design decisions limited to this view.

### View Packet-Based Documentation

For views that require division into *view packets* to accommodate complexity, each view will be documented with the following structure:

* **Section 3.i: Name of view**
* **Section 3.i.1: View description** – Aligns with the viewpoint described in Section 1.5.
* **Section 3.i.2: View packet overview** – Lists all view packets within the view and explains why this set is comprehensive.
* **Section 3.i.3: Architecture background** – Describes relevant design decisions, requirements, and architectural choices specific to this view.
* **Section 3.i.4: Variability mechanisms** – Notes architectural mechanisms specific to this view, if applicable.
* **Section 3.i.5: View packets** – Each view packet follows the outline below:
  + **Section 3.i.5.j: View packet #j**
    - **Section 3.i.5.j.1: Primary presentation** – Depicts the core elements and their relationships.
    - **Section 3.i.5.j.2: Element catalog**:
      * **Elements** – Details the elements in the view packet.
      * **Relations** – Additional relations or constraints.
      * **Interfaces** – Visible software interfaces for each element.
      * **Behavior** – Any significant behavior of elements or interactions.
      * **Constraints** – Any specific constraints not covered earlier.
    - **Section 3.i.5.j.3: Context diagram** – Shows the scope and external interactions of this view packet.
    - **Section 3.i.5.j.4: Variability mechanisms** – Lists available configuration mechanisms.
    - **Section 3.i.5.j.5: Architecture background** – Rationale for design decisions specific to this packet.
    - **Section 3.i.5.j.6: Relation to other view packets** – References to related packets.

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

## 1.7 Relationship to Other SADs

Not applicable.

## 1.8 Process for Updating this SAD

Not applicable.

# Architecture Background

## 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.

### 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.

### 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.

### 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. Other quality attributes, such as availability and maintainability, were evaluated using methods like ATAM(Architecture Tradeoff Analysis Method,), 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.

### 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.

**Considered and Rejected Alternatives**:

**Frontend Frameworks**: React and Angular were considered, but Vue.js was chosen for simplicity and ease of integration.

**Databases**: MySQL and MongoDB were reviewed, but PostgreSQL was selected for handling complex transactions and ensuring data consistency.

**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.

### Analysis Results

ISTO ESTAVA TUDO MAL MAS JÁ ESTÁ CERTO NO DOCUMENTO RPINCIPAL

### 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)**:

**Quiz Creation and Management**: 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.

### Summary of Background Changes Reflected in Current Version

This section provides an overview of the architectural changes and their rationale since the initial release of the Software Architecture Document (SAD) for the Quizzes Tutor project. These changes have been influenced by ongoing analysis, requirement adjustments, and optimization opportunities identified through testing and stakeholder feedback.

**Improved Modularity**: Based on maintainability analysis with tools like SonarQube, certain parts of the code were refactored to improve modularity. This change enhances the ease of future updates and reduces interdependencies, allowing components to be updated independently.

**Enhanced Security Measures**: In response to stakeholder concerns and security assessments, the system now incorporates additional data protection measures, particularly in user authentication and data storage. This change was driven by the need to better protect sensitive information and align with updated security standards.

**Performance Optimization**: Performance testing revealed areas where system response times could be improved. As a result, database queries were optimized, and certain backend processes were restructured. These optimizations were necessary to ensure that the system can handle increased user load during peak times, enhancing scalability and user experience.

**Scalability Adjustments**: As part of the scalability strategy, the architecture has been updated to allow easier scaling by adding backend server instances as needed. This flexibility was essential to accommodate growing usage and was informed by load testing and anticipated growth.

These updates reflect a commitment to maintaining a robust and adaptive architecture that meets evolving project needs and aligns with quality standards in performance, security, and maintainability.

## 2.3 Product Line Reuse Considerations

Not applicable.

