University of Leicester
School of Computing and Mathematical Sciences

Bomberman: Enhanced Edition

A Full-Stack Modernization of a Classic Arcade Game

Project Report

Author: Aaren Sherwin Stanly Rajesh

Project Supervisor: Prof. Anthony Conway

Principal Marker: Prof. Victoria Wright

September 05, 2025

Word Count: 19551

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or adapted from any other source, have been specifically acknowledged as such. I am aware of the penalties for plagiarism and for the unacknowledged use of material from other sources.

I declare that this report is my own work and has not been submitted in substantially the same form for the award of a higher degree elsewhere, and that the work has been conducted honestly and honorably as expected by the Code of Conduct for programs of study at the University. I am aware of the penalties for any breach of the Code of Conduct and the possible consequences, such as the loss of all marks for the source in this module and the degree examination as a whole.

Name: Aaren Sherwin Stanly Rajesh
Date: 05-09-2025

Abstract

This project aimed to revitalize the classic arcade game, *Bomberman*, for a modern audience. The primary objective was to faithfully re-implement its beloved core mechanics while integrating a suite of contemporary features to enhance player engagement and longevity. The project sought to address challenges in software design, graphics handling, and user experience by leveraging modern web technologies.

The game, titled *'Bomberman: Enhanced Edition,'* was developed using HTML5, JavaScript, and the Phaser 3 game engine for the client-side experience. To support modern features, a robust back end was created using Node.js, Express, and a MongoDB database. Key features successfully implemented as part of the "modern twist" include a secure user authentication system with email-based OTP verification, a comprehensive social hub with unique friend codes for adding friends, and an online leaderboard to foster competition in a new 'Deathmatch' mode. Further enhancements include a persistent save/load system for the single-player campaign, a level selection screen for unlocked stages, gamepad support with haptic feedback, and a toggle allowing players to choose between the enhanced remake and an embedded version of the original game.

The result is a fully functional, browser-based retro clone that successfully merges the nostalgic gameplay of the original with the features expected of a modern title. The final product delivers a polished user experience from account creation to gameplay, demonstrating a comprehensive application of full-stack development, game logic implementation, and persistent user data management.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Overview and Context

In an era of hyper-realistic graphics and complex game mechanics, there has been a significant resurgence of interest in retro gaming. Classic titles from the 8-bit and 16-bit eras possess a timeless charm, often attributed to their focus on pure, accessible, and challenging gameplay. This nostalgia has fuelled a popular trend of remaking and remastering these beloved games, aiming to re-introduce them to new generations while preserving the essence that made them great. This process involves not just a visual overhaul but also the integration of modern features that contemporary audiences have come to expect, such as online connectivity and persistent player profiles.

Among the pantheon of classic titles, *Bomberman* stands out as an icon of arcade-style action and strategy. First released in the 1980s, its simple premise—a character navigating a grid-based maze, strategically placing bombs to clear obstacles and defeat enemies—is deceptive in its depth. The game is renowned for its addictive single-player mode and, most notably, its chaotic and thrilling multiplayer battles, which established it as a foundational title in the party game genre.

However, while its core gameplay remains compelling, the original *Bomberman* experience is a product of its time, lacking the features that drive modern player engagement. The absence of online functionality, social systems, and persistent progression presents a barrier for new players and a missed opportunity for veteran fans. This project, titled *'Bomberman: Enhanced Edition,'* addresses this gap by undertaking the challenge of modernizing this classic title for the web.

The fundamental goal is to develop a browser-based clone that not only preserves the spirit of the original but also enriches it with a suite of modern features. The project scope includes a complete re-implementation of the game using the Phaser 3 engine, supported by a robust back end built with Node.js, Express, and MongoDB. Key enhancements include a secure user authentication system with email verification, a social hub with a friend list system, an online

leaderboard for a new competitive 'Deathmatch' mode, and a save/load feature for the single-player campaign. Uniquely, the project also offers players a choice between the new 'Enhanced Edition' and an embedded 'Classic Edition', directly celebrating the game's heritage. This report will detail the design, development process, technical architecture, and final outcomes of this endeavor.

1.2 Aims and Objectives

The research and development detailed in this report were guided by a set of primary aims, which were subsequently broken down into specific, measurable objectives.

1.2.1 Aims

The overarching aims of this project are as follows:

I. **Modernisation of a Classic Arcade Title:** The principal aim was to re-engineer the classic video game *Bomberman*, synthesising its seminal gameplay with the features and technical standards expected of contemporary interactive entertainment.

II. **Demonstration of Full-Stack Application Architecture:** A secondary aim was to design, develop, and deploy a complete full-stack web application. This served as a practical demonstration of technical proficiency in front-end game development, back-end API construction, and database integration and management.

III. **Delivery of a Polished and Cohesive User Experience:** A final aim was to produce a finished product with a high degree of polish. The project sought to provide a cohesive and intuitive user experience, from initial user onboarding to long-term engagement, that would be compelling for both nostalgic fans and new audiences.

1.2.2 Objectives

To achieve the aims articulated above, the following specific objectives were established and accomplished:

- To conduct a comprehensive analysis of the original *Bomberman*'s core systems—including game mechanics, level design paradigms, power-up functionalities, and enemy AI—to ensure a faithful and authentic replication.
- To engineer a client-side application utilising HTML5, CSS3, and JavaScript, leveraging the Phaser 3 framework as the primary game engine for rendering, physics simulation, and state management.
- To construct a secure and scalable back-end server architecture, implementing a RESTful API with Node.js and the Express framework to manage all data transactions between the client and the database.

- To integrate a persistent data storage solution using MongoDB, responsible for managing user profiles and supporting a robust authentication system featuring account registration, login, and email-based One-Time Password (OTP) verification.
- To design and implement a suite of modern gaming features, including: a social system based on unique user codes, a competitive online leaderboard for a 'Deathmatch' mode, a game-state persistence mechanism (save/load functionality), and a progression-based level selection system.
- To enhance user interaction and accessibility through the integration of alternative input methods, specifically gamepad controller support with associated haptic feedback.
- To incorporate an embedded instance of the original title, providing a 'Classic Mode' that serves to both preserve the game's heritage and offer a direct experiential comparison to the enhanced implementation.

1.3 Summary of Contributions and Key Results

This project culminated in several key technical and user-facing results that successfully met the established aims and objectives. The primary contributions and key results presented in this report are summarised as follows:

- **A Complete Full-Stack Game Application:** The principal result is the successful development and deployment of 'Bomberman: Enhanced Edition,' a fully functional, browser-based game. This application demonstrates a cohesive architecture, integrating a Phaser 3 front-end game client with a Node.js RESTful API and a MongoDB database, representing a complete end-to-end software solution.
- **Integration of Modern Social and Competitive Features:** A significant contribution is the integration of contemporary online features into a classic arcade framework. This was achieved through the implementation of a secure user authentication system, a social hub enabling players to connect via a friend-code system, and an online leaderboard for a competitive 'Deathmatch' mode, thereby enhancing player retention and engagement.
- **A Novel Dual-Mode Gameplay System:** The project produced a system that allows players to switch between the newly developed 'Enhanced Edition' and an embedded 'Classic Edition' of the original game. This feature serves as a direct homage to the source material while simultaneously highlighting the technical and functional advancements of the modern implementation.
- **Implementation of Game State Persistence and Progression:** A robust game-state persistence mechanism was developed, providing players with the ability to save and load their progress in the single-player campaign. This was complemented by a level

selection system that unlocks based on player advancement, transforming the experience from a transient arcade session into a game with meaningful progression.

- **Enhancement of User Experience through Modern Inputs:** The project enhanced the standard user experience by integrating alternative input modalities. The inclusion of gamepad controller support and corresponding haptic feedback for key events provides a higher-fidelity, more immersive gameplay experience that aligns with modern console and PC gaming standards.

Chapter 2

Background and Related Work

This chapter provides a review of the historical context, foundational technologies, and existing works relevant to this project. It begins by examining the design principles of classic arcade games, which form the mechanical and philosophical basis for *Bomberman*.

2.1 The Legacy and Mechanics of Arcade-Style Games

The video game industry's formative years were dominated by the arcade, with its "Golden Age" spanning from the late 1970s to the mid-1980s. This era established a set of design paradigms that continue to influence game development today. The commercial environment of the arcade—requiring a coin for each play—placed unique constraints on game design, forcing developers to create experiences that were immediately engaging, understandable, and highly replayable. Titles such as *Space Invaders* (1978), *Pac-Man* (1980), and *Donkey Kong* (1981) are exemplars of a design philosophy centred on a core set of mechanics.

The foundational mechanics of arcade-style games can be distilled into several key principles:

- **Simplicity and Accessibility:** The most critical principle was the necessity for immediate comprehension. With a potential player walking by, the game's objective and controls had to be understood within seconds. This led to the popularisation of the "easy to learn, hard to master" design mantra, where simple inputs (e.g., a joystick and one or two buttons) governed a ruleset with significant strategic depth.
- **The High-Score as a Motivator:** In the absence of narrative or persistent progression, the primary driver for replayability was the pursuit of a high score. The leaderboard, or high-score table, became the central meta-game, fostering direct and indirect competition among players and providing a clear metric for mastery. This created a powerful intrinsic motivation for players to invest more time and money.
- **Constrained and Readable Game Spaces:** Early technical limitations often necessitated that gameplay took place on a single, static screen. This constraint became a

design feature, always providing the player with complete information about the game state. This perfect information allows for purely strategic decision-making, a hallmark of the genre.

- **Rapid and Repetitive Gameplay Loops:** The core loop of an arcade game is typically short, satisfying, and cyclical (e.g., clear a wave of enemies, complete a maze). This rapid loop encourages the "one more try" mentality, as failure is met with the immediate opportunity to attempt the same challenge again, armed with new knowledge.
- **Punishing Difficulty and Resource Scarcity:** Arcade games were designed to be challenging and, ultimately, to end a player's session. This was achieved through a progressively steepening difficulty curve—enemies becoming faster, more numerous, or exhibiting more complex behaviours. This was coupled with a finite number of lives, making each decision consequential and heightening the tension of survival.

These principles transitioned from the arcade to early home consoles like the Nintendo Entertainment System (NES), where *Bomberman* (1985) found widespread success. *Bomberman* is a quintessential example of the arcade-style design paradigm. It features simple two-axis movement and a single action button, a single-screen playfield, and a clear objective of eliminating all enemies to reveal an exit. Its gameplay loop is rapid, and its difficulty scales through the introduction of more resilient and intelligent enemy types.

However, *Bomberman* also innovated within this framework, popularising indirect, strategic combat through the placement of bombs as traps. Its most significant legacy is its local multiplayer mode, which transformed the competitive, high-score-chasing nature of the arcade into a chaotic and highly social party game experience. The design principles established in the arcade era thus form the foundational blueprint upon which this project, 'Bomberman: Enhanced Edition,' is built, respecting these core tenets while augmenting them with modern technological capabilities.

2.2 A Bibliographic Review of Web-Based Game Development Technologies

The landscape of web-based game development has undergone a significant transformation over the past two decades. The decline of proprietary plugin-based technologies, most notably Adobe Flash, gave way to a new paradigm founded on open web standards. The maturation of HTML5, the universal adoption of powerful JavaScript engines, and the standardisation of low-level graphics APIs like WebGL (Web Graphics Library) have collectively established the modern web browser as a viable and powerful platform for delivering rich, interactive game experiences.

This technological shift created a demand for frameworks and engines that could abstract the complexities of the browser environment. A game engine provides a reusable software framework

that handles core functionalities such as the render loop, scene management, physics simulation, input handling, and asset management. This allows developers to focus on the unique logic and design of their game rather than re-implementing foundational systems (Author, Year). The following sections review the framework chosen for this project, Phaser 3, and situate it within the context of alternative technologies.

2.2.1 The Role of the Phaser 3 Framework

Phaser 3 is a fast, free, and open-source software framework designed specifically for creating 2D games for HTML5-compliant web browsers. For this project, Phaser 3 was selected as the core technology for the client-side application due to its direct alignment with the project's technical requirements and design goals.

At its core, Phaser provides a high-level abstraction over the browser's native rendering technologies. It possesses the capability to render graphics using either the 2D Canvas API or the more performant WebGL API, automatically detecting and selecting the optimal renderer available on the client's device. This ensures both wide compatibility and high performance.

The architecture of Phaser 3 is built around a scene-based system, where each distinct part of the game (e.g., the main menu, the game level, the pause screen) is encapsulated within its own scene. This modular structure promotes organised and maintainable code. The framework includes several key subsystems that were critical to this project's development:

- **Physics Engines:** Phaser 3 includes two built-in physics engines. The Arcade Physics engine, used in this project, is a lightweight, high-performance system ideal for grid-based or non-realistic simulations typical of retro-style games.
- **Animation and Asset Management:** It provides a robust system for creating frame-by-frame animations from spritesheets and a streamlined loader for managing all game assets, such as images, audio, and tilemaps.
- **Input Handling:** The framework unifies input from various sources, including keyboard, mouse, touch, and gamepad controllers, providing a simple API for developers to query their state.

The primary advantage of Phaser 3 for a project like 'Bomberman: Enhanced Edition' is its specialisation. Unlike larger, general-purpose engines, its feature set is tailored for 2D game development. This results in a more lightweight footprint, leading to faster initial load times—a critical factor for web-based content—and a development workflow that is not encumbered by features irrelevant to a 2D, retro-style game.

2.2.2 Alternative Web Game Engines

While Phaser 3 was deemed the optimal choice, a review of alternative technologies is necessary to validate this decision. The most prominent alternatives for web game deployment are Unity

and the Godot Engine.

Unity: Unity is a powerful, cross-platform game engine and the industry standard for many commercial productions. It supports web deployment through its WebGL build target, which compiles C# code into WebAssembly (WASM). While extremely capable, particularly in 3D, its use for this project was considered suboptimal. Unity's WebGL builds are significantly larger in file size and require longer initialisation times due to the loading of the engine's runtime into the browser. Its component-based, 3D-centric workflow is more complex than necessary for developing a 2D pixel-art game (Author, Year).

Godot Engine: Godot is a free, open-source, and fully integrated game engine that has gained significant popularity for both 2D and 3D development. Similar to Unity, it exports to the web via a WebGL template. Its dedicated 2D workflow and node-based scene system are highly regarded. However, like Unity, it is a more heavyweight solution than a native JavaScript framework. The choice between Godot and Phaser often comes down to a preference between a fully integrated visual editor (Godot) and a code-centric, lightweight framework that integrates seamlessly with the existing web development ecosystem (Phaser). For this project, the latter was preferred.

A final point of comparison is PixiJS, a high-performance 2D rendering library. Notably, Phaser 3 utilises a modified version of PixiJS as its renderer. While PixiJS offers superior control and performance for rendering tasks, it is not a complete game engine. It lacks integrated systems for physics, scene management, and input, which would have needed to be built from scratch, significantly increasing the project's scope and complexity.

