

Gaming Room - Draw It or Lose It: Software Design Document

# **CS 230 completed software design document** Version 4.0

## Table of Contents

[**CS 230 Project Software Design Template** 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 | 09/17/2025 | DALE davenport | Initial draft based on project requirements |
| 2.0 | 10/02/2025 | DALE davenport | Updated Evaluation section for Project Two |
| 3.0 | 10/16/2025 | DALE davenport | Completed Recommendations section for Project Three with comprehensive analysis of all six required areas |
| 4.0 | 10/21/2025 | DALE davenport | **completed software design document** |

## [Executive Summary](#_sbfa50wo7nsh)

This document outlines the software design for ‘Draw It or Lose It,’ a web-based game developed for The Gaming Room by Creative Technology Solutions (CTS). The Gaming Room aims to expand its existing Android game to a multi-platform web environment. This design focuses on meeting the client’s software requirements, including unique game, team, and player names, and ensuring only a single instance of the game service exists in memory at any given time. The proposed solution leverages object-oriented programming principles and design patterns to create a robust, scalable, and maintainable application. Critical information for the client includes understanding the architectural decisions made to support a web-based, distributed environment, and the implications for future development and deployment across various operating systems and mobile platforms. The document details the domain model, design constraints, and provides recommendations for operating platforms, system architectures, storage, memory management, distributed systems, and security to ensure a successful transition and future growth for ‘Draw It or Lose It.

## Requirements

The system must ensure that only a single instance of the game service exists in memory at any given time, which is achieved through the singleton design pattern. It must also enforce uniqueness for all game, team, and player names to prevent conflicts. The game state must remain consistent and synchronized across multiple platforms and players, with relationships modeled such that a GameService manages multiple games, each game contains multiple teams, and each team includes multiple players. To support this, the iterator pattern is used to efficiently search existing games and teams when adding new entries. Additionally, the system must provide real-time updates, including drawing actions, timers, and chat, across connected clients.

From a non-functional standpoint, the system must be scalable, using microservices and distributed systems to handle an expanding user base, and reliable, preventing conflicting game states while ensuring resilience through mechanisms like load balancing, retries, and circuit breakers. Performance is also critical, with low latency achieved through optimized communication protocols such as WebSockets and API Gateway. Security is addressed through a multi-layered strategy including OAuth 2.0 or OpenID Connect for authentication, role-based access control for authorization, TLS/SSL for data in transit, and strong encryption for data at rest, alongside secure coding practices. The platform must be portable across Windows, Linux, macOS, and mobile devices, and maintainable through the use of object-oriented principles, modular microservices, and clear, well-documented APIs.

## [Design Constraints](#_2et92p0)

The primary design constraint for developing “Draw It or Lose It” in a web-based distributed environment is the need to ensure a consistent and synchronized game state across multiple platforms and players. This requires careful management of data and application logic to prevent inconsistencies and ensure a seamless user experience. The implications of this constraint on application development are significant. For instance, the application must be designed to handle concurrent requests from multiple users, manage shared resources effectively, and maintain data integrity. The use of a singleton pattern for the GameService is a direct response to this constraint, ensuring that all game-related operations are managed through a single, centralized point of control. This prevents the creation of multiple, conflicting game instances and simplifies the management of game state. Furthermore, the requirement for unique game and team names necessitates the implementation of a mechanism to check for existing names before creating new ones. The iterator pattern is employed to efficiently search through the lists of games and teams, ensuring that name uniqueness is enforced without compromising performance. The choice of a web-based architecture also introduces constraints related to network latency and bandwidth, which must be considered when designing the communication protocols between the client and server. The application must be optimized to minimize data transfer and handle network interruptions gracefully to provide a responsive and reliable gaming experience.

## [System Architecture View](#_ilbxbyevv6b6)

The system architecture for "Draw It or Lose It" encompasses both the logical structure of software components and the physical infrastructure that supports the distributed gaming platform. Understanding this architecture is essential for visualizing how components interact, how data flows through the system, and how the platform scales to meet user demand.

### Logical Architecture

The logical architecture of "Draw It or Lose It" follows a microservices-based, three-tier model that separates concerns across presentation, application, and data layers. The presentation tier consists of client applications running on diverse platforms including web browsers on desktop computers, mobile applications on iOS and Android devices, and potentially future platforms. These clients are responsible solely for rendering the user interface, capturing user input, and displaying game state. The presentation layer communicates with backend services exclusively through standardized APIs and WebSocket connections, maintaining a clear separation between user interface concerns and business logic.

