

Draw It or Lose It

# **CS 230 Project Software Design**

Version 3.0

## Table of Contents

[**CS 230 Project Software Design** 1](#_Toc194608193)

[Table of Contents 2](#_Toc194608194)

[Document Revision History 2](#_Toc194608195)

[Executive Summary 3](#_Toc194608196)

[Requirements 3](#_Toc194608197)

[Design Constraints 4](#_Toc194608198)

[System Architecture View 5](#_Toc194608199)

[Domain Model 5](#_Toc194608200)

[Evaluation 6](#_Toc194608201)

[Recommendations 9](#_Toc194608202)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 03/23/25 | Rick Goshen | Initial draft of the software design document completed. Includes Executive Summary, Requirements, Design Constraints, and Domain Model sections with alignment to project guidelines, UML diagram, and industry best practices. |
| 2.0 | 04/03/25 | Rick Goshen | Expanded platform and tools evaluation section with detailed analysis of server-side scalability, client compatibility, and development tool licensing across all target platforms. Refined design constraints to address performance, rendering, timing, and data limitations expected in a cross-platform, browser-based game environment. |
| 3.0 | 04/15/2025 | Rick Goshen | Expanded the Recommendations section to include operating platform selection, OS architecture, storage and memory management, distributed system design, and security strategy—aligned with Draw It or Lose It deployment goals across platforms. |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room has tasked Creative Technology Solutions with designing a web-based version of their existing Android game, Draw It or Lose It. This game challenges teams to guess puzzles based on progressively revealed drawings, with gameplay spanning four timed rounds. As the game expands to web platforms, it must support multiple teams, multiple players per team, unique naming for games and teams, and enforce a single active game instance in memory at a time.

To address these requirements, we propose a scalable, object-oriented design that leverages standard software design patterns, most notably the Singleton pattern, to ensure only one game instance exists at any time. The solution also incorporates inheritance, composition, and encapsulation principles to manage game state and object relationships efficiently.

This design will form the foundation of a robust, maintainable, and platform-independent application architecture. The proposed implementation aligns with industry best practices and will enable The Gaming Room to extend its game across multiple platforms while maintaining consistent performance and game logic integrity.

## Requirements

The following functional and technical requirements have been defined for developing the web-based version of Draw It or Lose It. These requirements ensure a scalable, interactive, and platform-ready application experience.

**Functional Requirements**

1. Multi-Team Support: The game must allow one or more teams to participate in a single game session.
2. Multi-Player Teams: Each team must be able to support multiple players.
3. Unique Naming System: Game and team names must be unique to prevent duplication and enable name validation during creation.
4. Game Flow Logic: The game consists of four rounds, each lasting one minute. At the 30-second mark, drawing clues are completed. Unsolved puzzles are available for guesses from the remaining teams within a 15-second window.
5. Team Turn Order: Teams must take turns guessing during their respective rounds; rules must enforce proper turn rotation and time limits.

**Technical Requirements**

1. Singleton Game Instance: The system must ensure that only one instance of the game service exists in memory at any time, which is achieved using the Singleton design pattern.
2. Unique Identifiers: Each game, team, and player object must have a unique identifier for tracking and retrieval.
3. Platform Compatibility: The application must be accessible via modern web browsers and support responsive layouts for mobile and desktop use.
4. Object-Oriented Structure: To support maintainability and extensibility, the codebase must follow object-oriented principles such as encapsulation, inheritance, and modular design.
5. Scalability: The design must accommodate the possibility of additional features (e.g., score tracking, chat, replay) without requiring significant refactoring.

## [Design Constraints](#_2et92p0)

The development of Draw It or Lose It as a web-based, distributed application introduces several design constraints that must be addressed to ensure technical feasibility, performance, and maintainability across platforms.

**Technical Constraints**

Single Instance Enforcement: Only one instance of the game service may exist in memory at any time. This requires implementing the Singleton design pattern, which limits instantiation and centralizes control.

Unique Entity Identification: All game-related entities (games, teams, players) must have unique identifiers to ensure data integrity and enable efficient retrieval and validation processes.