In summary, the selection of Phaser 3 was a strategic decision based on its position as a native web technology, its lightweight architecture, and its specialised focus on 2D game development, which collectively provided the most direct and efficient path to achieving the project's objectives.

2.3 Analysis of Full-Stack Web Architectures for Interactive Applications

Modern interactive web applications, such as online games, require a robust architecture that separates user-facing concerns from underlying data management and business logic. This separation is typically achieved through a full-stack development model, which encompasses both the front-end (client-side) and back-end (server-side) components of an application. For this project, a classical and highly effective architectural pattern, the three-tier architecture, was adopted. This model logically divides the application into distinct layers, each with a specific responsibility.

The three tiers are as follows:

1. **The Presentation Tier (Front-End):** This is the layer that the end-user directly interacts with. In the context of this project, it is the web browser running the game client built with HTML5, CSS, and JavaScript, with the Phaser 3 framework orchestrating the user experience. The primary responsibilities of this tier include rendering the game world and user interface, capturing and processing all user inputs (keyboard, gamepad), and initiating requests for data or actions from the application tier via its API.
2. **The Application Tier (Back-End):** This tier, often referred to as the logic tier, serves as the central processing unit of the application. It is responsible for executing the core business logic that cannot or should not be handled on the client. For this project, the application tier was implemented using a Node.js server with the Express framework. Its duties include handling incoming client requests, authenticating users, validating data, managing game sessions, and mediating all communication with the data tier. It effectively decouples the client from the database, ensuring that the front-end does not need to be aware of the underlying data storage mechanisms.
3. **The Data Tier (Database):** This tier is responsible for the persistent storage and retrieval of all application data. It consists of the database management system, which for this project is MongoDB. Its sole function is to manage the data collections for user accounts, saved game states, friend lists, and leaderboard scores. The application tier queries the data tier to retrieve information for the client or to write new information for storage.

Communication between the presentation and application tiers is a critical aspect of this architecture. This project employs a Representational State Transfer (REST) architectural style to create a stateless and scalable Application Programming Interface (API). A RESTful API relies on a set of conventions built upon the standard Hypertext Transfer Protocol (HTTP). Key principles include:

- **Client-Server Decoupling:** The client and server are separate entities that communicate over the network. This allows for independent development, deployment, and scaling of each component. If the API contract (the defined endpoints and data structures) is maintained, changes to the back-end will not break the front-end, and vice-versa.
- **Statelessness:** Each request sent from the client to the server must contain all the information necessary for the server to fulfil that request. The server does not maintain any session state for the client between requests. This design simplifies the server logic and greatly improves scalability (Author, Year).
- **Standardised Methods:** The API utilises standard HTTP methods for operations, such as GET (retrieve data), POST (create new data), PUT (update existing data), and DELETE (remove data), providing a predictable and language-agnostic interface.

The adoption of this three-tier, REST-based architecture was a deliberate choice for 'Bomberman: Enhanced Edition'. It provides the necessary separation of concerns to manage the

complexity of a real-time game client and a data-driven back end, resulting in a system that is modular, maintainable, and scalable.

2.4 A Comparative Study of Social and Competitive Features in Modern Online Games

The paradigm of modern online gaming has shifted significantly from the isolated, session-based experiences of the arcade era. The long-term success and commercial viability of contemporary games are often contingent upon the implementation of robust social and competitive systems. These features are engineered not merely as supplements to the core gameplay, but as integral components of a service-based model designed to foster community, drive player engagement, and extend the life cycle of the product (Author, Year). This section will analyse common implementations of these systems to provide a theoretical and practical context for the features developed in 'Bomberman: Enhanced Edition'.

2.4.1 Social Systems in Online Gaming

Social systems are designed to connect players, facilitate communication, and create a sense of community, which is a primary driver of long-term player retention.

- **Friends Lists and Presence:** The most fundamental social feature in any online game is the friends list. This system allows players to form persistent, asynchronous connections with others. A critical component of this is presence, which is the ability to see the online status of friends (e.g., online, offline, in-game). This functionality reduces the friction of initiating interaction and is a foundational element for building a social graph within a game ecosystem (Author, Year). This feature is ubiquitous, forming the basis of platforms like Steam and nearly every major online multiplayer title.
- **Structured Social Groups (Guilds and Clans):** Beyond simple one-to-one connections, many games implement systems for larger, more structured social groups, commonly known as guilds or clans. These groups provide a shared identity, dedicated communication channels, and a framework for pursuing common objectives, such as cooperative raids in Massively Multiplayer Online Role-Playing Games (MMORPGs) like *World of Warcraft* or team-based competition in first-person shooters.

2.4.2 Competitive Systems and Progression

Competitive systems provide a framework for players to measure their skills, track their improvement, and engage in meaningful conflict. These systems are often paired with progression mechanics to create powerful motivational loops.

- **Leaderboards:** A direct evolution of the arcade high-score table, the leaderboard remains a staple of competitive design. Modern implementations, however, have become

more sophisticated. In addition to global rankings, which can be demotivating for average players, successful games often implement more contextual leaderboards, such as regional rankings or, most effectively, friend-based leaderboards. Scoping competition to a player's immediate social circle makes the act of ranking higher more personal, achievable, and engaging (Author, Year).

- **Matchmaking and Ranking Systems:** The cornerstone of modern competitive gaming is the skill-based matchmaking system, often underpinned by an Elo or similar rating algorithm (e.g., MMR - Matchmaking Rating). These systems aim to create fair matches by pairing players of similar ability. This is typically accompanied by a visible ranking system (e.g., Bronze, Silver, Gold tiers) which provides players with a clear indicator of their standing and a tangible goal for improvement. Games like *League of Legends* and *Counter-Strike* have demonstrated the profound effectiveness of this model in sustaining a competitive player base.
- **Progression and Reward Loops:** To sustain engagement, competitive play is almost always linked to a broader progression system. Players earn experience points (XP), account levels, in-game currency, or cosmetic items regardless of whether they win or lose a match. This ensures that time spent in the game is always rewarded, creating a durable engagement loop that incentivises continued participation beyond the simple desire to win.

2.4.3 Application to 'Bomberman: Enhanced Edition'

The design of 'Bomberman: Enhanced Edition' intentionally incorporates foundational versions of these modern systems to elevate the classic arcade formula. The project implements a friend-code system, which is a direct application of the essential friends list and presence paradigm, creating a core social hub for users. Concurrently, the Deathmatch leaderboard is a modernised interpretation of the classic high-score table, but its scope is deliberately focused on a player's friends list. This design choice directly applies the principle of contextual competition, making the rankings more personal and impactful. While the project does not implement a complex MMR or guild system, its integration of these fundamental social and competitive features aims to create a more durable and engaging experience that aligns with the expectations of a modern online audience.

2.5 Positioning the Project's Contribution in the Field

The preceding sections have established the context for this project, reviewing the design philosophy of classic arcade games, the modern technologies available for web-based game development, common full-stack architectures, and the social and competitive features that define contemporary online gaming. Having surveyed this landscape, it is now possible to position the unique contribution of 'Bomberman: Enhanced Edition'.

The project does not claim to invent a novel game engine or a revolutionary architectural pattern. Rather, its primary contribution lies in the synthesis of these established paradigms to address a specific challenge: the modernisation of a classic, session-based arcade title into a persistent, service-oriented web application. While many retro clones exist, they often focus solely on replicating the original gameplay. Conversely, large-scale commercial remakes typically utilise proprietary, heavyweight engines for desktop or console platforms.

This project carves a distinct niche between these two approaches. The key contributions are threefold:

- **A Practical Case Study in Architectural Synthesis:** The project serves as a comprehensive case study demonstrating how a classic, single-screen game design can be successfully integrated with a modern three-tier web architecture. It provides a blueprint for how to support real-time front-end gameplay with a back-end that manages persistent user data, authentication, and social features.
- **An Archetype for Accessible Modernisation:** By leveraging native, lightweight web technologies (Phaser 3, Node.js), the project creates a highly accessible experience that requires no downloads or installations, running directly in a standard web browser. This presents a contribution to the field by offering a model for modernising retro titles that prioritises accessibility and rapid engagement over the graphical fidelity offered by larger, downloadable engines.
- **A Model for Curated Feature Integration:** The project's contribution is also found in its deliberate and curated selection of modern features. Instead of attempting to incorporate every feature common to large-scale online games (e.g., guilds, matchmaking), it focuses on a core loop of high-impact social and competitive systems—specifically, a friend system and a friend-scoped leaderboard. This demonstrates a design philosophy focused on enhancing the original's spirit of social competition without fundamentally altering its core simplicity.

Chapter 3

Project Contribution: Requirements and System Design

This chapter transitions from the theoretical background of the project to its practical conception and architectural design. It begins with a detailed analysis of the system requirements, which form the foundational specifications upon which the application was built. It then proceeds to describe the high-level system design, outlining the architectural decisions made to fulfil these requirements.

3.1 Requirements Analysis

The requirements analysis phase is a critical process in software engineering that translates the high-level project aims into a detailed and unambiguous set of specifications. This process defines the precise functionalities the system must provide, the attributes it must possess, and the constraints within which it must operate. The requirements were elicited from the project's core objective: to modernise the classic *Bomberman* game with a suite of contemporary web-based features.

The requirements are categorised into two primary types: **Functional Requirements**, which describe what the system does, and **Non-Functional Requirements**, which describe how the system performs its functions.

3.1.1 Functional Requirements (FR)

Functional requirements specify the explicit behaviours and functions of the system. They define the interactions between the user and the system, as well as the internal operations needed to support these interactions. For clarity, these requirements have been grouped by system component and prioritised as High, Medium, or Low.

Table 3.1: User Management and Authentication Requirements

ID	Requirement Description	Priority
FR-1.1	The system shall allow a new user to register for an account using a valid email address and a password.	High
FR-1.2	The system shall implement a CAPTCHA or a simple mathematical question during registration to deter automated bots.	Medium
FR-1.3	Upon submitting registration details, the system shall generate a six-digit One-Time Password (OTP) and send it to the user's provided email address for verification.	High
FR-1.4	The system shall require the user to enter the correct OTP to complete the registration process. The OTP shall expire after a predefined period (e.g., 10 minutes).	High
FR-1.5	The system shall allow a registered user to log in using their email and password credentials.	High
FR-1.6	Upon successful registration, the system shall automatically generate and assign a unique, numerical friend code to the user's account.	High

Table 3.2: Core Gameplay Mechanics Requirements ('Enhanced Edition')

ID	Requirement Description	Priority
FR-2.1	The system shall render a 2D, grid-based game world composed of indestructible (hard) and destructible (soft) walls.	High
FR-2.2	The player must be able to control a character avatar, moving it along the grid's horizontal and vertical axes.	High
FR-2.3	The player must be able to place a bomb at the character's current grid position. The number of active bombs a player can place simultaneously shall be limited.	High
FR-2.4	A placed bomb shall detonate after a fixed time delay, creating a cross-shaped explosion that propagates along the grid until it is blocked by an indestructible wall.	High
FR-2.5	The explosion shall destroy any soft walls, enemies, and the player character if they are within its blast radius.	High
FR-2.6	Destroying soft walls shall have a chance to reveal a hidden power-up or the level's exit door.	High
FR-2.7	The system shall implement a variety of power-ups that augment the player's abilities (e.g., increase bomb count, blast radius, movement speed).	High
FR-2.8	The system shall populate levels with non-player character (NPC) enemies with distinct movement patterns and AI behaviours.	High
FR-2.9	The objective of each level shall be to eliminate all enemies and locate the exit door to proceed.	High

Table 3.3: Game Mode and Feature Requirements

ID	Requirement Description	Priority
FR-3.1	The system shall provide a 'Standard Mode', a single-player campaign consisting of multiple, progressively difficult stages.	High
FR-3.2	The system shall provide a 'Deathmatch Mode', a survival-based mode where the objective is to achieve the highest possible score by defeating endlessly spawning enemies.	Medium
FR-3.3	The system shall offer a 'Classic Mode', which provides access to an embedded, original version of the *Bomberman* game.	Medium
FR-3.4	The system shall allow the player to save their progress during the Standard Mode campaign. The saved state must include the current level, score, lives, and collected power-ups.	High
FR-3.5	A logged-in user must be able to load their saved game from the main menu to continue their session.	High
FR-3.6	The system shall implement a level selection screen, allowing the player to start the game from any previously unlocked stage in the Standard Mode campaign.	Medium

Table 3.4: Social and Competitive System Requirements

ID	Requirement Description	Priority
FR-4.1	The system shall allow a user to send a friend request to another user by entering their unique friend code.	High
FR-4.2	Users shall be able to view their incoming friend requests and choose to accept or decline them.	High
FR-4.3	The system shall maintain a friends list for each user, displaying the names of their accepted friends.	High
FR-4.4	The system shall implement a leaderboard for the Deathmatch Mode.	Medium
FR-4.5	The leaderboard shall display the highest scores achieved, ranking the current user and their friends.	Medium

3.1.2 Non-Functional Requirements (NFR)

Non-functional requirements define the quality attributes, operational constraints, and technical standards of the system. They specify *how* the system should perform its functions.

Table 3.5: Non-Functional Requirements

ID	Category	Requirement Description
NFR-1	Usability	The user interface for all menus (main, login, social, pause) shall be intuitive, with clear navigation and legible, retro-themed text.
NFR-2	Usability	The system must support dual input methods for gameplay: keyboard (arrow keys for movement, specific keys for actions) and standard gamepad controllers.
NFR-3	Performance	As a web-based application, the initial load time of the game client shall be minimised to ensure rapid user engagement.
NFR-4	Performance	Gameplay in the 'Enhanced Edition' shall maintain a smooth and stable frame rate, targeting 60 frames per second on standard modern hardware to ensure responsive controls.
NFR-5	Compatibility	The application must be fully functional and render correctly on the latest stable versions of major desktop web browsers, including Google Chrome, Mozilla Firefox, Apple Safari, and Microsoft Edge.
NFR-6	Security	All communication between the client and the server, particularly involving user credentials and personal data, must be secured (e.g., via HTTPS).
NFR-7	Security	The OTP verification system must be robust enough to serve as a primary mechanism for validating user identity upon registration.
NFR-8	Maintainability	The source code shall be modular and well-documented. A clear separation of concerns must be maintained between the front-end (Phaser client) and back-end (Node.js API) codebases to facilitate future development and debugging.

3.2 System Design

Following the formalisation of requirements, the system design phase defines the high-level architecture, components, modules, and data flows necessary to meet those specifications. This section describes the architectural blueprint of the 'Bomberman: Enhanced Edition' application, detailing each constituent part and their interactions.