The application tier comprises multiple independent microservices, each responsible for a specific domain of functionality. The API Gateway microservice serves as the single entry point for all client requests, handling cross-cutting concerns including authentication, authorization, request routing, rate limiting, and protocol translation. Behind the gateway, specialized microservices handle distinct responsibilities: the Authentication Service manages user registration, login, session management, and token validation; the Game State Management Service implements the core GameService singleton pattern, maintaining active games and coordinating game flow; the Matchmaking Service handles team formation and player assignment to games; the Real-time Communication Service manages WebSocket connections for live drawing updates, timer synchronization, and chat functionality; the Image Service retrieves and serves stock drawing images from object storage; the Leaderboard Service tracks scores, maintains rankings, and generates statistics; and the Notification Service manages alerts and updates sent to players. Each microservice operates independently with its own lifecycle, can be deployed and scaled separately, and communicates with other services through well-defined APIs and message queues.

The data tier encompasses multiple storage systems optimized for different data types and access patterns. A distributed PostgreSQL database cluster stores structured relational data including user profiles, game states, team information, and player records. The database is sharded horizontally across multiple nodes to distribute load and ensure high availability. Object storage, implemented through solutions like Amazon S3, Google Cloud Storage, or self-hosted MinIO, stores unstructured data including the library of stock drawing images and potentially user-generated content. A distributed caching layer using Redis or Memcached sits between the application tier and persistent storage, caching frequently accessed data to reduce database load and improve response times. Message queues using Apache Kafka or RabbitMQ facilitate asynchronous communication between microservices, decoupling services and improving system resilience.

### Physical Architecture and Deployment Topology

The physical architecture describes how software components are deployed across computing infrastructure. In a production environment, the system operates across multiple physical or virtual servers, potentially distributed across geographic regions for performance and redundancy. The client tier operates on user-owned devices worldwide, ranging from personal computers to smartphones and tablets, connecting to the system over the public internet through HTTPS and secure WebSocket connections.

The entry point to the backend infrastructure consists of a Content Delivery Network that caches static assets including JavaScript files, CSS stylesheets, and frequently accessed game images at edge locations worldwide, reducing latency by serving content from locations geographically close to users. Behind the CDN, a Global Load Balancer distributes incoming traffic across multiple regional deployments based on geographic proximity and current system load, implementing DNS-based routing to direct users to the optimal data center.

Within each regional deployment, the infrastructure follows a consistent pattern. An Application Load Balancer receives traffic from the global load balancer and distributes it across multiple instances of the API Gateway service. The load balancer performs health checks on API Gateway instances, automatically removing failed instances from the rotation and adding them back once they recover. The API Gateway layer typically runs on multiple server instances for redundancy and horizontal scalability, with the number of instances adjusted dynamically based on traffic patterns.

The microservices layer operates within a container orchestration platform, specifically Kubernetes, which manages the deployment, scaling, and operation of application containers. Each microservice runs in one or more containerized instances called pods. Kubernetes automatically schedules pods across available worker nodes in the cluster, balancing resource utilization and ensuring high availability by distributing replicas of the same service across different physical servers. The cluster consists of master nodes that manage the overall system and worker nodes that run application workloads. For high availability, multiple master nodes operate in an active-active configuration with automated failover.

The data tier employs multiple strategies for reliability and performance. The PostgreSQL database cluster consists of multiple nodes, with one or more primary nodes handling write operations and multiple replica nodes serving read operations. Replication can be configured for both high availability, where replicas can be promoted to primary in case of failure, and read scaling, where read queries are distributed across replicas. Database servers typically run on dedicated physical or virtual machines with high-performance SSD storage to meet the I/O demands of active gameplay. Object storage for images operates as a distributed system with built-in redundancy, storing multiple copies of each object across different physical locations within and potentially across data centers. The caching layer using Redis runs as a cluster with multiple nodes, providing both high availability through replication and scalability through sharding.

The message queue infrastructure, whether Apache Kafka or RabbitMQ, operates as a distributed cluster with multiple broker nodes. Messages are replicated across multiple brokers for durability, ensuring that temporary failure of individual nodes doesn't result in message loss. Producers and consumers connect to the cluster through load-balanced endpoints, with the messaging system automatically routing operations to available brokers.

