

Draw It or Lose It

# **CS 230 Project Three Software Design**

Version 1.0

## Table of Contents

[**CS 230 Project Three Software Design** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 3**](#_Toc115077323)

[**Domain Model 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 10/16/2025 | Jeremy Schultz | Completed initial software design document with project requirements and proposed solution. |

## [Executive Summary](#_sbfa50wo7nsh)

I was asked by The Gaming Room to help take their Android-only game, *Draw It or Lose It*, and make it work across multiple platforms. The goal is to build a modern, responsive web app that runs smoothly on both desktop and mobile devices, with a back end that can handle thousands of players at once. My plan is to keep the server-side setup simple, make sure every game, team, and player has a unique name, and use a single GameService to manage active games and assign IDs. In this document, I go over Linux, macOS, Windows, and mobile platforms to figure out which options work best for hosting, development, and compatibility so The Gaming Room can make the right call moving forward.

## Requirements

***Business Requirements:***

* Move from Android-only to a cross-platform, web-delivered experience.
* Allow teams and players to create/join games easily from desktop and mobile browsers (and optionally native shells).
* Enforce unique names for games, teams, and players.
* Keep a single authoritative game service so duplicate game instances don’t cause confusion.

***Technical Requirements:***

* Support multiple teams and players per game; each entity has a unique identifier.
* Provide a responsive HTML front-end that works across major browsers and on mobile devices.
* Provide a scalable server-side web API to manage game state and coordinate rounds.
* Design with production concerns in mind: persistence, concurrency, horizontal scaling, secure communication.

## [Design Constraints](#_2et92p0)

* Right now, the prototype stores everything in memory, but the final version will need a real database or shared cache to keep data safe and consistent.
* The current setup checks for unique names by looping through lists, which works for testing, but in production we’ll need proper database constraints to handle larger scale.
* The prototype isn’t thread-safe yet, so we’ll need to make sure it can handle multiple requests at the same time without issues using locks, transactions, or other concurrency controls.
* There’s no data persistence right now, meaning everything resets when the server restarts. We’ll fix that by adding a relational database like PostgreSQL for the main game data.
* To scale across multiple servers or containers, we’ll use external storage for sessions and state (like a database or Redis) and design the app to stay stateless.

## [System Architecture View](#_ilbxbyevv6b6)

I’m suggesting a simple three-tier web setup:

* **Presentation (Client):** A responsive HTML and JavaScript front end that works in any modern browser, whether on desktop (Linux, macOS, Windows) or mobile (Android, iOS). We could also wrap it as a progressive web app (PWA) or lightweight native app if needed.
* **Application (Server):** This layer handles the main logic using REST APIs and WebSockets to manage games, teams, players, and live drawing updates. It’ll be mostly stateless, with important data saved to a database or cache.
* **Data (Storage):** We’ll use PostgreSQL for core game data and Redis (or Memcached) for fast, temporary storage like game states and leaderboards. Images and other assets will live in cloud storage, such as S3 or a CDN, for better performance.

The client and server will communicate over HTTPS, and real-time game actions like timers and drawing updates will be sent securely through WebSockets.

## [Domain Model](#_8h2ehzxfam4o)

The domain involves the following main classes/entities:

* **Game:** id, name, teams, state (waiting, active, finished), settings (roundLength, roundsCount).
* **Team:** id, name, players, score.
* **Player:** id, name, connection/session id.
* **GameService:** Singleton responsible for game lifecycle management, id generation, and coordination of timers and round events.

The model enforces unique names at the appropriate scope and uses IDs as the canonical references. In production these uniqueness rules are enforced at the database/schema level and validated by the service.