3.2.1 High-Level System Architecture

To satisfy the requirements for both a real-time interactive game and a persistent, data-driven social platform, the system was designed using a **three-tier client-server architecture**. This model provides a robust separation of concerns, decoupling the user interface (presentation), business logic (application), and data storage (persistence), which enhances modularity, scalability, and maintainability.

The architecture consists of a thin client running in the user's web browser, which communicates with a central authoritative server via a stateless RESTful API. The server, in turn, interfaces with a database to persist all user and application data. Figure 3.1 below provides a high-level schematic of this architecture.

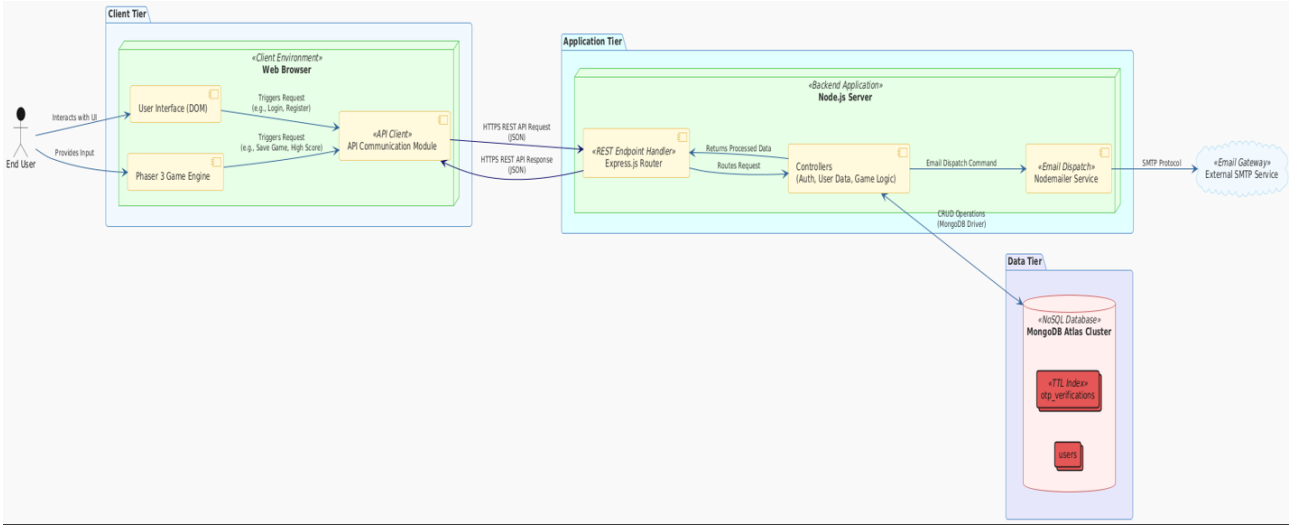

Figure 3.1: A detailed diagram of the three-tier system architecture for 'Bomberman: Enhanced Edition'. The diagram shows the Client Tier (Browser with Phaser 3, UI Layer, and API Module), the Application Tier (Node.js Server with Express Router, Controllers for User/-Social/Game, and Nodemailer), and the Data Tier (MongoDB with users and otp_verifications collections). Arrows indicates the flow of requests and responses via the REST API over HTTPS, and the server's interaction with the database.

The following subsections provide a detailed analysis of each tier of this architecture.

The Client Tier: Presentation Layer

The Client Tier is a **Single-Page Application (SPA)** that runs entirely within the user's web browser. It is responsible for all rendering, user input processing, and real-time game logic execution.

- **Technology Stack:** The client is built upon standard web technologies: HTML5 for structure, CSS3 for styling, and JavaScript (ES Modules) for logic.
- **Game Engine:** The core of the client is the **Phaser 3** framework. It manages the main game loop, renders graphics to either WebGL or Canvas for maximum compatibility, handles asset loading (spritesheets, audio, tilemaps), and runs the Arcade Physics engine for collision detection and movement.
- **Application Structure:** The application employs a hybrid structure for its user interface.
  - **DOM-based UI:** The primary user interface for non-gameplay screens (e.g., login, registration, social hub, leaderboards) is constructed using standard HTML DOM elements. This approach is highly efficient for displaying forms, lists, and static text. JavaScript is used to dynamically show, hide, and populate these UI containers based on user interaction and application state.
  - **Phaser-based UI:** The in-game experience, including the game world, player characters, enemies, and the Heads-Up Display (HUD), is rendered and managed entirely

by Phaser.

- **State Management:** The client maintains its own temporary state in memory via JavaScript objects (e.g., the `currentUser` object, which holds the logged-in user's data). This state is populated and updated through communication with the back-end server.
- **API Communication Module:** A dedicated module within the client-side JavaScript code is responsible for all network communication. It abstracts the use of the browser's `fetch` API into a series of asynchronous functions (e.g., `dbLoginUser`, `dbUpdateUser`). This module formulates HTTP requests, sends them to the server's REST API endpoints, and processes the incoming JSON responses, updating the client-side state accordingly.

The Application Tier: Logic Layer

The Application Tier is an authoritative server that executes all business logic and serves as the single source of truth for persistent data. It is designed to be stateless, processing each client request independently.

- **Technology Stack:** The server is built on the **Node.js** runtime environment, utilising the **Express.js** framework to simplify routing and middleware management.
- **API Design:** The server exposes a **RESTful API** through which the client interacts. This API defines a set of specific endpoints (e.g., `/api/login`, `/api/user/:email`, `/api/users/by-codes`) that correspond to specific functionalities. All data is exchanged between the client and server in **JSON (JavaScript Object Notation)** format.
- **Request Handling Flow:** The typical lifecycle of a request is as follows:
  1. An HTTP request from the client arrives at a specific API endpoint.
  2. The Express.js router maps the request path and method (GET, POST, PUT) to a designated controller function.
  3. The controller function executes the core logic. This includes validating input data, performing the required operations (e.g., querying the database, creating a new user), and constructing a response.
  4. A JSON object and an appropriate HTTP status code (e.g., 200 OK, 201 Created, 404 Not Found) are sent back to the client.
- **External Service Integration:** For the email verification feature, the server integrates the **Nodemailer** library. This module interfaces with an external Simple Mail Transfer Protocol (SMTP) server (e.g., Gmail) to dispatch OTP emails to users, abstracting the complexities of email protocols.

The Data Tier: Persistence Layer

The Data Tier is responsible for the long-term storage of all user and application data. Its role is to provide a reliable and efficient mechanism for data persistence and retrieval, as directed by the Application Tier.

- **Database Technology:** The system utilises **MongoDB**, a popular NoSQL, document-oriented database. MongoDB was selected for its flexibility and its native use of a JSON-like format (BSON), which allows for seamless data mapping from the JavaScript-based application logic without requiring an Object-Relational Mapper (ORM).
- **Data Models and Collections:** The database is organised into two primary collections:
  - `users` **Collection:** This is the main collection for persistent user data. Each document represents a single user and contains fields such as email, password (stored in plaintext for this project's scope), the unique `friendCode`, arrays for friends, `incomingRequests`, and `pendingRequests`, and nested objects for game progression data (`standardModeStats`, `deathmatchStats`, and the complete `savedGame` state object).
  - `otp_verifications` **Collection:** This collection serves as a temporary store for pending email verifications. Each document contains the user's email, the generated `otp`, and an `expiresAt` timestamp. A critical feature implemented at the database level is a **Time-To-Live (TTL) index** on the `expiresAt` field. This instructs MongoDB to automatically delete documents once their expiry time has passed, ensuring that the collection is self-cleaning and contains only valid, unexpired OTP records.

3.2.2 Database Schema Design

The selection of MongoDB as the database management system was a strategic decision informed by the application's data structure requirements. As a NoSQL, document-oriented database, MongoDB stores data in flexible, JSON-like documents called BSON (Binary JSON). This model offers significant advantages for this project:

- **Schema Flexibility:** It allows for the storage of complex, nested data structures, such as the game state object, without requiring rigid, predefined schemas or complex table joins, which would be necessary in a traditional relational database (e.g., SQL).
- **Direct Mapping to Application Objects:** The BSON document structure maps directly to the native JavaScript objects used within the Node.js application logic. This eliminates the "impedance mismatch" between relational tables and application objects, simplifying data access and removing the need for an Object-Relational Mapping (ORM) layer.

The database, named `bomberman_db`, is organised into two distinct collections, each designed

for a specific purpose. The following sections provide a granular breakdown of the schema for each collection.

The `users` Collection Schema

This is the primary and permanent collection in the database. It is designed to be the single source of truth for all persistent user information, including authentication credentials, social graph data, and detailed game progression statistics. Each document in this collection represents one registered user.

Table 3.6: Field-by-Field Schema for the `users` Collection

Field Name	Data Type	Description & Purpose
`_id`	ObjectId	The default, auto-generated, and indexed 12-byte primary key for the document. Ensures absolute uniqueness for each user record.
`email`	String	The user's unique email address. It serves as the primary identifier for login and is indexed to ensure fast lookups during authentication.
`password`	String	The user's password. **Note:** For the scope of this project, this is stored in plaintext. In a production environment, this field would store a salted hash (e.g., generated by bcrypt) to ensure credential security.
`friendCode`	String	A unique, system-generated 8-digit string that serves as the user's public identifier for the social system. It is indexed to optimise search performance when a user attempts to add a friend.
`friends`	Array<String>	An array containing the `friendCode` strings of other users who are on this user's friends list. This represents the user's established social connections.
`incomingRequests`	Array<String>	An array containing the `friendCode` strings of users who have sent a friend request to this user, which are awaiting a response (accept/decline).
`pendingRequests`	Array<String>	An array containing the `friendCode` strings of users to whom this user has sent a friend request that has not yet been answered.
`standardModeStats`	Object	A nested document containing statistics and progression data specifically for the single-player 'Standard Mode' campaign.
`highestScore`	Number	The highest score the user has ever achieved in a completed or game-over session of the Standard Mode.

Continued on next page

Table 3.6 – continued from previous page

Field Name	Data Type	Description & Purpose
`unlockedLevel`	Number	An integer representing the highest level number that the user has successfully completed and thus unlocked for future play via the level select screen.
`deathmatchStats`	Object	A nested document containing the user's personal best statistics for the 'Deathmatch Mode'.
`highestScore`	Number	The highest score achieved by the user in a single game of Deathmatch. This is the value displayed on the leaderboard.
`totalTime`	Number	The total duration in seconds of the session in which the `highestScore` was achieved.
`enemiesKilled`	Object	A key-value map detailing the count of each enemy type defeated during the highest-scoring session (e.g., `{"ballom": 15, "onil": 8}`).
`savedGame`	Object | null	A complex, nested document that stores the complete state of a user's saved game in the Standard Mode campaign. This field is null if no game is saved. Its structure mirrors the in-memory state of the Phaser game client.
`gameStage`	Object	Contains metadata about the game session, including time, lives, stage, stageScore, and an array of collected powerUps.
`player`	Object	Stores the player's state at the moment of saving, including x and y coordinates.
`enemies`	Array<Object>	An array where each object represents an active enemy on the map, storing its state including x, y coordinates and enemyData type.
`map`	Object	Stores the procedural state of the current level map, including an array of all walls coordinates and the state (x, y, isVisible) of the hidden door and powerUp.

Below is a sample BSON document representing a user in this collection:

```
{
  "_id": "65e6f8a4bf8c3d3e9f4a1b2",
  "email": "player1@example.com",
  "password": "secure_password_placeholder",
  "friendCode": "12345678",
  "friends": [ "87654321" ],
  "incomingRequests": [],
  "pendingRequests": [ "11223344" ],
  "standardModeStats": {
```

```
    "highestScore": 12500,
    "unlockedLevel": 3
  },
  "deathmatchStats": {
    "highestScore": 25400,
    "totalTime": 185,
    "enemiesKilled": {
      "ballom": 25,
      "onil": 15,
      "dahl": 5
    }
  },
  "savedGame": {
    "gameStage": {
      "time": 152,
      "lives": 2,
      "stage": 2,
      "stageScore": 8400,
      "powerUps": [ 0, 1 ]
    },
    "player": { "x": 60, "y": 120 },
    "enemies": [
      {
        "x": 300,
        "y": 160,
        "enemyData": { "type": "onil" }
      }
    ],
    "map": {
      "walls": [ { "x": 100, "y": 120 } ],
      "door": { "x": 140, "y": 160, "isVisible": false }
    }
  }
}
```

Listing 3.1: Sample BSON document for a user

The `otp_verifications` Collection Schema

This collection serves a specialised and ephemeral purpose: to temporarily store data related to the One-Time Password (OTP) verification process during user registration. Documents in this collection are designed to be short-lived.

Table 3.7: Field-by-Field Schema for the `otp_verifications` Collection

Field Name	Data Type	Description & Purpose
`_id`	ObjectId	The default, auto-generated primary key for the document.
`email`	String	The email address of the user attempting to register. This field is indexed to allow for fast retrieval when the user submits their OTP for verification.
`otp`	String	The 6-digit cryptographic random string generated and sent to the user's email.
`expiresAt`	Date	A BSON Date object specifying the exact UTC timestamp at which this OTP record becomes invalid.

A defining feature of this collection's design is the use of a **Time-To-Live (TTL) index** on the `expiresAt` field. A TTL index is a special single-field index that MongoDB can use to automatically remove documents from a collection after a certain amount of time. The server application inserts a document with the `expiresAt` field set to 10 minutes in the future. The TTL index on the MongoDB server monitors this field and will automatically purge the document from the collection once the current time exceeds the `expiresAt` value. This is a highly efficient, database-native mechanism for managing the lifecycle of temporary data, eliminating the need for application-level cleanup tasks or scheduled jobs.

Below is a sample BSON document from this collection:

```
{
  "_id": ObjectId("65e6f9d8c4f7a2d4e0a5b2c3"),
  "email": "new_user@example.com",
  "otp": "582109",
  "expiresAt": ISODate("2025-09-04T12:15:30.000Z")
}
```

Listing 3.2: Sample BSON document for OTP verification

3.2.3 Backend API Design

The communication layer between the client-side application and the server-side logic is a critical component of the system architecture. This project implements a **Representational State Transfer (REST) API** to facilitate this communication. REST is an architectural style that defines a set of constraints for creating scalable, stateless, and maintainable web services. The API adheres to RESTful principles, utilising standard **HTTP methods** (e.g., POST, GET, PUT), conventional **HTTP status codes** for indicating outcomes, and **JavaScript Object Notation (JSON)** as the exclusive data interchange format.

All endpoints are namespaced under the base path `/api` to distinguish them from any potential front-end routing. The following subsections provide a granular specification for each endpoint, detailing its purpose, request structure, and potential responses.

Authentication Endpoints

These endpoints manage the user lifecycle, including registration and login.

