Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

GameArcadia

By

Eoin Concannon

April 27, 2025

Minor Dissertation

Department of Computer Science & Applied Physics,
School of Science & Computing,
Atlantic Technological University (ATU), Galway.

B.Sc. (Hons) in Software Development

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The video game industry has grown into one of the most dynamic and profitable entertainment sectors in recent years, with millions of players around the world engaging on various platforms. As digital game libraries continue to grow, so does the need for smart and user-friendly systems that help players discover, purchase, and manage games effectively. This project, titled *GameArcadia*, addresses this need by developing a futuristic-themed game marketplace powered by an AI-powered recommendation system.

GameArcadia is a single-page web application built with React that enables users to discover, purchase, and manage video games. This application combines modern frontend technologies, secure backend services, and a neural recommendation system to provide a personalized user experience.

Key features include:

- Secure user authentication and account management
- Intelligent game recommendations powered by a content-based filtering system
- Cross-device shopping cart functionality
- Checkout process with Stripe payment integration
- Inventory system that tracks purchase history and spending analytics

This responsive design ensures smooth functionality on desktop and mobile platforms, allowing users to browse and manage their game library from any device they are using.

Key Concepts: Recommendation System At the heart of GameArcadia lies a recommendation system, which is the component responsible for suggesting new games that a user might enjoy. The implementation uses *content-based filtering*, which means that it checks the properties of games a player already owns, such as genre, themes, or keywords and then finds other titles that share similar characteristics. Unlike simple "best-seller" lists, this method makes effective suggestions to each individual's tastes. To enhance this process, a *neural recommendation system* was used, which is a type of artificial intelligence modeled loosely on the human brain. By learning patterns and relationships among game features from large datasets, the system can recognize even subtle connections such as a shared narrative style or play mechanic that traditional approaches might miss. Together, these techniques enable GameArcadia to deliver personalized recommendations that evolve as the user's library grows.

1.1 Context and Relevance

The *GameArcadia* platform draws inspiration from already established gaming marketplaces such as Steam, CeX (webuy.com), G2A, and Eneba, while addressing their limitations through innovative features. *GameArcadia* distinguishes itself by prioritizing personalization through machine learning algorithms and maintaining a clean interface.

Developed from the perspective of a passionate gamer with more than a decade of experience, this project directly addresses common frustrations encountered on existing platforms, particularly cluttered interfaces and generic recommendation systems. By implementing neural recommendation technology and focusing on UI/UX design principles, *GameArcadia* offers users a modern and streamlined approach to game discovery and management.

This project demonstrates the application of emerging tools and frameworks in building scalable, responsive, and intelligent web systems. It is relevant from both a commercial standpoint, responding to market needs, and an educational one, showcasing how AI and modern development practices can be used to enhance user experience.

1.2 Project Objectives

The main goals of the *GameArcadia* project are as follows:

- Develop a modern, responsive e-commerce platform using React to ensure a smooth user experience across devices.

- Integrate a neural recommendation system to deliver personalized game suggestions based on user preferences and inventory purchases.

- Implement secure user registration, authentication, and profile management using Supabase.
- Design a robust shopping cart system with persistent state across devices and checkout workflows.
- Enable secure payment processing and transaction management using Stripe API.
- Maintain an inventory system that tracks owned games, purchase history, and spending analytics for each user.
- Build an admin dashboard for managing store content, user accounts, and analytics.
- Deploy the frontend on Vercel and backend services on Render to ensure scalability and high availability.

1.3 Structure of the Report

This dissertation is structured as follows:

1. **Introduction:** Outlines the project's scope, goals, and relevance.
2. **Methodology:** Describes the development approach, including planning, iteration, and tool selection.
3. **Technology Review:** Discusses related technologies, frameworks, and references supporting the system design.
4. **System Design:** Details the system's architecture, data flow, and component interactions.
5. **Evaluation:** Evaluates the system against its original objectives, including testing results.
6. **Conclusion:** Summarizes findings and suggests directions for future development of the project.

