Norwegian University of
Science and Technology

TDT4240 — Software Architecture

Group 13 — Architecture

DrawGuess

Android

Sjøberg, Isak Olav
Weidel, Jacob
Færestrand, Benjamin
Trouchet, Ninon Lisa
Raeesi, Arya

Chosen COTS: Firebase Firestore, Socket.IO, OkHttp WebSocket

Primary Quality Attribute: Modifiability

Secondary Quality Attribute: Usability and Performance

February 24, 2025

Contents

1 Introduction

1.1 Description of the Project

This document will outline a set of functional requirements and architectural attributes for a multiplayer game in Android. All documentation will be kept up to date, thus having a *Changes* section in section 10.

1.2 Description of the Game Concept

DrawGuess is a multiplayer Pictionary-style game in which players take turns drawing and guessing within set time limits. The game features a competitive point-based system and a turn-based multiplayer game with real-time interactions in the guessing phase.

The game follows a structured sequence where players are assigned as either a *guesser(s)* or *drawer*. The *drawer* will sketch a given word, while the *guesser(s)* attempt to identify it. After each round, points are given based on response time and correctness. This process repeats for several rounds, until a final leaderboard appears. Figure 1 illustrates this workflow.

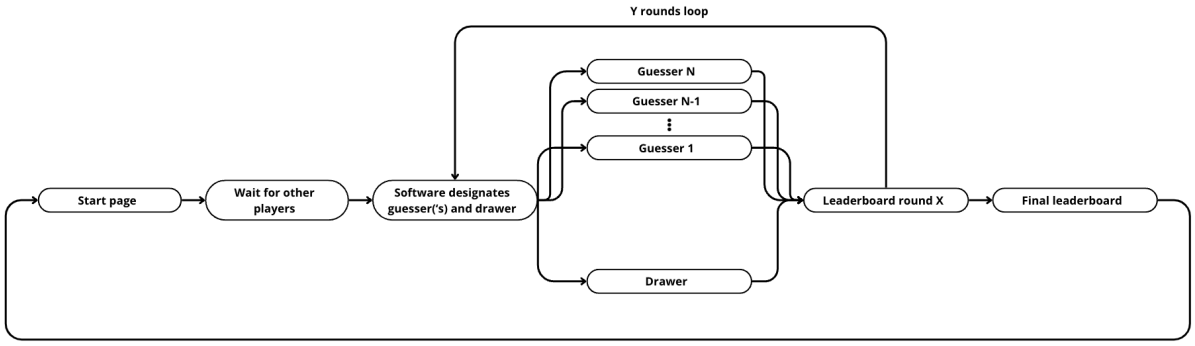

Figure 1: Game flowchart. The game consists of Y total rounds, looping until X (completed rounds) equals Y. In each round, one player is the *drawer* while N players take on the role of *guessers*.

With Figure 1 in mind, Figure 2 shows an illustration of the game's start page.

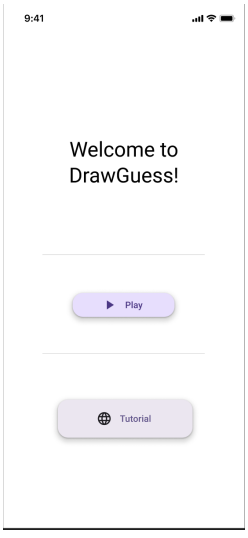

Figure 2: Illustration of the DrawGuess start page.

After clicking *Play*, the game will find opponents, and split up our user and opponents to one *drawer*, and one or more *guessers*.

Figure 3 shows the *guessers* screen during a round. The left screen shows the *guessers* screen before the *drawer* has finished drawing, while the right screen shows the *guessers* screen after finished drawing. When the drawing has appeared on the *guessers* screen, there's a timer to finish drawing. Faster correct guesses give more points, while wrong guesses give 0 points. If a *guesser* guesses correctly, the *drawer* will receive more points depending on how fast a *guesser* guessed correctly.