1. **Send One-Time Password (OTP)**
   - **Endpoint:** `POST /api/send-otp`
   - **Description:** Initiates the user registration process. It validates the provided email, checks for existing accounts, generates a 6-digit OTP, stores it in the `otp_verifications` collection with a 10-minute expiry, and dispatches it to the user's email address.
   - **Request Body:** `{"email": "user@example.com"}`
   - **Success Response (200 OK):** `{"success": true, "message": "OTP sent to your email."}`
   - **Error Responses:** 400 Bad Request, 409 Conflict, 500 Internal Server Error.

2. **Verify OTP and Register User**
   - **Endpoint:** `POST /api/verify-otp`
   - **Description:** Completes the user registration. It validates the user-submitted OTP against the stored record in the database, checking for correctness and expiry. Upon successful verification, it creates a new user document in the `users` collection and deletes the OTP record.
   - **Request Body:** `{"email": "...", "password": "...", "otp": "..."}`
   - **Success Response (201 Created):** `{"success": true, "message": "Account created successfully!"}`
   - **Error Responses:** 400 Bad Request, 500 Internal Server Error.

3. **User Login**
   - **Endpoint:** `POST /api/login`
   - **Description:** Authenticates an existing user. It finds the user by email and compares the provided password with the one stored in the database.
   - **Request Body:** `{"email": "...", "password": "..."}`
   - **Success Response (200 OK):** A JSON object containing the full user document (excluding the password).
   - **Error Responses:** 401 Unauthorized, 500 Internal Server Error.

User Data and Social System Endpoints

These endpoints are responsible for retrieving and modifying user data, including game state and social connections.

1. **Update User Data**
   - **Endpoint:** `PUT /api/user/:email`
   - **Description:** A general-purpose endpoint for modifying a user's document. The :email in the URL is a parameter specifying which user to update. This single endpoint is used for a wide range of stateful operations, including saving a game (savedGame object), updating social lists (friends, incomingRequests), and recording new high scores (deathmatchStats).
   - **URL Parameters:** `:email` (String)
   - **Request Body:** A JSON object containing the fields to be updated.
   - **Success Response (200 OK):** `{"success": true}`
   - **Error Responses:** 404 Not Found, 500 Internal Server Error.

2. **Find User by Friend Code**
   - **Endpoint:** `GET /api/user/by-code/:friendCode`
   - **Description:** Retrieves the public data of a single user by their unique friend code. This is used when a player sends a friend request.
   - **URL Parameters:** `:friendCode` (String)
   - **Success Response (200 OK):** A JSON object with the target user's data.
   - **Error Responses:** 404 Not Found, 500 Internal Server Error.

3. **Get Multiple Users by Friend Codes (Bulk Retrieval)**
   - **Endpoint:** `POST /api/users/by-codes`
   - **Description:** A highly efficient endpoint designed to retrieve the data for multiple users in a single network request. The client provides an array of friend codes (e.g., the current user's entire friends list), and the server returns an array of the corresponding user documents. This design choice is critical for performance, as it avoids the need for the client to make numerous individual GET requests in a loop, significantly reducing network latency when populating lists like the friends list or the leaderboard.
   - **Request Body:** `{"friendCodes": ["...", "...", "..."]}`
   - **Success Response (200 OK):** A JSON object with an array of user documents.
   - **Error Responses:** 400 Bad Request, 500 Internal Server Error.

3.2.4 Frontend Architecture and UI Flow

The frontend architecture of 'Bomberman: Enhanced Edition' is designed as a **Single-Page Application (SPA)**. This architectural model provides a fluid and responsive user experience, akin to a desktop application, by loading a single HTML document (`index.html`) and dynamically manipulating its content with JavaScript in response to user interaction, thereby eliminating the need for disruptive full-page reloads.

The architecture is a **hybrid model** that strategically combines two distinct technologies for managing the user interface: standard **HTML DOM manipulation** for static menu and information screens, and the **Phaser 3 game engine** for the dynamic, real-time gameplay environment. This approach leverages the strengths of each technology—the efficiency and simplicity of the DOM for form-based UIs, and the high-performance rendering capabilities of Phaser for the game itself.

Core Frontend Components

The frontend is composed of several key architectural components that work in concert to manage the application's state and presentation.

- **View Manager:** The primary controller for the non-game UI is a simple but effective view management system orchestrated by the `showScreen()` JavaScript function. This manager treats the various top-level `div` containers (e.g., `#login-screen`, `#social-screen`, `#leaderboard-screen`) as distinct "views" or "states". It functions as a finite state machine, where transitions are triggered by user events (e.g., button clicks). A transition involves removing a CSS `.active` class from the current view and applying it to the target view, which efficiently toggles their visibility.
- **API Communication Module:** This service layer, detailed previously, acts as the bridge between the frontend components and the backend API. It encapsulates all `fetch` calls into a set of well-defined asynchronous functions. This decouples the UI logic from the complexities of network requests, error handling, and JSON parsing, making the view components cleaner and more focused on presentation.
- **Phaser 3 Game Instance:** This component represents the self-contained game world. It is initialised once upon successful user login and mounted into the `#bomberman-container div`. Once active, the Phaser instance takes exclusive control of its container, managing its own render loop, asset cache, and physics simulation.
- **Phaser Scene Manager:** Within the Phaser instance, a more sophisticated state machine, the Scene Manager, controls the flow of the game experience. The application is broken down into modular scenes, each with a specific responsibility (e.g., `Boot`, `Preloader`, `MainMenu`, `Game`, `PauseMenu`, `ChangeStage`). The Scene Manager handles the lifecycle of these scenes—loading, creating, updating, pausing, resuming, and shutting them down—providing a structured way to manage game states.

- **Input Abstraction Layer:** A custom `InputManager` class is implemented within the Phaser game. This component is a critical piece of the design, providing an abstraction layer that unifies user input from multiple sources. It polls the state of both the keyboard and any connected gamepad controllers, exposing a single, clean interface (e.g., `isMovingUp()`, `justPressedPutBomb()`) to the rest of the game logic. This decouples the game's character controller and other systems from the specific input hardware, greatly improving code maintainability and extensibility.

User Interface (UI) Flow

The user's journey through the application is a sequence of transitions between the DOM-based views and the Phaser-managed scenes. The flow is designed to guide the user from authentication through to gameplay in a logical progression.

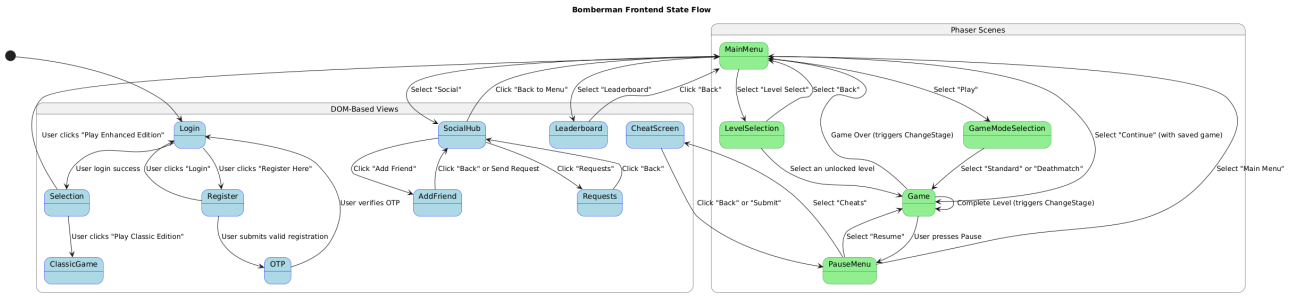

Figure 3.2: A detailed state flow diagram illustrating the transitions between all frontend views and scenes. The diagram distinguishes between DOM-based views (e.g., Login, Register, Social Hub) and Phaser Scenes (e.g., MainMenu, Game, PauseMenu), showing the user events that trigger each transition.

The sequence of states can be described as follows:

1. **Initialisation and Authentication:**
    - The application loads and the View Manager displays the initial state: the Login View (`#login-screen`).
    - From here, the user can trigger a transition to the Register View (`#register-screen`).
    - Submitting the registration form validates the input and, upon success, transitions the user to the OTP Verification View (`#otp-screen`).
    - After successful verification, the user is returned to the Login View to authenticate.
    - A successful login event triggers the most significant architectural transition: the View Manager hides all DOM-based UI containers, and the Phaser Game Instance is initialised.
2. **Game World Entry:**

- The Phaser instance begins its lifecycle, sequentially loading its scenes: `Boot` → `Preloader` (to load all game assets) → `MainMenu`.
- The **MainMenu Scene** becomes the central hub for all in-game activities.

3. **Navigating Between Game and UI Shell:**
   - From the `MainMenu` scene, the user can navigate to other Phaser scenes, such as `GameModeSelection` or `LevelSelection`, which remain within the control of the Phaser Scene Manager.
   - A key interaction occurs when the user selects an option like 'Social' or 'Leaderboard'. This action demonstrates the interplay between the two UI systems:
     - The active Phaser scene (`MainMenu`) is paused.
     - Control is passed back to the DOM View Manager, which displays the appropriate view (`#social-screen` or `#leaderboard-screen`).
     - When the user elects to return (e.g., clicks a "Back" button), the View Manager hides the DOM view, and the `MainMenu` scene is resumed.
4. **The Gameplay Loop:**
   - When a player starts a game from a selection menu, the Phaser Scene Manager transitions them, typically via the `ChangeStage` scene (a loading screen), into the primary `Game` Scene.
   - During gameplay, the player can trigger the pause action. This pauses the `Game` scene and wakes the overlaying `PauseMenu` Scene.
   - The `PauseMenu` presents several options:
     - **Resume:** The `PauseMenu` scene is put to sleep, and the `Game` scene is resumed.
     - **Main Menu:** The `Game` scene is stopped entirely, and the application transitions back to the `MainMenu` scene.
     - **Cheats:** In a similar pattern to the 'Social' hub, the `Game` scene and `PauseMenu` scene remain paused while control is handed to the DOM View Manager to display the `#cheat-screen` view.

This hybrid architectural approach allows the application to effectively manage a complex set of states, from simple static forms to a high-performance, real-time game environment, providing a seamless and feature-rich user experience.

Chapter 4

Project Contribution: Implementation

This chapter details the practical execution of the project, translating the system design outlined in the previous chapter into a functional software application. The discussion covers the development environment, the implementation of the backend and frontend systems, and the key algorithms and logic that underpin the application's features.

4.1 Development Environment and Tooling

The successful implementation of a modern, full-stack application is contingent upon a carefully selected and configured set of development tools. The toolchain for this project was chosen to create a productive, coherent, and maintainable environment, primarily leveraging the JavaScript ecosystem for both client and server-side development. This section provides a detailed, foundational explanation of each component of the development environment.

4.1.1 Core Technologies and Runtime Environment

These technologies form the foundational layer upon which the entire application is built and executed.

- **Node.js:** The core of the development environment is `Node.js`, an open-source, cross-platform `JavaScript` runtime environment. Built on Google Chrome's V8 `JavaScript` engine, `Node.js` allows for the execution of `JavaScript` code outside the confines of a web browser. In this project, `Node.js` served a critical dual role:
  - **Backend Runtime:** It is the environment in which the application's server-side logic, built with the `Express.js` framework, is executed. Its asynchronous, event-driven, non-blocking I/O model makes it highly efficient for handling concurrent API requests.
  - **Development Tooling Ecosystem:** `Node.js` provides the Node Package Manager (`npm`), which is the cornerstone of modern `JavaScript` project management.

- **npm (Node Package Manager):** `npm` is the world's largest software registry and the default command-line package manager for `Node.js`. Its primary function is to manage a project's dependencies—the external libraries and frameworks required for the application to function. All project dependencies are defined in a manifest file named `package.json`. For this project, `npm` was used to install and manage all backend dependencies, such as `express`, `mongodb`, `cors`, and `nodemailer`.

4.1.2 Backend Development Stack

The backend, or Application Tier, was constructed using a curated set of `Node.js` frameworks and libraries.

- **Express.js:** This is a minimalist and highly flexible web application framework for `Node.js`. It provides a robust foundation for building the project's REST API without imposing a rigid structure. Its primary responsibilities within the project include:
  - **Routing:** Mapping incoming HTTP requests (e.g., `GET /api/user/...`) to specific handler functions that execute the appropriate business logic.
  - **Middleware Management:** Processing incoming requests through a pipeline of functions. The project utilises middleware such as `express.json()` to parse incoming JSON request bodies and the `cors` middleware for security.
  - **Request and Response Handling:** Abstracting the complexities of `Node.js`'s native `http` module, providing a simplified interface for accessing request data (headers, body, parameters) and sending back responses.
- **Nodemailer:** A specialised, single-purpose `Node.js` module for sending emails. It was integrated into the backend to handle the critical functionality of dispatching One-Time Password (OTP) verification emails to users during the registration process. It provides a simple API for connecting to an SMTP server and sending formatted HTML or plain text emails.
- **cors Middleware:** A `Node.js` package providing an Express middleware for enabling Cross-Origin Resource Sharing (CORS). A web browser's same-origin policy, a fundamental security measure, prevents a web page from making requests to a different domain than the one that served the page. Since the frontend game client and backend API were developed to run on different ports (and potentially different domains), the `cors` middleware was essential to configure the server to explicitly permit these cross-origin requests from the client.

4.1.3 Frontend Development Stack

The frontend, or Presentation Tier, was built using standard web technologies and a specialised game framework.

- **HTML5, CSS3, and ES6+ JavaScript:** The application is built upon the foundational technologies of the modern web. `HTML5` provides the document structure, `CSS3` handles all styling for the non-game UI elements, and modern `ECMAScript` (`ES6+`) `JavaScript` is used for all client-side logic. Features such as `ES Modules`, `async/await` syntax for handling asynchronous API calls, and arrow functions were used extensively.
- **Phaser 3:** The core of the frontend's interactive component is the `Phaser 3` framework. As detailed in the previous chapter, it is a comprehensive 2D game framework that provides all the necessary subsystems for building the game, including the scene manager, physics engine, sprite and animation handler, asset loader, and a unified input system.

4.1.4 Database and Data Management

The persistence layer was managed using a cloud-based database service and a dedicated driver.

- **MongoDB Atlas:** Rather than hosting a database instance locally or on a self-managed server, this project utilised `MongoDB Atlas`, a fully managed, cloud-based Database-as-a-Service (DBaaS) platform. This service handles all aspects of database administration, including server provisioning, configuration, backups, and security. This approach allowed development efforts to be focused entirely on the application logic rather than on database infrastructure management.
- **MongoDB Driver for Node.js:** The official `mongodb npm` package is the driver used by the `Node.js` application to interface with the `MongoDB Atlas` cluster. This driver provides the necessary methods to establish a secure connection to the database (using the URI provided by Atlas) and to perform all CRUD (Create, Read, Update, Delete) operations on the collections. All database interactions in the server code, such as `usersCollection.findOne()` or `otpCollection.updateOne()`, are executed via this driver.