### Network Topology and Communication Patterns

The network topology describes how components communicate across the distributed system. Clients communicate with the system through encrypted connections, with HTTPS used for standard request-response operations and secure WebSocket connections (WSS) used for real-time bidirectional communication. These connections terminate at the load balancer, which distributes them to API Gateway instances.

Within the backend infrastructure, microservices communicate through multiple mechanisms depending on the communication pattern required. Synchronous request-response communication between services uses RESTful HTTP APIs or gRPC for higher performance scenarios, with service discovery mechanisms enabling services to locate each other dynamically. Asynchronous communication for operations that don't require immediate responses uses message queues, where producer services publish messages to topics or queues that consumer services process independently. Real-time communication for features like drawing synchronization and chat uses WebSocket connections managed by the dedicated Real-time Communication Service, which maintains persistent connections to clients and coordinates message distribution.

All internal communication within the backend infrastructure occurs over a private network, isolated from the public internet. Microservices within the Kubernetes cluster communicate through the cluster's internal network using Kubernetes Services, which provide stable network endpoints backed by one or more pod instances. Traffic between services can be encrypted using service mesh technologies like Istio or Linkerd, which implement mutual TLS authentication ensuring both parties in a communication verify each other's identity.

Database access follows a hub-and-spoke pattern where microservices that require database access connect to the database cluster through connection pools that manage and reuse database connections efficiently. Services query databases primarily through their assigned shard or replica, with the sharding logic embedded in the data access layer determining which database node handles each operation. The caching layer sits logically between microservices and databases, with services checking the cache before querying the database and populating the cache with query results.

### Scalability and High Availability Architecture

The architecture is designed explicitly for horizontal scalability, meaning capacity is increased by adding more instances rather than upgrading individual servers. Each tier of the application can scale independently based on specific demands. The API Gateway scales by adding more instances behind the load balancer as traffic increases. Individual microservices scale independently, with frequently used services like the Real-time Communication Service potentially running dozens of instances while less frequently used services might run only a few instances. The Kubernetes orchestration platform can automatically scale services based on metrics like CPU utilization, memory usage, or custom metrics like active player count.

High availability is achieved through redundancy at every level. Multiple instances of each component ensure that failure of any single instance doesn't disrupt service. Load balancers continuously monitor component health and automatically route traffic away from failed instances. Databases employ replication with automated failover, where replica nodes can be quickly promoted to primary when failures occur. The entire system can be deployed across multiple availability zones within a data center or across multiple geographic regions, providing resilience against data center outages and enabling disaster recovery capabilities.

### Security Architecture

Security is integrated throughout the architecture rather than being a separate layer. Network security implements multiple perimeter defenses: firewalls restrict access to only necessary ports and services; Virtual Private Cloud (VPC) configuration segments networks with public subnets for load balancers and API Gateways and private subnets for microservices and databases; and network access control lists and security groups define fine-grained rules for what traffic is permitted between components. Application security enforces authentication and authorization at the API Gateway before requests reach internal services, encrypts all communication both external and internal using TLS, and implements secure coding practices across all microservices to prevent common vulnerabilities. Data security encrypts data both in transit using TLS and at rest using database encryption and encrypted storage volumes, implements strict access controls limiting which services can access which data stores, and maintains comprehensive audit logging of all data access and modifications.

This comprehensive system architecture provides "Draw It or Lose It" with a robust, scalable, and secure foundation capable of supporting thousands of concurrent players across multiple geographic regions while maintaining the responsive, real-time gameplay that is central to the gaming experience. The architecture's modularity and use of industry-standard patterns and technologies position the platform for future growth and evolution as new features are added and user demand increases.

## [Domain Model](#_8h2ehzxfam4o)

provided UML class diagram illustrates the domain model for the ‘Draw It or Lose It’ game application. This model is designed to be both scalable and maintainable, adhering to key object-oriented programming principles. The diagram consists of seven classes: ProgramDriver, SingletonTester, Entity, GameService, Game, Team, and Player.

The Entity class serves as a foundational base class, encapsulating the common attributes of id (a unique long integer) and name (a String). This is a clear example of abstraction, as it simplifies the design by factoring out common properties into a single, reusable component. The Game, Team, and Player classes all inherit from the Entity class, demonstrating the principle of inheritance. This relationship is depicted in the UML diagram by the open arrows pointing from Game, Team, and Player to Entity. By inheriting from Entity, these classes automatically acquire the id and name attributes, as well as the associated getId(), getName(), and toString() methods, promoting code reuse and reducing redundancy.

