

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 3.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_heading=h.da06aszhp7jb)

[**Table of Contents 2**](#_heading=h.m5pfg6dkwp6r)

[**Document Revision History 2**](#_heading=h.5qnvt9p6wkvb)

[**Executive Summary 3**](#_heading=h.z1x09htcjk16)

[**Design Constraints 3**](#_heading=h.27f2dmix5rth)

[**Domain Model**](#_heading=h.9bdz4ndhuq0x) **4**

[**Evaluation**](#_heading=h.h3fv4m8gba04) **5-7**

[**Recommendations**](#_heading=h.5iadh6ad454p) **8-9**

**References 10**

## [Document Revision History](#_heading=h.96olx9t9a07f)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 07/22/2025 | Nathan Vanderpool | Initial release. Completed sections: executive summary, design constraints, and domain model |
| 2.0 | 08/03/2025 | Nathan Vanderpool | Redone the design constraints and domain model. Evaluation for the development requirements for client-side, server-side, and development tools for multiple platforms (Mac, Linux, Windows, and Mobile Devices) |
| 3.0 | 08/17/2025 | Nathan Vanderpool | Recommendations section completed. |

## [Executive Summary](#_heading=h.2qx5l05kc1z7) (Project One)

The Gaming room has tasked a software engineer to design a web-based version of a game that is done similarly to the game, “Draw it or Lose It,” a 1980s TV game show. The game needs to meet the client’s key requirements which are that there must be multiple teams, able to do multiplayer, no empty teams when matchmaking, unique/dynamic game experience and team names, and also without creating duplicate instances of any game in the program’s memory. This can be done with the use of a singleton pattern to have objects that are created not have multiple instances of itself, and an iterator pattern to prevent any conflicts with things like team names and the team members.

## [Design Constraints](#_heading=h.x5c7hmuwwqoe) (Project One)

Design constraints that may arise during the game application development:

1. **Web-Based Environment**: The application operates in a web-based environment, requiring compatibility with major browsers (e.g., Google Chrome, Firefox) and platforms (Windows, Mac, Linux, and mobile devices) while ensuring smooth performance.
2. **Uniqueness**: The application maintains a database of active players, teams, and ongoing games to prevent duplicate game instances. To ensure uniqueness, it uses permutations and combinations of identifiers, such as a color paired with a group name (e.g., Crimson Crocodiles). Concerns exist regarding the balance of client-side and server-side processing, as this may affect performance based on users' hardware and internet connectivity, even for simple checks.
3. **Scalability**: The game must support web-based communication to accommodate large teams and numerous concurrent games, depending on the player base size. This demands robust database management and server-side optimizations, potentially offloading minor tasks to the client-side (user's hardware) for efficiency.
4. **Security**: Random lobbies of varying player sizes should incorporate security measures from the outset to facilitate adaptation and streamline development, preventing potential performance issues later.

## [Domain Model](#_heading=h.40z58gfgqij6) (Project One)

The process of creating the games, teams, and players begins with a main driver class, as indicated by the application/UML diagram. The actual construction of these entities will be carried out by the GameService class, which will implement a singleton design pattern, ensuring only one instance of the GameService class exists in memory during execution. The GameService class checks whether an instance has been created through the getInstance() method and only creates one if it is not stored in memory. In addition, the constructor is set to private which will help in preventing the creation of any instances. The driver class calls the addGame() method. This method employs the iterator pattern to prevent the duplicate names of Game objects and only then is a unique instance added to the List games. A team will then be added to the game which is the functionality of the addTeam() method. This approach also applies the iterator pattern so that duplication of naming for the objects created for Team is avoided, and then it is further added to the List teams collection as well. For the reason that a team now needs players, the addPlayer() method is called. Once again, this method also applies the iterator pattern to avoid duplication in the naming of the objects created, this time for Player, and subsequently, these are added to the List players collection. Now, using the Entity class with the three subclasses (Game, Team, and Player), we add two attributes (id and name). The entity class is used to ensure that null objects are not created and that only the overloaded constructors are used.

This UML diagram shows the following multiple Object-oriented program (OOP) techniques:

* **Inheritance** is implemented in the Entity class, serving as a base class for multiple subclasses, utilizing overloaded constructors.
* **Abstraction**, also applied in the Entity class, encapsulates attributes (id and name), isolating them from subclasses to prevent sharing of common attributes.
* **Encapsulation** is achieved in GameService through private methods, contrasting with its public methods, enabling self-management without reliance on other objects and controlling access to class states via getter and setter methods.
* **Polymorphism** is demonstrated by each Entity subclass overriding the toString method for its specific needs.

