Capstone Project: Ft_Transcendence

Group Members' Full Names:

Calvin Hon
Muhammad Ali Danish
Mahad Abdullah
Nguyen The Hoach

Group Members' Intra Logins:

chon
mdanish
maabdull
honguyen

Date of Submission: February 2, 2026

Abstract

The evolution of online gaming has created a demand for platforms that provide seamless real-time interaction while maintaining high standards for security and data integrity. This report presents **ft_transcendence**, a full-stack multiplayer Pong platform engineered to resolve the performance and security challenges inherent in distributed systems. The system employs a microservices architecture consisting of ten Docker containers — Auth, User, Game, Tournament, Blockchain (Hardhat node), Blockchain-Deploy, Blockchain-Service, Frontend (NGINX), Redis, and Vault — orchestrated via Docker Compose for production-grade deployment.

Methodologically, the project leverages WebSockets for authoritative 60 FPS gameplay, Babylon.js for 3D rendering, and Solidity smart contracts on a Hardhat node to ensure immutable tournament record-keeping. Security is achieved through a ”Defense-in-Depth” strategy, utilizing NGINX with ModSecurity WAF for network protection and HashiCorp Vault for centralized secret management.

Final implementation results confirm 100% compliance with all subject requirements, validated through comprehensive manual testing. This project demonstrates the effective synthesis of modern web technologies and rigorous engineering practices, serving as a secure, high-fidelity foundation for competitive distributed gaming.

Contents

List of Figures

List of Tables

List of Abbreviations

API Application Programming Interface

AI Artificial Intelligence

DB Database

FPS Frames Per Second

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

OWASP Open Web Application Security Project

REST Representational State Transfer

SDLC Software Development Life Cycle

SPA Single-Page Application

SQL Structured Query Language

SQLi SQL Injection

Chapter 1

Introduction

1.1 Project Overview

ft_transcendence is a production-ready, full-stack multiplayer Pong platform designed to deliver real-time competitive gameplay, social features, tournaments with immutable blockchain recording, and comprehensive system observability. The platform accommodates multiple players, with extensible architecture supporting AI opponents and campaign progression.

1.2 Project Objectives

1.2.1 Primary Objectives

1. Implement a server-authoritative Pong game with real-time WebSocket synchronization at 60 FPS
2. Deliver a secure, scalable microservices architecture supporting concurrent multiplayer sessions
3. Provide tournament management with blockchain-based result recording for immutability
4. Ensure production-grade security with WAF, secrets management, and layered defense
5. Support multiple access patterns (web SPA)

1.2.2 Quality Metrics

- **Functional Completeness:** 100% subject compliance
- **Security:** Zero critical vulnerabilities, WAF protection active
- **Code Quality:** TypeScript strictness enabled, consistent standards

Chapter 2

Software Development Life Cycle (SDLC)

2.1 SDLC Approach

The project followed an iterative, incremental SDLC model with five phases:

2.1.1 Planning & Requirements Analysis

- Review official subject requirements document (ft_transcendence v16.1)
- Identify mandatory features, major modules, and minor modules
- Define user stories and acceptance criteria for each feature

2.1.2 Architectural Design

- Design microservices topology: auth, user, game, tournament, blockchain and vault services
- Select technology stack: Fastify + TypeScript + SQLite
- Plan deployment strategy: Docker Compose with reverse proxy (NGINX)
- Define security architecture: WAF, Vault

2.1.3 Implementation (Iterative)

- Develop core services in parallel
- Integrate game logic with real-time WebSocket support
- Implement security features incrementally

2.1.4 Deployment & Evolution

- Containerization and Docker Compose orchestration
- Production deployment and optimization

2.2 SDLC Flowchart

The following flowchart illustrates the Software Development Life Cycle process:

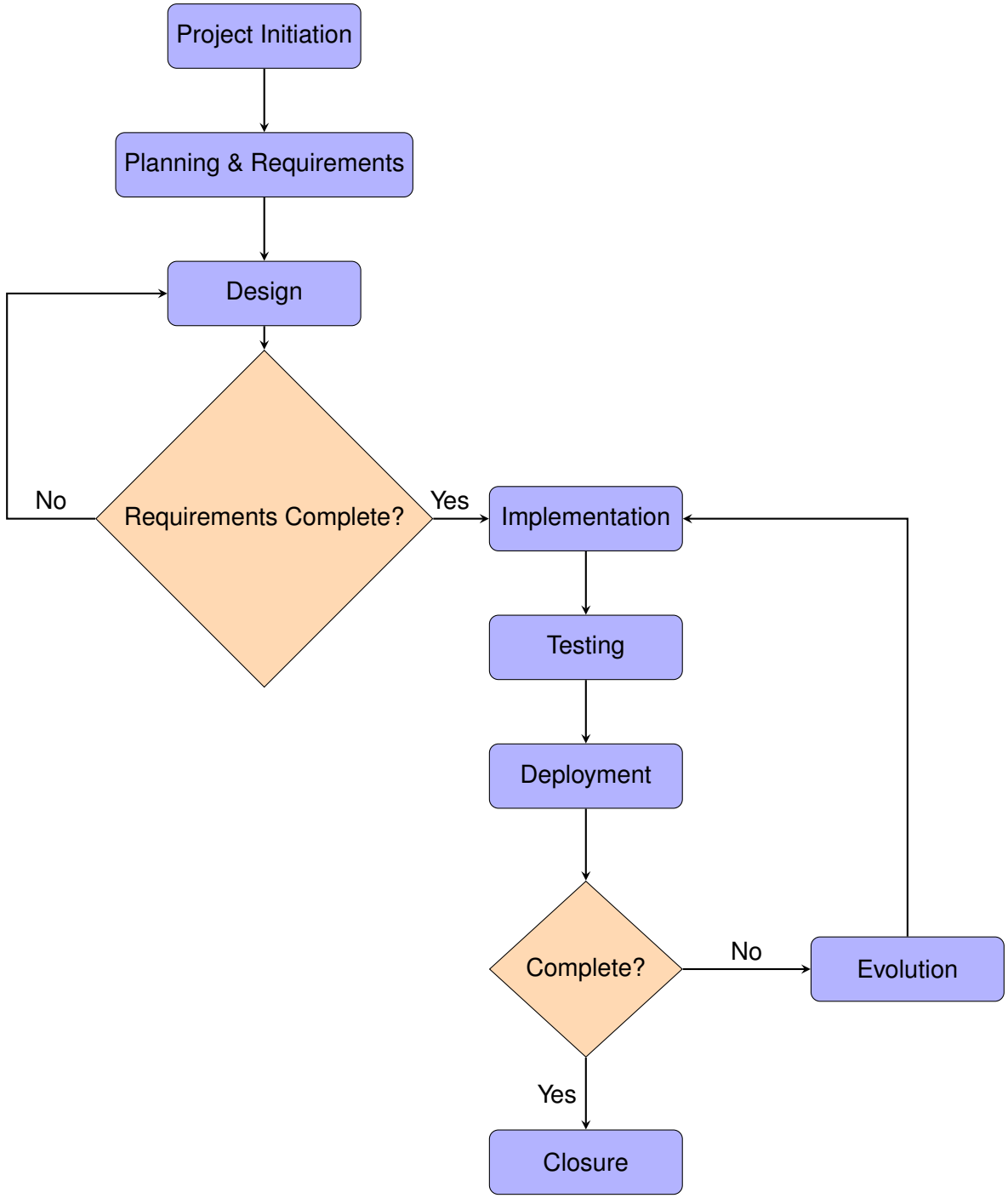

Figure 2.1: SDLC Flowchart: Development process overview

2.3 Project Timeline and Gantt Chart

The project was executed in 4 phases over 14 weeks with assigned leads:

- **Phase 1 (Planning & Design):** 3 weeks
- **Phase 2 (Core Development):** 5 weeks
- **Phase 3 (Security & Blockchain):** 6 weeks
- **Phase 4 (Testing & QA):** 7 weeks

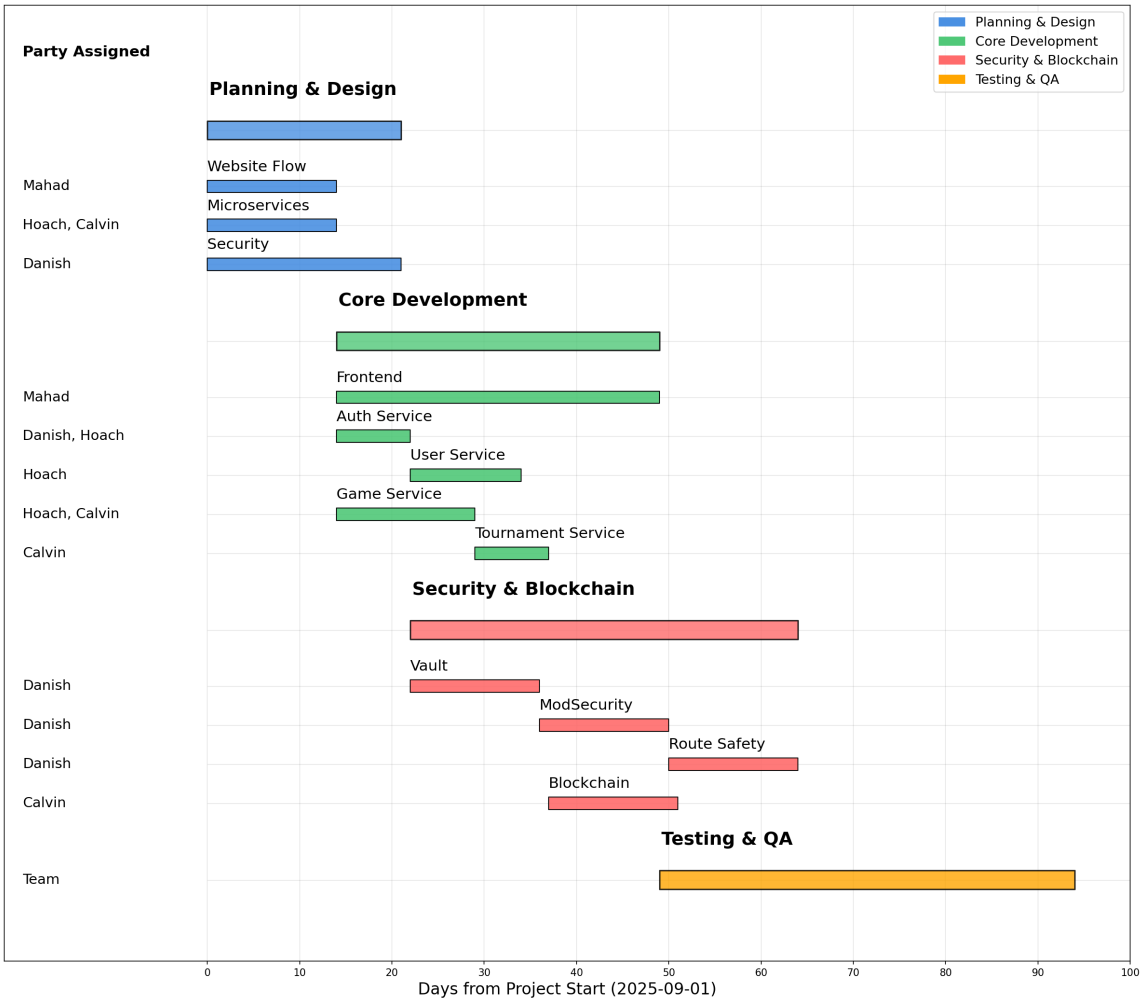

Figure 2.2: Project Gantt Chart: Timeline and resource allocation

2.4 Risk Management

2.4.1 Risk Matrix

The risk matrix below provides a framework for assessing the level of risk for potential project setbacks based on the product of likelihood and impact. This enabled the team to prioritize solutions for higher risk issues.

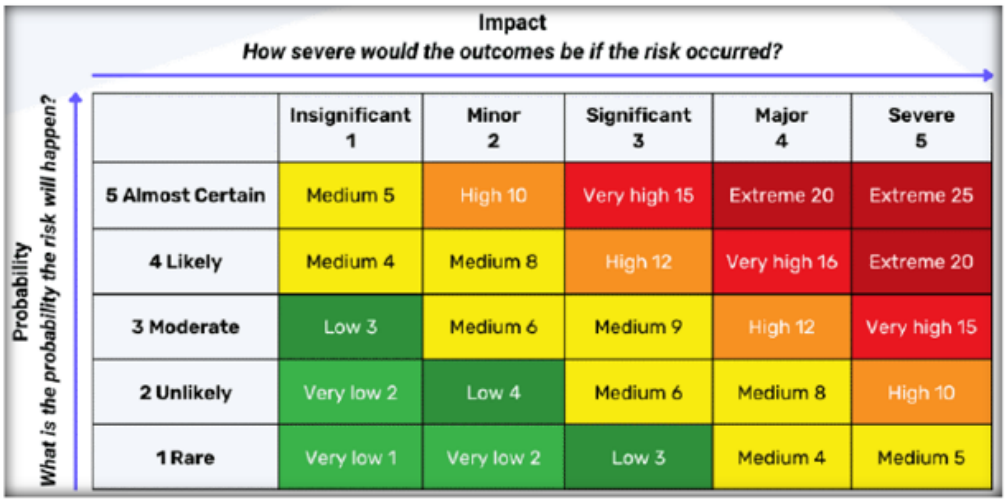

Table 2.1: Risk Matrix

2.4.2 Risk Register

ID	Description	Likelihood	Impact	Severity	Owner	Mitigation
1	Server downtime during peak testing	2	4	8	DevOps (Mahad & Hoach)	Monitoring, alerts, automated restarts
2	SQL injection attempt in legacy code	1	5	5	Security Team (Danish)	Parameterized queries + WAF rules
3	Data leak via misconfigured logs	2	4	8	Development Team (Hoach)	Redact PII in logs
4	OAuth provider downtime	1	3	3	QA Team (Danish)	Alternative login methods (email)
5	Unauthorized tournament/ blockchain route use (forged results)	3	5	15	Security + Blockchain Teams (Danish & Calvin)	Session auth + X-Microservice-Secret; contract onlyOwner (Vault-held key)

Table 2.2: Risk Assessment Matrix

Chapter 3

Requirement Analysis

Requirements specify what the system must do and how it achieves those goals. Detailed implementation, UI/UX, and architecture are described in the Design chapter.