3. Views

**CONTENTS OF THIS SECTION**: 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:
   1. What is the primary functional responsibility assigned to each module?
   2. What other software elements is a module allowed to use?
   3. Which modules are related to others by generalization or specialization?

The subtypes included in this category are:

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

1. **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:
   1. What are the major executing components, and how do they interact?
   2. Which parts of the system can operate in parallel, and how does data flow through the system?

The subtypes included in this category are:

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

1. **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:
   1. Where is each software element deployed in terms of physical or virtual infrastructure?
   2. How are resources assigned to development and operations teams?

The subtypes included in this category are:

* 1. **Execution View**: Maps software components to the execution infrastructure, such as physical servers or cloud environments, to support real-time operation.
  2. **Development View**: Shows the organization of development artifacts, such as source code, to facilitate team coordination and management.

**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 document are as follows:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 | ISTO ESTAVA MAL | yes | no | no |
| Uses View | Functional dependencies between modules | ISTO ESTAVA MAL | yes | no | no |
| Data Model View | Data entities and relationships | ISTO ESTAVA MAL | yes | no | no |
| Call-Return View | Control flow between components | ISTO ESTAVA MAL | no | yes | no |
| Repository View | Data storage components and access connectors | |  | | --- | |  | | ISTO ESTAVA MAL | | no | yes | no |
| Execution View | Software components to execution infrastructure | ISTO ESTAVA MAL | no | no | yes |
| Development View | Development artifacts and source code organization | ISTO ESTAVA MAL | no | no | yes |

### 3.1 Module View

#### 3.1.1 Decomposition View

* **View Description**
* The Decomposition View presents the hierarchical structure of the system, organizing it into modules and submodules. This view highlights the responsibilities assigned to each module, providing a clear understanding of the system’s division into functional parts.
* **View Packet Overview**

This view has been divided into the following view packets for convenience of presentation:

* + **Frontend Module**: Manages user interactions and interface elements.
  + **Backend Module**: Manages core business logic and data processing.
  + **Database Module**: Responsible for persistent data storage and management.
* **Architecture Background**

The modular organization enables parallel development, easier maintenance, and scalability. It allows each module to evolve independently, provided interfaces remain consistent.

* **Variability Mechanisms**

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

**View Packets**

##### **Frontend Module (View Packet #1)**

* **Primary Presentation**: Diagram showing QuizComponent, UserInterface, ResultDisplay.
* **Element Catalog**:
  + **Elements**: QuizComponent, UserInterface, ResultDisplay.
  + **Relations**: Dependency on backend APIs.
  + **Interfaces**: RESTful APIs to connect with backend.
  + **Behavior**: Provides interactive functionalities for users.
  + **Constraints**: Compatibility with multiple browsers.
* **Context Diagram**: Shows interactions between the frontend, users, and backend.
* **Variability Mechanisms**: Theme customization and language support.
* **Architecture Background**: Built with Vue.js for responsiveness and ease of use.
* **Related View Packets**: Backend Module.

##### **Backend Module (View Packet #2)**

* **Primary Presentation**: Diagram illustrating QuizService, UserController, and AuthService.
* **Element Catalog**:
  + **Elements**: QuizService (quiz logic), UserController (user interactions), AuthService (authentication).
  + **Relations**: Interfaces with frontend and database.
  + **Interfaces**: RESTful APIs for frontend communication.
  + **Behavior**: Manages requests, quiz processing, and data retrieval.
  + **Constraints**: Scalability and security protocols.
* **Context Diagram**: Depicts backend interactions with frontend and database.
* **Variability Mechanisms**: Scalability through load balancing.
* **Architecture Background**: Developed with Spring Boot for modularity.
* **Related View Packets**: Frontend Module, Database Module.