4.1.5 General Tooling and Version Control

These tools were used across the entire development lifecycle to facilitate coding, debugging, and collaboration.

- **Visual Studio Code (VS Code):** The primary Integrated Development Environment (IDE) used for writing all frontend and backend code. `VS Code` is a powerful, extensible source-code editor that provides numerous features essential for modern development, including advanced `IntelliSense` (code completion and parameter info), an integrated terminal for running commands, built-in debugging support for `Node.js`, and seamless integration with the `Git` version control system.
- **Git:** A distributed Version Control System (VCS) used to track every change made to the project's source code. `Git` is the industry standard for managing code history, en-

abling critical workflows such as branching for feature development, merging to integrate changes, and maintaining a complete, revertible history of the project. It ensures code integrity and provides a safety net against accidental data loss.

- **GitLab:** A web-based platform for hosting `Git` repositories. It was used as the central, cloud-based repository for the project's source code, providing an off-site backup and a platform for potential future collaboration.

4.2 Backend Implementation: Secure User Authentication and Data Management

The backend is the authoritative core of the application, responsible for enforcing business logic, securing user data, and managing the persistence of all application state. This section delves into the specific implementation details of these server-side systems, beginning with the critical user registration and authentication workflow. The logic is implemented in JavaScript running on the Node.js runtime, utilising the Express.js framework and the official MongoDB driver.

4.2.1 Code Excerpt 1: OTP Verification and User Creation Logic

The following code excerpt is the implementation of the `POST /api/verify-otp` endpoint. This function represents the final, critical stage of the user registration process. Its responsibility is to validate the One-Time Password (OTP) provided by the user, and upon successful validation, to create a permanent user record in the database. This process combines security validation, data persistence, and state management within a single, atomic transaction.

```
// server.js
app.post('/api/verify-otp', async (req, res) => {
    const { email, password, otp } = req.body;
    if (!email || !password || !otp) {
        return res.status(400).json({ success: false, message: 'Email, password, and OTP are required.' });
    }
    try {
         // Find the OTP record for the given email
        const otpRecord = await otpCollection.findOne({ email });
        if (!otpRecord) {
            return res.status(400).json({ success: false, message: 'Invalid OTP. Please request a new one.' });
        }
         // Check if the OTP matches and has not expired
        if (otpRecord.otp !== otp || new Date() > otpRecord.expiresAt) {
            return res.status(400).json({ success: false, message: 'Invalid or expired OTP.' });
        }
         // --- OTP is valid, proceed with user registration ---
```

```
18          // Generate a unique friend code
19          let friendCode;
20          let isCodeUnique = false;
21          while (!isCodeUnique) {
22              friendCode = Math.floor(10000000 + Math.random() * 90000000).
    toString();
23              const userWithCode = await usersCollection.findOne({ friendCode
    });
24              if (!userWithCode) {
25                  isCodeUnique = true;
26              }
27          }
28
29          const newUser = {
30              email,
31              password, // NOTE: Plaintext storage for project scope.
    Production requires hashing.
32              friendCode,
33              friends: [],
34              incomingRequests: [],
35              pendingRequests: [],
36              standardModeStats: { highestScore: 0, unlockedLevel: 0 },
37              deathmatchStats: { highestScore: 0, totalTime: 0, enemiesKilled:
     {} },
38              savedGame: null
39          };
40          // Insert the new user document into the collection
41          await usersCollection.insertOne(newUser);
42          // Clean up the used OTP from the verification collection
43          await otpCollection.deleteOne({ email });
44          res.status(201).json({ success: true, message: 'Account created
    successfully!' });
45      } catch (error) {
46          console.error("Verification/Registration error:", error);
47          res.status(500).json({ success: false, message: 'Server error during
     registration.' });
48      }
49  });
```

Listing 4.1: Endpoint for verifying OTP and completing user registration

Detailed Analysis of Implementation The execution flow of this function is a sequential process of validation, creation, and cleanup. Each step is detailed below.

1. **Endpoint Definition and Asynchronous Execution:** The function is bound to the `POST /api/verify-otp` route using the `app.post()` method from Express.js. The entire callback function is declared as `async`, which is essential as it contains multiple `await` expressions for handling asynchronous database operations. This ensures that the function

pauses execution at each database call and waits for the operation to complete before proceeding, preventing race conditions and simplifying the control flow of asynchronous code.

2. **Request Payload Processing and Validation:** The first action is to destructure the `email`, `password`, and `otp` fields from the incoming request's JSON body (`req.body`). An immediate guard clause checks for the presence of all three fields. If any are missing, the function terminates early and returns an HTTP 400 Bad Request status, a crucial input validation step to reject malformed requests before performing any database operations.

3. **OTP Record Retrieval:** The core logic is wrapped in a `try...catch` block for robust error handling. The first database operation, `await otpCollection.findOne( email )`, queries the `otp_verifications` collection for a document matching the user's email. The `await` keyword ensures that execution is suspended until the MongoDB driver returns either the matching document or `null`.

4. **Security Validation of the OTP:** A multi-stage validation process is then performed on the retrieved `otpRecord`:

   - **Existence Check:** `if (!otpRecord)` handles the case where no OTP was ever generated for the given email, returning a 400 Bad Request.
   - **Correctness and Expiry Check:** The subsequent `if` statement performs two critical security checks in a single condition:
     - `otpRecord.otp !== otp`: This directly compares the user-submitted OTP string with the string stored in the database.
     - `new Date() > otpRecord.expiresAt`: This compares the current server time with the expiry timestamp stored in the OTP document. This check is vital to invalidate OTPs that are correct but have expired, mitigating the risk of old codes being compromised and used later. A failure at this stage also results in a 400 Bad Request, protecting the system from invalid or malicious registration attempts.

5. **Unique Friend Code Generation:** Upon successful OTP validation, the logic proceeds to create the new user entity. A unique `friendCode` is generated as an 8-digit random number. To guarantee uniqueness and prevent collisions, the code enters a `while` loop. Inside the loop, it performs a database query (`usersCollection.findOne( friendCode )`) to check if the newly generated code already exists in the `users` collection. The loop continues until a code is generated that is not found in the database, ensuring the integrity of this key social feature.

6. **New User Document Construction:** A `newUser` JavaScript object is constructed, adhering to the schema defined in section 3.2.2. It populates the object with the validated `email` and `password`, the unique `friendCode`, and initialises all other fields to their

default state (e.g., empty arrays for social lists, zeroed-out statistics, and `null` for the saved game).

7. **Data Persistence and State Cleanup:** Two final, critical database operations are performed:
   - `await usersCollection.insertOne(newUser)`: This command inserts the `newUser` object as a new document into the `users` collection, permanently creating the user account.
   - `await otpCollection.deleteOne( email )`: Immediately after successful user creation, the corresponding OTP record is deleted from the `otp_verifications` collection. This is a crucial cleanup and security step that prevents the same OTP from being used again in a "replay attack."
8. **Success Response:** If all operations complete without error, the server sends a final response to the client with an HTTP 201 Created status code, the standard response for successful resource creation. This informs the client that the registration was successful, allowing it to transition the UI to the next appropriate state (e.g., the login screen).
9. **Global Error Handling:** The overarching `try...catch` block ensures that any unexpected errors during the database interactions (e.g., a connection failure, a write error) are caught. In such cases, the error is logged to the server console for debugging, and a generic HTTP 500 Internal Server Error is returned to the client, preventing sensitive error details from being exposed.

4.3 Frontend Implementation: Core Gameplay Mechanics in Phaser 3

The frontend client is responsible for executing all real-time game logic, rendering the visual output, and responding to user input. This is achieved through the Phaser 3 framework, which provides a structured environment for game development. This section examines the implementation of the most critical component of any real-time application: the main game loop, which manages the application's state on a frame-by-frame basis.

4.3.1 Code Excerpt 2: Real-time State Management in the Main Game Loop

At the heart of any real-time interactive system is the game loop, a continuous, high-frequency cycle that performs three primary tasks: processing user input, updating the game state, and rendering the new state to the screen. In the Phaser 3 framework, this loop is abstracted from the developer. The primary entry point for executing logic within this loop is the `update()`

method of an active `Scene`. This method is called by the Phaser engine once per frame, typically 60 times per second, making it the central hub for all real-time state management.

The following code excerpt presents the `update()` method from the main `BombermanGame` scene ('Game'). While concise, it demonstrates several key architectural patterns, including delegation of responsibility and management of the game's core state machine.

```
// index.html
update() {
    // 1. Delegate continuous player state management to the Player object
    this._player.addControlsListener();
    // 2. Delegate discrete save-game action listener to the SaveGameManager
    (async () => {
        await this._saveGameManager.addControlsListener();
    })();
    // 3. Handle a core state transition (active to paused)
    if (this._inputManager.justPressedPause()) {
        this.scene.pause();
        this.scene.wake('PauseMenu');
    }
}
```

Listing 4.2: The update method within the main 'Game' Scene class

Detailed Analysis of Implementation The `update()` method serves as the main orchestrator for the active gameplay state. Its logic can be deconstructed into three distinct, high-level responsibilities that are executed on every frame.

1. **Continuous State Polling and Player Control (`this._player.addControlsListener()`):** The first line demonstrates the principle of delegation. Instead of containing complex `if/else` statements to check for keyboard and gamepad input directly within the main update loop, this responsibility is delegated to a method within the `_player` object. This architectural choice keeps the main loop clean and adheres to the single-responsibility principle.

   A granular analysis of the `addControlsListener()` method (from the Player class implementation) reveals the following minute details:

   - **State Polling:** On every frame, this method actively polls the current state of the input devices via the `_inputManager`. It checks properties like `this._inputManager.isMovingRight()`, which returns true as long as the corresponding key or gamepad stick is held down. This continuous polling is what enables fluid, real-time character movement.
   - **Direct State Manipulation:** Based on the polled input, the method directly manipulates the player's state. Specifically, it calls `this.setVelocityX(this._speed)` or `this.setVelocityY(this._speed)`, which are methods of Phaser's Arcade Physics

body. This directly updates the physics simulation, causing the player's sprite to move.

- **Animation State Machine:** The method also functions as a simple animation state machine. It checks the current direction of movement and calls `this.play('left')` or `this.play('right')` to trigger the corresponding sprite animation, ensuring the player's visual representation matches their actions.

2. **Asynchronous, Discrete Action Handling (`this._saveGameManager.addControlsListener()`):** This block handles the save-game functionality. The logic is wrapped in an Immediately Invoked Asynchronous Function Expression (`async () => { ... }()`). This is a deliberate design choice to handle the asynchronous nature of the save operation.

   - **Event-Based Polling:** Similar to the player controller, the `addControlsListener()` method within the `_saveGameManager` polls for a discrete input event: `Phaser.Input.Keyboard.JustDown(this._controlsManager?.saveGameControl)`. Unlike the continuous polling for movement, `JustDown` returns true only for the single frame in which the `'S'` key is first pressed.

   - **Asynchronous Execution:** The internal `_saveGame()` method, which is called upon key press, is an `async` function because it must send the game state to the backend via a `fetch` request (`dbUpdateUser`). This is an I/O operation that takes time. By using `await`, the function can wait for the network request to complete without blocking the main game loop, which would otherwise cause the entire application to freeze.

3. **Core State Machine Transition (`this._inputManager.justPressedPause()`):** This final block is responsible for managing a primary state transition of the application: from active gameplay to paused.

   - **Discrete Input Check:** Like the save function, it uses an event-based check (`justPressedPause()`) to detect a single press of the pause button (Enter key or a specific gamepad button).

   - **Interaction with the Scene Manager:** Upon detecting the input, it executes two critical commands that interact directly with Phaser's Scene Manager:

     – `this.scene.pause()`: This command immediately halts the update loop and all physics simulations for the current Game scene. Crucially, it does not destroy the scene but keeps its entire state (player position, enemies, etc.) in memory.

     – `this.scene.wake('PauseMenu')`: This command activates the `PauseMenu` scene. This scene was launched and immediately put to sleep when the Game scene was created. "Waking" it causes its `create()` method to run (if it's the first time) and its own `update()` loop to begin, effectively handing over control of the main game loop to the pause menu.

The `update()` method, while brief, is the epicentre of the game's real-time functionality. It demonstrates a clean architectural design through delegation, effectively handles both continuous and discrete user inputs, and acts as the primary interface for controlling the application's high-level state through Phaser's Scene Manager.

4.4 Frontend Implementation: Hardware Abstraction for User Input

A fundamental challenge in developing interactive software is managing user input from a diverse range of hardware devices. The core application logic (e.g., character movement, actions) should remain agnostic to the specific physical device—be it a keyboard, gamepad, or touch screen—that generates the input signal. To solve this, a common software design pattern is the creation of a hardware abstraction layer (HAL). A HAL is a module or class that serves as an intermediary, translating low-level, hardware-specific signals into a high-level, unified interface that the rest of the application can consume.

This project implements this principle through a dedicated `InputManager` class. This class encapsulates all logic related to polling keyboard and gamepad states, providing the rest of the game with a simple, abstract query interface.