3.0.1 Functional Requirements

Functional requirements specify *what* the system must do from the user's perspective. (See Design chapter for detailed UI, wireframes, and flows.)

User Management & Authentication

- FR-1: Users shall register with email and password
- FR-2: Users shall authenticate via local credentials
- FR-3: Users shall manage profiles (username, avatar, bio)

Gameplay & Real-Time Features

- FR-4: System shall provide a real-time Pong game with smooth rendering
- FR-5: Players shall control paddles via keyboard input
- FR-6: Game state shall synchronize across connected clients in real-time
- FR-7: System shall detect collisions, score updates, and game end conditions
- FR-8: Players shall access multiple game modes: campaign, arcade, tournament

Social Features

- FR-9: Users shall add and remove friends
- FR-10: Users shall view match history with detailed statistics
- FR-11: System shall display player profiles

Tournament Management

- FR-12: Users shall create and configure tournaments
- FR-13: System shall manage tournament bracket progression
- FR-14: Tournament results shall be recorded immutably
- FR-15: Users shall view tournament standings and schedules

Usability & User Experience

- FR-16: Interface shall be responsive across desktop and mobile devices
- FR-17: 3D mode shall gracefully degrade to 2D rendering
- FR-18: User onboarding shall take $<$5 minutes for new users
- FR-19: Error messages shall be clear and actionable

3.0.2 Technical Requirements

Technical requirements specify *how* the system shall achieve functional goals. (See Design chapter for architecture diagrams and implementation details.)

Architecture & Infrastructure

- TR-1: Backend shall implement microservices architecture (6 services: auth, user, game, tournament, blockchain, vault)
- TR-2: Each microservice shall operate independently with its own database (SQLite)
- TR-3: Services shall communicate via REST API and WebSocket protocols
- TR-4: NGINX reverse proxy shall route traffic and enforce HTTPS
- TR-5: System shall be deployable via Docker Compose
- TR-6: Services shall be independently deployable with automatic restart on failure
- TR-7: Architecture shall support horizontal scaling with load balancing

Technology Stack

- TR-8: Backend: Node.js 18+ with Fastify v4 framework
- TR-9: Language: TypeScript with strict mode enabled
- TR-10: Frontend: Vite + TypeScript with vanilla DOM APIs
- TR-11: Database: SQLite 3 (optimized with prepared statements)
- TR-12: Real-time communication: WebSocket protocol
- TR-13: Blockchain: Solidity with Hardhat framework
- TR-14: 3D Graphics: Babylon.js for game rendering

Security Requirements

- TR-15: All HTTP traffic shall enforce HTTPS with TLS 1.2+
- TR-16: Sensitive headers shall include Secure and HttpOnly flags
- TR-17: Web Application Firewall (ModSecurity) shall block OWASP Top 10 attacks
- TR-18: All SQL queries shall use parameterized statements
- TR-19: Passwords shall be hashed with bcrypt (cost factor 10+)
- TR-20: Secrets shall be managed via HashiCorp Vault
- TR-21: Input validation shall enforce type and length constraints
- TR-22: System shall implement defense-in-depth security layers

Performance & Reliability

- TR-23: Game loop shall execute at 60 FPS with server-authoritative architecture
- TR-24: WebSocket messages shall be sent at 50 ms intervals
- TR-25: API response time shall be smaller than 200 ms for 95th percentile
- TR-26: System shall support 100+ concurrent WebSocket connections per instance
- TR-27: Database queries shall complete within 100ms under normal load
- TR-28: Database shall handle 1000+ concurrent connections
- TR-29: System uptime shall be 99.9% during operational hours
- TR-30: Database transactions shall maintain ACID properties
- TR-31: Blockchain records shall be immutable and verifiable

Code Quality & Maintainability

- TR-32: Documentation shall be comprehensive and up-to-date
- TR-33: Logging shall provide sufficient debugging information

Chapter 4

Design

4.1 System Architecture

The system employs a microservices architecture and the complete deployment consists of 10 Docker containers orchestrated via Docker Compose:

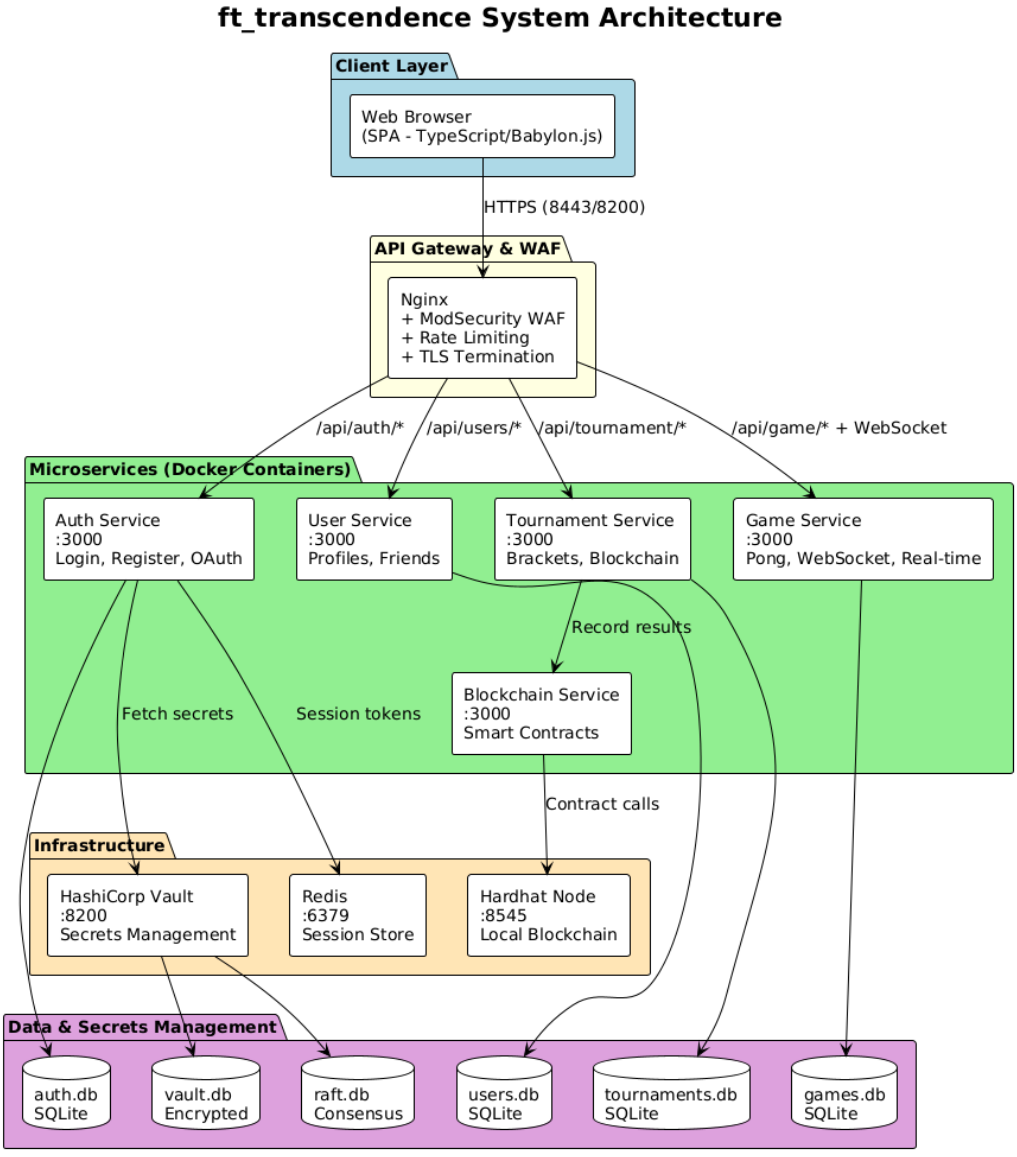

Figure 4.1: System Architecture with Microservices, API Gateway, and Persistent Storage

4.2 Data Model

Each microservice manages its own SQLite database:

4.2.1 Auth Service Database (auth.db)

- `users`: id, username, email, password_hash, oauth_provider, created_at, last_login

4.2.2 User Service Database (users.db)

- `user_profiles`: id, user_id, display_name, avatar_url, is_custom_avatar, bio, country, campaign_level, games_played, games_won, win_streak, tournaments_won, friends_count, xp, level, created_at, updated_at
- `friends`: user_id, friend_id, created_at

4.2.3 Game Service Database (games.db)

- `games`: id, player1_id, player2_id, player1_score, player2_score, status, started_at, finished_at, winner_id, game_mode, team1_players, team2_players, tournament_id, tournament_match_id
- `game_events`: id, game_id, event_type, event_data, timestamp

4.2.4 Tournament Service Database (tournaments.db)

- `tournaments`: id, name, current_participants, status, created_by, created_at, started_at, finished_at, winner_id
- `tournament_matches`: id, tournament_id, round, match_number, player1_id, player2_id, winner_id, player1_score, player2_score, status, played_at
- `tournament_participants`: id, tournament_id, user_id, alias, avatar_url, joined_at, eliminated_at, final_rank

4.3 Security Design

The system implements a comprehensive, defense-in-depth security architecture following industry best practices and OWASP guidelines. The security model encompasses six distinct layers, each providing specific protections against various attack vectors.

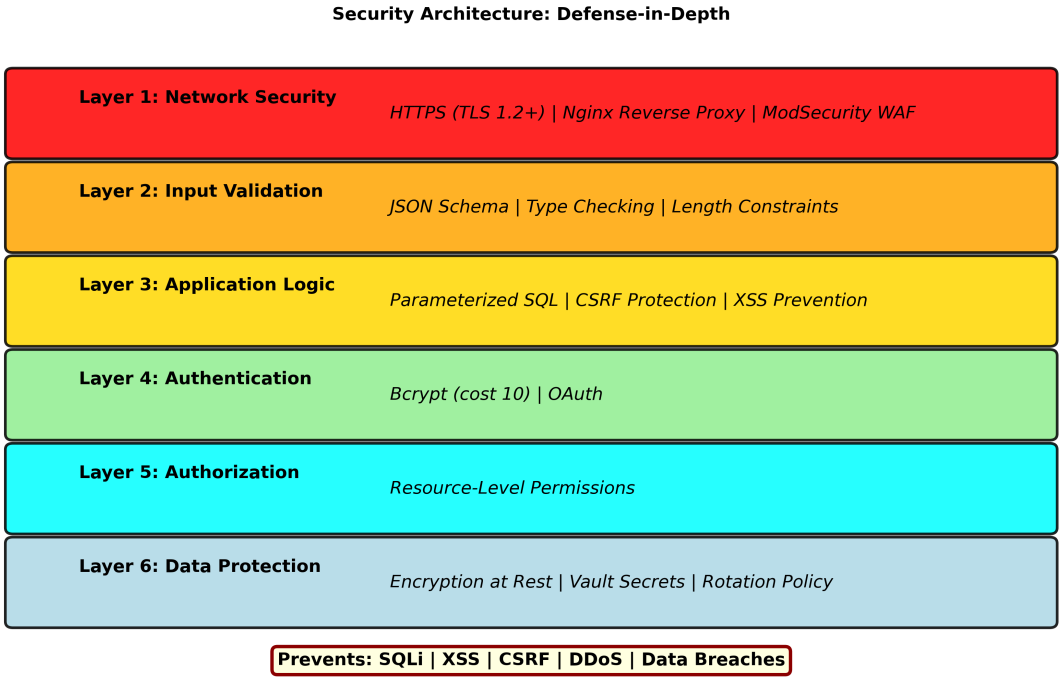

Figure 4.2: Defense-in-Depth Security Architecture with Six Protective Layers

4.3.1 Layer 1: Network Security

HTTPS and TLS Implementation

All communication channels are secured with HTTPS using TLS 1.2+ protocols. The system implements:

- **TLS 1.2/1.3 Enforcement:** NGINX configured to reject TLS 1.1 and lower
- **HSTS Headers:** Strict-Transport-Security with max-age=31536000
- **Certificate Validation:** Mutual TLS authentication between services

Web Application Firewall (WAF)

ModSecurity v3 engine is integrated as an inline module within the NGINX reverse proxy, utilizing the OWASP Core Rule Set (CRS) for real-time traffic inspection to block common attacks.

DDoS Mitigation

NGINX provides basic DDoS mitigation through connection management and upstream health monitoring.

4.3.2 Layer 2: Transport Security

Mutual TLS (mTLS) Between Services

All inter-service communication uses mutual TLS authentication.

Session Security

Redis-backed session storage with TLS encryption:

- **Secure Session Storage:** Sessions stored in Redis with TLS encryption
- **Session Encryption:** All session data encrypted in transit and at rest
- **Session Timeout:** Automatic session expiration and cleanup

4.3.3 Layer 3: Application Security

Input Validation and Sanitization

Comprehensive input validation using manual checks and sanitization.

SQL Injection Prevention

All SQL queries use parameterized statements with ‘?‘ placeholders.