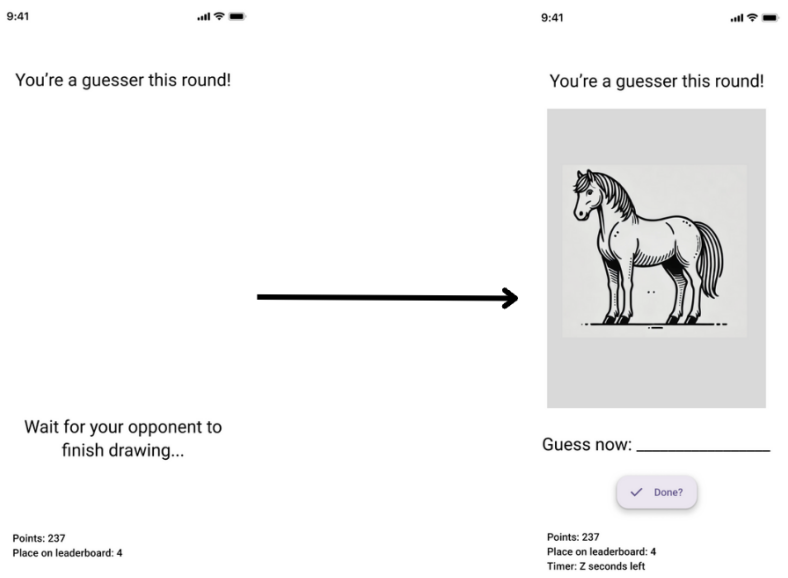

Figure 3: Illustration of one round for a *guesser*. Note that the horse on the right screen was created using AI [1].

Furthermore, Figure 4, shows the *drawer's* screen during a round. The left screen shows the *drawer's* screen before drawing, and the right screen shows the screen after drawing. Faster drawing gives more points if a *guesser* is able to guess correctly. If a *guesser* isn't able to guess correctly, the drawer will receive 0 points. For mathematical expressions regarding point distribution, see appendix A.

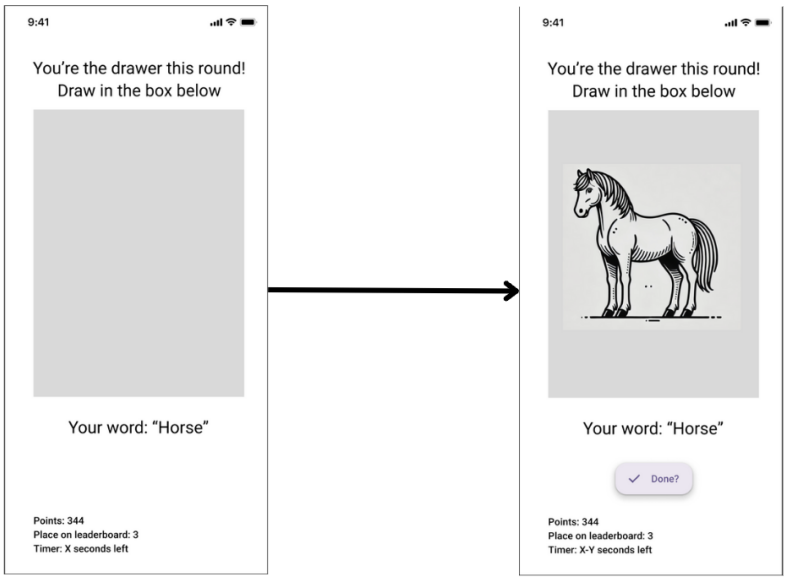

Figure 4: Illustration of one round for a *drawer*. The timer starts with X seconds left, and in the illustration the drawing is done with X-Y seconds left. Note that the horse on the right screen was created using AI [1].

After the *guessers* and *drawer* are done, the game goes to the round X leaderboard (see Figure 5).

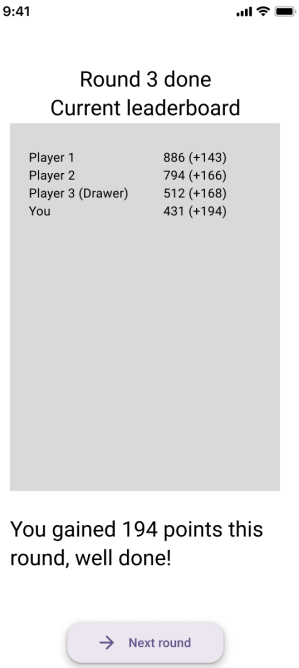

Figure 5: Illustration of the leaderboard after a round. In this example it's the third round, and our player is in last place.

Finally, after Y rounds (see Figure 1), the final leaderboard appears (see Figure 6). After the final leaderboard, our player can return to the start page when clicking *Homepage* in Figure 6.