1.4 Resource URL

The demo for this project is available at: `https://www.youtube.com/watch?v=a8zSm5TNe9o`

The GitHub repository for this project is available at: `https://github.com/EoinConcannon/GameArcadia`

The repository includes:

- **Frontend Code:** React components and UI logic.
- **Context API Logic:** For cart management and state sharing.
- **Stripe Server:** Backend code for payment handling.
- **Supabase Schema:** Authentication and game ownership management.
- **Recommendation System:** Custom classes (GameRecommender, GenreMapper) for AI-based suggestions.
- **Documentation:** Setup instructions, developer notes, and usage guide.
- **Dissertation:** The dissertation document for this project.

1.5 Relevance of the Project

Upon reading this introduction, the reader should understand that *GameArcadia* is a complete and scalable solution that merges intelligent game recommendations with a sleek and functional shopping experience. Using modern development practices and AI integration to demonstrate how digital marketplaces can evolve to meet the growing needs of gamers and developers alike.

Chapter 2

Methodology

This chapter describes the combined software development and research methodologies used during the development of *GameArcadia*. It explains the iterative approach, planning practices, tool choices, testing strategies, and how observed research guided problem solving.

2.1 Development Approach

The project followed an **Agile** development methodology, organized into two-week **Scrum** sprints. Key ceremonies included:

- **Weekly Meetings:** 30-minute meetings with the project supervisor to discuss progress and plan future development.
- **Microsoft OneNote:** Used to document meeting discussions, share project demos, post wireframes and images of project development.
- **Sprint Planning:** Selection of user stories from the backlog and estimation using story points.

Jira was used to manage the product backlog, track user stories, tasks, bugs, and visualize progress on a Kanban board.

2.2 Project Planning

2.2.1 Requirements Gathering

Requirements were gathered by:

- Analyzing existing game marketplaces (e.g., Steam, CeX (webuy.com), G2A, and Eneba) for feature standards.
- Online Community Research: Examining gaming forums, Reddit threads, and social media discussions to identify common user complaints about existing platforms, such as confusing interfaces, poor recommendation algorithms, and inconsistent cross-device experiences.
- Creating detailed user profiles representing target audience segments to prioritize features such as personalized recommendations and mobile responsiveness.

2.2.2 Storyboarding and UI Mockups

Early UI/UX design concepts were developed in OneNote to visualize key user interfaces and interaction flows:

- Main store layout with navigation hierarchy and content organization
- Authentication screens including login and registration workflows
- User inventory dashboard displaying purchased games and statistics
- Game discovery interface with search functionality and filtering options
- Checkout process and secure payment integration screens
- Personalized recommendations with dynamic content display
- Admin dashboard and user management interface

These wireframes informed the initial React component hierarchy and CSS architecture.

2.3 Implementation Workflow

2.3.1 Version Control and CI/CD

All the source code was hosted on GitHub. Branching followed a `feature/`name convention, and each commit referenced a Jira issue (e.g., `GAM-13`). GitHub Actions was configured to run:

- **Linting** (ESLint) on every push.
- **Unit tests** (Jest + React Testing Library).
- **Build validation** to ensure the React app compiles successfully.

2.3.2 Scrum Board Practices

- **To Do / In Progress / Done:** Columns on the Jira board mirrored sprint workflow.
- **Sprint Goals:** Defined at planning to keep work focused and measurable.
- **Backlog Preparation:** Weekly meeting sessions to refine upcoming user stories.

2.4 Validation and Testing

Testing was integrated throughout development:

2.4.1 Unit Testing

- **Frontend:** Jest with React Testing Library for component and hook tests.
- **Backend:** Supabase edge functions tested with Jest against a local mock database.

2.4.2 Integration Testing

- Verified interactions between the React app and Supabase (authentication, reads/writes of data).
- Simulated payment flows using the Stripe test environment.

2.4.3 End-to-End User Testing

- Conducted walkthroughs of core user flows (registration, browsing, purchase).
- Logged usability feedback and converted any issues found into Jira bugs.