Cross-Site Scripting (XSS) Protection

Multiple layers of XSS prevention:

- **Content Security Policy (CSP):** Strict CSP headers enforced
- **X-XSS-Protection:** Browser-based XSS filtering enabled
- **Input Sanitization:** All user inputs sanitized before rendering

Cross-Site Request Forgery (CSRF) Protection

CSRF protection via SameSite cookie attributes and origin validation.

4.3.4 Layer 4: Authentication & Authorization

Password Security

Industry-standard password hashing and validation.

Remote Authentication (OAuth)

OAuth 2.0 integration with Google for alternative authentication.

4.3.5 Layer 5: Data Protection

Secrets Management with HashiCorp Vault

Centralized secrets management for all sensitive data. Key secrets managed in Vault:

- **API Keys:** OAuth provider secrets and external service keys
- **Session Secrets:** Cryptographically secure session signing keys
- **TLS Certificates:** Automated certificate lifecycle management

Database Security

SQLite databases with additional security measures:

- **Prepared Statements:** All queries use parameterized execution
- **Single Connection Per Service:** Each service maintains one SQLite connection via the shared common package
- **Access Control:** Database files with restricted permissions

4.3.6 Layer 6: Monitoring & Logging

Security Event Logging

Comprehensive logging of security-relevant events.

Health Monitoring

Automated health checks for all security components:

- **Certificate Expiry Monitoring:** Automatic renewal alerts
- **Vault Connectivity:** Health checks for secrets management
- **WAF Status:** ModSecurity rule effectiveness monitoring

4.3.7 Layer 7: Incident Response

Security Headers Implementation

Comprehensive security headers configuration implemented within the `nginx.conf` file.

Container Security

Docker security best practices implementation:

- **Minimal Images:** Alpine Linux base images for reduced attack surface
- **Secret Management:** Environment variables and Vault retrievals for sensitive configuration
- **Resource Limits:** Memory limits per service to prevent resource exhaustion

4.3.8 Security Testing and Validation

The security implementation is validated through comprehensive testing:

WAF Effectiveness Testing

Tests verify ModSecurity rule effectiveness:

- **SQL Injection Attempts:** Parameterized query validation
- **XSS Payload Testing:** Input sanitization verification
- **Path Traversal:** File system access control validation

Vault Integration Testing

Secrets management functionality validation:

- **Secret Retrieval:** Automated secret access testing
- **Certificate Management:** PKI certificate lifecycle testing

HTTPS/TLS Testing

Transport security validation:

- **Certificate Validation:** SSL/TLS handshake verification
- **Cipher Suite Testing:** Supported cipher suite validation
- **HSTS Compliance:** Security header presence verification

4.3.9 Security Compliance

The implementation achieves compliance with multiple security standards:

OWASP Top 10 Coverage

- **A01:2021 - Broken Access Control:** Session validation
- **A02:2021 - Cryptographic Failures:** TLS 1.2+ and bcrypt hashing
- **A03:2021 - Injection:** Parameterized queries and input validation
- **A04:2021 - Insecure Design:** Defense-in-depth architecture
- **A05:2021 - Security Misconfiguration:** Automated configuration validation

Industry Best Practices

- **Zero Trust Architecture:** Every request authenticated and authorized
- **Least Privilege:** Minimal permissions for all components
- **Fail-Safe Defaults:** Secure defaults with explicit allow rules
- **Defense in Depth:** Multiple security layers for redundancy

4.4 Blockchain Integration

The ft_transcendence platform uses blockchain technology to provide immutable tournament result recording, improving transparency and reducing the risk of result manipulation. The implementation runs on a local Hardhat network inside Docker and is suitable for development, testing, and staging demonstrations.

4.4.1 Blockchain Architecture

The blockchain implementation consists of four coordinated components:

1. **Hardhat Local Network:** Dockerized Ethereum-compatible network used for local execution.
2. **Solidity Smart Contract:** On-chain tournament ranking storage with owner-restricted write access.
3. **Deployment Step:** Automated contract deployment that persists the deployed address for service startup.
4. **Blockchain Service:** Internal API that receives rankings and submits transactions to the smart contract.

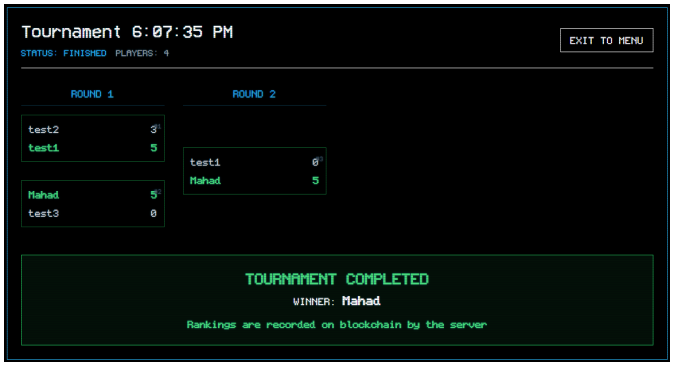

Figure 4.3: Blockchain Record: Tournament Result Verification on Immutable Ledger

4.4.2 Smart Contract Implementation

The smart contract stores tournament rankings by tournament identifier and player identifier. Writes are restricted to the contract owner, and each recorded rank emits an event for traceability.

Contract Features

- **Immutability:** Tournament results cannot be altered once recorded
- **Access Control:** Only the authorized owner account can record results
- **Batch Submission:** Multiple player ranks can be recorded in a single transaction
- **Event Logging:** Transparent event emission for result verification

4.4.3 Hardhat Development Environment

The project uses Hardhat as its local blockchain runtime for development and validation. The runtime is configured for an internal Docker network and deterministic local chain ID behavior.

Deployment Automation

Contract deployment is automated during container startup. The deployed contract address is persisted and consumed by the blockchain service, preventing manual copy/paste steps and reducing integration drift.

4.4.4 Blockchain Service Architecture

The blockchain service exposes an internal API for tournament result recording.

Service Components

- **Network Connectivity:** Connection to the local Hardhat node over the internal Docker network
- **Key Management:** Runtime retrieval of signing key material from HashiCorp Vault
- **Contract Binding:** Initialization from deployed address and compiled contract artifact
- **Transaction Confirmation:** Return of mined transaction hash for traceability

Service Behavior

At startup, the service validates required blockchain artifacts and deployment metadata, then initializes contract access. During requests, payloads are validated, rankings are submitted in batch, and transaction hashes are returned to upstream services.

4.4.5 Tournament Integration

Tournament results are automatically recorded to blockchain upon completion:

Integration Flow

1. Tournament matches complete and final rankings determined
2. Tournament service calls blockchain service with player rankings
3. Blockchain service submits transaction to smart contract
4. Transaction hash returned and stored in tournament database
5. Results become immutable and verifiable on blockchain

4.4.6 Blockchain Security Measures

Private Key Management

- **Vault Storage:** Private keys stored securely in HashiCorp Vault
- **Runtime Retrieval:** Keys loaded at service startup, not persisted
- **Access Control:** Microservice authentication via shared secrets
- **Audit Logging:** All blockchain operations logged with transaction details

Transaction Security

- **Authorization Checks:** Requests are accepted only from authenticated sessions or trusted internal callers
- **Transaction Confirmation:** Responses include mined transaction hashes for verification
- **Input Validation:** Tournament identifier and ranking payloads are validated before submission
- **Failure Signaling:** Failed submissions return explicit error responses for upstream handling

4.4.7 Blockchain Testing and Validation

Validation activities cover contract deployment, transaction submission, and end-to-end integration from tournament completion to recorded on-chain result.

Contract Testing

- **Compilation and Deployment Checks:** Validation that contract artifacts and deployment metadata are generated
- **Functional Validation:** Verification of rank recording and retrieval behavior
- **Integration Validation:** End-to-end checks from tournament flow to on-chain transaction
- **Security Review:** Manual review of access control and input assumptions

Service Testing

Service tests validate API authorization, payload validation, and transaction result handling.

4.4.8 Blockchain Performance Optimization

Gas Optimization

- **Batch Operations:** Multiple rankings recorded in single transaction
- **Efficient Storage:** Mapping-based storage for straightforward lookup
- **Minimal Computation:** Simple ranking storage without complex logic

Network Efficiency

- **Local Network:** Hardhat provides fast local blockchain operations for development and testing
- **Async Processing:** Non-blocking blockchain operations in tournament flow
- **Preloaded Metadata:** Contract address and artifact data loaded at service startup

4.4.9 Blockchain Monitoring and Observability

Transaction Monitoring

- **Transaction Hashes:** All blockchain operations tracked with unique identifiers
- **Event Logging:** Smart contract events logged for audit trails
- **Performance Metrics:** Gas usage and transaction time monitoring
- **Error Tracking:** Failed transactions logged with detailed error information

Health Checks

Automated health monitoring for blockchain components:

- **Network Connectivity:** Hardhat node availability monitoring
- **Contract Accessibility:** Smart contract address validation
- **Service Health:** Blockchain service API responsiveness

4.4.10 Blockchain Deployment and Operations

Docker Integration

The blockchain node, deployment step, and blockchain service are containerized and orchestrated together. Service startup is dependency-aware: the node becomes healthy first, contract deployment completes, then the API service starts with mounted artifacts and deployment metadata.

Production Considerations

- **Current Scope:** Local-chain deployment focused on development, testing, and subject validation
- **Public-Network Migration:** Requires external RPC providers, secure key lifecycle controls, and chain-specific gas policies
- **Operational Hardening:** Requires enhanced observability, alerting, and incident procedures for sustained production use
- **Resilience Planning:** Requires deployment/versioning strategy for contract upgrades and rollback-safe service releases

This blockchain integration provides tournament result immutability and transparency, ensuring competitive outcomes can be independently verified. The implementation demonstrates practical blockchain integration for a microservice system in development and staging contexts, with a clear path to additional production hardening.

4.5 Microservices Architecture

The ft_transcendence platform implements a comprehensive microservices architecture designed for scalability, maintainability, and fault isolation. The system consists of 10 containerized services orchestrated through Docker Compose using a `docker-compose.yml` file, with each service handling specific responsibilities and communicating through well-defined APIs.

4.5.1 Deployment Topology

The microservices architecture follows domain-driven design principles with clear separation of concerns:

1. **Vault Service:** HashiCorp Vault for secrets management and TLS certificate issuance
2. **Redis Service:** In-memory data store for session management with mutual TLS
3. **Auth Service:** User authentication and authorization with Redis-backed sessions
4. **User Service:** User profile management and social features
5. **Game Service:** Real-time game logic and WebSocket communication
6. **Tournament Service:** Tournament management and bracket generation
7. **Blockchain (Hardhat):** Local Ethereum-compatible blockchain node
8. **Blockchain-Deploy:** One-shot container that deploys smart contracts on startup
9. **Blockchain Service:** Smart contract interaction API for tournament result recording
10. **Frontend (NGINX):** Vanilla TypeScript SPA with Babylon.js 3D rendering

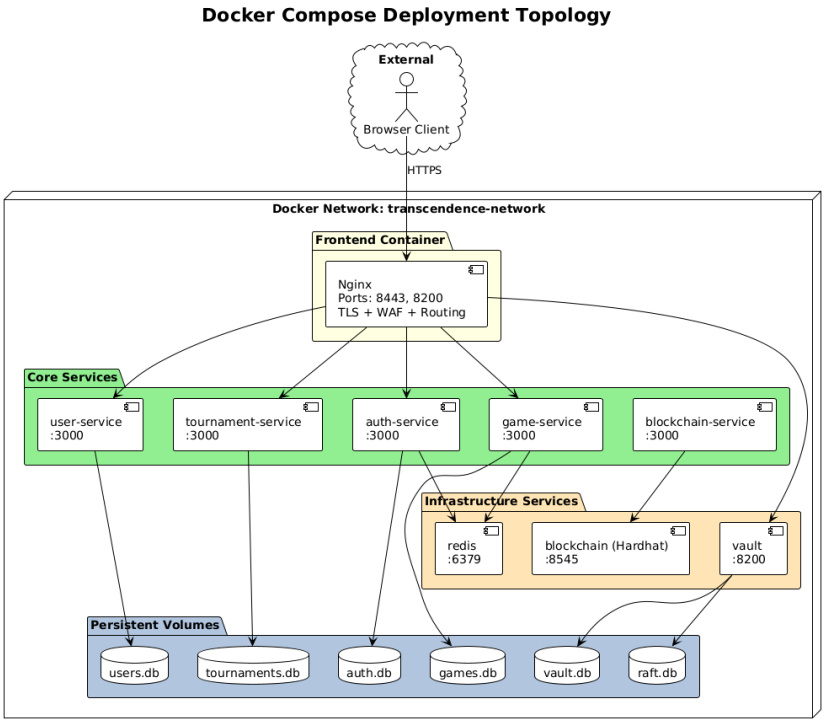

Figure 4.4: Deployment Topology: Service Dependencies and Communication Flow

4.5.2 Service Communication Patterns

Services communicate through multiple protocols optimized for different use cases:

- **HTTP/HTTPS APIs:** RESTful communication between services using Fastify framework
- **WebSocket Connections:** Real-time game state synchronization
- **Database Sharing:** SQLite databases with service-specific schemas
- **Shared Volumes:** Persistent data storage with bind mounts
- **Environment Variables:** Configuration management through .env files

4.5.3 Service Health Monitoring

Each service implements comprehensive health checks with automatic restart policies:

- **Health Endpoints:** HTTP health checks on service-specific ports
- **Dependency Validation:** Services use `depends_on` with health check conditions before starting
- **Resource Limits:** Memory constraints per service (256MB for backend services, 128MB for frontend)
- **Startup Probes:** Extended startup periods for complex services (Vault, Blockchain)

4.5.4 Production Deployment Considerations

The microservices architecture supports production deployment with:

- **Load Balancing:** NGINX reverse proxy for service distribution
- **Service Discovery:** Internal DNS resolution within Docker network
- **Configuration Management:** Environment-based configuration
- **Logging Aggregation:** Centralized logging through Docker Compose
- **Monitoring Integration:** Health check endpoints for external monitoring

This microservices architecture provides the foundation for a scalable, maintainable platform with clear service boundaries, comprehensive testing, and production-ready deployment capabilities.