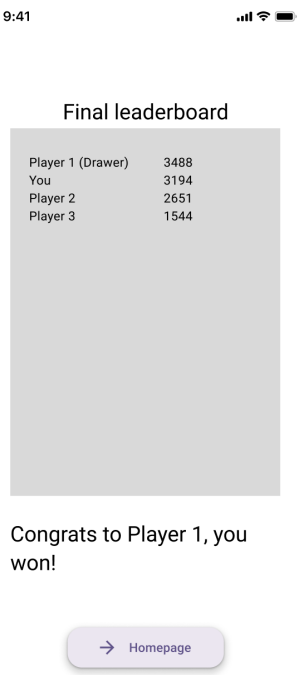

Figure 6: Illustration of the final leaderboard.

In short, in DrawGuess players start in the *start page*, play for Y rounds and return to the *start page* after the *Final leaderboard* appears (see Figure 1).

1.3 Report Structure

This report entails several parts of DrawGuess. The report structure is as follows:

1. **Introduction:** an overview of DrawGuess, including its purpose and core gameplay mechanics.
2. **Architectural Drivers / Architecturally Significant Requirements (ASRs):** specifies the key drivers used for DrawGuess.
3. **Stakeholders and Concerns:** overview of stakeholders and concerns for DrawGuess.
4. **Architectural Viewpoints:** an overview of the logical-, process-, development- and physical view of DrawGuess.
5. **Architectural Tactics:** an overview of the architectural tactics used for achieving the quality attributes chosen for DrawGuess.
6. **Architecture and Design Patterns:** specifications about the architectural and design patterns utilized in DrawGuess.
7. **Architectural Views:** showcasing the different architectural views of DrawGuess, including several diagrams.
8. **Consistency among Architectural Views:** highlights how architectural views are consistently used logically along with each other.
9. **Architectural Rationale:** an overview of the rationale behind the architectural choices of DrawGuess.

10. **Issues:** highlights challenges encountered during development and their impact on the project.
11. **Changes:** documents modifications made to the report.
12. **Contributions:** details individual team members contributions to the project.
13. **Bibliography:** lists sources and references used throughout the project documentation.
14. **Appendix:** supplementary material such as diagrams, formulas, or additional explanations.

2 Architectural Drivers / Architecturally Significant Requirements (ASRs)

The architecture of DrawGuess is driven by key functional and quality requirements that ensure the system meets performance, usability, and modifiability goals. These drivers define the essential capabilities and constraints of the system, shaping the overall software architecture.

2.1 Functional Requirements

The functional requirements define the core features that enable DrawGuess to operate as a turn-based multiplayer experience with real-time interactions during the guessing phase.

2.1.1 Turn-Based Multiplayer with Real-Time Guessing

The game follows a turn-based structure where one player is assigned as the *drawer* each round. The *drawer* privately creates an image based on a given word and submits it by clicking "done?" Once submitted, the completed drawing is revealed to all other players simultaneously. At this stage, the game transitions into a real-time guessing phase, where players compete to identify the word as quickly as possible. Guesses are processed dynamically, with immediate feedback on correctness. Additionally, players can see when others have completed their guesses. A high-score system tracks player performance across multiple rounds, ensuring a competitive and engaging experience.

2.1.2 Drawing Submission and Display

Unlike fully real-time drawing games, the drawing phase in DrawGuess is private to the *drawer*. The system captures and stores the drawing until submission, at which point it is displayed to all other players. This eliminates the need for continuous real-time updates during the drawing phase while ensuring a smooth transition into the guessing stage.

2.1.3 Guessing Mechanism

Once the drawing is revealed, players must submit their guesses as quickly as possible. The system evaluates guesses dynamically, comparing them against the assigned word and providing immediate feedback. To maintain an engaging gameplay experience, guess validation must be near-instantaneous.

2.1.4 Scoring System and Server Component

A backend server manages and stores game session data, including player scores, session history, and game settings. This ensures that high scores persist across multiple sessions and that game data is retained after a match concludes. The backend must be scalable to handle multiple game instances simultaneously.

2.2 Quality Requirements

Beyond functional capabilities, the system must meet quality attributes that ensure long-term maintainability, ease of use, and optimized performance.