##### **Database Module (View Packet #3)**

* **Primary Presentation**: Database schema with main tables Users, Quizzes, and Results.
* **Element Catalog**:
  + **Elements**: Users, Quizzes, Results.
  + **Relations**: Foreign key constraints ensure data integrity.
  + **Interfaces**: SQL or ORM interfaces for backend communication.
  + **Behavior**: Manages data storage and ensures integrity.
  + **Constraints**: Enforces data consistency and backup routines.
* **Context Diagram**: Shows database interactions with the backend.
* **Variability Mechanisms**: Options for replication and clustering for high availability.
* **Architecture Background**: Built on PostgreSQL for transactional support.
* **Related View Packets**: Backend Module.

#### 3.1.2 Uses View

* **View Description**
* The Uses View focuses on the functional dependencies between modules, showing how different parts of the system work together to achieve functionality. This view helps identify critical module dependencies for successful system operation.
* **View Packet Overview**

The system has been divided into the following view packets based on their functional dependencies:

* + **Frontend-Backend Dependency**: The frontend depends on the backend for data processing and business logic.
  + **Backend-Database Dependency**: The backend depends on the database for data storage and retrieval.
* **Architecture Background**

Understanding these dependencies is crucial for maintenance and upgrades, as changes in one module might affect the other. This view helps ensure that any critical dependencies are well-defined and managed to prevent integration issues.

* **Variability Mechanisms**

These dependencies allow each module to be updated or replaced independently, provided the interfaces remain consistent.

**View Packets**

##### **Frontend-Backend Dependency (View Packet #1)**

* **Primary Presentation**: Dependency diagram illustrating how the frontend interacts with the backend to retrieve and submit data.
* **Element Catalog**:
  + **Elements**: Frontend Module and Backend Module.
  + **Relations**: "Depends-on" relationship where the frontend relies on the backend to handle data processing.
  + **Interfaces**: RESTful API endpoints for communication between frontend and backend.
  + **Behavior**: The frontend sends requests to the backend for data such as quiz content and user scores.
  + **Constraints**: Ensures frontend functionality even if backend services are updated, provided the API remains stable.
* **Context Diagram**: Illustrates the data exchange between the frontend and backend modules.
* **Variability Mechanisms**: API structure is designed to support backward compatibility, allowing for frontend upgrades with minimal impact.
* **Architecture Background**: Emphasizes clear communication channels for seamless data flow from frontend to backend.
* **Related View Packets**: Backend-Database Dependency.

##### **Backend-Database Dependency (View Packet #2)**

* **Primary Presentation**: Diagram showing how the backend relies on the database to persist data such as quiz results and user profiles.
* **Element Catalog**:
  + **Elements**: Backend Module and Database Module.
  + **Relations**: "Depends-on" relationship where the backend relies on the database for data persistence.
  + **Interfaces**: SQL queries or ORM for data access.
  + **Behavior**: The backend stores and retrieves data from the database as needed for application logic.
  + **Constraints**: Database integrity and transaction management to ensure reliable data storage.
* **Context Diagram**: Depicts the interaction between the backend and database, showing data flows for storage and retrieval.
* **Variability Mechanisms**: The database structure allows for replication and clustering, ensuring availability and load distribution.
* **Architecture Background**: Optimizes backend-database communication to support efficient data handling.
* **Related View Packets**: Frontend-Backend Dependency.

#### 3.1.3 Data Model View

* **View Description**

The Data Model View illustrates relationships between data entities within the system, ensuring a consistent approach to data storage and access.

* **View Packet Overview**
  + **Users Table**: Contains user account information.
  + **Quizzes Table**: Stores quiz questions and answers.
  + **Results Table**: Tracks user quiz scores and responses.
* **Architecture Background**

Structured for data integrity and performance, supporting complex data queries and ensuring data consistency.

* **Variability Mechanisms**

Adaptable schema to support future data model changes with minimal disruptions.