4.6 3D Frontend Implementation

The ft_transcendence platform features an innovative 3D user interface built with Babylon.js, providing an immersive gaming experience that transcends traditional 2D web applications. The 3D frontend combines modern web technologies with advanced 3D rendering techniques.

4.6.1 Immersive Office Environment

The application features a unique ”Immersive Office” concept where the user interacts with the application through a virtual computer monitor situated within a 3D rendered 90s-style office cubicle. This design choice transforms the standard web interface into a diegetic element of the game world, enhancing immersion.

The 3D environment serves as more than just a background; it is the primary container for the application. When the user navigates the application, they are effectively looking at the screen of the virtual monitor.

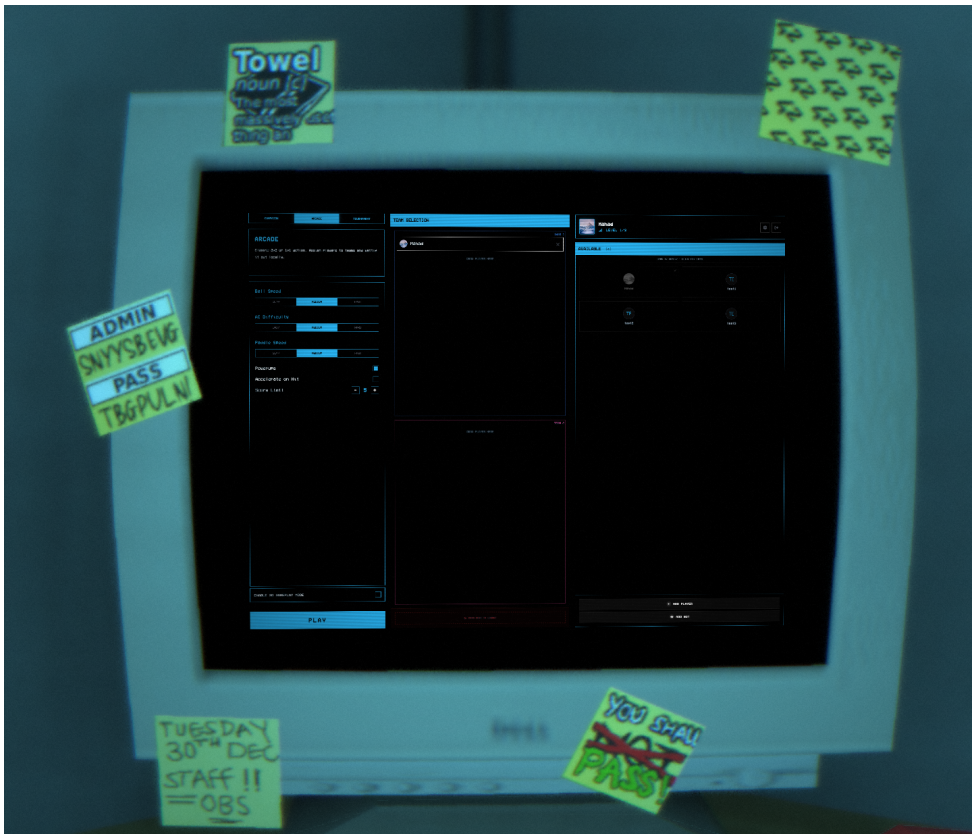

Figure 4.5: Immersive Office: Main Menu displayed on the Virtual Monitor

4.6.2 Story and Lore Integration

Upon the first launch, the user is presented with a narrative sequence that establishes the setting. The camera acts as the user's viewpoint, capable of panning dynamically between different points of interest, such as the monitor (for gameplay and UI) and ”Lore” items (like newspapers) scattered around the desk.

This seamless transition is managed by the `BabylonWrapper`, which interpolates camera positions to create smooth, cinematic movements between these interaction points, making the UI feel like an integrated part of the story.

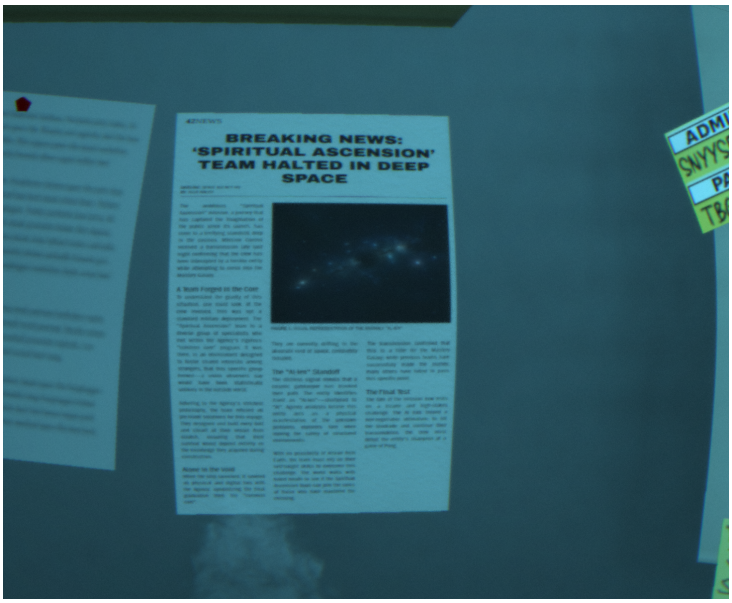

Figure 4.6: Story Integration: Interactive Newspaper providing Narrative Context

4.6.3 Babylon.js Integration Architecture

The 3D frontend implementation uses a singleton pattern with conditional initialization to manage the scene, camera, and post-processing effects. The helper methods `panToLore()` and `panToMonitor()` handle the cinematic transitions.

4.6.4 3D Game Rendering and Environmental Effects

The 3D Pong game is not an isolated overlay but plays out physically within the 3D scene. The game board is positioned effectively ”inside” the virtual monitor.

Environmental Lighting Interaction

A key feature of the 3D mode is the interplay between game elements and the environment. The ball and paddles are equipped with dynamic light sources. As the ball moves across the field, it casts real-time light onto the surrounding office desk and objects, creating a grounded and realistic effect. The virtual monitors also emit a glow that reflects off the desk surface.

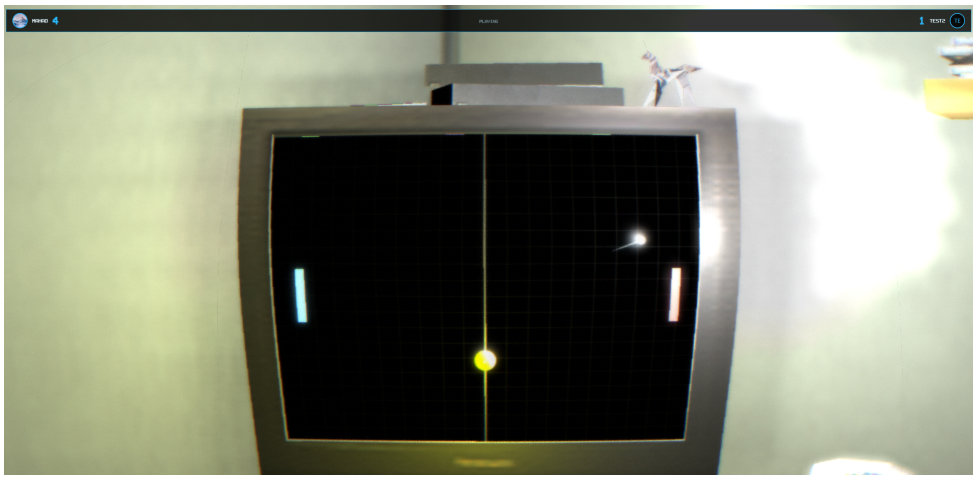

Figure 4.7: 3D Arcade Mode: Rendering ”Inside” the Virtual TV Screen

4.6.5 Real-time 3D Synchronization

The 3D renderer synchronizes with WebSocket game state updates:

- **Coordinate Mapping:** 2D game coordinates mapped to 3D world space
- **Smooth Interpolation:** Ball and paddle movement with easing functions
- **Visual Effects:** Dynamic lighting, particle trails, and glow effects

4.6.6 HTML Mesh Integration

To achieve the ”game within a monitor” effect for standard UI pages, the system employs Babylon.js's `HtmlMesh`. This allows the existing DOM-based interface to be projected onto a 3D plane within the scene, maintaining full interactivity (clicking, scrolling) while undergoing 3D perspective transformations.

4.6.7 Post-Processing Effects

Advanced visual effects enhance the retro gaming aesthetic:

- **Ambient Occlusion:** SSAO for realistic shadow rendering
- **Depth of Field:** Lens effects for cinematic camera work
- **Fog Effects:** Atmospheric depth cueing
- **Glow Layers:** Neon lighting effects for retro aesthetic

4.6.8 Performance Optimizations

The 3D implementation includes comprehensive performance optimizations:

- **Conditional Rendering:** 3D mode only enabled when WebGL is available
- **Resource Management:** Proper cleanup and disposal of 3D resources
- **Memory Limits:** Texture compression and efficient mesh usage
- **Fallback Support:** Graceful degradation to 2D rendering

This 3D frontend implementation provides an innovative, immersive gaming experience while maintaining performance and accessibility standards.

4.7 Wireframes and User Interface Design

Wireframes provide visual representations of application screens, illustrating layout, functionality, and user navigation flow. The design follows human-computer interaction principles with intuitive navigation and clear visual hierarchy.

4.7.1 Authentication Flow Wireframes

Login Interface

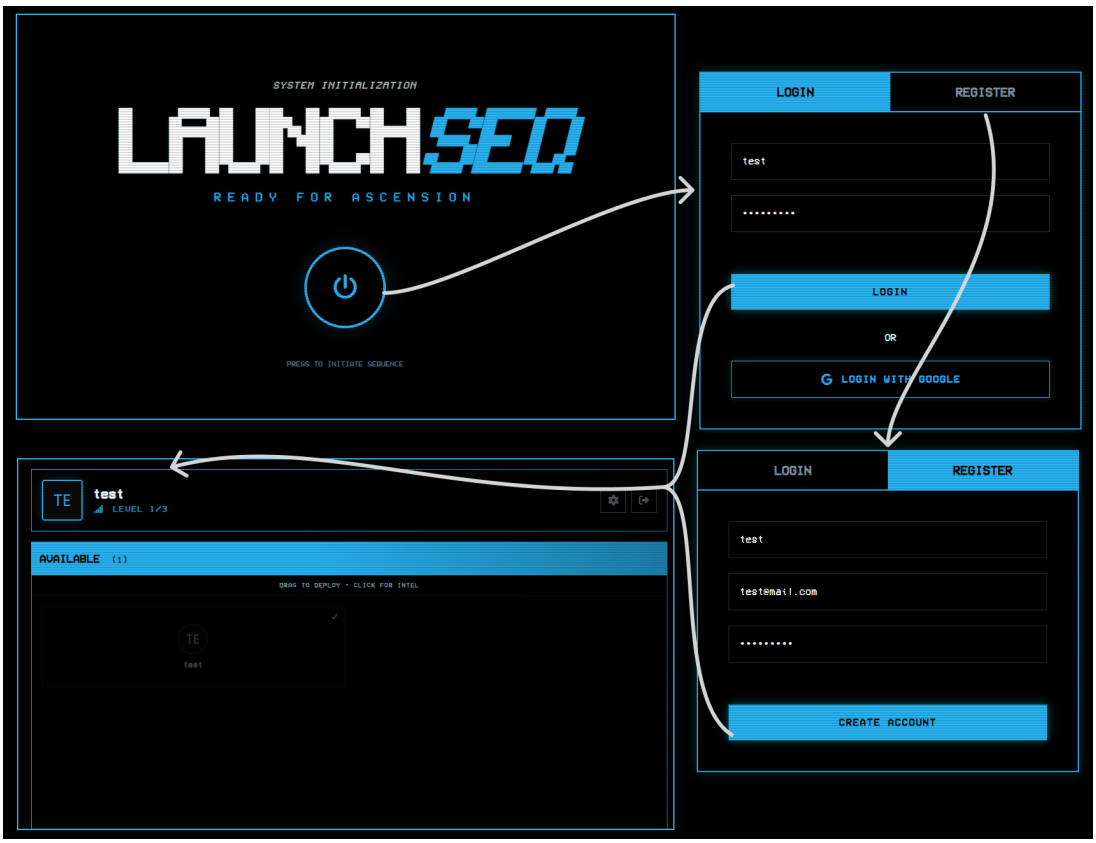

Figure 4.8: Authentication User Flow: Diagram illustrating the user journey through login and registration screens