2.2.1 Modifiability

The architecture is designed with flexibility in mind, allowing for future modifications without significant restructuring. It should support the introduction of new drawing tools and brush styles, the addition of alternative game modes such as timed challenges or cooperative play, and the ability to modify scoring rules dynamically through configuration files instead of hardcoded logic. Furthermore, the system should accommodate different types of word lists, including custom user-generated words. A modular approach, with clearly defined APIs between components, ensures that updates can be implemented with minimal impact on other parts of the system.

2.2.2 Usability

The game must provide an intuitive and accessible user experience for both *drawers* and *guessers*. A responsive UI is essential, ensuring smooth handling of user input without lag. Clear visual feedback should be provided when a guess is correct, incorrect, or partially correct. The layout must be well-structured, ensuring that essential game elements such as the drawing area, guess input, and timer are easily visible. Additionally, error handling mechanisms must be in place to guide users in case of connectivity issues or invalid inputs.

2.2.3 Performance

Responsiveness is critical to the success of DrawGuess. The architecture must minimize latency and optimize resource usage to ensure smooth gameplay. While the drawing phase does not require real-time transmission, the guessing phase demands low network latency, requiring efficient WebSocket-based communication rather than traditional HTTP polling. Guess processing should be optimized to occur in under 200ms to maintain a fast-paced gaming experience. Data transmission should be efficient, sending drawing inputs as compressed vector data instead of raw images to reduce bandwidth usage [2]. The backend infrastructure must support load balancing, distributing concurrent game sessions across multiple server instances when required.

By addressing these functional and quality requirements, the architecture of DrawGuess ensures a responsive, scalable, and easily extendable system, providing an engaging multiplayer experience with a balance between turn-based structure and real-time interactions.

3 Stakeholders and Concerns

The development of DrawGuess involves multiple stakeholders, each with unique concerns and requirements that influence the system's architecture. Addressing these concerns is essential to ensure a robust, scalable, and user-friendly solution that meets functional and quality requirements.

3.1 Developers

The development team is responsible for designing, implementing, and maintaining the software. Their primary concern is ensuring that the system architecture remains modular, maintainable, and scalable. A well-structured codebase with clear separation of concerns allows for efficient development, reduces technical debt, and makes it easier to introduce new features or fix bugs. The developers also require well-documented APIs and coding standards to facilitate collaboration among team members. Furthermore, automated testing and continuous integration practices are crucial for maintaining software quality and ensuring that new code does not introduce regressions.

3.2 Players

Players are the end-users of DrawGuess, and their primary concern is the overall user experience. They expect the game to be intuitive, engaging, and responsive, with minimal latency in real-time interactions. The drawing experience should be smooth, and guesses should be processed immediately to maintain a sense of competition. Players also value a fair scoring system that accurately rewards performance and provides leaderboards for competitive gameplay. Additionally, reliability is a key concern; unexpected disconnections, lag, or crashes can severely impact user satisfaction. A well-designed user interface with clear instructions and feedback mechanisms enhances usability and player retention.

3.3 Course Staff and Evaluators

Since this project is part of an academic course, course staff and evaluators play an important role in assessing its architectural decisions and implementation quality. Their main concern is whether the architecture aligns with best practices in software engineering, particularly regarding modifiability, usability, and performance. The evaluators need clear and well-structured documentation that outlines design decisions, trade-offs, and justifications for architectural choices. They also assess whether the system follows the stated requirements, including the ability to extend the game with new features and modifications.

4 Architectural Viewpoints

Table 1: Architectural Viewpoints.

View	Purpose	Stakeholders	Notation
Logical View	Describes system functionality and major components (e.g., game logic, network, UI).	Developers, TAs, ATAM evaluators, End-users	UML Class Diagrams
Process View	Shows runtime interactions (e.g., drawing broadcast, guess processing, score updates).	Developers, ATAM evaluators	Sequence and Activity Diagrams
Development View	Details module organization for parallel development (e.g., separate UI, network, game logic packages).	Development team, TAs	Package/Component Diagrams
Physical View	Maps software components to hardware (e.g., Android devices, cloud server deployment).	Developers, Deployment Engineers	Deployment Diagrams

5 Architectural Tactics

This section provides an overview of the architectural tactics employed to achieve the quality attributes of modifiability, usability, and performance in the DrawGuess game.