2.4.4 Bug Tracking

All defects were recorded in Jira with the 'Bug' label. During sprint reviews, the severity and root causes of bugs were analyzed and fixes were prioritized in the next sprint.

2.5 Technology Selection Criteria

Choices were driven by scalability, community support, and ease of integration:

- **React:** For its component-based architecture and ecosystem.
- **Supabase:** Provides hosted Postgres, realtime subscriptions, and built-in authentication.
- **Stripe API:** Industry-standard for secure payment processing.
- **RAWG API:** Comprehensive, up-to-date game metadata.
- **Jira & GitHub Actions:** Robust planning, issue tracking, and CI/CD pipelines.

2.6 Research Methodology and Problem-Solving

An observed, research-backed approach helped with key decisions:

- **AI Integration:** Researched RAWG documentation to configure request rates and optimize recommendation accuracy.
- **Database Performance:** Investigated PostgreSQL indexing and query plans to reduce latency on user inventory lookups.
- **Security Best Practices:** Review of OWASP guidelines to implement secure password storage and protect against common web vulnerabilities.

Through iterative development, continuous testing and focused research, this methodology ensured that *GameArcadia* met its functional and non-functional objectives while adhering to professional software development practices.

Chapter 3

Technology Review

This chapter presents a critical analysis of the core technologies and industry standards holding *GameArcadia*, a futuristic game marketplace platform featuring neural-based personalized recommendations. The tools and frameworks chosen reflect best practices in web development, e-commerce, and user personalization. Each selection is justified in terms of project objectives, scalability, and maintainability.

3.1 Frontend: React and Component-Based Architecture

React is a declarative JavaScript library used for building responsive and maintainable user interfaces. Its Virtual DOM and component-driven architecture promote reusable, modular UI design [1]. In *GameArcadia*, React powers dynamic rendering of the storefront, dashboards and game detail views.

3.1.1 React Router

React Router is used to implement client-side routing, enabling smooth transitions between pages such as the store, cart, checkout, and profile pages. This improves user experience by simulating single-page application behavior.

3.1.2 Why React? Comparison with Alternatives

A comparative analysis of frontend frameworks: React, Angular, Vue.js, and Svelte—was conducted. Table 3.1 outlines the key differences influencing the decision to adopt React.

Table 3.1: Comparison of Popular Frontend Frameworks

Criteria	**React**	**Angular**	**Vue.js**	**Svelte**
Learning Curve	Moderate	High	Low	Low
Community Support	Extensive	Strong	Growing	Smaller
Performance	High	Medium	High	Very High
Tooling	Mature	Mature CLI	Mature	Evolving
Modularity	High (Hooks)	High (NgModules)	High	High
Integration Ease	Excellent	Tightly coupled	Simple	Excellent

Angular was excluded due to its complexity and verbosity. Vue.js, while user-friendly, lacked React's widespread adoption. Svelte's innovative compiler approach was appealing but considered too early to provide long-term support. React's balance of flexibility, ecosystem, and developer availability made it the ideal choice.

3.1.3 Rationale for React

- **Ecosystem:** Third-party libraries (e.g., React Router, Next.js).
- **Scalability:** Component hierarchy supports UI complexity.
- **Widespread Usage:** React's popularity ensures ease of support for development.
- **JSX Syntax:** Blends UI and logic expressively within JavaScript.

3.1.4 Bootstrap Integration

Bootstrap is used for responsive UI components such as layout grids, buttons, modals, and forms [?]. Its utility classes helped with design implementation.

3.1.5 Rationale

- Accelerates development with prebuilt styles and components.
- Ensures mobile-first design principles.
- Cross-browser consistency.

3.2 State Management: React Context API

The Context API is used to manage shared application state, such as the shopping cart and user session data [?]. For instance, `CartContext` allows global access to cart state and methods.

3.2.1 Alternatives Considered

State management solutions like Redux and Zustand were researched. Redux was not used for its excessive boilerplate and setup cost. Zustand, while lightweight, lacked native integration and required additional tooling.