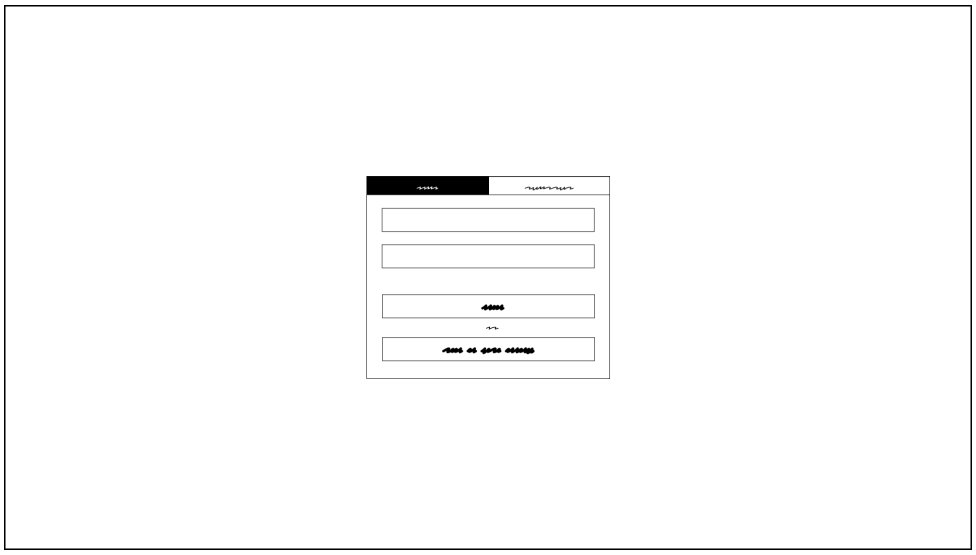

Figure 4.9: Login Interface Wireframe: Email/password authentication layout

Key elements:

- ”Register” tab for new user registration
- Email and password input fields with validation
- ”Login” button
- Google OAuth integration button
- Error message display area

Registration Interface

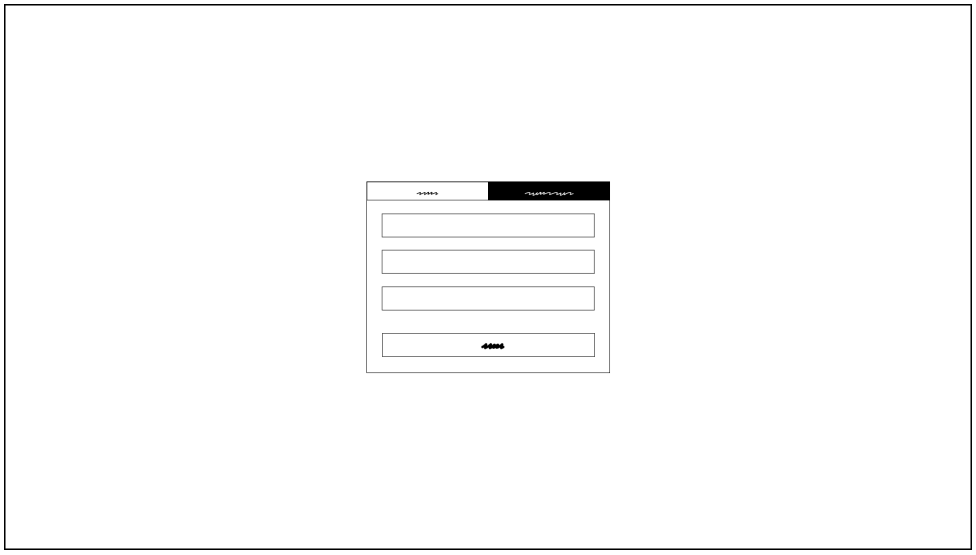

Figure 4.10: Registration Interface Wireframe: New user account creation form layout

Key elements:

- ”Login” tab for existing user authentication

- Username, email, and password fields
- ”Register” button
- Error message display area

4.7.2 Main Navigation and Menu Wireframes

Main Menu Interface

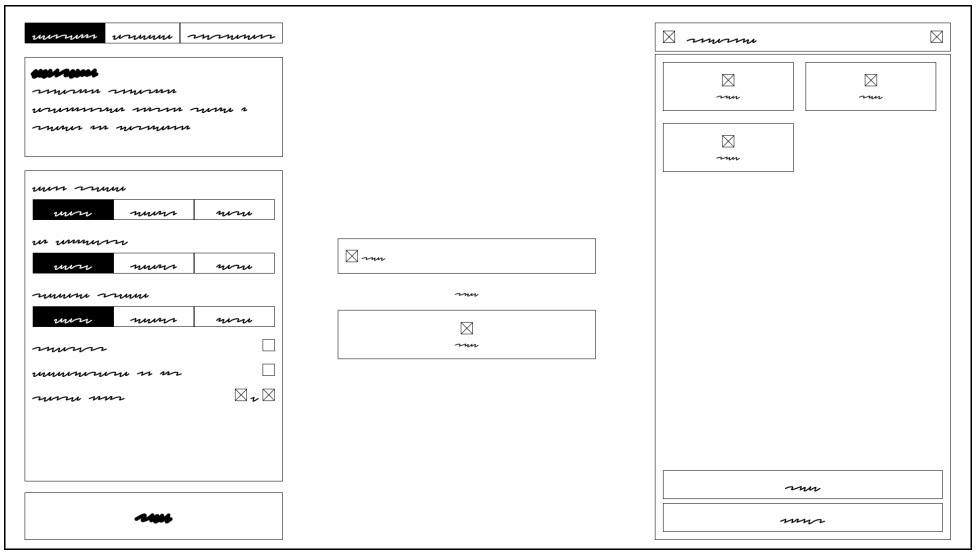

Figure 4.11: Main Menu Wireframe: Navigation hub with game modes and profile access

Key elements:

- Game mode buttons: Campaign, Arcade, Tournament
- User profile section with avatar and stats
- Settings and logout options

4.7.3 Game Interface Wireframes

Game Mode Selection

Key elements:

- Difficulty level selector (Easy, Medium, Hard)
- Ball speed adjustment slider
- Paddle size configuration
- AI opponent toggle (for campaign mode)
- ”Start Game” button

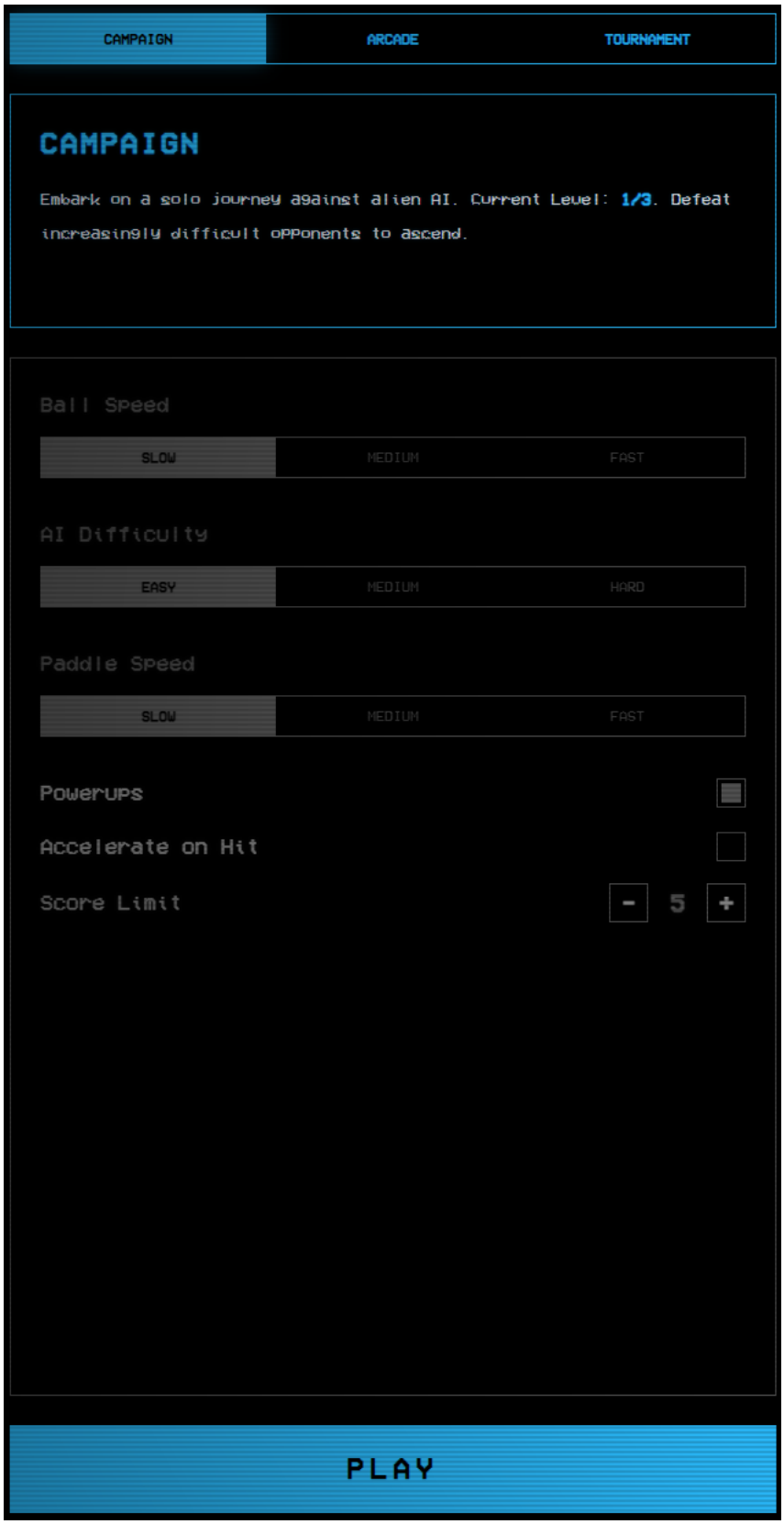

Figure 4.12: Game Mode Selection: Difficulty and settings configuration

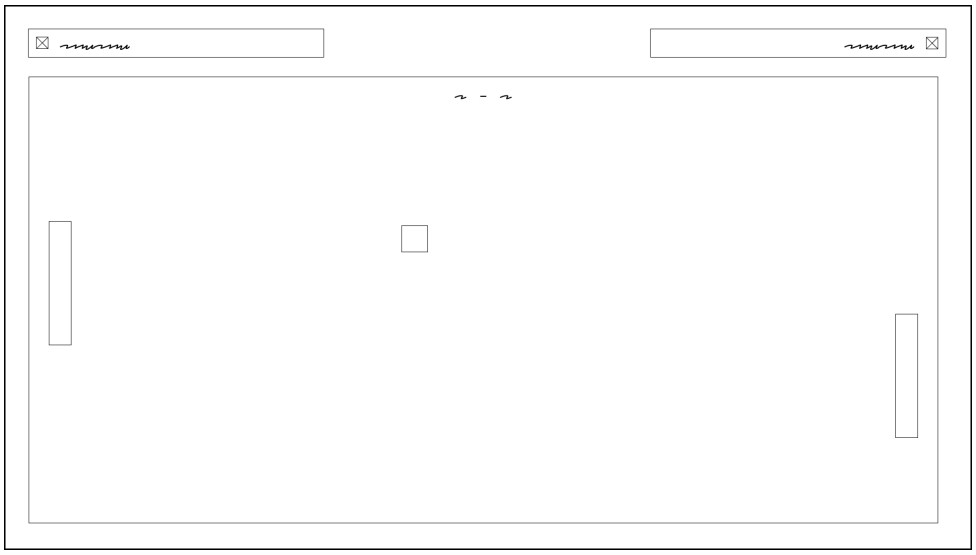

Figure 4.13: Gameplay Interface Wireframe: Pong match layout with HUD elements

Gameplay Interface

Key elements:

- Game canvas/board area
- Real-time score display (Player 1 vs Player 2)
- Game status text indicator
- 3D/2D rendering toggle feedback

Multiplayer Arcade Mode

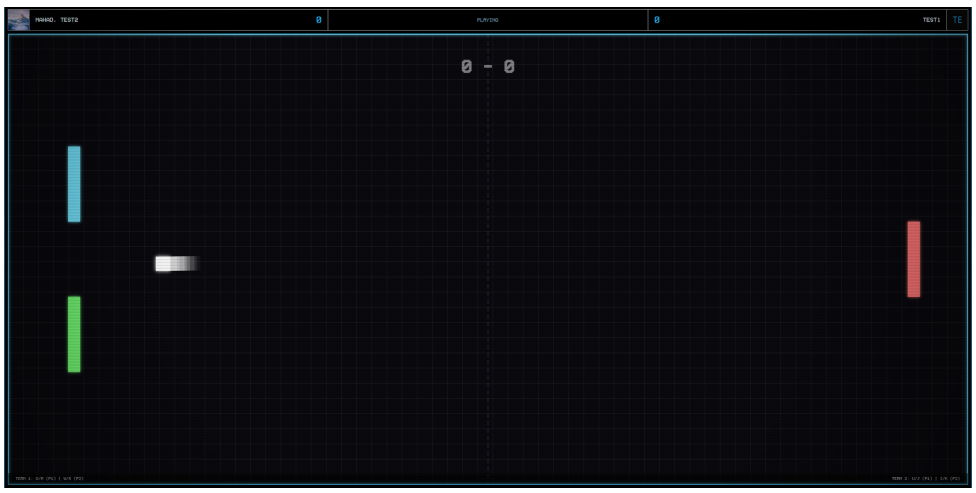

Figure 4.14: Multiplayer Arcade: Real-time competitive gameplay

Key elements:

- Player identification (avatars/names)
- Real-time input synchronization
- Disconnect/reconnect handling

4.7.4 Tournament Interface Wireframes

Tournament Bracket View

Figure 4.15: Tournament Bracket: Match scheduling and progression visualization

Key elements:

- Interactive bracket visualization
- Current match highlighting
- Participant status indicators
- Blockchain recording status message
- Player elimination tracking

Tournament Mode Selection

Key elements:

- ”Create Tournament” button
- Available tournaments list
- Tournament details (players, prize, status)
- Join/registration functionality
- Tournament rules display