The GameService class is implemented as a singleton, a creational design pattern that ensures only one instance of the class can exist in memory at any given time. This is a critical requirement for the application, as it centralizes the management of game state and prevents inconsistencies that could arise from multiple, independent game service instances. The getInstance() method provides a global point of access to the singleton instance. The GameService class maintains a list of active Game objects and is responsible for creating, retrieving, and managing games. The relationship between GameService and Game is a one-to-many association, as one GameService can manage multiple Game instances (indicated by the ’0…\*’ multiplicity).

The Game class represents a single game session. It contains a list of Team objects, reflecting the one-to-many relationship between a Game and its participating Teams. Similarly, the Team class holds a list of Player objects, representing the one-to-many relationship between a Team and its members. These relationships demonstrate the principle of composition, where a Game is composed of Teams, and a Team is composed of Players.

The ProgramDriver class contains the main() method, which serves as the entry point for the application. It utilizes a SingletonTester class to verify the correct implementation of the GameService singleton pattern. This separation of concerns, where the ProgramDriver is responsible for application startup and the SingletonTester is dedicated to testing a specific design pattern, is a good software engineering practice.

To fulfill the requirement of unique game and team names, the addGame() and addTeam() methods will employ the iterator pattern. This pattern provides a way to access the elements of an aggregate object (in this case, the lists of games and teams) sequentially without exposing its underlying representation. By iterating through the existing games and teams, the application can efficiently check for name conflicts before adding new entries, thus ensuring the uniqueness of names as required by the client.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac operating systems, primarily macOS Server, can host web applications, but it is not a common choice for large-scale production environments due to its niche market share and higher hardware costs. While it offers a Unix-based foundation, its server-side ecosystem is less mature compared to Linux or Windows Server. Licensing costs for macOS Server are typically minimal (e.g., ~$20 for the app itself [1]), but the primary cost is the dedicated Apple hardware, which can be significantly higher than commodity server hardware. Scalability can be a concern for high-traffic applications. | Linux is a highly popular and robust choice for hosting web-based applications due to its open-source nature, stability, and strong community support. Distributions like Ubuntu Server, CentOS, and Red Hat Enterprise Linux offer excellent performance and security. It provides a wide array of web servers (Apache, Nginx) and database options. Licensing costs for the operating system itself are generally zero for open-source distributions. For enterprise-grade support and features (e.g., Red Hat Enterprise Linux, Ubuntu Pro), costs can range from $349 - $1,299 per year for self-support to premium subscriptions [2, 3], making it a cost-effective solution for scaling. | Windows Server is a strong contender for hosting web applications, especially for organizations already invested in the Microsoft ecosystem. It offers robust features, good integration with .NET applications, and strong enterprise support. IIS (Internet Information Services) is its native web server. Licensing costs for Windows Server can be significant, often based on cores or client access licenses (CALs), which can increase with scale. A 16-core license for Windows Server 2022 Standard can range from $800 to $1,680 (one-time purchase) [4, 5], with Datacenter editions being considerably more expensive (e.g., ~$6,771 [6]). CALs can add further costs (e.g., $220 per user CAL [7]). | Mobile devices are not designed to act as server-side hosts for web applications. Their primary function is as client devices. While it's technically possible to run a local web server on a mobile device for development or very niche use cases, it is entirely unsuitable for hosting a scalable, public-facing web application due to limitations in processing power, memory, network connectivity, battery life, and security. Therefore, there are no viable server-based deployment methods or associated licensing costs for mobile devices in this context. |
| **Client Side** | Developing for Mac clients involves creating responsive HTML interfaces that run within web browsers like Safari, Chrome, or Firefox. Considerations include ensuring cross-browser compatibility and optimizing for various screen sizes and resolutions. The development process would leverage standard web technologies (HTML, CSS, JavaScript) and frameworks. Costs primarily involve developer time and expertise in web development. An average web developer salary in the US is around $83,000 - $99,000 per year [8, 9], or an hourly rate of $45 - $60 [10, 11]. Minimal additional platform-specific costs beyond standard testing environments. | For Linux desktop clients, the approach is identical to Mac and Windows: a responsive HTML interface delivered via a web browser (e.g., Firefox, Chrome). The development considerations are focused on web standards and cross-browser