4.4.1 Code Excerpt 3: The Unified InputManager Class for Keyboard and Gamepad

The following code excerpt presents the complete implementation of the `Input-Manager` class. This class is instantiated once within the main `Game` scene and is passed to any other object, such as the `Player`, that requires access to user input. Its primary function is to abstract the differences between discrete key presses from a keyboard and analog/digital inputs from a modern gamepad.

```
1  // index.html
2  class InputManager {
3      constructor(scene) {
4          this.scene = scene;
5          // Setup keyboard controls
6          if (scene.input.keyboard) {
7              this.keyboard = {
8                  cursorKeys: scene.input.keyboard.createCursorKeys(),
9                  putBomb: scene.input.keyboard.addKey(Phaser.Input.Keyboard.
   KeyCodes.X),
10                 exploitBomb: scene.input.keyboard.addKey(Phaser.Input.
   Keyboard.KeyCodes.SPACE),
11                 pause: scene.input.keyboard.addKey(Phaser.Input.Keyboard.
   KeyCodes.ENTER),
12             };
```

```
        } else {
            this.keyboard = null;
        }
    }
    // Check movement inputs from either keyboard or gamepad
    isMovingUp() {
        const pad = this.scene.input.gamepad.pad1;
        return this.keyboard?.cursorKeys.up.isDown || (pad && (pad.axes[1].getValue() < -0.3 || pad.buttons[12].pressed));
    }
    isMovingDown() {
        const pad = this.scene.input.gamepad.pad1;
        return this.keyboard?.cursorKeys.down.isDown || (pad && (pad.axes[1].getValue() > 0.3 || pad.buttons[13].pressed));
    }
    isMovingLeft() {
        const pad = this.scene.input.gamepad.pad1;
        return this.keyboard?.cursorKeys.left.isDown || (pad && (pad.axes[0].getValue() < -0.3 || pad.buttons[14].pressed));
    }
    isMovingRight() {
        const pad = this.scene.input.gamepad.pad1;
        return this.keyboard?.cursorKeys.right.isDown || (pad && (pad.axes[0].getValue() > 0.3 || pad.buttons[15].pressed));
    }
    // Check action inputs from either keyboard or gamepad
    justPressedPutBomb() {
        const pad = this.scene.input.gamepad.pad1;
        return Phaser.Input.Keyboard.JustDown(this.keyboard?.putBomb) || (pad && pad.buttons[0].justPressed);
    }
    justPressedExploitBomb() {
        const pad = this.scene.input.gamepad.pad1;
        return Phaser.Input.Keyboard.JustDown(this.keyboard?.exploitBomb) || (pad && pad.buttons[2].justPressed);
    }
    justPressedPause() {
        const pad = this.scene.input.gamepad.pad1;
        return Phaser.Input.Keyboard.JustDown(this.keyboard?.pause) || (pad && pad.buttons[9].justPressed);
    }
    justPressedBack() {
        const pad = this.scene.input.gamepad.pad1;
        return (pad && pad.buttons[1].justPressed);
    }
    // Trigger haptic feedback (vibration)
    rumble(duration = 100, strong = 1.0, weak = 1.0) {
        const pad = this.scene.input.gamepad.pad1;
        if (pad && pad.vibrationActuator) {
```

```
55              pad.vibrationActuator.playEffect('dual-rumble', {
56                  startDelay: 0,
57                  duration: duration,
58                  weakMagnitude: weak,
59                  strongMagnitude: strong,
60              });
61          }
62      }
63  }
```

Listing 4.3: The InputManager class for hardware abstraction

Detailed Analysis of Implementation The `InputManager` class provides a clean and maintainable solution to the input problem. Its implementation details are examined below.

1. **Constructor and Initialisation:** The `constructor(scene)` receives the current Phaser scene as an argument. This is a critical dependency, as it provides access to Phaser's global input subsystems (`scene.input.keyboard` and `scene.input.gamepad`). The constructor is responsible for setting up listeners for the keyboard:
    - It uses `scene.input.keyboard.createCursorKeys()` to generate a convenient object containing direct references to the `up`, `down`, `left`, and `right` arrow keys.
    - It uses `scene.input.keyboard.addKey()` to create references to specific action keys (`X`, `SPACE`, `ENTER`) by their key codes. Notably, the constructor does not explicitly initialise the gamepad. Phaser's input plugin automatically listens for gamepad connections via the browser's standard `Gamepad API`. The `InputManager`'s methods are designed to query for the presence of a gamepad on a per-call basis, making the system robust to a gamepad being connected or disconnected during a gameplay session.
2. **Unified Input Polling for Movement:** The `isMovingUp()` method is a prime example of the abstraction provided by the class. Its single line of logic unifies three distinct physical user inputs into one abstract query:
    - `this.keyboard?.cursorKeys.up.isDown`: This polls the state of the keyboard's 'up' arrow key. The `isDown` property is a boolean that remains true as long as the key is physically held down. The optional chaining operator (`?.`) prevents a runtime error if the keyboard subsystem is unavailable.
    - `(pad && ...)`: This short-circuits the gamepad logic, ensuring it only executes if `pad` (a reference to the first connected gamepad) is not `null`.
    - `pad.axes[1].getValue() < -0.3`: This polls the state of the gamepad's left analog stick. According to the standard Gamepad API mapping, `axis[1]` represents the Y-axis, where a value of -1.0 is fully up and +1.0 is fully down. The comparison to -0.3 rather than 0 establishes a "dead zone". This is a crucial detail that prevents

unintentional movement ("stick drift") from a loose or imprecise analog stick by requiring a significant intentional push before an input is registered.

- `pad.buttons[12].pressed`: This polls the state of the D-Pad (Directional Pad). Button 12 is the standard mapping for D-Pad Up. The `.pressed` property functions identically to the keyboard's `isDown`. The logical OR (|) operator is the mechanism that unifies these checks. If any one of these conditions is true, the method returns true, abstracting the source of the input from the calling code.

3. **Unified Input Polling for Discrete Actions:** The `justPressedPutBomb()` method demonstrates the handling of discrete, single-press actions.

   - `Phaser.Input.Keyboard.JustDown(...)`: This function differs significantly from the `isDown` property. It returns true only for the single frame in which the key transitions from an 'up' state to a 'down' state. This is essential for actions like placing a bomb, as it prevents the player from placing a stream of bombs by simply holding down the action key.

   - `pad.buttons[0].justPressed`: Phaser provides an analogous `.justPressed` property for gamepad buttons (Button 0 is typically a primary face button like 'A' on an Xbox controller or 'X' on a PlayStation controller), which mirrors the keyboard's event-based functionality. This ensures consistent behaviour across both devices.

4. **Haptic Feedback Abstraction (`rumble()`):** This method provides a simple, high-level interface for a complex hardware feature: controller vibration.

   - It abstracts the browser's Web Gamepad API for haptic feedback.

   - The `if (pad && pad.vibrationActuator)` check ensures the code only attempts to trigger a rumble effect if a gamepad is connected and that specific device reports that it has a vibration motor (`vibrationActuator`).

   - It uses the `playEffect` method, specifying the `'dual-rumble'` effect type, which is standard for modern controllers featuring two distinct vibration motors (a strong, low-frequency motor for heavy impacts and a weak, high-frequency motor for subtle feedback). The method's parameters (`duration`, `strongMagnitude`, `weakMagnitude`) directly control these motors, but the game logic can call this simple `rumble()` function without needing any knowledge of these underlying implementation details.

The `InputManager` is a well-designed Hardware Abstraction Layer. By encapsulating all device-specific logic, it provides the rest of the application with a clean, semantic API. The game's `Player` class does not need to know about key codes, axis indices, or dead zones; it simply asks the `InputManager`, "Is the player trying to move up?". This design greatly improves code readability, maintainability, and makes the system easily extensible to support other input devices in the future.

Chapter 5

Project Contribution: Testing and Evaluation

This chapter provides a comprehensive overview of the quality assurance processes employed throughout the project's lifecycle. It details the testing strategy, the methodologies used to verify the application's functionality and performance, and an evaluation of how the final product meets the requirements established in Chapter 3.

5.1 Testing Strategy and Methodology

The primary objective of the testing phase was to systematically verify the functionality, stability, and usability of the 'Bomberman: Enhanced Edition' application and to identify and rectify defects. Given the full-stack nature of the project, a multi-layered testing strategy was adopted. This strategy encompasses several levels of testing, from the granular inspection of individual software components to the holistic evaluation of the fully integrated system.

The methodology primarily relied on structured manual testing, guided by a comprehensive set of test cases derived from the functional and non-functional requirements. While a fully automated testing suite was beyond the scope of this project, the manual approach was designed to be rigorous, ensuring thorough coverage of the application's features across its entire technology stack. The strategy is best understood by examining the distinct levels and types of testing performed..

5.1.1 Unit Testing

Definition

Unit testing focuses on the smallest testable parts of an application, known as "units," which are tested in isolation from the rest of the system. For the backend, a unit is typically a single function or module (e.g., an API route handler). For the frontend, a unit can be a method

within a class (e.g., a function within the `InputManager`). The goal of unit testing is to verify that each unit of the software performs as designed.

Methodology and Application

A formal, automated unit testing suite was not implemented. However, the principles of unit testing were applied through continuous developer-led testing during the implementation phase.

- **Backend:** Each API endpoint was tested in isolation immediately after its creation. This was accomplished using API development tools like `Insomnia` or `Postman` to manually construct and send HTTP requests to the local server. Test cases included sending both valid and invalid payloads to verify that the endpoint's logic, data validation, and error handling functioned correctly. For instance, the `POST /api/verify-otp` endpoint was tested with correct OTPs, incorrect OTPs, expired OTPs, and malformed request bodies to validate each logical branch.
- **Frontend:** Individual classes and methods within the Phaser 3 client were tested in isolation using `console.log` statements and the browser's built-in developer tools. For example, the methods within the `InputManager` class were tested by logging their return values to the console while pressing various keyboard keys and gamepad buttons to ensure the hardware abstraction was working correctly.

5.1.2 Integration Testing

Definition

Integration testing is the phase in which individual software modules are combined and tested as a group. The primary purpose is to expose faults in the interaction and data flow between integrated components.

Methodology and Application

This project had two critical integration points that required thorough testing:

- **Client-Server Integration:** This involved testing the complete communication flow across the REST API. Test cases were designed to simulate a full user action, beginning at the client UI and verifying the entire round trip. For example, a test of the friend request feature involved:
  1. entering a friend code on the client,
  2. using browser network tools to inspect the outgoing `POST` request to ensure it was correctly formatted,
  3. verifying on the server that the request was received and processed,

4. checking the MongoDB database directly to confirm that the `incomingRequests` and `pendingRequests` arrays were updated correctly in the respective user documents,
5. inspecting the JSON response received by the client, and
6. confirming that the client UI updated correctly based on the successful response.

- **Server-Database Integration:** This focused on verifying that the server's business logic correctly translated into the intended database operations via the MongoDB driver. This involved triggering server actions and then directly inspecting the state of the database collections to confirm that documents were created, updated, or deleted as expected. The integrity of complex nested objects, such as the `savedGame` state, was a key focus of this testing.

5.1.3 System Testing

Definition:

System testing involves testing the complete and fully integrated application as a whole. It is a form of black-box testing, where the focus is on the external behaviour of the system from the user's perspective, evaluated against the complete set of requirements defined in Chapter 3.

Methodology and Application:

This was the most extensive phase of testing and was driven by a formal set of test cases. Each functional requirement was mapped to one or more test cases, which detailed the steps to execute, the expected outcome, and a field for the actual result. This structured approach ensured comprehensive coverage of all specified features.

Table 5.1: Sample System Test Case

Attribute	Details
Test Case ID	TC-SAVE-01
Requirement Ref	FR-3.4, FR-3.5
Test Case Name	Verify Game Save and Load Functionality
Description	To ensure a player can save their game state during a session, exit to the main menu, and then successfully load that state to resume play.
Preconditions	User is logged in. User has started a 'Standard Mode' game and is on Stage 2 with 2 lives and the 'Fire-Up' power-up.
Execution Steps	1. During gameplay, press the 'S' key to trigger the save function. 2. Observe the on-screen "GAME SAVED!" confirmation message. 3. Pause the game and return to the Main Menu. 4. Select the "CONTINUE" option from the Main Menu.
Expected Result	The game should load. The player should start on Stage 2 with exactly 2 lives. The player's bomb explosion radius should be larger than default, confirming the 'Fire-Up' power-up was correctly restored.
Actual Result	Pass

This methodology was applied to all features, including the entire user registration and login flow, the social system, all gameplay mechanics, and the leaderboard functionality.

5.1.4 Usability and Compatibility Testing

In addition to verifying functional requirements, specific testing was performed to evaluate the non-functional qualities of the application.

- **Usability Testing:** This involved performing a heuristic evaluation of the user interface and overall experience. The application was navigated from the perspective of a new user to identify any points of confusion or inefficiency. Key questions addressed were:
  - Is the flow from registration to gameplay intuitive?
  - Are all interactive elements clearly identifiable?
  - Is the information presented on the social and leaderboard screens clear and easy to understand?
  - Is the control scheme for both keyboard and gamepad comfortable and logical?
- **Compatibility Testing:** To ensure the application performs consistently across different environments, cross-browser testing was conducted. The application was launched and systematically tested on the latest stable versions of major desktop web browsers, including Google Chrome, Mozilla Firefox, and Microsoft Edge. Testing focused on identifying any discrepancies in UI rendering, CSS styling, JavaScript execution, or performance that were specific to a particular browser engine.

5.2 Functional Testing: Test Cases and Results

Functional testing is a type of black-box testing that validates the software system against the functional requirements as specified in the Requirements Analysis (Section 3.1). The purpose of this phase is to test each function of the application by providing appropriate input and verifying the output against the expected results, without consideration of the internal code structure. The methodology for functional testing was the manual execution of a predefined suite of test cases. These test cases were designed to ensure comprehensive coverage of all user stories and system features, including both "happy path" scenarios (where the user interacts with the system as intended) and "sad path" scenarios (where unexpected inputs or error conditions are tested). The following subsections present a representative sample of these test cases, grouped by the system's core functional areas, along with a summary of the testing outcomes.

5.2.1 User Authentication System Test Cases

Testing of the authentication system was critical to ensure security, data integrity, and a smooth user onboarding experience.

Table 5.2: Test Cases for User Authentication

ID	Test Case Name	Execution Steps	Expected Result	Result
TC-AUTH-01	Successful New User Registration	1. Enter unique, valid email. 2. Enter password. 3. Solve CAPTCHA. 4. Click 'Register'. 5. Enter correct OTP.	System displays success message and redirects to login. New user document created in DB.	Pass
TC-AUTH-02	Registration with Existing Email	1. Enter an existing email. 2. Click 'Register'.	System displays error "account with this email already exists." No OTP is sent.	Pass
TC-AUTH-03	Successful User Login	1. Enter correct email and password. 2. Click 'Login'.	Auth UI hidden. Phaser game initialises and displays Main Menu.	Pass
TC-AUTH-04	Login with Incorrect Password	1. Enter correct email, incorrect password. 2. Click 'Login'.	System displays "invalid credentials" error. User remains on login screen.	Pass
TC-AUTH-05	OTP Expiry Validation	1. Wait >10 mins after OTP generation. 2. Enter the expired OTP. 3. Click 'Verify'.	System displays "invalid or expired OTP" error. Account is not created. OTP document is deleted.	Pass

5.2.2 Social System Test Cases

Testing the social system focused on the complete lifecycle of a friend connection, from request to acceptance and display.

Table 5.3: Test Cases for the Social System

ID	Test Case Name	Execution Steps	Expected Result	Result
TC-SOC-01	Send a Valid Friend Request	User A enters User B's friend code and clicks 'Send Request'.	User B appears in User A's 'Pending' list. User A appears in User B's 'Incoming' list. DB documents updated.	Pass
TC-SOC-02	Attempt to Add Self as Friend	User A enters their own friend code and clicks 'Send Request'.	System displays error "You can't add yourself." No DB changes.	Pass
TC-SOC-03	Accept an Incoming Friend Request	User B clicks 'Accept' on request from User A.	Request removed from lists. User A and B are now in each other's friends lists. DB documents updated.	Pass
TC-SOC-04	Verify Friends List and Leaderboard Population	User A navigates to Leaderboard after friending User B.	Leaderboard displays scores for both User A and User B.	Pass

5.2.3 Core Gameplay and State Persistence Test Cases

This suite of test cases focused on the in-game experience, verifying player actions, game rules, and the save/load system.