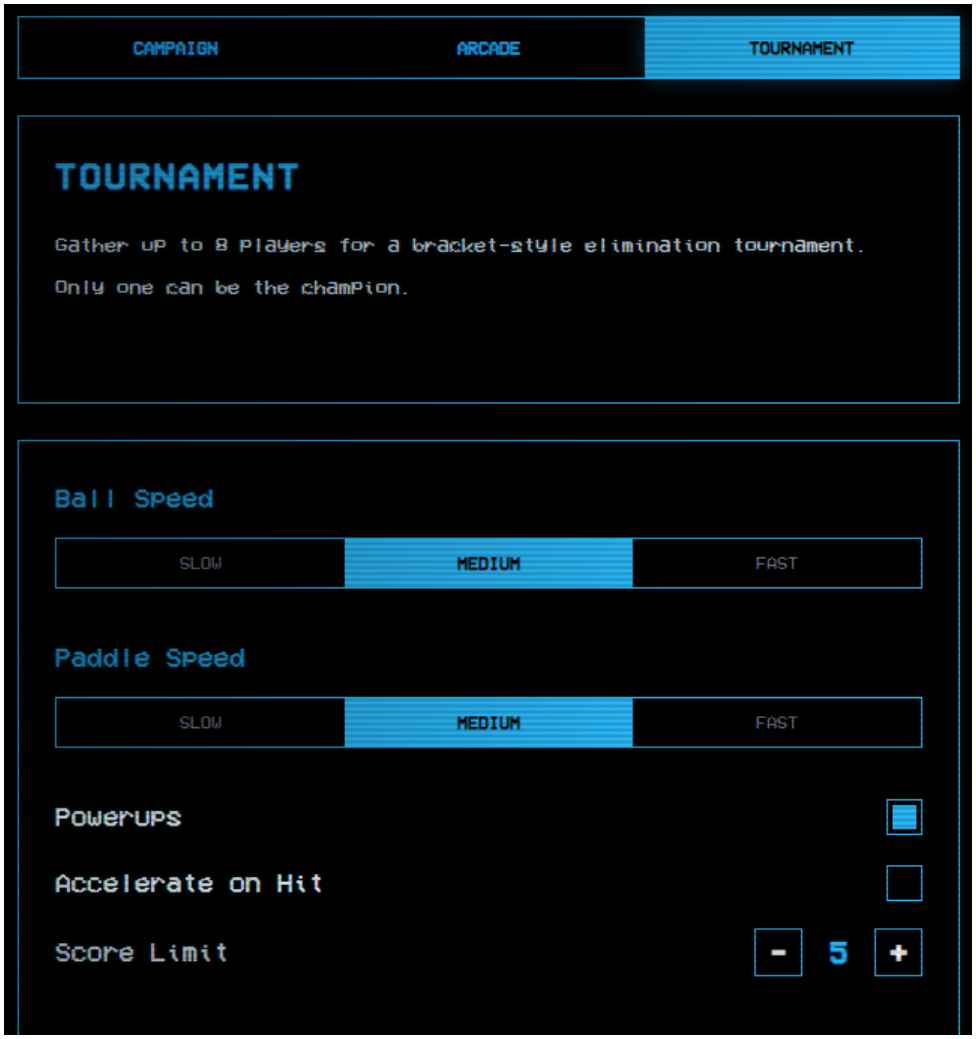

Figure 4.16: Tournament Mode Selection: Tournament creation and joining interface

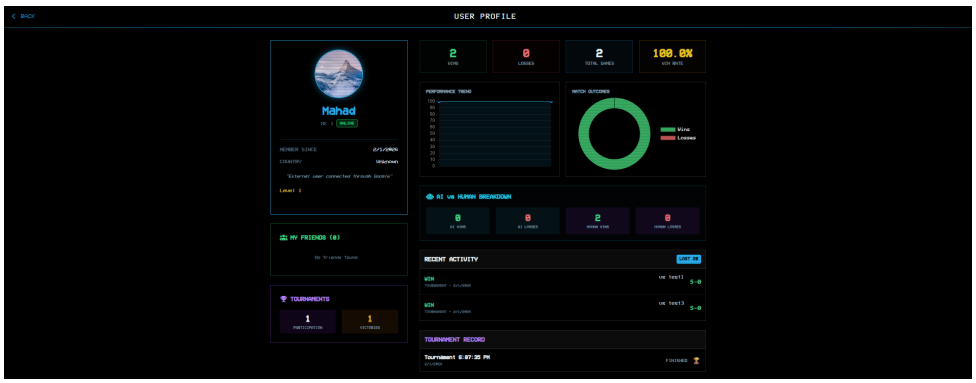

Figure 4.17: User Dashboard: Profile statistics and achievements

4.7.5 User Profile and Social Features

Dashboard and Profile

Key elements:

- User avatar and bio
- Win/Loss statistics visualization
- Match history with expansion details
- Friend list and social actions

4.7.6 3D Environment Integration

3D Monitor Interface

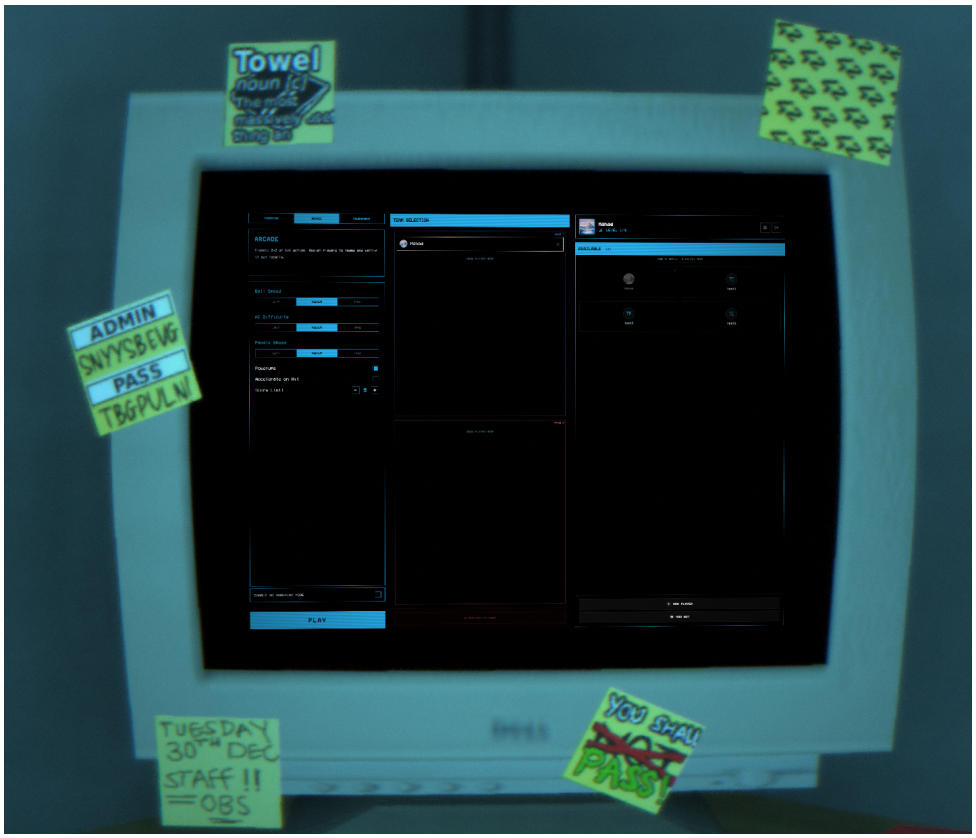

Figure 4.18: 3D Monitor Interface: Main menu projected on virtual screen

Key elements:

- 3D office environment context
- Virtual monitor displaying UI
- Interactive HTML mesh projection
- Environmental lighting effects
- Camera controls for 3D navigation

3D Arcade Game Mode

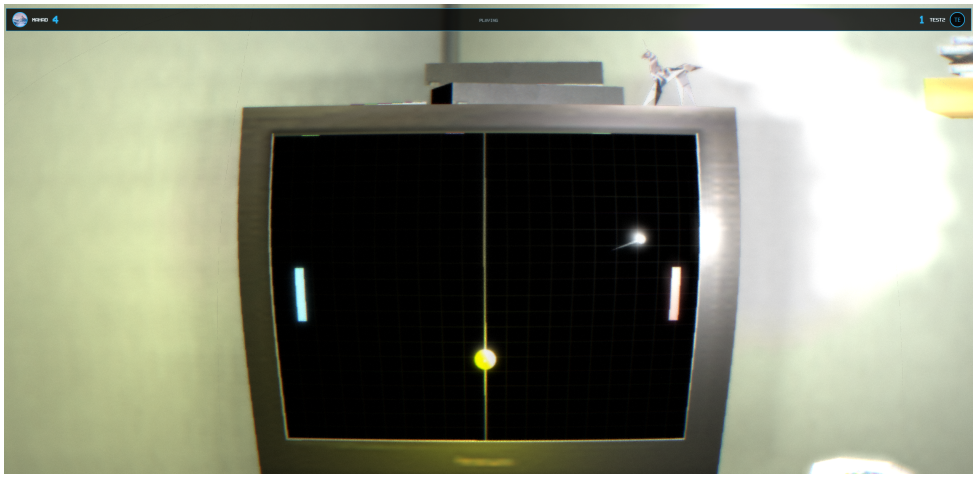

Figure 4.19: 3D Arcade Mode: Game rendering within virtual TV screen

Key elements:

- Game rendered ”inside” virtual monitor
- 3D ball and paddle physics
- Dynamic lighting from game elements
- Environmental interaction effects
- Retro aesthetic with modern 3D effects

3D Newspaper/Lore View

Key elements:

- Newspaper mesh in 3D space
- Interactive reading experience
- Camera transitions and animations
- Contextual game information
- Immersive narrative elements

4.7.7 Wireframe Design Principles

Responsive Design Considerations

- **Desktop-First Design:** Optimized for mouse and keyboard interaction
- **High-Performance Layout:** Fixed-width canvas for consistent game rendering
- **Clear Visual Hierarchy:** Prioritized game elements and status indicators

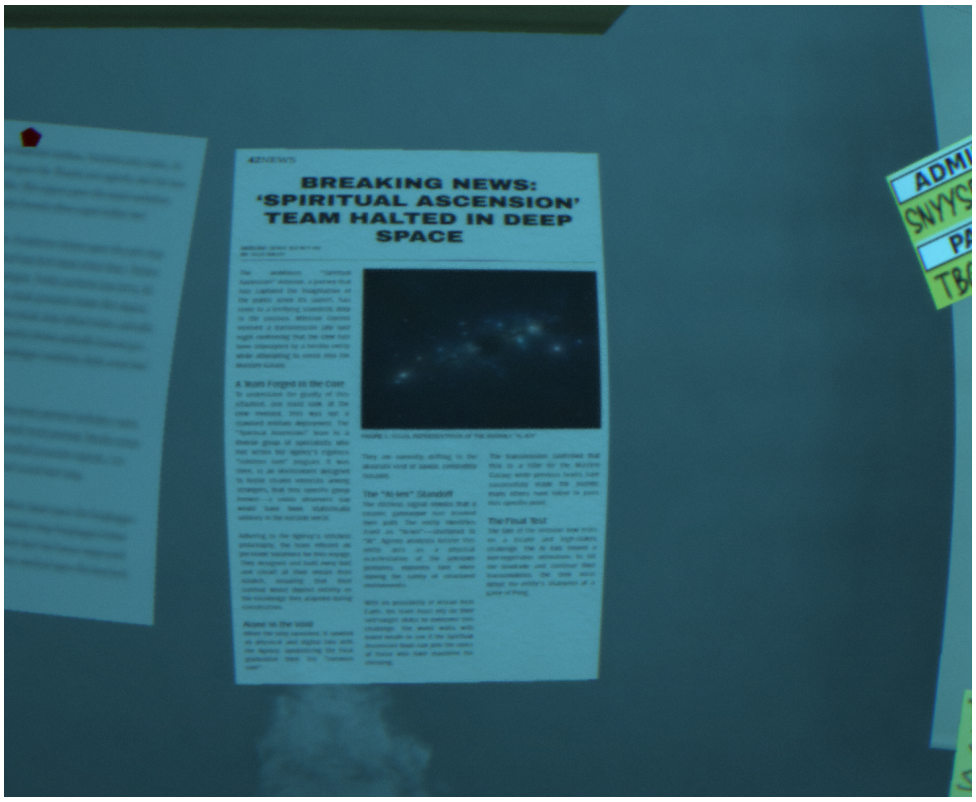

Figure 4.20: 3D Newspaper View: Interactive environmental storytelling

User Experience Guidelines

- **Intuitive Navigation:** Clear information hierarchy and logical flow
- **Visual Consistency:** Unified color scheme and typography
- **Feedback Systems:** Loading states, success/error messages, and progress indicators
- **Performance Focus:** Optimized for 60 FPS gameplay and responsive interactions

3D Integration Principles

- **Optional Enhancement:** 3D mode as progressive enhancement over 2D
- **Performance Fallback:** Automatic degradation to 2D rendering when needed
- **Contextual Immersion:** 3D environment enhances rather than complicates gameplay
- **Technical Stability:** WebGL detection and error handling

4.8 Flowcharts

Flowcharts illustrate the primary user interactions, system processes, and data flow across key components such as the homepage, settings, player profile, and game sections, providing a comprehensive overview of the project's functional workflow.