Object-Oriented Design: The system architecture must follow object-oriented principles such as inheritance, encapsulation, and composition. These principles add complexity but improve modularity and reuse.

Platform Independence: The application must be web-based and browser-agnostic, meaning all design decisions—from language selection to UI rendering—must support cross-platform compatibility. Browser differences in rendering canvas elements and handling animations impose limitations on how drawing features can be implemented. Additionally, timing mechanisms such as turn countdowns and drawing time limits must operate independently of server state and may vary across browsers or device performance.

**Environmental Constraints**

Web-Based Distributed Environment: The game must function reliably in a distributed architecture, where clients interact over HTTP through stateless protocols. This constrains session-based state management and necessitates careful handling of concurrency and shared resources. Stateful elements like game rounds, scores, and active users must be tracked on the client side or passed via APIs since persistent server-side infrastructure is not guaranteed in early deployment

Client Network Variability: Users will connect from devices with varying network speeds and hardware capabilities, requiring lightweight front-end components and efficient backend communication. Graphical rendering and canvas updates must be optimized to avoid lag or jitter, especially on low-powered mobile devices.

**Project-Related Constraints**

No Existing Web Infrastructure: The Gaming Room is transitioning from an Android-only product and currently lacks a web development environment. The design must assume no pre-existing web services, APIs, or hosting capabilities.

Scalability for Future Features: While the initial scope is focused on gameplay mechanics, the design must leave room for future enhancements like scoreboards, user accounts, chat features, or analytics without requiring significant redesign. Scorekeeping and leaderboard features must initially function with local or session-based data, as persistent storage may not be available in early iterations.

## [System Architecture View](#_ilbxbyevv6b6)

Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and storage aspects is also necessary to understand the overall architecture and should be provided.

## [Domain Model](#_8h2ehzxfam4o)

The domain model for *Draw It or Lose It* consists of seven primary classes that define the structure and relationships between the core entities of the game. The model demonstrates clear object-oriented design, applying principles such as inheritance, encapsulation, and composition to meet the software requirements efficiently.

**Class Relationships and Structure**

At the top of the hierarchy is the *Entity* class, which contains shared attributes *id* and *name*. This class is extended by *Game*, *Team*, and *Player*, allowing for code reuse and centralized control of common functionality. This use of inheritance reduces duplication and maintains consistency across all game components.

The *GameService* class acts as the core controller and is implemented using the singleton design pattern. It stores a list of all *Game* instances and uses internal counters such as *nextGameId*, *nextTeamId*, and *nextPlayerId* to generate unique identifiers. This ensures that only one instance of *GameService* exists in memory at a time, aligning with the design constraint requiring a single active game service.

The *Game* class manages a list of *Team* objects, and each *Team* manages a list of *Player* objects. These are examples of composition, where:

* A *Game* has one or more *Teams*
* A *Team* has one or more *Players*

Each class provides constructors and public methods for controlled interaction. For example, the *Game* class includes the *addTeam()* and *getGameCount()* methods, while the *Team* class provides *addPlayer()*. These methods encapsulate logic for managing internal collections and ensure safe access to object data.

The *ProgramDriver* class contains the application’s entry point through the *main()* method and uses the *SingletonTester* class to verify that only one instance of *GameService* is ever created and used.

**Object-Oriented Programming Principles**

* **Inheritance**: The *Entity* class is extended by *Game*, *Team*, and *Player*, supporting shared behavior and structure.
* **Encapsulation**: Internal fields such as *id*, *name*, and lists of teams or players are kept private and accessed via public getter methods like *getId()* and *getName()*.
* **Composition**: The *Game* class contains *Teams*, and *Teams* contain *Players*, modeling real-world relationships and enabling nested object management.
* **Singleton Pattern**: The *GameService* class is designed as a singleton, ensuring centralized control, shared access, and consistent state management across the application.

This domain model provides a modular, scalable structure that directly supports the project requirements and can be extended for future features such as scoring, user authentication, or real-time multiplayer functionality.