**View Packets**

##### **Users Table (View Packet #1)**

* **Primary Presentation**: Schema diagram showing columns for user details.
* **Element Catalog**:
  + **Elements**: User attributes such as user\_id, name, email.
  + **Relations**: Linked to Results by user\_id.
  + **Interfaces**: SQL/ORM for data manipulation.
  + **Behavior**: Stores and retrieves user information.
  + **Constraints**: Enforces unique emails for accounts.
* **Context Diagram**: Shows interaction with backend for user data requests.
* **Variability Mechanisms**: Fields can be expanded with new attributes.
* **Architecture Background**: Designed for efficient user management.
* **Related View Packets**: Quizzes Table, Results Table.

##### **Quizzes Table (View Packet #2)**

* **Primary Presentation**: Schema diagram showing quiz details.
* **Element Catalog**:
  + **Elements**: Quiz attributes like quiz\_id, title, questions.
  + **Relations**: Related to Results for tracking responses.
  + **Interfaces**: SQL/ORM for backend access.
  + **Behavior**: Stores quiz content for retrieval.
  + **Constraints**: Validates quiz data consistency.
* **Context Diagram**: Shows interactions with backend for quiz retrieval.
* **Variability Mechanisms**: Can add new fields as needed.
* **Architecture Background**: Supports modular quiz creation and storage.
* **Related View Packets**: Users Table, Results Table.

##### **Results Table (View Packet #3)**

* **Primary Presentation**: Schema diagram for recording quiz outcomes.
* **Element Catalog**:
  + **Elements**: Result attributes like result\_id, user\_id, quiz\_id, score.
  + **Relations**: Foreign key relationships with Users and Quizzes.
  + **Interfaces**: SQL/ORM for backend access.
  + **Behavior**: Logs quiz results per user.
  + **Constraints**: Ensures referential integrity with Users and Quizzes.
* **Context Diagram**: Shows interaction with backend for result storage.
* **Variability Mechanisms**: Support for additional result metrics.
* **Architecture Background**: Structured for quick retrieval of user results.
* **Related View Packets**: Users Table, Quizzes Table.

**3.2Component-and-Connector View**

#### 3.2.1 Call-Return View

* **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 after task completion. This view is typical for client-server interactions where components invoke functions and expect responses.
* **View Packet Overview**

This view includes the following packets:

* + **User Interaction Component**: Manages communication between the frontend and backend, where the frontend calls backend services to retrieve or submit data.
  + **Backend Processing Component**: Handles core logic and processes requests from the frontend, returning data or processing results.
* **Architecture Background**

This view supports a structured, request-response model, which is fundamental for interactive applications like *Quizzes Tutor*. By organizing components in a call-return structure, the architecture ensures that each component performs specific tasks and returns control efficiently.

* **Variability Mechanisms**

The backend can be scaled horizontally to support a higher number of concurrent requests. Additionally, caching mechanisms can be introduced to reduce redundant calls and improve performance.

**View Packets**

##### **User Interaction Component (View Packet #1)**

* **Primary Presentation**: Diagram showing the frontend components, such as QuizComponent and ResultDisplay, calling backend services.
* **Element Catalog**:
  + **Elements**: QuizComponent (manages quiz display and interactions), ResultDisplay (shows quiz results).
  + **Relations**: "Calls" relationship where the frontend components make requests to backend APIs.
  + **Interfaces**: RESTful APIs between frontend and backend for data retrieval and submission.
  + **Behavior**: The frontend sends a request to the backend, waits for the response, and then updates the interface based on the data received.
  + **Constraints**: The frontend must handle response delays gracefully, and ensure compatibility with backend API versions.
* **Context Diagram**: Shows the interaction flow between frontend components and backend services for retrieving and displaying quiz data.
* **Variability Mechanisms**: Support for caching of frequently accessed data, reducing the need for repeated calls to backend services.
* **Architecture Background**: Built using a client-server model, where the frontend (client) calls the backend (server) as needed for data updates and processing.
* **Related View Packets**: Backend Processing Component.