5.1 Modifiability

5.1.1 Encapsulation and Abstraction

The system leverages explicit interfaces to encapsulate components such as network communication and drawing input. This approach ensures that changes to the implementation of one component do not affect other parts of the system, promoting modularity and ease of maintenance.

5.1.2 Low Coupling and High Cohesion

The architecture organizes the system into well-defined modules, which is shown in Figure 7 in the Architectural Views chapter. Each module focuses on a single responsibility (high cohesion) and minimizes dependencies between components (low coupling). This structure simplifies maintenance and reduces the impact of changes.

5.2 Usability

5.2.1 Consistent UI Design

The system maintains a uniform and intuitive interface. This consistency reduces the learning curve for new users and enhances their overall experience.

5.2.2 Real-Time Feedback

The game provides immediate feedback to users through visual cues and notifications. For example, players are informed of successful guesses. This ensures clarity and engagement throughout gameplay.

5.3 Performance

5.3.1 Efficient Data Transmission

The system uses asynchronous messaging via firebase to minimize latency during the real-time guessing phase and leaderboard updates, while drawings are stored and revealed upon submission to optimize performance.

6 Architecture and Design Patterns

This section outlines the key architectural and design patterns utilized in the development of DrawGuess. These patterns provide reusable solutions to common problems, enhance system modifiability, and improve the game's overall maintainability and performance. The architectural patterns focus on the high-level structure of the system, while the design patterns address more specific implementation details.

6.1 Architectural Patterns

6.1.1 Client-Server Pattern

The game employs a client-server architecture with a centralized server or cloud-based service, and an application running the player's device [3]. The server manages critical functionalities such as hosting game sessions, maintaining high scores, and synchronizing game state between players. This pattern ensures reliability, scalability, and efficient management of player interactions, enabling a smooth gaming experience across devices.

6.1.2 Model-View-Controller (MVC)

The MVC pattern is used to separate the user interface (View), the underlying data (Model), and the business logic (Controller) [3]. This separation facilitates easier updates and testing by isolating different components. For instance, changes to the user interface do not affect the logic or data, and vice versa. This pattern is crucial for maintaining a clean, modular, and testable codebase.

6.2 Design Patterns

6.2.1 Creational Patterns

Singleton Pattern
The Singleton pattern ensures that only one instance of specific classes exists throughout the application's lifecycle. Examples include `GameSessionManager` and `ConfigurationManager`, which centralize control over game sessions and application-wide settings. While Singleton may not enhance modifiability directly, it simplifies implementation and reduces redundancy.

Factory Method Pattern
The Factory Method pattern dynamically creates objects based on runtime conditions. It is used to instantiate key components such as `Player`, `GameSession`, and `Drawing`. For instance, players are assigned roles (Drawer or Guesser) at the start of each game. This pattern supports modifiability by abstracting object creation.

6.2.2 Behavioral Patterns

State Pattern
The State pattern manages transitions between game phases, such as *Waiting for Players*, *Drawing Phase*, and *Guessing Phase*. Each state has its specific behavior, ensuring clear separation and simplifying the game logic.

7 Architectural Views

7.1 Logical View

Figure 7 illustrates the main classes and their relationships in DrawGuess. These classes are split into different modules according to their responsibilities, with a focus on separating game logic from user interface elements and shared services.

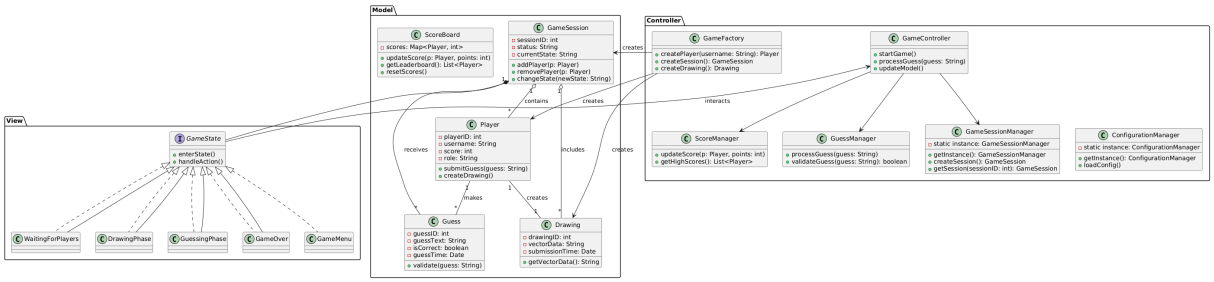

Figure 7: Class Diagram.