**"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)

To ensure the “Draw It or Lose It” application is scalable, accessible, and compatible across devices, the server-side and client-side requirements were evaluated across macOS, Linux, Windows, and mobile platforms. Additionally, development tools and technical requirements were analyzed to understand the impact on the development team and project cost.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | **Characteristics:** Unix-based OS with support for Apache and Nginx.  **Advantages:** Built-in developer tools, Unix shell, and stability.  **Weaknesses:** No official macOS server environment supported by cloud providers; not viable for scalable hosting.  **Licensing Costs (to The Gaming Room):** Requires Apple hardware; no standalone server license; high infrastructure cost due to hardware. | **Characteristics:** Open-source, industry-standard for server deployments.  **Advantages:** Scalable, secure, lightweight, and fully supported by cloud providers.  **Weaknesses:** CLI-heavy and requires technical expertise; variation across distributions.  **Licensing Costs (to The Gaming Room):** Free for most distros; optional support available (Red Hat: ~$349–$1,299/year). | **Characteristics:** Commercial OS with full support for Java and .NET applications.  **Advantages:** Integration with Microsoft tools; familiar to enterprise environments.  **Weaknesses:** Higher resource usage and licensing costs; more frequent updates.  **Licensing Costs (to The Gaming Room):** Windows Server Standard: ~$1,069; Datacenter: ~$6,155; CALs may apply. | **Characteristics:** Not suitable for server hosting.  **Advantages:** None applicable.  **Weaknesses:** Limited hardware, no cloud or enterprise support.  **Licensing Costs (to The Gaming Room):** Not applicable; mobile OS cannot be used for server deployment. |
| **Client Side** | **Cost:** No software licensing cost; Apple hardware required.  **Time:** Must test in Safari and Chrome on Retina displays.  **Expertise:** Familiarity with WebKit (Safari engine) and macOS accessibility features.  Compatibility requires use of CSS media queries, responsive layouts, and testing for high-DPI screens and macOS gestures. | **Cost:** Free tools; browser and CLI-based testing supported.  **Time:** Testing across multiple Linux distros may increase QA overhead.  **Expertise:** Developers must handle variation in rendering engines and desktop environments.  Compatibility requires strict standards compliance and open-source testing tools. | **Cost:** Windows license may be needed for testing; tools are otherwise free.  **Time:** Testing required across Edge, Chrome, and Firefox.  **Expertise:** Considerations include high-DPI support, accessibility, and touch features on hybrid devices.  Compatibility requires validation of layout, scaling, and input behavior. | **Cost:**  Android: Free testing tools.  iOS: Requires Apple Developer Account ($99/year) and hardware.  **Time:** High testing overhead across screen sizes, orientations, and mobile browsers.  **Expertise:** Developers must use responsive web design and mobile-first layouts, and validate touch input and performance on real devices.  Compatibility requires proper testing with Chrome (Android) and Safari (iOS), handling mobile gestures, and optimizing for limited screen real estate. |
| **Development Tools** | **Languages & Tools:** Java, HTML, CSS, JavaScript; Safari Web Inspector, VS Code, IntelliJ\*, Xcode (for iOS testing).  **Impact:** Apple hardware required for full testing; shared testing devices may be needed.  **Licensing Costs:** Free tools; Xcode requires Apple hardware. | **Languages & Tools:** Java, HTML, CSS, JavaScript, Bash; VS Code, Eclipse, IntelliJ\*, Firefox Dev Tools.  **Impact:** Highly efficient for back-end/server development; ideal for CI/CD and scripting.  **Licensing Costs:** Free and open-source tools; optional support available. | **Languages & Tools:** Java, HTML, CSS, JavaScript, PowerShell; VS Code, Eclipse, IntelliJ\*, Edge Dev Tools.  **Impact:** Accessible to general development teams; supports all required front-end and back-end workflows.  **Licensing Costs:** Windows license (~$100–200); all major dev tools are free. | **Languages & Tools:** HTML, CSS, JavaScript; Chrome DevTools, Android Studio, Safari Web Inspector.  **Impact:** Requires extensive mobile testing; tools needed for layout, touch, and performance validation.  **Licensing Costs:**  Android: Free.  iOS: Requires Apple Developer Account ($99/year). |

**\*** IntelliJ IDEA Community Edition is free and supports Java development but lacks advanced features like Spring Boot support, database tools, and version control integrations. IntelliJ IDEA Ultimate (commercial edition) is available for ~$169/year (1st year) per developer, with discounts for subsequent years. WebStorm (JetBrains’ dedicated HTML/CSS/JS IDE) offers more complete web front-end tooling but is only available as a paid product, starting at ~$69/year per developer.