**"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](#_heading=h.c83rjt69iaay) (Project Two)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | **Characteristics:**   * Unix-based OS, primarily for desktops, with limited server capabilities * GUI-focused, polished user experience * macOS Server deprecated in 2022 (Apple, 2022)   **Advantages:**   * Stable, compatible with open-source tools (Tanenbaum & Bos, 2020) * Intuitive GUI for small teams or developers * Popular for local testing of web applications * Built-in protections (Apple, 2022)   **Disadvantages:**   * Lacks dedicated server tools post-macOS Server * Expensive Apple hardware (Silberschatz et al., 2022) * Not designed for large-scale or cloud deployments * Restricted to Apple hardware * Limited server-focused documentation | **Characteristics:**   * Open-source OS * Command-line centric, with optional GUI support. * Highly customizable, widely used in server environments (W3Techs, 2025)   **Advantages:**   * Free and no licensing fees * ideal for large-scale deployments * Lightweight, optimized for servers, with high uptime (Tanenbaum & Bos, 2020) * Strong permissions, community auditing, and fewer malware threats (Pfleeger et al., 2023) * Supports diverse hardware and software, compatible with Node.js, Python, PHP * Extensive documentation and enterprise support * Excels in cloud and containerized environments (Vaughan-Nichols, 2023)   **Disadvantages:**   * Command-line interface can be challenging for beginners * Some proprietary hardware may lack drivers * Less polished GUI tools compared to Windows * Variations across distributions can cause inconsistencies | **Characteristics:**   * Proprietary OS with server editions (e.g., Windows Server 2022) * GUI-focused, integrated with Microsoft’s ecosystem * Supports enterprise software and tools (Microsoft, 2023)   **Advantages:**   * Intuitive GUI reduces management learning curve * Seamless integration * Comprehensive Microsoft support and updates * Broad driver support for enterprise hardware * Robust tools like Remote Desktop and Server Manager. * Strong for enterprise network management (Microsoft, 2023).   **Disadvantages:**   * High licensing fees for Windows Server (Silberschatz et al., 2022) * Heavier footprint reduces performance on low-end hardware * Larger attack surface, requires active patching (Pfleeger et al., 2023) * Less customizable than Linux * Smaller open-source community, reliance on paid support | **Characteristics:**   * OS designed for smartphones/tablets (iOS by Apple, Android by Google) * Optimized for low-power, touch-based devices * Not built for server-side hosting   **Advantages:**   * Compact, suitable for edge computing * Energy-efficient for small-scale deployments * Lightweight server apps * Built-in cellular/Wi-Fi support   **Disadvantages:**   * Limited CPU, RAM, and storage (Silberschatz et al., 2022) * Unsuitable for high-traffic applications * Minimal server software support * Prone to interruptions (battery, thermal throttling) * Sandboxing restricts server functionality (Pfleeger et al., 2023) * Lacks robust server management tools |
| **Client Side** | **Cost**   * **Licensing and Hardware**: Apple Developer Program membership costs $99/year for Mac App Store distribution. Development requires Mac hardware, increasing initial investment (Apple, 2025). * **Tooling**: Xcode is free, but third-party tools (e.g., testing frameworks, CI/CD systems) may incur additional costs. * **Maintenance**: Frequent macOS updates necessitate ongoing testing and compatibility updates.   **Time**   * **Development**: Xcode streamlines development, but App Store review processes (1-2 weeks) extend timelines (Apple, 2025). * **Testing**: Testing across macOS versions and hardware (e.g., Intel vs. Apple Silicon) adds time.   **Expertise**   * **Skills**: Developers need Swift/Objective-C expertise and knowledge of macOS APIs (e.g., AppKit). Familiarity with Apple’s Human Interface Guidelines is essential (Apple, 2025). * **Learning Curve**: Moderate, due to Xcode’s intuitive design, but mastering Apple’s ecosystem requires experience. | **Cost**   * **Open-Source Advantage**: Open-source tools (e.g., Visual Studio Code) minimize licensing costs. * **Distribution Challenges**: Supporting multiple Linux distributions (e.g., Ubuntu, Fedora) increases testing and packaging costs due to varying dependencies (Corbet et al., 2023). * **Hardware**: Cost-effective hardware can be used, but server infrastructure for testing may add expenses.   **Time**   * **Development**: Open-source tools enable rapid prototyping, but configuring environments for multiple distributions is time-intensive (Corbet et al., 2023). * **Testing**: Diversity of Linux distributions significantly increases testing time.   **Expertise**   * **Skills**: Proficiency in C/C++, Python, or Java and familiarity with Linux tools (e.g., Bash, Docker) are required. Knowledge of package managers is critical (Corbet et al., 2023). * **Learning Curve**: High, due to Linux’s diversity and lack of a unified framework. | **Cost**   * **Licensing**: Microsoft Visual Studio Community edition is free, but Professional/ Enterprise versions for larger teams are paid (Microsoft, 2025). * **Tooling**: Enterprise tools (e.g., SQL Server) increase costs. * **Compatibility**: Supporting multiple Windows versions requires additional testing resources.   **Time**   * **Development**: Familiarity with .NET or Windows frameworks speeds coding, but legacy system integration may cause delays (Microsoft, 2025). * **Testing**: Supporting multiple Windows versions requires moderate testing time due to standardization.   **Expertise**   * **Skills**: Expertise in C#, .NET, or C++ and Windows APIs (e.g., Win32, UWP) is necessary. Visual Studio knowledge is beneficial (Microsoft, 2025). * **Learning Curve**: Moderate, with well-documented tools, but legacy system integration can be complex. | **Cost**   * **iOS**: Apple Developer Program ($99/year) and Mac hardware are required. App Store fees for in-app purchases may apply (Apple, 2025). * **Android**: Google Play Store requires a one-time $25 fee. Device fragmentation increases testing costs (Google, 2025). * **Cross-Platform Tools**: Frameworks like Flutter or React Native reduce costs via code reuse, but premium features or libraries may add expenses.   **Time**   * **iOS**: Swift and Xcode enable fast development, but App Store reviews add delays (Apple, 2025). * **Android**: Android Studio supports quick development, but testing across diverse devices and Android versions extends timelines (Google, 2025). * **Cross-Platform**: Frameworks like Flutter reduce initial development time but may require platform-specific optimizations, adding 20-30% to timelines (Fowler, 2024).   **Expertise**   * **iOS**: Requires Swift/Objective-C expertise and iOS SDK knowledge (e.g., UIKit, SwiftUI). Apple’s design guidelines are critical (Apple, 2025). * **Android**: Requires Java/Kotlin proficiency and Android SDK knowledge. Handling device fragmentation is essential (Google, 2025). * **Cross-Platform**: Expertise in Flutter (Dart) or React Native (JavaScript) is needed, with platform-specific optimization skills (Fowler, 2024). * **Learning Curve**: High, due to dual-platform expertise and frequent SDK updates. |
| **Development Tools** | **Programming Languages**   * **Swift**: Best for native macOS apps (e.g., utilities, productivity tools). Uses SwiftUI or AppKit. * **Objective-C**: For older macOS apps or legacy code. Uses AppKit. * **Python**: For cross-platform or scripting tools. Uses PyQt or Tkinter. * **C/C++**: For high-performance apps (e.g., games). Uses Qt or SDL. * **JavaScript/TypeScript**: For cross-platform apps (e.g., VS Code). Uses Electron or Tauri. * **Java**: For enterprise apps. Uses JavaFX or Swing. * **Rust**: For secure, high-performance apps. Uses Tauri or Cocoa bindings.   **Tools (IDEs and Others)**   * **Xcode**: Main IDE for Swift/Objective-C, with UI design, debugging, and App Store submission. * **VS Code**: For Python, JavaScript, Rust, or C++. Lightweight, with extensions. * **JetBrains IDEs**: PyCharm (Python), IntelliJ IDEA (Java), CLion (C++). * **Qt Creator**: For C++ cross-platform apps using Qt. * **Homebrew**: Installs libraries/dependencies. * **CMake**: Manages C++ builds. * **Git**: Version control. * **PyInstaller/Platypus**: Packages Python apps as .app. * **create-dmg**: Creates macOS disk images for distribution. * **Sparkle**: Adds auto-updates. | **Programming Languages**   * **C/C++**: For system programming, kernel modules, high-performance apps. * **Python**: For scripting, web apps, data science, automation. * **Java**: For enterprise apps, backend services. * **Go**: For microservices, cloud-native apps. * **Rust**: For secure system tools, performance-critical apps. * **JavaScript (Node.js)**: For web servers, APIs. * **Bash**: For automation, deployment scripts.   **Tools (IDEs and Others)**   * **Visual Studio Code**: Lightweight, supports many languages, extensible. * **Eclipse**: Strong for Java, C/C++, project management. * **IntelliJ IDEA**: Best for Java, Kotlin, smart code tools. * **PyCharm**: Specialized for Python, web, and data science. * **Geany**: Lightweight, fast for smaller projects. * **Vim/Emacs**: Customizable editors for server-side coding. * **Build**: GCC, Make/CMake for compiling code. * **Version Control**: Git for code tracking. * **Debugging**: GDB for runtime debugging. * **Containerization**: Docker for consistent deployments. * **CI/CD & Deployment**: Jenkins, AWS CodeDeploy, Ansible for automation and deployment. | **Programming Languages**   * **C#**: Best for .NET apps (WPF, WinForms). * **C++**: For high-performance, native apps using Win32. * **Python**: For scripting or GUI apps, packaged with PyInstaller. * **JavaScript/TypeScript**: For cross-platform apps via Electron. * **Java**: For desktop apps with Swing/JavaFX. * **Rust**: For secure, low-level apps.   **Tools (IDEs and Others)**   * **Visual Studio**: Best for C#, C++, .NET development. * **Visual Studio Code**: Lightweight, supports Python, JavaScript, etc. * **JetBrains Rider**: For .NET and C#. * **PyCharm**: For Python apps. * **MSBuild**: Compiles .NET projects. * **Windows Installer (MSI)**: Creates installers. * **PyInstaller**: Packages Python apps. * **Git**: Version control. * **Windows SDK**: For API access. * **Azure DevOps/GitHub Actions**: For CI/CD. | **Programming Languages**  ***Android (Native):***   * **Kotlin**: Language for Android. Concise syntax, null safety, and modern features like coroutines for asynchronous programming. Ideal for new apps. * **Java**: For maintaining legacy Android apps or libraries. Robust but less concise.   ***iOS (Native):***   * **Swift**: Modern language for iOS, iPadOS, and macOS apps. Fast, safe (with strong type-checking), and integrates seamlessly with Apple’s frameworks like UIKit and SwiftUI. * **Objective-C**: Older language for iOS, now largely replaced by Swift but used in legacy codebases or specific low-level tasks.   ***Cross-Platform:***   * **JavaScript (React Native)**: For developers with familiarity of web development. Used to build apps with near-native performance, sharing up to 90% of code between iOS and Android. * **Dart (Flutter)**: UI-focused framework, known for its fast performance and widget-based UI design. Flutter’s single codebase compiles to native code for both platforms. * **C# (.NET MAUI/Xamarin)**: For enterprise apps, leveraging C# for code sharing. .NET MAUI is useful for strong Windows integration.   **Tools (IDEs and Others)**   * **Android Studio**: For Android (Kotlin/Java) and Flutter (Dart) * **Xcode**: For iOS (Swift/Objective-C) * **Visual Studio Code**: For cross-platform (JavaScript, Dart) * **Visual Studio**: For .NET MAUI/Xamarin (C#) * **IntelliJ IDEA**: For Android and multiplatform projects * **Build Tools**: Gradle (Android), CocoaPods/Swift Package Manager (iOS), npm (JavaScript) * **Testing/Emulators**: Android Emulator, iOS Simulator, Appium, Espresso, XCUITest * **Backend Services**: Firebase, AWS Amplify, Azure * **CI/CD**: GitHub Actions, Jenkins, Buddy * **Design**: Figma, Adobe XD * **Version Control**: Git (GitHub/GitLab) |