##### **Backend Processing Component (View Packet #2)**

* **Primary Presentation**: Diagram showing backend components, such as AuthService and QuizService, processing requests from the frontend and returning responses.
* **Element Catalog**:
  + **Elements**: AuthService (handles user authentication), QuizService (manages quiz logic and scoring).
  + **Relations**: "Calls" relationship from frontend to backend, with backend returning control upon task completion.
  + **Interfaces**: RESTful endpoints exposed to the frontend for secure data access.
  + **Behavior**: The backend processes requests, validates inputs, retrieves or updates data as needed, and returns the result to the frontend.
  + **Constraints**: Security requirements for data access and response times for a smooth user experience.
* **Context Diagram**: Illustrates the backend components receiving requests from the frontend and interacting with the database to fulfill the requests.
* **Variability Mechanisms**: Backend load balancing and horizontal scaling to handle high traffic volumes.
* **Architecture Background**: Implemented with Spring Boot, allowing modular and efficient call-return interactions for dynamic processing.
* **Related View Packets**: User Interaction Component.

#### 3.2.2 Repository View

* **View Description**

The Repository View focuses on the system’s data storage and management, with a central repository for storing and retrieving persistent data. This view is essential for understanding how data is shared, accessed, and managed across multiple components.

* **View Packet Overview**

This view includes the following packets:

* + **Data Storage Component**: Central data repository (database) used for storing quiz data, user profiles, and results.
  + **Data Access Component**: Backend components that interact with the data repository to read and write data as needed.
* **Architecture Background**

The repository structure supports data integrity, consistency, and efficient access, especially in environments where multiple users are accessing or modifying data simultaneously.

* **Variability Mechanisms**

The repository supports replication and clustering to improve data availability and performance, ensuring high availability and load distribution.

**View Packets**

##### **Data Storage Component (View Packet #1)**

* **Primary Presentation**: Database schema diagram showing primary tables (e.g., Users, Quizzes, Results).
* **Element Catalog**:
  + **Elements**: Users (stores user information), Quizzes (stores quiz data), Results (stores quiz results).
  + **Relations**: "Stores" and "retrieves" relationships between backend components and the data repository.
  + **Interfaces**: SQL or ORM interfaces for backend access to the database.
  + **Behavior**: Provides persistent data storage, ensuring that data is available even after sessions end.
  + **Constraints**: Must support high read/write performance, and ensure data integrity through transaction management.
* **Context Diagram**: Shows data repository interactions with backend components for read/write operations.
* **Variability Mechanisms**: Database replication and clustering to ensure data is always available and can handle high traffic.
* **Architecture Background**: Based on PostgreSQL, chosen for its robustness and support for large-scale data management.
* **Related View Packets**: Data Access Component.

##### **Data Access Component (View Packet #2)**

* **Primary Presentation**: Diagram of backend components (AuthService, QuizService) accessing data in the repository.
* **Element Catalog**:
  + **Elements**: AuthService (retrieves user information for authentication), QuizService (reads/writes quiz data).
  + **Relations**: "Accesses" relationship between backend services and data storage.
  + **Interfaces**: SQL queries or ORM methods used by backend components to interact with the database.
  + **Behavior**: Backend services perform CRUD operations on the database to manage application data.
  + **Constraints**: Ensures secure data access and maintains consistency in concurrent operations.
* **Context Diagram**: Shows backend service interactions with the data repository for storing and retrieving data.
* **Variability Mechanisms**: Database connection pooling for efficient access, allowing multiple concurrent connections.
* **Architecture Background**: Designed to optimize data access through efficient query handling and data caching mechanisms.
* **Related View Packets**: Data Storage Component.