4.8.1 System Workflow Overview

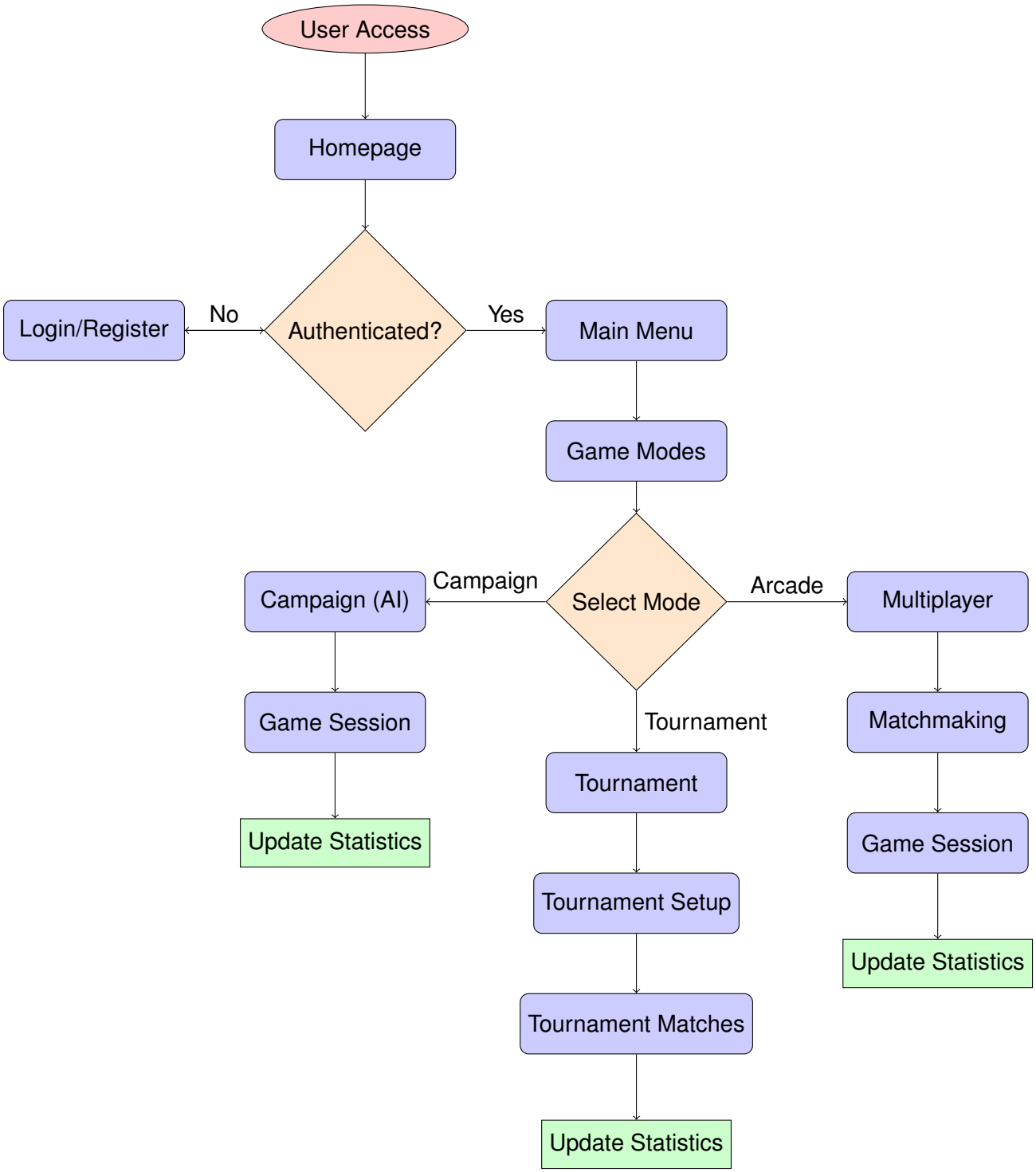

Figure 4.21: System Workflow: Complete user journey from access to gameplay completion

4.8.2 User Authentication Flow

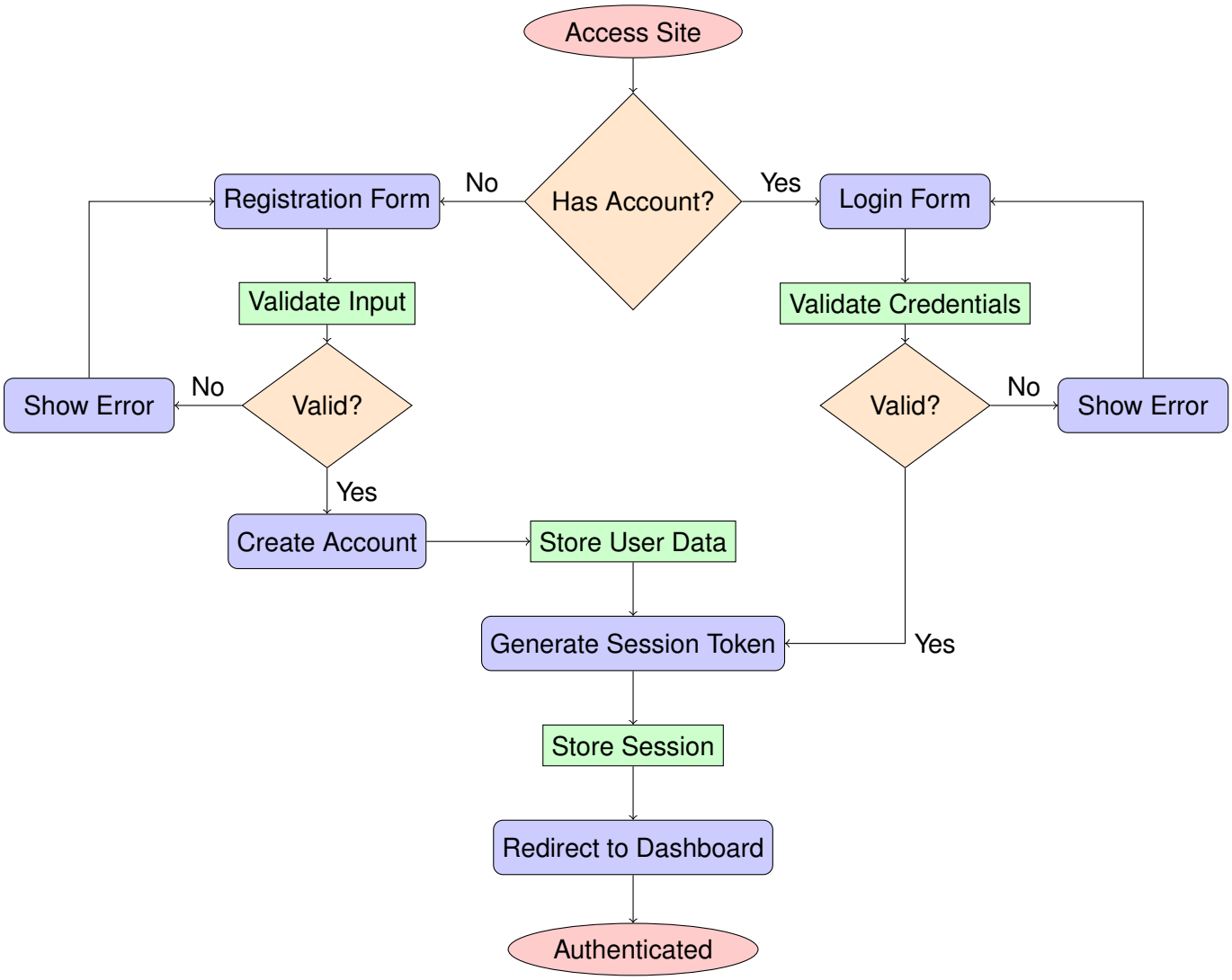

Figure 4.22: Authentication Flow: User registration and login process with validation

4.8.3 Game Session Flow

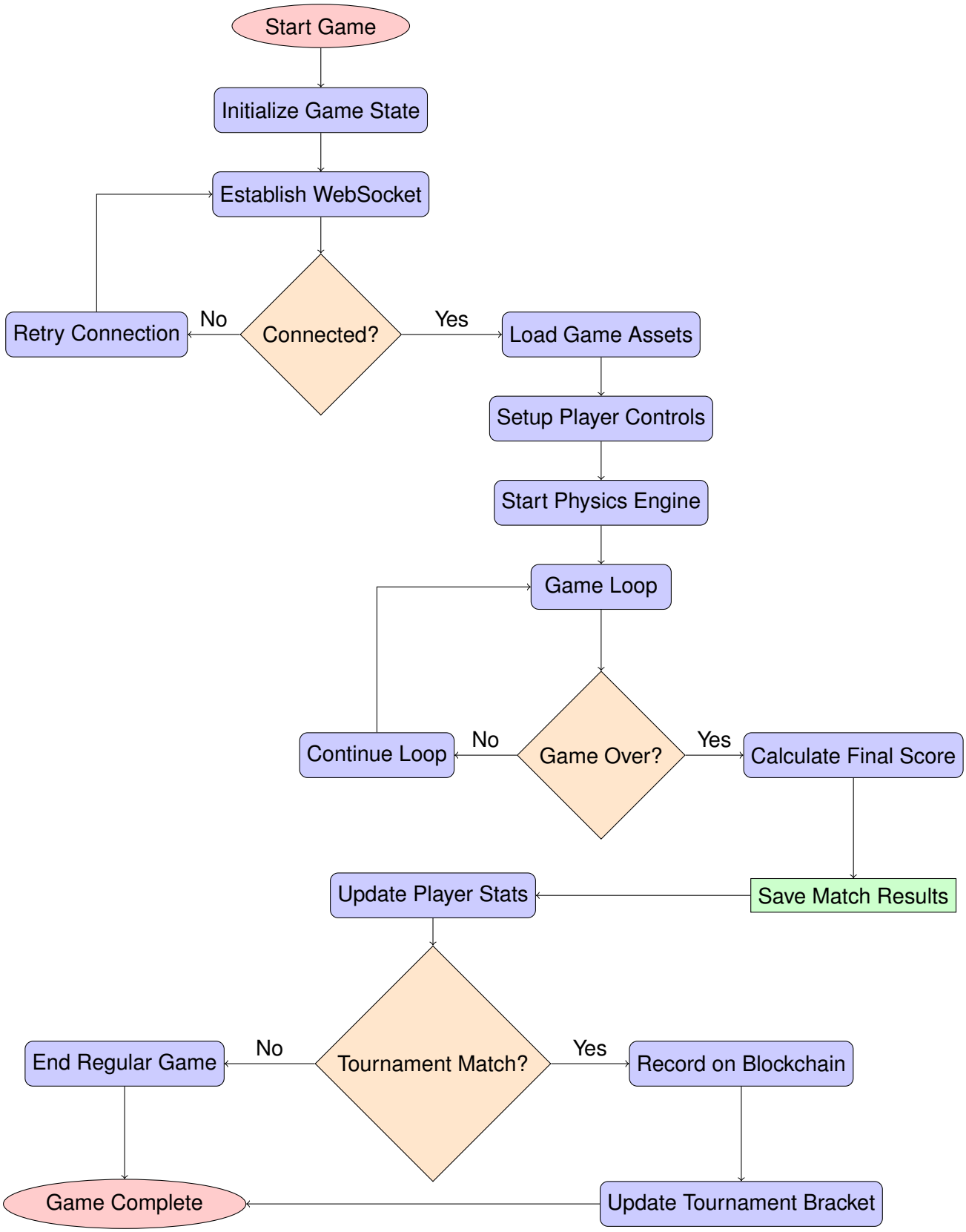

Figure 4.23: Game Session Flow: Complete game lifecycle from initialization to completion

Chapter 5

Implementation

The implementation follows a microservices architecture with five backend services communicating via REST APIs and WebSocket connections, supported by infrastructure containers (Vault, Redis, NGINX, Hardhat). The system achieves full compliance with all subject requirements, completing the mandatory with bonus – implementing 9 major and 4 minor modules. All 10 containers are orchestrated via Docker Compose for production deployment.

5.1 Technology Stack Summary

Component	Technology	Version
Backend	Fastify + Node.js + TypeScript	4.29.1 / 18+ / 5.9.3
Database	SQLite 3	5.1.6
Frontend Build	Vite	5.0.8
Real-Time	WebSocket	(Fastify plugin)
Auth	Bcrypt	(npm package)
Blockchain	Hardhat + Solidity	2.22.17
Secrets	HashiCorp Vault	1.21.1
API Gateway	NGINX + ModSecurity	1.29.4
Containers	Docker Compose	5+

Table 5.1: Technology Stack

5.2 Backend Framework

All five backend microservices use Fastify v4 with TypeScript strict mode:

- `auth-service`: User registration, login
- `user-service`: Profiles, friendships
- `game-service`: Server-authoritative Pong game logic, WebSocket real-time sync
- `tournament-service`: Tournament management, blockchain integration
- `blockchain-service`: Tournament result recording

5.3 Infrastructure Services

Core infrastructure is provided by dedicated supporting containers:

- `vault`: Secret management and TLS certificate issuance
- `redis`: Session storage backend
- `frontend` (NGINX + ModSecurity): Reverse proxy, WAF enforcement, and SPA delivery
- `blockchain` (Hardhat): Local blockchain node for smart contract execution
- `blockchain-deploy`: One-shot contract deployment container

5.4 Frontend Architecture

Modern TypeScript Single-Page Application (SPA) built with a component-based architecture and a clear service layer. The frontend implements client-side routing and progressive enhancement to provide a responsive, app-like UX while maintaining accessibility and graceful degradation.