## Recommendations (Project Three)

1. **Operating Platform**:

To facilitate the expansion of Draw It or Lose It—a multiplayer drawing game requiring dependable server hosting, cross-platform client support, and scalability across various computing environments—I recommend Linux as the primary server-side operating platform. Its open-source nature, compatibility with cloud and containerized systems, and ability to support diverse hardware make it well-suited for scaling the game to multiple client platforms, including desktops, mobile devices, and cloud-based instances. For client-side development, cross-platform frameworks ensure compatibility with macOS, Windows, Android, and iOS, enabling growth without platform-specific restrictions.

1. **Operating Systems Architectures**:

Linux is an open-source, Unix-like operating system with a command-line-driven architecture, optional graphical interfaces, and extensive customizability, making it a top choice for server environments powering over 80% of web servers (W3Techs, 2025). Its kernel manages critical functions such as process scheduling, memory management, and I/O operations, while user-space components provide flexibility. Architecturally, Linux employs a monolithic kernel with modular extensions (e.g., loadable drivers), enabling efficient, lightweight deployments. Compared to macOS (GUI-focused, hardware-limited), Windows (proprietary, resource-heavy), or mobile operating systems (low-power, sandboxed), Linux excels in distributed environments, supporting scalable deployments on cloud platforms like AWS or Azure (Vaughan-Nichols, 2023; Tanenbaum & Bos, 2020).