## Recommendations

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

1. **Operating Platform**:

The recommended operating platform for deploying *Draw It or Lose It* is **Ubuntu Server LTS**, a widely used Linux-based system that offers strong performance, scalability, and platform compatibility.

Linux is open-source and free to use, with no licensing fees. This dramatically reduces the total cost of ownership compared to commercial server operating systems like Windows Server, which require upfront and recurring licensing costs (Silberschatz et al., 2018). The cost-effectiveness of Linux aligns well with The Gaming Room’s budget-conscious expansion goals..

In terms of scalability and flexibility, Ubuntu Server supports modern deployment models, including virtualization and containerization (e.g., Docker, Kubernetes). This enables The Gaming Room to host thousands of concurrent game sessions efficiently and scale their services horizontally as player demand increases (Linux.com, n.d.).

Linux has long been adopted in the web and gaming industries and is the foundation of platforms such as Steam and AWS EC2. The global development community ensures that the system is constantly updated, supported, and backed by extensive documentation and tools—all freely available.

Because *Draw It or Lose It* is designed to run in modern web browsers, the technology stack (Java, HTML5, CSS3, JavaScript) must be fully supported by the hosting platform. Ubuntu Server is highly compatible with Java 8 and widely used web servers like Apache and Nginx, as well as databases such as MySQL and PostgreSQL.

Security is another major advantage. Ubuntu allows system administrators to apply fine-grained access controls, firewall rules, and data encryption measures. This control is critical for protecting user data across distributed client environments.

Finally, Linux is well-suited to distributed system architectures. It supports APIs, microservices, and cloud-based deployment strategies that are essential for delivering a consistent game experience across web and mobile clients (Arsov, 2022; Linux.com, n.d.).

Ubuntu Server LTS offers the cost savings, reliability, scalability, and distributed architecture support necessary to power *Draw It or Lose It* into its next phase of growth across platforms.

1. **Operating Systems Architectures**:

Ubuntu Server is built on the monolithic Linux kernel and supports a modular, layered architecture optimized for high-performance, scalable, and distributed systems—ideal for hosting modern web applications like *Draw It or Lose It*.

The Linux kernel uses a monolithic architecture, meaning all core functionalities (process management, memory management, device drivers, and file systems) run in kernel space. However, Linux retains modularity by allowing components to be dynamically loaded or unloaded as needed. This design provides the performance advantages of a monolithic system with the flexibility of a modular one.

Ubuntu separates user space from kernel space, isolating application processes from system-level operations. This means game services like matchmaking, session management, and image rendering will run in user space, minimizing the risk of a system-wide failure due to a single service crash.

The system supports preemptive multitasking and uses the Completely Fair Scheduler (CFS) to handle thread prioritization. This ensures that multiple services and game sessions can run simultaneously without blocking each other, enabling real-time responsiveness for concurrent players.

Ubuntu Server is compatible with multiple file systems (e.g., ext4, XFS, Btrfs) and supports virtual memory through paging and swap space. It also offers full support for containers (Docker, LXC) and virtual machines (KVM), allowing the application to scale horizontally and deploy isolated microservices.

Finally, Ubuntu’s modular networking stack includes tools like iptables/nftables, Netplan, and NetworkManager. This enables custom firewall rules, virtual private network (VPN) support, IPv6 compatibility, and HTTP/2—all of which are essential for the secure and efficient communication required in distributed gaming environments.