compatibility. Expertise in web development is key, and costs are primarily related to development effort and testing across different Linux distributions and browser versions. The cost for developer time is similar to Mac clients, with an average web developer salary in the US around $83,000 - $99,000 per year [8, 9], or an hourly rate of $45 - $60 [10, 11]. | Windows desktop clients will access the application through standard web browsers (e.g., Edge, Chrome, Firefox). The development effort focuses on creating a modern, responsive HTML interface that adheres to web standards. This ensures compatibility across different Windows versions and browser choices. Development costs are mainly associated with skilled web developers and thorough testing to ensure a consistent user experience. The cost for developer time is similar to Mac and Linux clients, with an average web developer salary in the US around $83,000 - $99,000 per year [8, 9], or an hourly rate of $45 - $60 [10, 11]. | Supporting iOS and Android mobile platforms requires a responsive web design approach, where the HTML interface adapts seamlessly to various screen sizes, orientations, and touch interactions. This involves using mobile-first design principles, responsive frameworks (e.g., Bootstrap, Materialize), and careful optimization for performance on mobile networks and devices. Development considerations include expertise in front-end web development, mobile UI/UX design, and extensive testing on a range of physical devices and emulators. While the core application is web-based, potential needs for native features (e.g., push notifications, camera access) might lead to considering Progressive Web Apps (PWAs) or hybrid app frameworks, which could introduce additional development time and expertise. Average mobile developer salaries in the US range from $110,000 - $130,000 per year [12, 13], or an hourly rate of $48 - $55 [14, 15]. Costs are primarily developer salaries and testing infrastructure. |
| **Development Tools** | For web-based client-side development targeting Mac, standard web development tools are used. This includes IDEs like Visual Studio Code, Sublime Text, or Atom, along with browser developer tools (Safari Developer Tools, Chrome DevTools). Programming languages are HTML, CSS, and JavaScript, often with frameworks like React, Angular, or Vue.js. These tools are generally free or have affordable licenses (e.g., Sublime Text one-time license is $99 [16]). The impact on a development team is minimal, as these are standard web development practices. | Similar to Mac, web development for Linux clients utilizes cross-platform IDEs such as Visual Studio Code, along with command-line tools and browser-specific developer tools. The core languages remain HTML, CSS, and JavaScript. Most development tools for Linux are open-source and free, reducing licensing costs to effectively $0. Development teams can leverage existing web development expertise without needing specialized Linux-only skills. | Web development for Windows clients also relies on cross-platform IDEs like Visual Studio Code, often complemented by Visual Studio (for .NET integration if applicable) and browser developer tools. HTML, CSS, and JavaScript are the primary languages. Many essential tools are free, though some enterprise-grade Microsoft tools (e.g., Visual Studio Enterprise) might have licensing costs ranging from $2,500 - $6,000 per user per year [17]. Development teams can maintain a unified web development workflow. | For mobile web development (iOS and Android), the same web development tools and languages (HTML, CSS, JavaScript, and frameworks) are used. IDEs like Visual Studio Code are common. Additionally, mobile-specific testing tools, emulators (e.g., Android Studio emulator, Xcode simulator), and potentially hybrid framework CLIs (e.g., React Native CLI, Ionic CLI) might be used. Most core web development tools are free. The impact on the development team is that they need expertise in responsive design, mobile performance optimization, and potentially hybrid app development if native features are integrated. This might require specialized front-end developers or training for existing teams. Licensing costs for tools are generally $0 for open-source options, with potential costs for specialized testing services or commercial UI/UX design tools. |

## Recommendations

1. **Operating Platform**:

Recommendation: For the server-side operating platform, Linux is strongly recommended. Given that “Draw It or Lose It” is transitioning to a web-based, multi-platform format, Linux offers the best balance of cost-effectiveness, flexibility, performance, and community support. Its open-source nature eliminates licensing costs, which can be substantial for commercial applications, especially as the game scales. The vast ecosystem of open-source tools, libraries, and frameworks available for Linux makes it an ideal environment for developing and deploying web applications. Furthermore, Linux distributions are known for their stability and security, crucial factors for a gaming application that needs to maintain high availability and protect user data. While the initial learning curve for command-line interfaces might be a consideration for teams unfamiliar with Linux, the long-term benefits in terms of operational cost, scalability, and robust performance far outweigh this initial investment. This choice will allow The Gaming Room to expand “Draw It or Lose It” to other computing environments efficiently and without vendor lock-in.