**"The Gaming Room UML diagram. The top of the diagram is labeled as com dot gamingroom. Test boxes are placed in two layers. The first layer has three text boxes and the second layer has four of them. In the first layer, the 'ProgramDriver' textbox points to 'SingletonTester' textbox. The 'ProgramDriver' textbox contains the text 'asterisk main round brackets.' The 'SingletonTester' textbox contains the text 'asterisk testSingleton round brackets.' The arrow between these two text boxes are labeled 'open two angle brackets uses close two angle brackets'. In the second layer, there are 'GameService', 'Game', 'Team', and 'Player' text boxes. The 'GameService' textbox has texts arranged in two layers. The first layer contains games colon List open angle bracket Game close angle bracket, nextGamesId colon long, nextPlayer Id colon long, nextTeamId colon long, and service colon GameService. The second layer contains GameService round brackets, getinstance round brackets colon GameService, addGame open parenthesis name colon String close parenthesis colon Game, getGame open parenthesis id colon long close open parenthesis colon Game, getGame open open parenthesis name colon String close open parenthesis colon Game, getGameCount round brackets colon int, getNextPlayerID round brackets colon long, and getNextTeamId round brackets colon long. The 'GameService' box is connected with the 'Game' textbox with a line labeled 'zero dot dt dot asterisk'.  The 'Game' textbox also contains text in two layers. The first layers contains the text teams colon List open angle bracket Team close angle bracket. The second layer has Game open round bracket id colon long comma name colon String close parenthesis, addTeam open parenthesis name colon String close parenthesis Team, toString round brackets colon String. The 'Game' textbox is connected with the 'Team' textbox with a line labeled 'zero dot dt dot asterisk'. The 'Team' textbox also contains text in two layers. The first layers contains the text players colon List open angle bracket Player close angle bracket. The second layer has Team open parenthesis id colon long comma name colon String close parenthesis, addPlayer open parenthesis name colon String close parenthesis colon Player, and toString round brackets colon String. The 'Team' textbox is connected with the 'Player' textbox with a line labeled 'zero dot dt dot asterisk'. It contains the text Player open parenthesis id colon long comma name colon String close parenthesis and toString round brackets colon String. The 'Game', the 'Team, and the 'Player' boxes point to the 'Entity' textbox in first layer. The 'Entity' textbox contains text in two layers. The first layer has the text id colon long and name colon String. The second layer has Entity round brackets, Entity open parenthesis id colon long comma name colon String close parenthesis, getId round brackets colon long, getName round brackets colon String, toString round brackets colon String.**

## [Evaluation](#_2o15spng8stw)

Below I evaluate Linux, macOS, Windows, and Mobile Devices against Server Side, Client Side, and Development Tools considerations.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS works great for local development since it has a Unix-style setup, strong developer tools like Xcode, and decent Docker support through Docker Desktop. That said, it’s not really used for large-scale web hosting. The hardware and licensing costs are higher, cloud providers don’t offer much support for macOS servers, and there’s a smaller community around running servers on it. It can handle small proof-of-concept projects, but it’s not a practical or cost-effective option for a scalable production setup. | Linux is pretty much the go to system for hosting web apps. It’s lightweight, stable, secure, and works with just about every cloud provider out there. Most Linux distributions like Ubuntu, Debian, or CentOS are free to use, which helps keep costs down. It also has great support for containers like Docker and orchestration tools like Kubernetes, making it easy to scale web services. The only real downside is that managing Linux servers can take some technical know-how, but using managed cloud services for things like databases or Kubernetes can make that a lot easier. | Windows Server is a solid, reliable platform that fits really well with Microsoft’s ecosystem like Active Directory, IIS, and .NET. It’s a good option for teams that already use a lot of Microsoft tools since it can easily host web apps built with .NET or similar frameworks. The downsides are the licensing costs, the fact that it usually uses more system resources, and that it doesn’t have as big of an open-source community as Linux does. | Mobile devices (Android/iOS phones and tablets) are not suitable as server hosts for a production web application. They can act as developer test clients or for small-scale demos but lack reliable networking, persistent availability, and the system management tooling required for production hosting. They are excluded as hosting platforms in any serious deployment plan. |
| **Client Side** | macOS users can play the game through modern browsers like Safari, Chrome, or Firefox, and everything should work fine since they all follow web standards. The main thing to watch for is Safari, it sometimes handles features like WebSockets or CSS a bit differently, so testing on it is a must. A native macOS app isn’t really needed since the responsive web version already covers everything. | Linux users will play through browsers like Firefox or Chrome, and everything should work fine. The user base is smaller, so testing might not be as extensive, but Linux has the benefit of being low cost and great for open-source tools. Just make sure to test across a few different browsers and distributions. | Windows represents the largest desktop market share; Chrome and Edge are the most common browsers. Windows compatibility testing is essential. Edge is Chromium-based, so behavior matches Chrome largely, but testing for OS-specific input methods, scaling (DPI) differences, and default security settings is needed. | Mobile users are a big priority, so the front end needs to work well on different screen sizes, touch controls, and slower connections. iOS can be a bit picky, especially with Safari and App Store rules, while Android is more flexible and supports PWAs better. We’ll just need plenty of testing across different devices and versions to keep things running smoothly. |
| **Development Tools** | Front-end usually uses JavaScript/TypeScript (React, Vue, Angular), while back-end can be Node.js, Java, Python, or Go. Tools include VS Code, IntelliJ, and Xcode for iOS. You’ll need a Mac to build and sign iOS apps. Xcode is free, but at least one mac is essential for iOS testing and builds. | Languages: Java (Spring Boot), Node.js, Python, Go, etc. Tools: VS Code, IntelliJ, Docker, command-line toolchain. Linux tooling is free and widely available. For server-side developers and DevOps, Linux is the most natural environment. There are no licensing costs for most open-source tools. If the team uses enterprise IDEs (IntelliJ Ultimate), there may be licensing costs per developer. | Languages/tools: .NET (C#), Visual Studio IDE, Node.js, Java. Visual Studio has both free (Community) and paid (Professional/Enterprise) editions. If the back end is .NET-based, Windows dev environments may be preferred; however .NET Core/.NET 6+ runs cross-platform so Windows-only development is not required. | For Android, use Kotlin/Java with free Android Studio. For iOS, Swift/Objective-C with Xcode on a Mac. Cross-platform options like React Native or Flutter save work but can have platform quirks. Most tools are free, though paid libraries or services could add costs. |