### 3.3 Allocation View

#### 3.3.1 Execution View

* **View Description**
* The Execution View illustrates how software components are mapped to the physical or virtual infrastructure required to run the system. This view provides insights into the distribution of components across servers or cloud resources to support real-time operations.
* **View Packet Overview**

This view has been divided into the following packets:

* + **Frontend Deployment**: Deployment of the frontend components on web servers and content delivery networks (CDNs).
  + **Backend Deployment**: Deployment of backend services on application servers with load balancers for scalability.
* **Architecture Background**

This infrastructure design leverages cloud and load balancing to ensure reliability and scalability, supporting multiple concurrent users and handling high traffic demands.

* **Variability Mechanisms**

Load balancing for backend deployment to handle varying loads, and CDN configurations for optimized frontend performance.

**View Packets**

##### **Frontend Deployment (View Packet #1)**

* **Primary Presentation**: Diagram showing the deployment of frontend components on web servers and CDNs.
* **Element Catalog**:
  + **Elements**: Web server instances hosting the frontend, CDN for static assets.
  + **Relations**: Frontend is served through a CDN to optimize delivery times and reduce server load.
  + **Interfaces**: HTTP/HTTPS for client communication.
  + **Behavior**: Manages requests from users, ensuring fast loading times by caching static assets and delivering content through the CDN.
  + **Constraints**: Requires high availability and minimal latency to maintain user experience under high load.
* **Context Diagram**: Shows the frontend's connections to the backend and interactions with end-user devices.
* **Variability Mechanisms**: CDN configuration allows content caching and load balancing for quick content delivery.
* **Architecture Background**: The frontend is distributed through a CDN, leveraging geographical proximity to reduce latency and improve user experience.
* **Related View Packets**: Backend Deployment.

##### **Backend Deployment (View Packet #2)**

* **Primary Presentation**: Diagram showing backend services deployed on application servers with load balancers for scalability.
* **Element Catalog**:
  + **Elements**: Backend server instances, load balancer.
  + **Relations**: Load balancer distributes requests across backend servers to manage traffic and ensure high availability.
  + **Interfaces**: HTTP/HTTPS APIs for communication with the frontend.
  + **Behavior**: Distributes incoming requests evenly across multiple servers, managing sessions and ensuring fault tolerance.
  + **Constraints**: Requires high reliability, secure API endpoints, and the ability to scale horizontally as demand increases.
* **Context Diagram**: Illustrates the backend's connections to the frontend and database, showing the load balancer and server instances.
* **Variability Mechanisms**: Load balancers allow the backend to scale horizontally, adding more server instances to handle peak loads.
* **Architecture Background**: The backend is designed to be scalable and resilient, with a load balancer to prevent any single server from becoming a bottleneck.
* **Related View Packets**: Frontend Deployment, Database Deployment.

#### 3.3.2 Development View

* **View Description**

The Development View presents the organization of the development artifacts, such as source code, to support team collaboration and version control. This view is essential for project management and coordination among the development team.

* **View Packet Overview**

This view is divided into the following packets:

* + **Source Code Organization**: Structure of source code repositories and folders for various modules.
  + **API Documentation**: Documentation setup for backend APIs to ensure smooth communication between frontend and backend teams.
* **Architecture Background**

A clear organization of code and documentation aids in team collaboration, facilitating version control and modular development. Using standardized tools and practices helps reduce integration issues.

* **Variability Mechanisms**

Modular code structure allows independent development and testing of each component, while documentation enables flexible onboarding of new team members.

**View Packets**

##### **Source Code Organization (View Packet #1)**

* **Primary Presentation**: Directory structure diagram showing folders for frontend, backend, and database code.
* **Element Catalog**:
  + **Elements**: Repositories for frontend, backend, and database code.
  + **Relations**: Modules are organized by functionality, enabling independent development.
  + **Interfaces**: Version control (e.g., Git) for managing changes and collaboration.
  + **Behavior**: Facilitates modular development, with separate directories for each component.
  + **Constraints**: Code structure should adhere to team standards for readability and modularity.
* **Context Diagram**: Illustrates the organization of code repositories and shows the flow of updates between frontend, backend, and database modules.
* **Variability Mechanisms**: Supports branching and merging in version control to allow concurrent feature development.
* **Architecture Background**: The project uses Git for version control, supporting multiple branches and collaborative work.
* **Related View Packets**: API Documentation.