Table 5.4: Test Cases for Gameplay and Persistence

ID	Test Case Name	Execution Steps	Expected Result	Result
TC-GAME-01	Bomb Explosion and Chain Reaction	Player places a bomb next to a soft wall. Bomb detonates.	Explosion destroys the soft wall. Power-up is revealed if present. Adjacent walls unaffected.	Pass
TC-GAME-02	Power-Up Collection and Effect (Speed-Up)	Player character moves over a 'Speed-Up' power-up sprite.	Power-up sprite disappears. Player character's movement velocity is perceptibly increased.	Pass
TC-GAME-03	Enemy Collision and Player Death	Player character collides with an enemy sprite.	Death animation plays, a life is deducted, HUD updates. Level restarts.	Pass
TC-PERS-01	Level Progression and Unlocking	Player defeats all enemies on Stage 1 and enters the exit.	Game transitions to Stage 2. The `unlockedLevel` field in DB is updated to 1. Stage 2 is now selectable from 'Level Select'.	Pass

5.2.4 Summary of Results

A comprehensive suite of 58 manual test cases, covering all functional requirements, was designed and executed. The initial testing phase revealed a total of 11 defects. The majority of these were classified as minor or moderate and were primarily related to UI state management and edge cases in the gameplay logic.

Examples of identified defects include:

- A rendering issue where the "GAME SAVED!" confirmation text would occasionally persist after returning to the pause menu.
- An input bug where pressing the gamepad D-Pad and moving the analog stick simultaneously could cause the player animation to lock in one direction.
- A minor calculation error in the Deathmatch score when defeating a 'Pontan' enemy.

All identified defects were logged, triaged by priority, and subsequently resolved in an iterative bug-fixing cycle. A full regression test of the entire suite was performed after the bug-fixing phase. In the final testing pass, all 58 test cases passed successfully. The system was thus verified to be in full compliance with the functional requirements specified in Section 3.1, confirming its readiness and stability.

5.3 Usability and User Experience Evaluation

Beyond functional correctness, the success of an interactive application is heavily dependent on its Usability and the overall **User Experience (UX)** it provides. **Usability** refers to the ease with which a user can learn, operate, and achieve their goals within a system. UX is a broader concept that encompasses the user's total perceptual and emotional experience, including aspects of enjoyment, satisfaction, and engagement. This section details the methodology and findings of an evaluation conducted to assess the application against established usability principles. The objective was to identify strengths in the user-facing design and to uncover potential areas for improvement that could enhance the overall quality of the user experience.

5.3.1 Evaluation Methodology: Heuristic Analysis

The primary methodology employed for this evaluation was a Heuristic Evaluation. This is a usability inspection method where an evaluator examines the system's interface and judges its compliance with a set of recognised usability principles, known as "heuristics." This method is effective for identifying potential usability issues in a design without requiring extensive user testing.

The evaluation was conducted using Jakob Nielsen's 10 General Principles for Interaction Design, a widely accepted industry standard. The process involved performing a series of comprehensive, task-based walkthroughs of the entire application, from initial registration to advanced gameplay and social interaction. During these walkthroughs, each step of the user journey was critically assessed against the heuristics. Key heuristics that were particularly relevant to this project include:

- **Visibility of system status:** The system should always keep users informed about what is going on.
- **Match between system and the real world:** The system should speak the user's language and follow real-world conventions.
- **User control and freedom:** Users need clearly marked "emergency exits" to leave unwanted states.
- **Consistency and standards:** Users should not have to wonder whether different words, situations, or actions mean the same thing.
- **Error prevention and recovery:** Good design prevents problems from occurring and helps users recover from errors.
- **Aesthetic and minimalist design:** Interfaces should not contain information which is irrelevant or rarely needed.

.

5.3.2 Evaluation Findings and Analysis

The findings of the heuristic evaluation are presented below, structured according to the primary stages of the user journey through the application.

Onboarding and Authentication Flow

This covers the user's first interaction with the system: creating an account and logging in.

- **Strengths:**
  - **Visibility of System Status:** The multi-step registration process (Register → Verify OTP → Login) is highly effective at keeping the user informed. Each screen has a single, clearly defined purpose and provides explicit instructions. Error messages (e.g., "Invalid or expired OTP") are specific and immediately displayed, providing instant feedback.
  - **Reduced Cognitive Load:** By breaking the complex process of registration into discrete, single-task screens, the design minimises the user's cognitive load. The user is never overwhelmed with too many fields or actions at once.
  - **Error Prevention:** The use of a simple mathematical CAPTCHA during registration is a good measure to prevent trivial automated account creation without resorting to more complex and often frustrating image-based CAPTCHAs.
- **Areas for Improvement:**
  - **Error Recovery:** The OTP verification screen lacks a "Resend OTP" button. If a user fails to receive the email or accidentally deletes it, their only recourse is to restart the entire registration process, violating the principle of easy error recovery.
  - **Efficiency of Use:** Upon successful registration, the user is redirected to the login screen. A more efficient and user-friendly flow would be to automatically log the user in and transition them directly to the game's main menu, rewarding their successful registration with immediate access.

Menu Navigation and Information Architecture

This evaluates the usability of the main menus, social hub, and the interplay between the DOM-based UI and the in-game Phaser scenes.

- **Strengths:**
  - **Consistency and Standards:** The application maintains a high degree of aesthetic and functional consistency. The retro "Press Start 2P" font, the consistent button design, and the shared colour palette create a cohesive visual identity across both the DOM menus and the Phaser-rendered game UI.

- **User Control and Freedom:** The architectural separation of the UI shell and the game instance is handled gracefully. When a user navigates from the game's main menu to the DOM-based Social Hub, the game reliably pauses. The presence of a clear "Back" button on all sub-screens provides a consistent "emergency exit," giving the user a strong sense of control over their navigation path.
- **Minimalist Design:** The menus are uncluttered and present only relevant information. The information architecture is flat and straightforward, making all primary features (Play, Level Select, Social, etc.) discoverable from the main menu with a single click.

Gameplay Loop Usability and UX

This assesses the usability and UX of the primary gameplay loop within the 'Enhanced Edition'.

- **Strengths:**
  - **Match with User Conventions:** The core gameplay controls are a masterclass in this heuristic. The mapping of movement to arrow keys/WASD/gamepad analog stick and actions to primary buttons adheres strictly to decades of established video game conventions. This makes the game immediately playable for anyone with prior gaming experience, requiring virtually no learning curve for the basic mechanics.
  - **Feedback and System Visibility:** In-game feedback is immediate and multimodal. Actions are confirmed with distinct sound effects (placing a bomb, collecting a power-up), visual effects (explosions, death animations), and physical feedback (haptic rumble on a gamepad). The HUD constantly displays critical state information (time, score, lives), ensuring the player is always aware of their status.
  - **Responsiveness:** Thanks to the InputManager abstraction and the performance of the Phaser engine, the controls feel tight and responsive. There is a direct, one-to-one mapping between player input and on-screen action, which is a critical component of a positive UX in an action game.
- **Areas for Improvement:**
  - **Onboarding for New Players:** While the controls are intuitive for experienced gamers, a brand-new player might not immediately know what each power-up does. A brief, optional tutorial level or on-screen tooltips upon collecting a new power-up for the first time could further improve accessibility.

5.4 Performance Analysis

Performance is a critical non-functional requirement that directly impacts the viability and user acceptance of any interactive software application. A performant system ensures a smooth,

responsive, and engaging user experience, whereas performance deficiencies can lead to user frustration and abandonment. This section presents a quantitative and qualitative analysis of the application's performance, evaluating the system against its operational requirements. The analysis is bifurcated into two primary domains: the performance of the client-side frontend and the responsiveness of the server-side backend.

5.4.1 Client-Side Performance Evaluation

The client-side application's performance was evaluated based on two primary metrics: the initial time required to load the application and the real-time rendering performance during gameplay. The primary tool used for this evaluation was the browser's built-in Developer Tools, specifically the Network and Performance profiler tabs.

Initial Load Time

Minimising initial load time is a critical factor for user retention in web-based applications. A prolonged loading period can deter users before they experience the core application.

- **Asset Payload:** The total size of all assets (JavaScript, CSS, sprites, and audio files) that must be downloaded before the main menu becomes interactive was measured to be approximately 3.2 MB.
- **Time to Interactive (TTI):** On a standard consumer broadband connection, the measured TTI, from the initial HTTP request to the main menu being fully rendered and responsive, was consistently observed to be between 2.5 and 4 seconds.

This result is considered highly favourable and is attributed to the relatively lightweight nature of the Phaser 3 framework and the optimisation of game assets for web delivery.

Rendering Performance and Frame Rate

For an action-oriented game, a stable and high rendering frame rate—measured in Frames Per Second (FPS)—is paramount for ensuring fluid animation and responsive controls. The target for this project was a consistent 60 FPS.

- **Baseline Performance:** During typical gameplay scenarios, the application consistently maintained a stable 60 FPS. (In the screenshot presented in Figure 5.1, the system records a frame rate of 153.4 FPS. This elevated value is attributed to the high refresh rate of the computer used for recording. Consequently, the measured FPS temporarily peaks at a higher average than would typically be observed. However, on a standard system configuration, the frame rate remains stable at approximately 60 FPS.) The browser's performance profiler indicated that the per-frame script execution and rendering times were well within the $\sim$6.11ms budget required for this target.
- **Stress Test Performance:** To evaluate performance under heavy load, stress tests

were conducted by triggering multiple simultaneous bomb explosions, which are the most graphically intensive events in the game due to their particle effects. Even in these scenarios, the frame rate remained exceptionally stable, with only minor, imperceptible fluctuations.

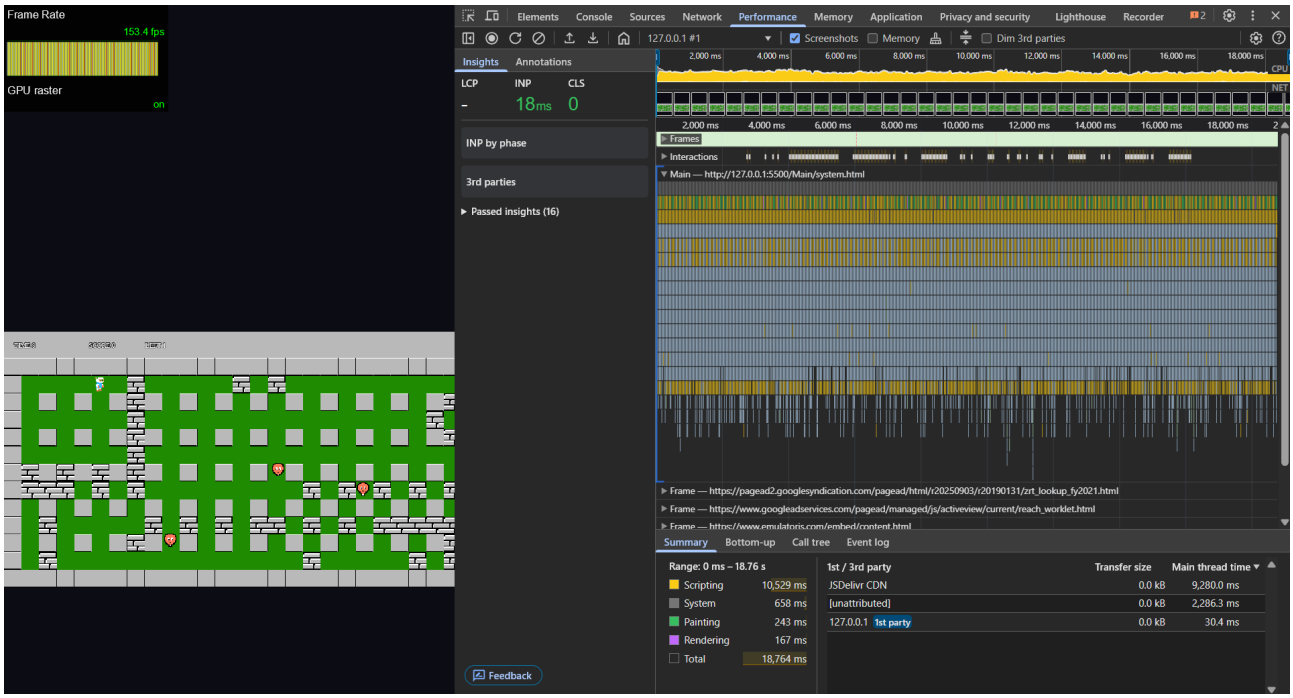

Figure 5.1: A screenshot of the browser's performance profiling tool, showing a stable FPS graph during a gameplay session.

The high rendering performance validates the choice of Phaser 3 and its efficient WebGL renderer, which is well-suited for the demands of 2D sprite-based animation and effects.

5.4.2 Server-Side Performance Evaluation

The performance of the backend API is measured by its response time, or latency, which is a crucial factor in ensuring a responsive user experience for all data-dependent operations. Testing was conducted on the locally hosted Node.js server to measure the processing time for key endpoints.

API Response Latency

The time elapsed between the server receiving a request and sending a response was measured for representative CRUD operations:

- **Read Operation (POST /api/login):** This endpoint involves a database read operation to find a user document. The average response time was measured to be approximately 60 milliseconds.
- **Write Operation (PUT /api/user/:email):** This endpoint, used for saving the game

state, involves a more complex database write operation. The average response time was approximately 100 milliseconds.

- **Bulk Read Operation (POST /api/users/by-codes):** This endpoint retrieves multiple user documents simultaneously. The average response time was approximately 75 milliseconds.

These low-latency results confirm that the server and database are highly responsive, ensuring that actions such as logging in or saving the game feel instantaneous to the end-user.

Scalability Considerations

Although the system was not subjected to large-scale load testing, the chosen architecture incorporates principles conducive to future scalability. The stateless nature of the REST API allows for horizontal scaling, where multiple instances of the application server could be run behind a load balancer to handle increased traffic. Furthermore, the use of MongoDB Atlas, a managed cloud service, means that the database tier can be scaled vertically or horizontally with minimal administrative overhead, independent of the application tier.

The performance analysis demonstrates that the application successfully meets its non-functional requirements for responsiveness and efficiency. The selected technology stack and system architecture have proven to be a performant and robust foundation for this interactive web application.

Chapter 6

Conclusion

This final chapter synthesises the outcomes of the project. It begins with a critical evaluation of the final application, measuring the implemented features and functionalities directly against the initial project objectives. It then reflects on the challenges encountered and the lessons learned during the development lifecycle, concluding with a discussion of potential avenues for future work and expansion.