The system follows the MVC architectural pattern by separating different modules to ensure better testability and modifiability. The components interact with the Firebase Firestore to ensure low-latency updates, with WebSockets handling communication lines between the components. This ensures consistency across all game clients.

Additionally, by structuring with distinct managers, we follow the Client-Server architecture. This gives DrawGuess clear separations between concerns, facilitating future modifications while maintaining an efficient and scalable architecture.

7.2 Process View

Figure 8 shows a high-level process flow for DrawGuess from the player's perspective.

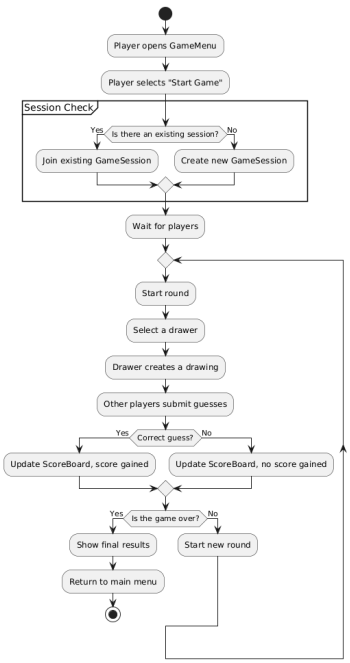

Figure 8: Activity Diagram.

When a user opens the `GameMenu` and clicks `Start Game`, they can either join an existing `GameSession` or create a new one. After all players have joined, the game selects a drawer and begins the drawing phase. The drawer creates a drawing privately, and once they are done, guessers submit their guesses in real-time. If a guess is correct, the system updates the `ScoreBoard`, and the round continues until time expires or everyone has finished guessing. If more rounds remain, the process repeats; otherwise, final results are displayed, and players can choose to return to the main menu. This activity diagram highlights the sequence of steps that guide players from starting a game session to viewing final scores.

7.3 Development View

Figure 9 shows how the codebase is divided into `Controller`, `Model`, and `View` packages, plus a database component. This clear separation minimizes coupling: the `Controller` layer handles logic (e.g., scoring and session flow), the `View` layer manages the user interface, and the `Model` contains domain objects like `Player` and `GameSession`. Such modular organization allows parallel development and straightforward maintenance, as changes in one package do not heavily impact others.

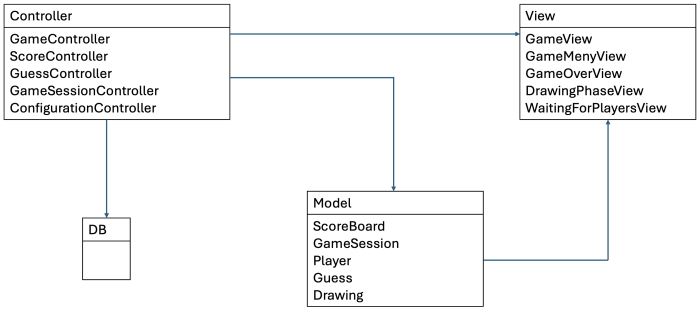

Figure 9: Activity Diagram.

7.4 Physical View

The physical view illustrates how the software components defined in the development view are deployed on actual hardware and infrastructure. The Android client, which includes both UI and logic modules, runs directly on mobile devices, delivering a responsive and user-friendly experience. Meanwhile, the backend server is hosted in the cloud, where Firebase Firestore is used. Communication between the client and server is efficiently handled by Socket.IO and OkHttp WebSocket, ensuring low latency. This deployment strategy meets the performance, scalability, and reliability required for a multiplayer game.