##### **API Documentation (View Packet #2)**

* **Primary Presentation**: Overview of API documentation setup for backend services.
* **Element Catalog**:
  + **Elements**: API endpoints documented with descriptions, usage examples, and response formats.
  + **Relations**: API documentation facilitates frontend-backend interaction by defining expected data flows.
  + **Interfaces**: Documentation tool (e.g., Swagger) for backend API endpoints.
  + **Behavior**: Allows frontend developers to interact with backend APIs based on documented guidelines.
  + **Constraints**: Documentation should be kept up-to-date with backend changes to avoid miscommunication between teams.
* **Context Diagram**: Shows the flow of information between documented API endpoints and their use by frontend developers.
* **Variability Mechanisms**: Supports versioning in API documentation to account for updates and backward compatibility.
* **Architecture Background**: API documentation is created and managed through Swagger to provide clear and accessible information for developers.
* **Related View Packets**: Source Code Organization.

# 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.

## General Relations Among Views

|  |
| --- |
| **CONTENTS OF THIS SECTION**: This section describes the general relationship among the views chosen to represent the architecture. Also in this section, consistency among those views is discussed and any known inconsistencies are identified. |

In the **Quizzes Tutor** project, architectural views provide unique perspectives on the system's structure, functionality, and behavior. These views are interrelated, and understanding their relationships is crucial for maintaining a coherent and consistent architecture. Below are the seven key relationships that link the views and ensure the system meets its design objectives.

Below are the **seven key relationships** between the views in the Quizzes Tutor project, detailing how they interact and complement each other.

### Relationships Between Views

1. **Decomposition View ↔ Uses View**
   1. **Relation**: Modules in the Decomposition View correspond to dependencies represented in the Uses View.
   2. **Reason**: This ensures that the defined modular structure supports the required functional dependencies between modules.
   3. **Example**: The "Quiz Management" module depends on the "User Authentication" module to verify users before granting access to quizzes.
2. **Decomposition View ↔ Data Model View**
   1. **Relation**: Modules in the Decomposition View interact with data entities defined in the Data Model View.
   2. **Reason**: To ensure that each module has access to the correct data structures it needs to function.
   3. **Example**: The "Quiz Submissions" module uses entities such as "Submissions", "Questions", and "Users" from the Data Model View.
3. **Uses View ↔ Data Model View**
   1. **Relation**: Dependencies between modules in the Uses View involve shared access to data entities described in the Data Model View.
   2. **Reason**: To verify that the data dependencies between modules are consistent with the underlying data model.
   3. **Example**: The "Results Calculation" module depends on data entities like "Answers" and "Grades" to calculate quiz scores.
4. **Call-Return View ↔ Repository View**
   1. **Relation**: Components in the Call-Return View interact with persistent storage mechanisms defined in the Repository View.
   2. **Reason**: To ensure that runtime calls to store or retrieve data are supported by the repository structure.
   3. **Example**: A component responsible for saving quiz results invokes repository operations to store them in the database.
5. **Call-Return View ↔ Data Model View**
   1. **Relation**: Components in the Call-Return View utilize data entities described in the Data Model View during runtime interactions.
   2. **Reason**: To ensure that the data entities used at runtime are correctly defined and consistent with the data model.
   3. **Example**: A component processing quiz answers uses data entities for "Questions" and "Answers".
6. **Execution View ↔ Development View**
   1. **Relation**: Development artifacts in the Development View are mapped to their execution environments in the Execution View.
   2. **Reason**: To ensure that every artifact, such as a code module or library, has a corresponding deployment location in the infrastructure.
   3. **Example**: The "Quiz API" module is implemented as a source code artifact and deployed on a specific server in the Execution View.
7. **Execution View ↔ Data Model View**
   1. **Relation**: Data entities in the Data Model View are stored and managed within the infrastructure defined in the Execution View.
   2. **Reason**: To ensure that the data storage mechanisms align with the system's infrastructure.
   3. **Example**: Entities like "Users" and "Quizzes" are mapped to a cloud-based database in the Execution View.