1. **Operating Systems Architectures**: Recommendation: The chosen operating platform (Linux) will primarily host a client-server architecture, specifically leveraging a microservices-based approach for the game’s backend. The client-server model is fundamental for web-based applications, where game logic and data reside on servers, and user interfaces run on various client devices (web browsers, mobile apps). A microservices architecture further refines this by breaking down the monolithic game application into smaller, independent services. For example, there could be separate services for user authentication, game state management, matchmaking, leaderboards, and image rendering. Each microservice can be developed, deployed, and scaled independently, offering significant advantages:

* Scalability: Individual services can be scaled up or down based on demand, optimizing resource utilization. For instance, if matchmaking sees a surge in activity, only that service needs more resources, not the entire application.
* Resilience: The failure of one microservice does not necessarily bring down the entire application. Other services can continue to function, improving overall system reliability.
* Flexibility: Different services can be developed using different programming languages or technologies best suited for their specific function, although for consistency, maintaining a primary language like Java (as per the existing prototype) across core services is advisable.
* Easier Maintenance and Development: Smaller codebases are easier to understand, maintain, and update. New features can be added to specific services without affecting the entire system.  
  On the client side, the architecture will be a thin client model, where web browsers or lightweight mobile applications handle the presentation layer and communicate with the backend microservices via APIs (e.g., RESTful APIs or WebSockets for real-time game updates). This ensures that the game is accessible across diverse platforms without requiring platform-specific heavy installations.

1. **Storage Management**: **Recommendation:** For storage management, a combination of a distributed relational database system and object storage is recommended. Given the web-based, multi-platform nature of “Draw It or Lose It,” a single, centralized database could become a bottleneck. A distributed database system, such as PostgreSQL configured with sharding or a similar horizontal scaling strategy, would be ideal. PostgreSQL is a powerful, open-source relational database known for its robustness, extensibility, and strong support for complex queries, making it suitable for storing structured game data like user profiles, game states, scores, and team information. Sharding would distribute data across multiple database servers, allowing for greater scalability and fault tolerance.   
     
   For the large library of stock drawings mentioned in the game description, object storage (e.g., Amazon S3, Google Cloud Storage, or a self-hosted MinIO instance on Linux) is highly recommended. Object storage is designed for storing and retrieving large amounts of unstructured data, offering high availability, durability, and cost-effectiveness. Storing images in object storage rather than a relational database will reduce database load, improve retrieval speeds, and simplify content delivery, especially when integrated with a Content Delivery Network (CDN) for global access.
2. **Memory Management**: **Recommendation:** Memory management for the “Draw It or Lose It” software, particularly for the Java application, will primarily rely on the Java Virtual Machine (JVM)’s heap management and be further optimized through container orchestration.  
     
   JVM Heap Management: Java applications manage memory within the JVM’s heap. The JVM automatically handles memory allocation and deallocation through its garbage collector. To optimize performance, the JVM can be configured with specific heap sizes (e.g., -Xms for initial heap size, -Xmx for maximum heap size) to prevent frequent garbage collection pauses or out-of-memory errors. Monitoring tools (e.g., JConsole, VisualVM) should be used to analyze heap usage and identify potential memory leaks or inefficiencies in the Java code. The singleton pattern used for GameService helps in memory efficiency by ensuring only one instance of this critical component exists, thus preventing redundant memory allocations for game state management.

Container Orchestration (e.g., Kubernetes): When deploying microservices on Linux, containerization (using Docker) and orchestration (using Kubernetes) will play a crucial role in memory management. Each microservice will run in its own container, allowing for isolated resource allocation. Kubernetes can be configured to set memory limits and requests for each container. Memory requests ensure that a container is guaranteed a minimum amount of memory, while memory limits prevent a container from consuming excessive memory and impacting other services on the same host. This fine-grained control over memory resources at the container level ensures efficient utilization of the underlying server hardware and prevents resource contention, contributing to the overall stability and performance of the distributed game application

1. **Distributed Systems and Networks**: **Recommendation:** To enable “Draw It or Lose It” to communicate effectively between various platforms in a distributed environment, a robust strategy involving an API Gateway, Message Queues, and WebSockets is essential.  
     