1. **Storage Management**:

For Linux-based servers, the ext4 filesystem paired with Logical Volume Manager (LVM) is recommended for storage management. ext4, a journaling filesystem, ensures data integrity during system crashes, supports large-scale storage (up to 1 exabyte), and uses extents for efficient handling of large files, making it suitable for storing game assets, user profiles, and session logs for Draw It or Lose It. LVM allows dynamic resizing of storage volumes without downtime and supports RAID for redundancy in multiplayer scenarios. This cost-effective solution outperforms macOS’s APFS (hardware-specific), Windows’ NTFS (higher resource demands), or mobile storage systems (capacity-limited) and integrates with cloud storage solutions like AWS S3 for scalability (Silberschatz et al., 2022).

1. **Memory Management**:

Linux’s memory management utilizes a virtual memory system with paging and swapping to optimize resource allocation for Draw It or Lose It. It organizes memory into pages (typically 4KB), mapping virtual to physical addresses via page tables to efficiently handle game processes like real-time drawing synchronization. Memory overcommitment allows allocation beyond physical RAM, with the Out-Of-Memory (OOM) killer prioritizing critical processes (e.g., game servers) during shortages. Swapping moves inactive pages to disk (e.g., swap partitions), while c-groups and namespaces isolate memory for containerized instances, ensuring one game session does not affect others. This approach provides high performance and uptime across varied hardware, surpassing macOS’s hardware-dependent unified memory, Windows’ heavier resource usage, or mobile OS’s thermally constrained limits (Tanenbaum & Bos, 2020; Silberschatz et al., 2022).

1. **Distributed Systems and Networks**:

To support cross-platform communication for Draw It or Lose It (e.g., between Android, iOS, and desktop clients), a microservices architecture on Linux servers is recommended, using WebSockets for real-time drawing updates and REST APIs for session management. Containerized deployments ensure fault tolerance for game logic, such as turn-based drawing. Dependencies include reliable network connectivity (managed by Linux’s TCP/IP stack with failover mechanisms), outage handling (mitigated by cloud redundancy in platforms like AWS EC2 with auto-scaling), and inter-component connections. On the client side, cross-platform networking capabilities handle device-to-server communication with retry mechanisms for unstable mobile networks. This setup minimizes single points of failure, offering greater flexibility than macOS’s limited scalability or Windows’ proprietary ecosystem (Vaughan-Nichols, 2023; Pfleeger et al., 2023).

1. **Security**:

To protect user data (e.g., profiles, drawings, and scores) across platforms, Linux’s robust security features should be utilized, including strict permissions (e.g., SELinux for mandatory access control), community-reviewed code, and reduced malware risks due to non-default root access. Data should be encrypted in transit with TLS/HTTPS and at rest with LUKS disk encryption to comply with standards like GDPR. For cross-platform authentication, OAuth/JWT should be implemented between clients and Linux servers, with firewalls to block unauthorized access. Linux supports regular kernel updates, AppArmor for process isolation, and audit logging for breach detection, outperforming Windows’ larger attack surface or mobile OS’s app-constrained sandboxing. Client-side, platform-specific security (e.g., Android’s scoped storage, iOS’s Keychain) can be integrated, with Linux servers acting as a secure hub to reduce vulnerabilities (Pfleeger et al., 2023; Tanenbaum & Bos, 2020).

**References**

Apple. (2025). Apple Developer Program. <https://developer.apple.com/programs/>

Apple. (2022). macOS Server discontinuation. Retrieved from <https://support.apple.com/en-us/HT208312>

Corbet, J., Kroah-Hartman, G., & McPherson, A. (2023). Linux kernel development: A study in open-source collaboration. Linux Foundation.

Fowler, M. (2024). Patterns of cross-platform development. Addison-Wesley.

Google. (2025). Android Developer Program. <https://developer.android.com/distribute>

Microsoft. (2025). Visual Studio Licensing. <https://visualstudio.microsoft.com/licensing/>

Microsoft. (2023). Windows Server 2022 overview. Retrieved from <https://www.microsoft.com/en-us/windows-server>

Pfleeger, C. P., Pfleeger, S. L., & Margulies, J. (2023). Security in computing (6th ed.). Pearson.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2022). Operating system concepts (10th ed.). Wiley.

Tanenbaum, A. S., & Bos, H. (2020). Modern operating systems (5th ed.). Pearson.

Vaughan-Nichols, S. J. (2023). Linux in 2023: The unstoppable force in server and cloud computing. IEEE Computer Society.

W3Techs. (2025). Usage statistics of operating systems for websites. Retrieved from <https://w3techs.com/technologies/overview/operating_system>