Ubuntu Server’s kernel-level performance, modularity, and robust user-kernel separation provide a highly secure and scalable architecture. These capabilities are critical to supporting *Draw It or Lose It* across distributed systems, where responsiveness, uptime, and modular service deployment are essential.

1. **Storage Management**:

For Ubuntu Server LTS, the most appropriate storage management solution for *Draw It or Lose It* includes a combination of the ext4 file system, a PostgreSQL relational database, a content delivery network (CDN), and optional cloud-based storage integration. This hybrid approach addresses both local performance and long-term scalability needs.

The ext4 file system, Ubuntu’s default, offers strong journaling, fast file access, and reliability. It is well suited for caching session data and storing temporary assets during active gameplay. It also supports large file sizes and efficient metadata management, which benefits the application’s need to load high-resolution image clues.

For managing structured data such as user accounts, teams, game histories, and scoreboards, PostgreSQL is a robust and scalable choice. It is open-source, well-integrated with Ubuntu, and supports advanced indexing and relational querying. These features ensure that game data can be retrieved and written efficiently, even as the user base grows.

The application’s heavy reliance on visual clues requires rapid access to a growing library of image files. To maintain performance, a content delivery network (CDN) should be used to serve static assets such as images. This reduces server load and latency by delivering content from edge locations geographically closer to users.

Finally, for long-term growth, the system should integrate with a cloud-based object storage solution such as Amazon S3 or Azure Blob Storage. These platforms offer redundancy, automatic backups, and elastic scaling, ensuring that the application can accommodate future data growth without compromising performance or security.

Ubuntu Server’s flexibility allows for seamless integration of these storage layers, providing The Gaming Room with a resilient, efficient, and scalable solution to support the ongoing development and expansion of *Draw It or Lose It*.

1. **Memory Management**:

Ubuntu Server LTS employs an advanced memory management system based on the Linux kernel, which uses a combination of paging, virtual memory, demand loading, memory caching, and garbage collection techniques. These methods provide efficient memory utilization for resource-heavy applications like *Draw It or Lose It*.

The system’s use of paging allows physical memory to be broken into fixed-size blocks, which are then mapped to virtual memory. This enables the execution of applications that exceed available RAM by temporarily offloading inactive memory pages to swap space. For *Draw It or Lose It*, which stores over 1.6 GB of high-resolution image assets, paging ensures that memory is used efficiently and that the game remains stable even when resource demands peak (Silberschatz et al., 2018).

Ubuntu also supports demand paging, where portions of files or programs are only loaded into memory when accessed. This is ideal for the game’s clue system, as it prevents the entire image library from being loaded into memory at once. Instead, only the relevant image is accessed during each round, conserving RAM and reducing latency.

The Linux kernel caches frequently accessed files in memory and supports memory pooling to minimize fragmentation and reuse memory blocks. This enhances performance during gameplay, particularly when users interact with common UI assets or repeat visual content across rounds.

Since the game is written in Java, it runs on the Java Virtual Machine (JVM), which uses configurable garbage collection algorithms to manage heap memory. Ubuntu allows the JVM to be tuned with options like the G1 or CMS collectors, enabling developers to optimize memory performance for high-throughput or low-latency scenarios, depending on gameplay demands.

Additionally, Ubuntu employs kernel-level memory protections such as the swapper daemon and Out-of-Memory (OOM) killer, which safeguard system stability by reclaiming memory from non-critical processes in extreme conditions. These features help maintain server uptime and prevent crashes under stress.

Ubuntu’s memory management system ensures that *Draw It or Lose It* delivers consistent, smooth gameplay across multiple sessions, even under the constraints of distributed deployment and variable user hardware.

1. **Distributed Systems and Networks**:

To support real-time communication between clients on various platforms, *Draw It or Lose It* must be developed as a distributed system—one in which independent software components across devices and servers operate as a unified experience. Ubuntu Server LTS provides the network infrastructure, tools, and compatibility required to support such a system.