## Recommendations

**Recommendations for The Gaming Room:**

**1. Operating Platform**

For The Gaming Room, I recommend using Linux as the main server platform. Linux is reliable, cost-effective, and has strong support from major cloud providers like AWS, Google Cloud, and Azure. It’s also open source, which keeps costs down compared to something like Windows Server. Linux works great with container technologies like Docker and orchestration tools such as Kubernetes, which makes scaling easier when more players join. Since the goal is to run Draw It or Lose It across multiple platforms, Linux gives the flexibility and performance needed to support that kind of setup.

**2. Operating System Architectures**

Linux uses a monolithickernel architecture, meaning the kernel handles system calls, device drivers, and file management directly. This helps keep the system fast and efficient since it reduces overhead when communicating with hardware or running processes. It’s very stable and secure, which is perfect for hosting a high-traffic web game. The Linux file structure and permission system also make it easier to isolate processes, which helps prevent one service from affecting others. That’s important for a distributed system like this, where the web server, database, and caching layers might all run separately.

**3. Storage Management**

For storage, I’d use PostgreSQL as the main database and Redis for caching. PostgreSQL is powerful and supports data integrity through transactions and constraints, which helps maintain unique names for games, teams, and players. Redis stores temporary data like active sessions or in-progress games, so everything feels fast and responsive. Both of these work great on Linux and can be hosted in the cloud for better uptime and scalability. Using a combination of persistent (PostgreSQL) and volatile (Redis) storage keeps the system balanced between speed and reliability.

**4. Memory Management**

Linux has strong memorymanagement built into its kernel. It uses techniques like paging, virtualmemory, and caching to make sure applications have access to memory efficiently without crashing the system. For Draw It or Lose It, this means the server can handle multiple game sessions at once without slowing down. When memory starts to fill up, Linux uses swap space and prioritizes active processes, which helps keep the most important game data running smoothly. If we containerize the app, Docker limits can also control how much memory each container uses to prevent overloads.

**5. Distributed Systems and Networks**

Since The Gaming Room wants Draw It or Lose It to connect across multiple platforms, a distributedsystem setup makes the most sense. Each part of the game (like the web server, database, and real-time communication server) can run as its own container or microservice. These are connected over a secure network using RESTAPIs for normal requests and WebSockets for real-time gameplay updates. Cloud load balancers can handle traffic between servers so if one goes down, another picks up the load automatically. Tools like Kubernetes help manage these containers and keep them running even if there’s a network issue or outage, which keeps gameplay reliable.

**6. Security**

Security is a huge priority. On Linux, user permissions, firewalls (like UFW or iptables), and encryption tools provide a strong base. The game should always use HTTPS for all communication, and sensitive data like passwords should be hashed using a strong algorithm such as bcrypt. For data moving between servers, SSL/TLS encryption keeps things secure. Access to the database and API should be restricted by role and IP range, and regular patches and updates should be applied to prevent vulnerabilities. Using a cloud-managed database also adds extra security through automatic backups and encryption at rest.