Browser back/forward navigation is handled via client-side routing:

- URL-based state management (`/game`, `/profile`)
- No page reloads; state preserved during navigation
- Progressive enhancement for accessibility
- `core/`: Core application infrastructure
  - `Api.ts`: Centralized API client for backend communication
  - `App.ts`: Main application controller and lifecycle management
  - `Router.ts`: Client-side routing with URL-based navigation
- `components/`: Reusable UI components
  - `AbstractComponent.ts`: Base component class with lifecycle hooks
  - `GameRenderer.ts`: Canvas-based Pong game rendering engine
  - Modal components: Login, Tournament, Password confirmation dialogs
- `pages/`: Page-level components for routing
  - Authentication: LoginPage, RegisterPage, OAuthCallbackPage
  - Game modes: GamePage, TournamentBracketPage, Campaign gameplay
  - User features: DashboardPage, ProfilePage, SettingsPage
  - System: MainMenuPage, LaunchSeqPage, ErrorPage
- `services/`: Business logic and external integrations
  - `AuthService.ts`: Authentication state and API calls
  - `GameService.ts`: Real-time game session management
  - `TournamentService.ts`: Tournament operations and blockchain integration
  - `AIService.ts`: AI opponent logic for campaign mode
  - `BlockchainService.ts`: Smart contract interactions
  - `ProfileService.ts`: User profile and statistics management
- `types/`: TypeScript type definitions and interfaces

Chapter 6

Testing

6.1 Test Results Summary

The ft_transcendence project achieves comprehensive test coverage with all manual tests completed:

- **Manual Testing:** All modules validated (100% coverage)
- **Test Categories:** User workflows, security checks, integration validation
- **Coverage Areas:** All microservices, security features, blockchain integration

Test Category	Status
Authentication Service	Manual Testing Completed
User Service	Manual Testing Completed
Game Service	Manual Testing Completed
Tournament Service	Manual Testing Completed
Blockchain Integration	Manual Testing Completed
Security Implementation	Manual Testing Completed
Microservices Communication	Manual Testing Completed
Frontend Components	Manual Testing Completed
Total:	**All modules validated**

Table 6.1: Module Test Results by Subject Category

6.2 Manual Testing Procedures

Manual testing validates user workflows and system functionality through hands-on verification:

6.2.1 Test Execution

1. Start all services: `make full-start`
2. Access the application at: `https://localhost:8443`
3. Perform end-to-end user scenarios manually
4. Verify functionality across different browsers and devices
5. Document any issues or deviations from expected behavior

6.2.2 Service Health Checks

```
# Check service availability
curl -k https://localhost:8443    # Frontend
curl -k https://localhost:8200    # Vault
```

6.2.3 Test Categories

- **Authentication:** Registration, login, Google OAuth flows
- **Gameplay:** Real-time Pong matches, controls, scoring
- **Social Features:** Friend management, profiles
- **Tournaments:** Creation, bracket management, blockchain recording
- **Security:** WAF protection, HTTPS enforcement, input validation
- **Microservices:** Inter-service communication and health endpoints
- **AI Opponent:** Campaign mode AI behavior and difficulty scaling
- **Game Logic:** Server-side Pong calculations and WebSocket stability

6.2.4 User Acceptance Test Scenarios

1. **User Registration:** Create account, create account with Google, complete profile
2. **Authentication:** Login with password, login with Google
3. **Gameplay:** Play quick match, verify real-time sync, check scoring
4. **Tournament:** Create tournament, manage bracket, record blockchain result

6.2.5 Integration Testing

Manual integration tests verify end-to-end functionality:

- User registration to gameplay flow
- Tournament creation to blockchain recording
- Multiplayer session synchronization

Chapter 7

Evolution

7.1 Current State

The ft_transcendence project is fully implemented, comprehensively tested with all manual tests completed, and production-ready for deployment. All subject requirements have been achieved. The system demonstrates a robust, scalable architecture capable of supporting real-time multiplayer gaming with enterprise-grade security and compliance features.

7.2 Future Enhancements

While the current implementation meets all specified requirements, several enhancement opportunities exist for future development:

7.2.1 Advanced Game Features

- **Spectator Mode:** Real-time match viewing with statistics overlay, allowing users to watch ongoing games without participating
- **Time-Limited Challenges:** Timed game modes with countdown-based scoring for faster-paced matches
- **Adaptive AI Learning:** Machine learning-based AI opponents that adapt to individual player strategies over time

7.2.2 Platform Expansion

- **Mobile Application:** Native iOS and Android apps with touch-optimized controls for on-the-go gameplay
- **In-Game Chat:** Real-time messaging between players during matches and in lobby areas
- **Esports Features:** Prize pool management, seasonal leagues, and professional tournament scheduling beyond the current bracket system

7.2.3 Technical Improvements

- **Global Distribution:** CDN integration and edge computing for reduced latency across regions

- **Advanced Analytics:** Player behavior tracking, heatmaps, and performance trend analysis beyond current win/loss statistics
- **Automated Testing:** Unit, integration, and end-to-end test suites to supplement current manual testing procedures
- **Database Migration:** Transition from SQLite to PostgreSQL for improved concurrency and scalability under high load

7.3 Limitations and Constraints

7.3.1 Current Limitations

- **Scalability Ceiling:** Current architecture supports hundreds of concurrent users but may require optimization for thousands
- **Resource Intensity:** 3D rendering and real-time physics demand significant client-side computing power
- **Browser Compatibility:** Advanced WebGL features may not be supported on older browsers or low-end devices
- **Storage Constraints:** SQLite databases in microservices may become performance bottlenecks at scale

7.3.2 Technical Debt Considerations

- **Monolithic Components:** Some services contain multiple responsibilities that could be further decomposed
- **Testing Coverage:** While comprehensive manual testing is complete, automated test coverage could be expanded
- **Documentation Updates:** API documentation and deployment guides require ongoing maintenance
- **Dependency Management:** Regular security audits and dependency updates are essential for production stability

Chapter 8

Conclusion

The ft_transcendence project aimed to develop a production-ready multiplayer Pong platform demonstrating modern software engineering practices, achieving 100% subject compliance with advanced features including real-time WebSocket gaming, blockchain tournament integrity, and enterprise security.

Working on ft_transcendence produced both interpersonal and technical growth. On the teamwork side we practiced collaborative planning, asynchronous development coordination, code review discipline, and cross-disciplinary communication between frontend, backend, and blockchain components. Those habits improved estimate accuracy, reduced integration friction, and made feature rollouts predictable. Technically, the project provided hands-on experience with TypeScript and Fastify-based microservices, WebSocket-driven server-authoritative real-time systems, Babylon.js 3D rendering, Solidity/Hardhat smart-contract development, Docker Compose orchestration, HashiCorp Vault for secrets, and practical WAF/HTTPS hardening — plus debugging and profiling techniques for low-latency gameplay.

The three most difficult challenges encountered were:

1. **Real-time server-authoritative synchronization:** Achieving smooth, deterministic gameplay under network jitter required careful physics timing, interpolation/extrapolation strategies, and robust reconnection handling.

2. **Blockchain integration and key management:** Automating contract deployment, ensuring deterministic testnet behavior, and securing signing keys via Vault while enabling service automation proved operationally complex.

3. **Secure containerized orchestration:** Combining mutual TLS between services, a ModSecurity-enabled NGINX gateway, and Vault-based secrets in a multi-container Docker Compose environment required careful startup ordering, health checks, and hardened defaults to avoid fragile boot-time or permission failures.

Planned improvements focus on reliability, scalability, and broader accessibility: add automated unit/integration/end-to-end test suites with CI pipelines to supplement the manual testing workflow; migrate service storage from per-service SQLite to a managed PostgreSQL deployment to improve concurrency and operational monitoring; introduce mobile/touch-friendly UI, spectator/chat features, and analytics dashboards; and evolve deployment to multi-host/container orchestration (Kubernetes or managed services) with CDN/edge delivery for global latency reduction.

The ft_transcendence project successfully demonstrates the application of modern software engineering principles to deliver a complex, production-ready gaming platform. The team implemented advanced features that validate the effectiveness of iterative development, comprehensive security, and quality assurance practices. The platform serves as both a technical achievement and educational case study for scalable software development.

Appendix A

Data Flow and System Diagrams

A.1 Game Match Data Flow

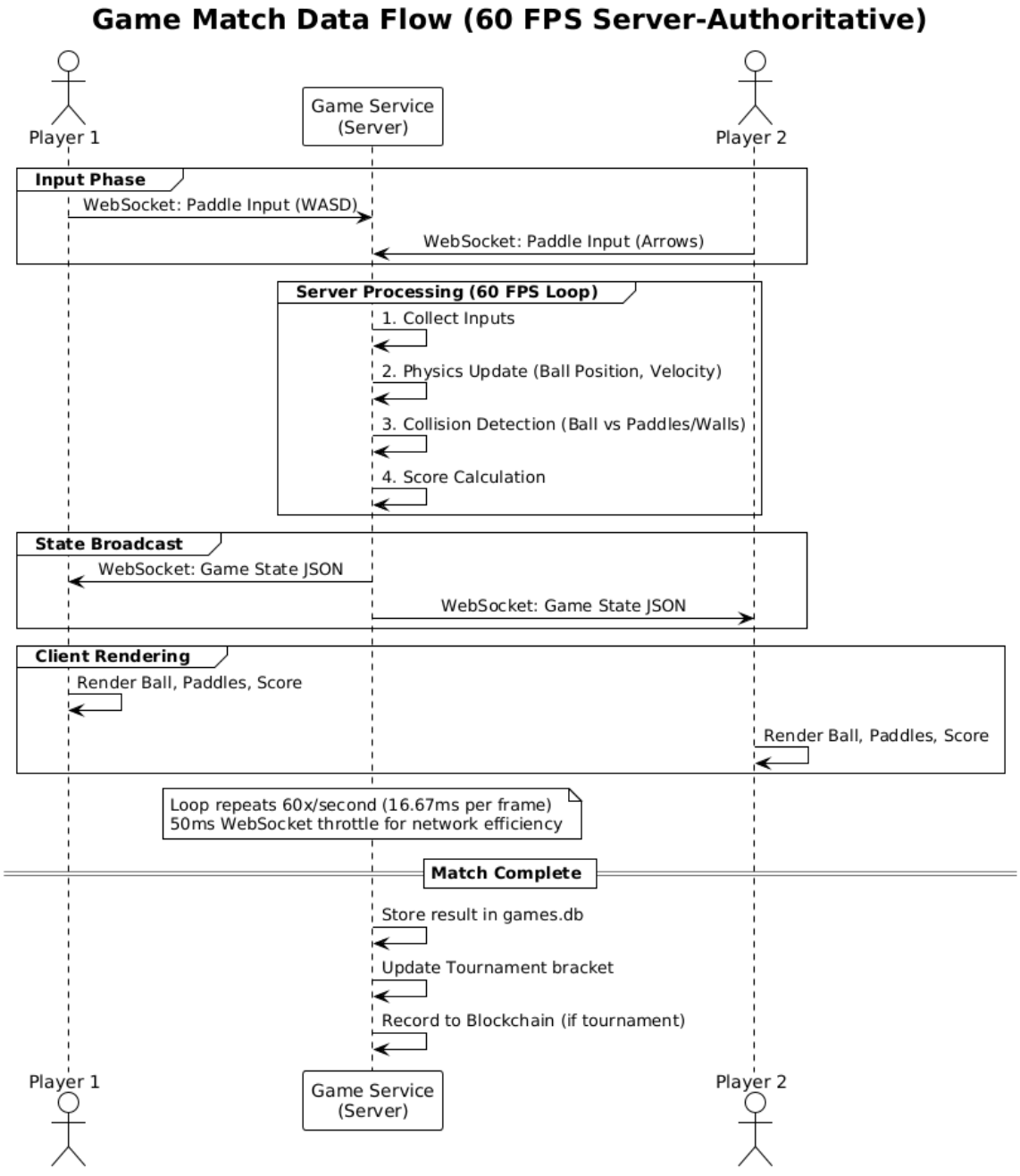

Figure A.1: Game Match Data Flow: From Player Input to Rendering and Persistence

Appendix B

Deployment & Operations

B.1 Quick Start

```
cd /mnt/d/H/42AD/ft_transcendence
make full-start        # Build and start all services
# Services available at https://localhost
```

B.2 Service URLs

- **Frontend SPA:** https://localhost:8443
- **Vault:** https://localhost:8200

B.3 Stopping Services

```
make full-stop         # Stop all containers
make full-clean        # Remove containers and volumes
```

Appendix C

Glossary

Blockchain Distributed ledger (Hardhat) for immutable tournament records

Microservices Independent services with own databases

Real-time Sync WebSocket state synchronization (16 ms intervals)

Server-Authoritative Game logic on server; clients send input only

SPA Single-Page Application; loaded once, updated via JavaScript

WAF Web Application Firewall (ModSecurity)

WebSocket Full-duplex communication protocol

Appendix D

References

[1] 42 School (2024) ft_transcendence Project Requirements (v16.1). 42 School Subject Documentation.

[2] OWASP Foundation (2024) OWASP Top 10 Web Application Security Risks. Available at: https://owasp.org/www-project-top-ten/ (Accessed: September 2025).

[3] Fastify Team (2023) Fastify Documentation. Available at: https://www.fastify.io/docs/latest/ (Accessed: September 2025).

[4] HashiCorp (2024) Vault Documentation. Available at: https://developer.hashicorp.com/vault/docs (Accessed: September 2025).

[5] Nomic Labs (2023) Hardhat Documentation. Available at: https://hardhat.org/docs (Accessed: September 2025).

[6] Trustwave (2024) ModSecurity Reference Manual. Available at: https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual (Accessed: September 2025).

[7] Microsoft (2023) TypeScript Documentation. Available at: https://www.typescriptlang.org/docs/ (Accessed: September 2025).

[8] OpenJS Foundation (2023) Node.js Documentation. Available at: https://nodejs.org/en/docs/ (Accessed: September 2025).

[9] SQLite Consortium (2023) SQLite Documentation. Available at: https://www.sqlite.org/docs.html (Accessed: September 2025).

[10] Docker Inc. (2023) Docker Documentation. Available at: https://docs.docker.com/ (Accessed: September 2025).

[11] Fette, I. and Melnikov, A. (2011) RFC 6455: The WebSocket Protocol. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc6455 (Accessed: September 2025).

[12] Dierks, T. and Rescorla, E. (2008) RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc5246 (Accessed: September 2025).