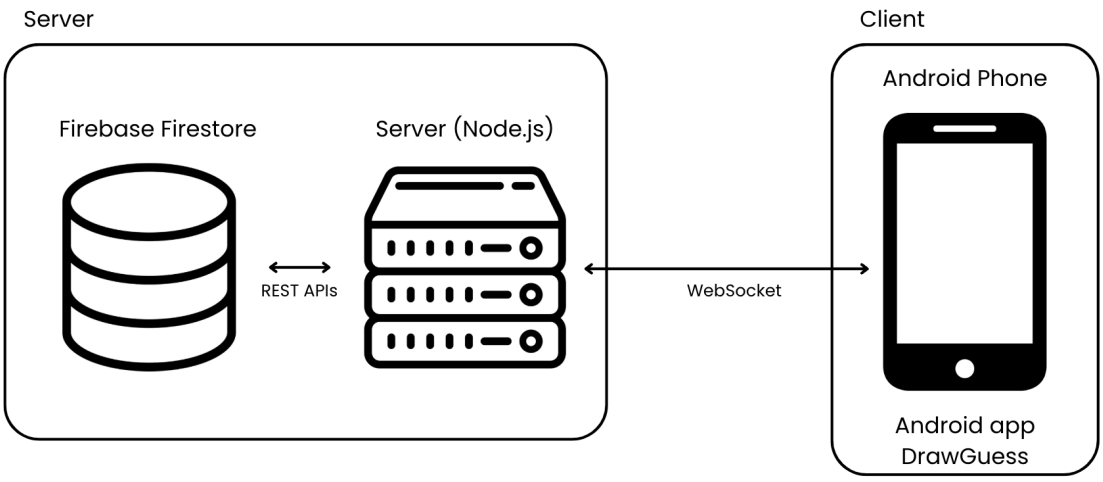

Figure 10: Physical View.

7.5 Sequence Diagram

Figure 11 illustrates a typical interaction in DrawGuess when a player submits a guess.

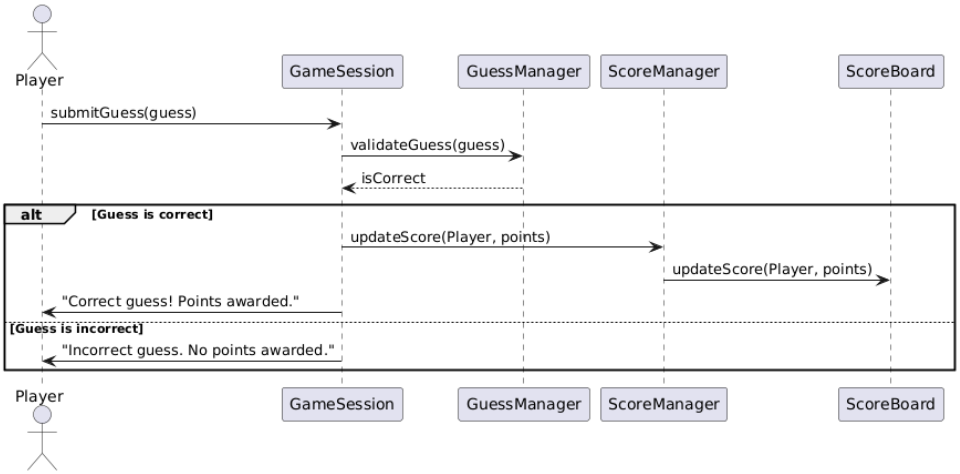

Figure 11: Sequence Diagram.

The `Player` first sends the guess to the `GameSession`, which passes it on to the `GuessManager` for validation. If the guess is correct, the `GuessManager` notifies the `ScoreBoard` to update the player's score, and the `ScoreBoard` is refreshed to reflect the new total. The player receives feedback such as "Correct guess!" indicating success. If the guess is incorrect, the `Player` is informed with a message like "Try again!" and no score update occurs. This sequence highlights how different components communicate in real time to manage scoring logic and provide immediate feedback to the user.

7.6 Inconsistencies

One observed inconsistency is that the Logical View separates guess validation and scoring into distinct managers (GuessManager, ScoreManager), but in the Process View, there is no clear indication that guess validation occurs independently of overall round management, causing

confusion about which component triggers scoring events. Similarly, the Development View shows a strict division between Controller and Model packages for drawing and guessing features, yet the Sequence Diagram sometimes combines these responsibilities into a single flow, implying tighter coupling than the Development View suggests.

8 Architectural Rationale

The architectural choices for DrawGuess are mainly driven by the need for modifiability, while still ensuring that the game is easy to use and performs well.

8.1 Modifiability

Modifiability is a key requirement for our game. To make future changes easier, we have core functionalities such as network communication and drawing input behind intuitive interfaces. This means that if new drawing tools or guessing words are introduced later, changes can be made without affecting other parts of the system. The architecture is organized into separate modules, each with a specific responsibility. Our design makes use of patterns such as the Client-Server model (with Firebase Firestore) and the Model-View-Controller (MVC) pattern, which help separate the user interface from the game logic. Together, these choices keep the system structure clear and allow for straightforward modifications.