# Referenced 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. |
| Referências Principais do Projeto | **GitHub Repository:** Quizzes Tutor. Disponível em <https://github.com/socialsoftware/quizzes-tutor>  **Vídeo Modelo de dominío:** Vídeo explicativo da arquitetura do Quizzes Tutor, disponível em [YouTube](https://www.youtube.com/watch?v=wEhEUCJpOpc) |
| Ferramentas de Análise e Métricas de Código | **SonarQube Documentation**: Plataforma de análise de código utilizada para avaliar a qualidade do código-fonte do projeto. Documentação disponível em (<https://docs.sonarqube.org/latest/>.) |
| **Referências para Exemplos e Estudo de Casos**: | **DESOSA**: Exemplos de projetos de arquitetura de software realizados por estudantes de mestrado na Universidade de Delft:  [**DESOSA 2019**](https://se.ewi.tudelft.nl/desosa2019/), [**DESOSA 2020**](https://desosa.nl/), [**DESOSA 2021**](https://2021.desosa.nl/), [**DESOSA 2022**](https://desosa2022.netlify.app/). |
|  |  |

# Directory

## Index

|  |
| --- |
| **CONTENTS OF THIS SECTION**: This section provides an index of all element names, relation names, and property names. For each entry, the following are identified:   * the location in the SAD where it was defined * each place it was used   Ideally, each entry will be a hyperlink so a reader can instantly navigate to the indicated location. |

## Glossary

|  |
| --- |
| **CONTENTS OF THIS SECTION**: This section provides a list of definitions of special terms and acronyms used in the SAD. If terms are used in the SAD that are also used in a parent SAD and the definition is different, this section explains why. |

|  |  |
| --- | --- |
| 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. |
| view packet | The smallest package of architectural documentation that could usefully be given to a stakeholder. The documentation of a view is composed of one or more view packets. |
| 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. |

## Acronym List

|  |  |
| --- | --- |
| API | Application Programming Interface; Application Program Interface; Application Programmer Interface |
| ATAM | Architecture Tradeoff Analysis Method |
| CMM | Capability Maturity Model |
| CMMI | Capability Maturity Model Integration |
| CORBA | Common object request broker architecture |
| COTS | Commercial-Off-The-Shelf |
| EPIC | Evolutionary Process for Integrating COTS-Based Systems |
| IEEE | Institute of Electrical and Electronics Engineers |
| KPA | Key Process Area |
| OO | Object Oriented |
| ORB | Object Request Broker |
| OS | Operating System |
| QAW | Quality Attribute Workshop |
| RUP | Rational Unified Process |
| SAD | Software Architecture Document |
| SDE | Software Development Environment |
| SEE | Software Engineering Environment |
| SEI | Software Engineering Institute  Systems Engineering & Integration  Software End Item |
| SEPG | Software Engineering Process Group |
| SLOC | Source Lines of Code |
| SW-CMM | Capability Maturity Model for Software |
| CMMI-SW | Capability Maturity Model Integrated - includes Software Engineering |
| UML | Unified Modeling Language |

**Appendices**

|  |
| --- |
| **CONTENTS OF THIS SECTION**: Appendices may be used to provide information published separately for convenience in document maintenance (e.g., charts, classified data, API specification). As applicable, each appendix is referenced in the main body of the document where the data would normally have been provided. Appendices may be bound as separate documents for ease in handling. If your SAD has no appendices, delete this page. |

* 1. **Heading 2 - Appendix**
  2. **Heading 2 - Appendix**