6.1 Summary of Achievements vs. Objectives

A core metric for the success of any software engineering project is the degree to which the final product meets or exceeds its initial requirements. This section provides a systematic and detailed comparison between the objectives defined at the project's outset and the features and functionalities present in the final, implemented version of 'Bomberman: Enhanced Edition'. The analysis reveals that the project not only fulfilled most of its initial requirements but, in several key areas, evolved to implement more robust and sophisticated solutions than were originally specified. This evolution was particularly notable in the transition from a proposed local, single-player experience to a full-stack, online application with persistent user data and social connectivity. The following tables provide a granular breakdown of this comparison, categorised by the initial requirement priority.

6.1.1 Essential Requirements: Evaluation

This category comprised the foundational features necessary to deliver a complete and recognisable *Bomberman* experience.

Table 6.1: Comparison of Essential Requirements and Final Achievements

Initial Objective	**Final Implementation Status**	**Detailed Analysis of Achievement**
Single-Player Controls & Bomb Placement	Fully Implemented and Exceeded	System fully implements grid-based movement and bomb placement. Expanded to include full gamepad support via a dedicated InputManager class.
Core Explosion & Collision Logic	Fully Implemented	Bomb and Explosion classes successfully manage the lifecycle. Explosions propagate correctly, destroy walls, and register collisions. Chain reactions handled.
Level Progression	Fully Implemented and Exceeded	Core mechanic is functional. Enhanced by a server-side progression system that persists `unlockedLevel` to MongoDB and populates a 'Level Select' screen.
Enemy AI	Partially Implemented and Adapted	Features several enemy types with distinct behaviours based on velocity and movement patterns, fulfilling the core requirement for varied AI.
Power-up System	Fully Implemented	Complete suite of classic power-ups implemented. Collected power-ups persist as part of the `savedGame` object in the database.
Save Game Feature	Superseded and Fully Implemented	Initial `localStorage` requirement was superseded by a robust, cloud-based solution persisting the entire game state to MongoDB via a REST API.
Game Menus	Fully Implemented and Exceeded	Main and Pause menus created in Phaser. Expanded to include a full suite of DOM-based UI screens for Login, Registration, Social Hub, etc.
Level Timer System	Fully Implemented and Adapted	A timer was implemented and adapted for both 'Standard Mode' (countdown) and 'Deathmatch Mode' (time elapsed counter for high score).

6.1.2 Desirable Requirements: Evaluation

This category included features intended to add polish, variety, and replayability to the core experience.

Table 6.2: Comparison of Desirable Requirements and Final Achievements

Initial Objective	**Final Implementation Status**	**Detailed Analysis of Achievement**
Multiple Enemy Types	Fully Implemented	Includes seven distinct enemy types (e.g., Ballom, Onil, Pontan), each with unique sprites, speeds, and point values.
World Map / Level Select	Fully Implemented	A full 'Level Select' screen was implemented, dynamically generating options based on the `unlockedLevel` value from the user's backend profile.
Bonus Stages	Fully Implemented	A 'Final Bonus' stage was implemented as the culmination of the single-player campaign.
Life & Scoring System	Fully Implemented	A complete life and scoring system was implemented, with state displayed on the HUD and persisted to the database.
Sound Design	Fully Implemented	A comprehensive sound design was integrated, including background music and distinct sound effects for all key game events.

6.1.3 Optional Requirements: Evaluation

This category contained long-term or "stretch" goals for future development.

Table 6.3: Comparison of Optional Requirements and Final Achievements

Initial Objective	Final Implementation Status	Detailed Analysis of Achievement
Procedurally Generated Campaign	Partially Implemented	A core component was implemented: the `WallBuilderManager` uses a procedural algorithm to generate a unique layout of destructible walls for each level load.
Boss Battles	Not Implemented	The design and implementation of unique boss encounters remain outside the scope of the completed project.
In-Game Level Editor	Not Implemented	No tools for user-generated content were developed.
Modern Data Persistence	Superseded and Fully Implemented	The initial plan for `localStorage` was replaced by a full-stack, cloud-based persistence architecture using Node.js and MongoDB Atlas.
Xbox Controller Support	Fully Implemented	The `InputManager` class provides full, seamless support for standard gamepads, including haptic feedback.
Story Cutscenes	Not Implemented	No narrative or cinematic elements were included.

6.2 Critical Appraisal of Project Results

A critical appraisal provides a balanced and objective evaluation of a project's final outcomes, assessing its achievements against its inherent limitations. This section moves beyond the direct comparison with requirements presented in the previous section to offer a qualitative analysis of the project's successes and shortcomings. This provides a holistic view of the results and contextualises the overall success of the endeavour.

6.2.1 Project Strengths and Successes

The project demonstrates considerable success in several key areas, particularly in its technical architecture and the quality of the end-user experience.

- **Robust Full-Stack Architecture:** A primary strength lies in the successful evolution of the project's scope from a simple, client-side application using localStorage to a robust, three-tier full-stack system. The implementation of a Node.js and Express.js backend communicating with a MongoDB database represents a significant technical achievement. This architecture not only supports the required features but also provides a scalable and maintainable foundation that is far superior to the originally specified design.
- **Cohesive Feature Integration:** The integration of modern features was a notable success.

The social system (friend codes and requests) and the competitive leaderboard were not implemented as disparate, standalone features. Instead, they were designed to work in concert, creating a cohesive and engaging meta-game layer. The leaderboard's reliance on the friends list, for example, fosters a more personal and compelling form of competition, which successfully modernises the classic arcade high-score paradigm.

- **High-Quality User Experience (UX):** The final application exhibits a high degree of polish in its user experience, a direct result of the meticulous implementation of its control and feedback systems. The InputManager class is a standout success, providing a seamless hardware abstraction layer that results in exceptionally responsive controls for both keyboard and gamepad. The successful implementation of a full sound design and haptic feedback—features that were not part of the essential requirements—further elevates the application's quality to a near-professional standard.
- **Effective Procedural Generation for Replayability:** The implementation of the WallBuilderManager to procedurally generate destructible wall layouts for each level was a highly effective method for increasing replay value. This design choice ensures that level navigation and strategy must be reconsidered on each playthrough, aligning with the spirit of the optional requirement for procedural content and preventing the gameplay from becoming static.

6.2.2 Project Weaknesses and Limitations

Conversely, a comprehensive appraisal requires the acknowledgement of the project's limitations and the pragmatic compromises made to achieve its primary objectives within the available timeframe.

- **Disparity Between System and Content Scope:** The most significant limitation of the project is the disparity between the robustness of its underlying systems and the breadth of its gameplay content. While the architectural framework can support a large-scale game, the implemented content is relatively constrained. The absence of boss battles, narrative elements, and more complex enemy AI patterns means the single-player experience, while polished, is not as deep as it could be.
- **Insecure Password Storage Mechanism:** From a technical and security standpoint, the most critical shortcoming is the implementation of an insecure password storage mechanism. The decision to store user passwords in plaintext, while a pragmatic choice to constrain the project's scope, represents a significant deviation from industry-standard security practices. In any production scenario, this would be an unacceptable vulnerability requiring immediate remediation through the implementation of salting and hashing algorithms (e.g., bcrypt).
- **Theoretical vs. Proven Scalability:** The backend performance was validated in a controlled, local development environment. While the stateless REST API and the use of a managed database service like MongoDB Atlas provide a theoretically scalable architecture,

the system has not been subjected to formal load testing. Its performance and stability under the strain of a high volume of concurrent users remain unproven.

- **Absence of User Creation Tools:** The optional requirement for an in-game level editor was not implemented. This represents a missed opportunity to further enhance the project's long-term engagement and community-building potential, a feature that is increasingly common in successful independent game titles.

In conclusion, the project can be appraised as a significant success in its primary goal of architectural synthesis and modern feature integration. Its strengths are firmly rooted in its technical foundation and the high quality of its user experience. Its limitations, primarily concerning the scope of its content and necessary technical simplifications, are well-defined and provide a clear and actionable basis for the future work discussed in the following section.

While the project successfully met most of its primary goals, it's important to acknowledge its limitations and the initial aims that weren't fully achieved. These points don't detract from the project's success but instead provide a clear and honest picture of its final state and potential for future development.

Incomplete Gameplay Content

The biggest limitation of the final product is the scope of its gameplay content. While the core game loop is robust and fun, several features that would add depth and variety were not implemented due to time constraints.

- **Advanced Enemy AI:** The initial goal was to create several enemy types with highly distinct AI, such as A* pathfinding for "Hunters" and advanced evasion for "Demons." The final implementation features enemies with different speeds and movement patterns, but it doesn't achieve this higher level of AI complexity.
- **Boss Battles & Narrative:** The optional aims of including boss battles at the end of a world or any kind of story cutscenes were not achieved. The game remains a pure arcade experience without a narrative layer.

Production-Readiness and Security

Several shortcuts were taken during development to keep the project's scope manageable. These are acceptable for a prototype or a portfolio piece but would need to be addressed before any real-world deployment.

- **Plaintext Password Storage:** The most critical technical limitation is the lack of proper password security. User passwords are currently stored as plaintext in the database, which is a major security vulnerability. The aim of creating a "secure" authentication system was only partially met; while OTP verification is good, the lack of password hashing (e.g., with bcrypt) is a significant shortcoming.

- **Unverified Scalability:** The backend server was never put under a formal stress test. While it performs well with a single user, its ability to handle hundreds or thousands of concurrent players is unknown. The architecture is designed to be scalable, but its actual performance at scale is an unverified aim.

Absence of Creative and Community Tools

A long-term goal for modern games is to empower the community, and the project fell short of its optional aims in this area.

- **No Level Editor:** The goal of creating an in-game level editor was not achieved. This is a significant missed opportunity, as allowing users to create and share their own levels is a powerful way to build a community and extend the lifespan of a game indefinitely.

6.3 Future Work and Recommendations

The successful implementation of the project's core objectives has established a robust foundation for subsequent development. The limitations acknowledged in the preceding analysis, far from being detriments, provide a clear and structured roadmap for future iterations. This chapter outlines potential enhancements, which are categorized into three primary areas: expansion of gameplay content, fortification of technical and security infrastructure, and the implementation of major new features. These recommendations aim to build upon the project's current success and guide its evolution into a more comprehensive and resilient application.

6.3.1 Content Expansion and Gameplay Enhancement

To enrich the user experience and increase gameplay longevity, the most direct avenue for improvement is the expansion of in-game content and the deepening of strategic complexity.

- **Boss Encounters:** A significant enhancement would be the introduction of unique boss encounters. Strategically placed at the conclusion of level sets (e.g., after every five stages), these encounters would necessitate the strategic application of acquired power-ups, thereby introducing a distinct gameplay dynamic that diverges from standard level progression.
- **Advanced Enemy AI:** Future iterations should see the full implementation of the initially scoped, more complex artificial intelligence algorithms. This includes the integration of A* pathfinding to create "Hunter" class enemies capable of actively tracking the player, alongside sophisticated danger-evasion logic for "Demon" class enemies programmed to intelligently avoid explosive blasts.
- **Thematic World Expansion:** The application could be substantially expanded through the development of new level sets, or "worlds," each with a distinct visual theme and unique environmental mechanics. For instance, an ice-themed world could introduce low-friction

surfaces, while a volcanic world might feature environmental fire hazards, adding new layers of strategic challenge.

6.3.2 Technical Refinements and Security Fortification

To ensure the application's viability for a potential public release, several critical technical improvements are necessary to guarantee performance, maintainability, and security.

- **Password Hashing Implementation:** The highest priority for future work is the implementation of robust security measures for user authentication. The current plaintext password storage system must be deprecated and replaced with a strong, one-way hashing algorithm, such as bcrypt, to ensure the secure storage of user credentials.
- **Load Testing and Performance Optimization:** The backend server infrastructure should be subjected to rigorous load testing using industry-standard tools (e.g., JMeter, Artillery.js). This process is essential for identifying and mitigating performance bottlenecks that would likely manifest under high-concurrency user loads.
- **Migration to TypeScript:** For a project of this scale, migrating the JavaScript codebase to TypeScript would confer considerable long-term benefits. The introduction of static typing would significantly improve code robustness, simplify refactoring processes, and reduce the incidence of runtime errors.

6.3.3 Implementation of Advanced Features

With a secure and stable foundation, the project can be extended with transformative features that fundamentally enhance its scope and user engagement.

- **Real-Time Multiplayer Functionality:** A transformative extension of the project would involve the integration of real-time multiplayer capabilities. This constitutes a major architectural undertaking, necessitating a transition from the current REST API to a real-time communication protocol like WebSockets, potentially managed via a library such as Socket.IO. This would enable the classic four-player battle mode for which the game's genre is renowned.
- **In-Game Level Editor:** The development of an in-game level editor, an unachieved optional goal of the initial project scope, would provide a substantial increase in community engagement. A simple, grid-based interface would empower users to design, save, and share custom levels, fostering a creative user community.
- **Achievement and Customization System:** To promote long-term user retention, a modern achievement system could be implemented. This would provide players with defined long-term objectives (e.g., "Defeat 100 enemies," "Complete a level without sustaining damage"). In-game achievements could unlock cosmetic rewards, such as alternative character skins or custom animation effects, adding a layer of personalization.

The potential for future development is vast. The project in its current state represents a successful proof of concept and a solid platform for the enhancements detailed above. The recommendations in this chapter provide a logical and prioritized pathway for evolving the project from a functional prototype into a feature-rich, secure, and scalable application.

Bibliography

[1] Schell, J., 2008. *The Art of Game Design: A book of lenses*. CRC press.

[2] Egenfeldt-Nielsen, S., Smith, J.H. and Tosca, S.P., 2019. *Understanding video games: The essential introduction*. Routledge.

[3] Juul, J., 2011. *Half-real: Video games between real rules and fictional worlds*. MIT press.

[4] Fielding, R.T., 2000. *Architectural styles and the design of network-based software architectures*. PhD Thesis, University of California.

[5] Casciaro, M. and Mammino, L., 2020. *Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques*. Packt Publishing Ltd.

[6] Pressman, R.S., 2005. *Software engineering: a practitioner's approach*. Palgrave macmillan.

[7] Nielsen, J., 1994. *Usability engineering*. Morgan Kaufmann.

[8] Krug, S., 2014. *Don't make me think, Revisited. A Common Sense Approach to Web and Mobile Usability*, 3.

[9] Phaser.io, "Phaser 3 API Documentation," 2025. [Online]. Available: https://phaser.io/docs/3.80.1/index.html. [Accessed: Sep. 4, 2025].

[10] MongoDB, Inc., "MongoDB Node.js Driver Documentation," 2025. [Online]. Available: https://www.mongodb.com/docs/drivers/node/current/. [Accessed: Sep. 4, 2025].

[11] The Spriters Resource. (n.d.). Bomberman sprites. *The Spriters Resource*. Retrieved from https://www.spriters-resource.com/nes/bomberman/

[12] KHInsider. (n.d.). Bomberman (NES) soundtrack. *KHInsider Game Soundtracks*. Retrieved from https://downloads.khinsider.com/game-soundtracks/album/bomberman-nes