   API Gateway: An API Gateway will serve as the single entry point for all client requests. It can handle request routing, composition, and protocol translation. For instance, mobile clients and web browsers would send requests to the API Gateway, which then directs them to the appropriate backend microservice. This centralizes concerns like authentication, rate limiting, and logging, simplifying client-side development and improving security. It also abstracts the internal microservice architecture from the clients, allowing the backend to evolve independently.  
     
   Message Queues (e.g., Apache Kafka, RabbitMQ): For asynchronous communication between microservices and to handle high-throughput events (like game actions, score updates), message queues are invaluable. They decouple services, allowing them to communicate without direct dependencies. If one service is temporarily unavailable, messages can be queued and processed once it recovers. This improves system resilience and scalability. For example, when a game round ends, a message can be sent to a queue, and a separate service can pick it up to update leaderboards or store game history without blocking the main game logic.  
     
   WebSockets: For real-time interactions critical to a game like “Draw It or Lose It” (e.g., drawing updates, timer synchronization, chat), WebSockets provide a persistent, full-duplex communication channel between the client and server. Unlike traditional HTTP requests, WebSockets allow the server to push updates to clients instantly, minimizing latency and providing a highly interactive user experience. This is crucial for synchronizing game state across multiple players and teams in real-time.

Dependencies and Resilience: The distributed nature introduces dependencies (connectivity, outages). To mitigate these: Service Discovery: Services need to find each other dynamically. Tools like Consul or Eureka can manage this. Circuit Breakers: Implement circuit breaker patterns (e.g., Hystrix) to prevent cascading failures when a service becomes unresponsive. Retries and Timeouts: Implement intelligent retry mechanisms and timeouts for inter-service communication. Load Balancing: Distribute incoming traffic across multiple instances of services to prevent overload. Monitoring and Alerting: Comprehensive monitoring of all services and network components is critical to detect and respond to issues quickly.

1. **Security**: **Recommendation:** Security is paramount for protecting user information and maintaining trust. A multi-layered security approach is recommended for “Draw It or Lose It” across all platforms:

Authentication (Who are you?):

* + OAuth 2.0 / OpenID Connect: Implement a robust authentication mechanism using industry standards like OAuth 2.0 and OpenID Connect. This allows users to securely log in, potentially using social logins (Google, Apple, etc.), and provides secure tokens (e.g., JWTs) for subsequent API calls. The API Gateway can enforce authentication for all incoming requests.
  + Multi-Factor Authentication (MFA): Offer MFA as an option for enhanced user account security.

Authorization (What can you do?):

* + Role-Based Access Control (RBAC): Implement RBAC to define what actions authenticated users can perform. For example, a “player” role can join games, while an “admin” role can manage game settings. This ensures that users only have access to the resources and functionalities they are permitted to use.
  + Least Privilege Principle: Grant users and services only the minimum necessary permissions to perform their tasks.

Data Encryption:

* + Encryption in Transit (TLS/SSL): All communication between clients and the API Gateway, and between microservices, MUST be encrypted using TLS/SSL (HTTPS). This protects data from eavesdropping and tampering during transmission over the network.
  + Encryption at Rest: Sensitive user data (e.g., passwords, personal information) stored in the database or object storage MUST be encrypted. Passwords should always be hashed and salted, never stored in plain text. Database encryption features should be utilized.

Secure Coding Practices:

* + Input Validation: Implement strict input validation on both client and server sides to prevent common vulnerabilities like SQL injection, cross-site scripting (XSS), and command injection.
  + Regular Security Audits: Conduct regular security audits, penetration testing, and vulnerability assessments of the application and infrastructure.

Operating Platform Security (Linux):

* + Firewalls: Configure network firewalls (e.g., ufw, iptables) to restrict access to only necessary ports and services.
  + Regular Updates: Keep the Linux operating system and all installed software (web servers, databases, Java runtime) up-to-date with the latest security patches.
  + Access Control: Implement strong access control mechanisms (e.g., sudo policies, SSH key-based authentication) for server access.
  + Intrusion Detection/Prevention Systems (IDS/IPS): Deploy IDS/IPS to monitor for and respond to malicious activities.

By implementing these security measures, The Gaming Room can build a robust and trustworthy platform for “Draw It or Lose It,” protecting user information and ensuring a secure gaming experience across all supported platforms.