8.2 Usability

Usability is another important quality for DrawGuess. The concerns provided by the MVC pattern means that changes to the user interface can be made without affecting the underlying logic. The design is presented in a clear and consistent structure. Real-time guessing feedback is provided to the player, which helps keep the gameplay engaging. Overall, these measures contribute to a user experience that is simple and straightforward.

8.3 Performance

Performance is essential to maintain a smooth and responsive gaming experience. To keep latency low, the architecture makes use of asynchronous communication protocols provided by Socket.IO and OkHttp WebSocket. This ensures that drawing updates and guess processing happen quickly. Although the added abstraction layers for modifiability may introduce some overhead, the system is designed to balance these trade-offs so that performance remains acceptable.

In summary, our revised architectural approach for DrawGuess uses updated patterns and clear separation of concerns to achieve a system that is both flexible and easy to maintain, while also providing a good user experience and reliable performance.

9 Issues

This section will be updated after future iterations.

10 Changes

Table 2 below outlines our change history.

Table 2: Change history.

Date	Change history	Comments
February 24, 2025	First released version.	None.

This section will be updated after future iterations.

11 Contributions

Table 3: Overview of contributions made by each group member.

Group member	Hours worked
Arya Raeesi	25
Isak Olav Sjøberg	17
Benjamin Færestrand	16
Jacob Weidel	16
Ninon Lisa Trouchet	0

Bibliography

[1] OpenAI, 2025. AI generated image of a horse using DALL-E.

[2] Khan, Nadeem: *From data to vectors: How vector databases revolutionize data storage*, 2024.

[3] Bass, Len, Paul Clements, and Rick Kazman: *Software Architecture in Practice*. Addison Wesley, 2021.

A Mathematical Expressions

For the point distribution system, mathematical expressions are needed. We decided to cap the maximum amount of points per round to 500 points. In addition, points are to be given as integers.

Note that the points given in the illustrations in section 1 don't follow the expressions, but were put there for better understanding of the game system.

There are essentially two expressions, one for a *guesser* and one for the *drawer*.

A.1 Mathematical Expression for Point Distribution for a *Guesser*

The score for a *guesser* depends on how quickly they guess the correct answer. Let T_{max} be the total amount of time per round, T_{guess} be the guessing time, $S_{max} = 500$ be the maximum score per round, and $S_{guesser}$ be the *guessers* score per round. This leaves us with the expression in Equation 1 below:

$$S_{guesser}(T_{guess}) = \begin{cases} \lfloor S_{max}(1 - \frac{\log(T_{guess}+1)}{\log(T_{max}+1)}) \rfloor, & 0 \leq T_{guess} \leq T_{max} \\ 0, & \text{Otherwise} \end{cases} \tag{1}$$

Note that the floor function is used to ensure integers being given as points.

A.2 Mathematical Expression for Point Distribution for the *Drawer*

Using the variables defined in section A.1, we can create a mathematical expression for the *drawer* too. Let S_{drawer} be the total points for the *drawer*, $N_{guessers}$ be the amount of *guessers*, and we can get the formula in Equation 2 below:

$$S_{drawer} = \lfloor \frac{1}{N_{guessers}} \sum_{i=1}^{N_{guessers}} (S_{guesser,i}(T_{guess,i})) \rfloor, \quad 0 \leq T_{guess,i} \leq T_{max} \quad \forall i \tag{2}$$

where $S_{guesser,i}(T_{guess,i})$ uses Equation 1 for all players, where i denotes each player in a sum.

Note that Equation 2 uses the same methodology as Equation 1, but adds an averaging and summation mechanism of all *guessers*. This ensures that the *drawer's* performance is evaluated based on the collective difficulty of the guessing process, making it a fair scoring system. By considering multiple *guessers'* response times, the game rewards clear and recognizable drawings rather than simply granting a fixed score per round.

Lastly, Equation 1 and 2 use the $\log_{10}$-base, since it will lead to extreme point reductions for larger T_{guess}. This will give *guessers* incentive to answer as quick as possible, leading to a more exciting and fast-paced game environment.