The recommended architecture for this application is a client-server model. Game logic and data management reside on the server, while clients (browsers or mobile apps) act as thin interfaces that interact with the server via a secure internet connection. Using RESTful APIs over HTTPS ensures interoperability across all target platforms—web, iOS, Android, Windows, and Mac—through a standardized communication model.

Ubuntu Server supports popular technologies such as Apache, Nginx, and Java-based frameworks like Spring Boot, which can handle API endpoints, session routing, and user authentication. Communication between clients and the backend should use a format such as JSON, which is lightweight and broadly supported.

To ensure reliable interaction between distributed components, the system must be tolerant of latency, network interruptions, and service dependencies. Stateless APIs help prevent session loss when a connection drops. Clients should implement retry logic, local caching, and background syncing to manage short-term connectivity issues. Additionally, using persistent session identifiers allows users to resume games even after disconnection.

The network layer should include a reverse proxy with load balancing (e.g., Nginx) to manage traffic and distribute requests across instances. For cloud scalability, container orchestration systems like Kubernetes can provide failover support and auto-scaling capabilities during peak usage.

To further support cross-platform reliability, mobile clients should store essential data—such as the last image shown, score, or user state—locally. When network service resumes, this data can be resynchronized with the backend, maintaining continuity.

Ubuntu Server's robust networking, API support, and compatibility with cloud orchestration tools ensure that *Draw It or Lose It* can function as a reliable distributed system across all supported platforms.

1. **Security**:

Protecting user data is critical to the success of *Draw It or Lose It*, particularly as the application communicates between multiple clients over the internet. Ubuntu Server LTS provides a hardened and flexible platform for implementing strong, modern security practices at both the system and application levels.

All communication between clients (mobile, web, desktop) and the server must be conducted over HTTPS using Transport Layer Security (TLS). This ensures that user credentials, scores, and session data are encrypted in transit, protecting them from interception or tampering during transmission.

The application should implement token-based authentication, such as JSON Web Tokens (JWT), to manage user identity across sessions. Tokens should be signed, expire after a set period, and be stored securely on the client side. This protects against session hijacking and ensures that credentials are not repeatedly transmitted.

Ubuntu Server includes a variety of tools and configurations to enhance server-side security. Access to system files is controlled through user permissions and sudo roles. Firewalls such as iptables or nftables can be configured to allow only necessary traffic, and intrusion prevention tools like Fail2Ban can block repeated failed login attempts. Secure Shell (SSH) access should use key-based authentication and disable password logins to reduce vulnerability.

To ensure security across platforms, the application must validate all client-side inputs and use secure storage mechanisms appropriate to the device. iOS clients should use Keychain, Android should use Keystore, and web clients should store temporary session data with expiration and sanitation controls. Cross-Origin Resource Sharing (CORS) should be configured to restrict which domains can access the backend APIs.

Sensitive data stored at rest, such as user progress or game metadata, should be encrypted using file-level encryption like LUKS or application-level techniques. PostgreSQL should use encrypted connections (SSL), hashed passwords, and database user roles with least-privilege access. Backups and media files should be stored in encrypted cloud buckets with access control policies.

Ubuntu Server’s security infrastructure, when paired with best practices in API security, encryption, and authentication, provides The Gaming Room with the tools necessary to safeguard user information while supporting a scalable, distributed game environment.

### References

Arsov, K. (2022). Microservices vs. SOA — Is there any difference at all? Microtica. https://medium.com/microtica/microservices-vs-soa-is-there-any-difference-at-all-4c4cbf594b57

Linux.com. (n.d.). What is a Linux server and why does your business need one? https://www.linux.com/what-is-linux-server/

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating system concepts essentials (2nd ed.). Wiley.

Wikipedia contributors. (n.d.-a). Memory management (operating systems). Wikipedia. <https://en.wikipedia.org/wiki/Memory_management_(operating_systems)>

Wikipedia contributors. (n.d.-b). Computer data storage. Wikipedia. <https://en.wikipedia.org/wiki/Computer_data_storage>

Difference Between. (n.d.). Difference between memory and storage. DifferenceBetween.net. <https://www.differencebetween.net/technology/difference-between-memory-and-storage/>