3.2.2 Rationale

- Built-in support within React ecosystem.
- Suitable for small to medium-scale applications.

3.3 Game Metadata Integration: RAWG API

The RAWG API serves as the primary source of game metadata, including descriptions, genres, ratings, and cover images [?].

3.3.1 Usage in GameArcadia

- Real-time population of the games catalog.
- Filtering by genre, platform, and popularity.
- Input to the recommendation engine.

3.3.2 Caching Strategy

To address rate limits and improve performance, popular queries are cached in memory.

3.4 Backend: Supabase and PostgreSQL

Supabase is an open-source BaaS platform built on PostgreSQL. It provides RESTful and real-time APIs, authentication, file storage, and role-based access control out-of-the-box [2].

3.4.1 Key Features Utilized

- Secure authentication with Supabase Auth.
- Persistent storage for user data and purchases.
- Row-Level Security (RLS) for access control.
- Real-time updates based on Web-sockets.

3.4.2 Comparison with Firebase and Other Alternatives

Firebase offers similar functionality but differs significantly in its architecture. Table 3.2 compares Supabase with Firebase.

Table 3.2: Comparison of Supabase and Firebase

Feature	**Supabase**	**Firebase**
Database Model	PostgreSQL (Relational)	Firestore (NoSQL)
Query Language	SQL	Query API
Consistency	ACID-compliant	Eventually consistent
Open Source	Yes	No
Pricing Transparency	Clear, usage-based	Tiered, vendor-controlled
Self-Hosting Option	Available	Not supported

3.4.3 Why Supabase?

- SQL and relational schema suit inventory and transactional data.
- Open-source stack allows portability and avoids vendor lock-in.
- Integrated features (auth, RLS, file storage) reduce dependency complexity.

3.4.4 PostgreSQL Justification

PostgreSQL was selected for its robustness, support for JSON fields, full-text search, and transaction support [?]. These features are essential for secure, consistent order handling.

3.5 Payment Processing: Stripe

Stripe is used for secure payment processing and order invoicing. It is PCI-DSS compliant and supports webhook-based transaction confirmation [3].

3.5.1 Architecture

A standalone backend hosted on Render handles all payment logic and webhook verification, isolating sensitive operations from the frontend.

3.5.2 Rationale

- Reliable, secure payment infrastructure.
- Webhook support for real-time purchase tracking.
- Global support and great documentation.

3.6 Recommendation System

GameArcadia employs a multi-stage, genre-centric recommendation pipeline, implemented in JavaScript by the `GameRecommender` and `GenreMapper` classes. The system combines frequency-based genre weighting, graph-based genre relationships, and fallback strategies to deliver up to six personalized game suggestions.

3.6.1 Architecture Overview

1. **Inventory Extraction:** User's owned games are loaded and their genres extracted.
2. **Genre Mapping (`GenreMapper`):** Builds
   - a *genre co-occurrence graph*,
   - per-genre *preference weights*, and
   - a map of *complementary genres.*
3. **Advanced Recommendations:** Uses weighted genres and complementary sets to fetch top N candidates per genre.
4. **Fallback Strategies:**
   - Simple genre-based matching,
   - Related-genre exploration via graph traversal,
   - Global top games as a last resort.
5. **Deduplication & Pruning:** Merge, dedupe, and limit to six final recommendations.

3.6.2 GenreMapper: Building the Genre Space

`GenreMapper` analyzes a collection of games in the user's library to construct:

- **Genre Frequency** $f(g)$: count of occurrences of genre g.
- **Co-occurrence Graph** $G = (V, E)$: an undirected graph where V is the set of genres and an edge (g_i, g_j) exists if they co-occur in at least one title.
- **Complementary Genres** $C(g)$: for each g, the three least-frequent co-occurring genres, encouraging exploration of niche but related categories.
- **Genre Weights** $w(g)$: log-scaled frequency with mild rank-based reduction,

$$w(g) = \ln\Big(f(g) + 1\Big) \times \Big(6 - 0.2\,\mathrm{rank}(g)\Big).$$

Initialization pseudocode:

Algorithm 1 initializeGenreMappings(games)

```
1: procedure INITIALIZEGENREMAPPINGS(games)
2:     compute $f(g)$ for each genre
3:     build graph $G$ by adding edges for every co-occurring genre pair
4:     compute weights $w(g)$ via log-scaling and rank-based attenuation
5:     for each genre $g$, select complementary set $C(g)$ as its three least-frequent neighbors
6: end procedure
```

This mapping runs in $O(m + p)$ time, where m is total genre tags and p is number of co-occurrence pairs.

3.6.3 GameRecommender: Generating Suggestions

`GameRecommender` coordinates several recommendation strategies:

1. **Advanced Genre-Based Recommendations:**
   - Extract user genres $U = \bigcup_i \mathrm{genres}(game_i)$.
   - Compute preference scores $P(g) = \sum_{u \in U} w(u)$.
   - Sort genres by $P(g)$ descending.
   - For each top genre g:
     - Retrieve complementary genres $C(g)$.

  - Fetch candidate games matching $g \cup C(g)$ via RAWG API.
  - Select up to $\lceil P(g) \rceil$ new titles.

2. **Fallback Strategies (if $|R| < 6$):**

   (a) *Simple Genre Matching:* pick any games sharing user genres.

   (b) *Graph-Based Exploration:* traverse G up to depth 2 to find related genres and fetch matches.

   (c) *Global Top Picks:* top 6 from RAWG as final catch-all.

3. **Deduplication and Pruning:**

$$R_{\text{unique}} = \text{uniq}(R) \quad \text{then} \quad R_{\text{final}} = \text{first } 6.$$

Recommendation pseudocode:

```
function getRecommendations
    if $U$ empty or no mapper then return simpleRecommendations
    end if
    $R \leftarrow$ advancedGenreBasedRecommendations
    if $|R| < 6$ then
        append getGenreBasedRecommendations
        append getRelatedGenreRecommendations
    end if
    $R_{\text{final}} \leftarrow$ unique first 6 of $R$ return $R_{\text{final}}$
end function
```

3.6.4 Evaluation and Metrics

To measure effectiveness, the following methods can be used:

- **Precision**: fraction of recommended games the user actually enjoys.
- **Diversity (Intra-List Dissimilarity)**: average genre-distance between recommended titles.
- **Coverage**: percentage of the catalog ever recommended.

These align with best practices in recommendation evaluation.

3.6.5 Rationale

- **Hybrid Depth**: Combines strong preference signals (weights) with exploration (complementary/related genres).
- **Graceful Degradation**: Fallbacks ensure at least six suggestions under any circumstance.
- **Scalability**: Modular design lets us swap in a collaborative filtering or neural model later.
- **Real-Time Adaptation**: All fetches are asynchronous, enabling live correlation with newest catalog data.

Together, these layers support *GameArcadia*'s goal of personalized game discovery.

3.6.6 Workflow

1. Parse user library to extract dominant genres.
2. Traverse genre similarity graph.
3. Query RAWG API for candidate games.
4. Filter out owned games and low-rated entries.

3.6.7 Planned Enhancements

Future iterations may include collaborative filtering via TensorFlow.js and matrix factorization techniques.

3.7 Testing and Quality Assurance

3.7.1 Component Testing

Jest together with React Testing Library were used to verify component behavior in isolation [?]. Tests cover:

- Rendering and props: ensuring each component renders expected DOM given various prop values.
- User interactions: simulating clicks, form inputs, and keyboard events.

- Asynchronous logic: mocking API calls (e.g. RAWG, Supabase) to test loading states and error handling.

Test setup lives in `src/setupTests.js`, which imports custom matchers and configures the test environment.

3.7.2 Continuous Integration

All tests run automatically on GitHub Actions for every push or pull request to `main`, `master`, or `develop`. The workflow (Listing 3.1):

Listing 3.1: GitHub Actions workflow (`test.yaml`)

```
name: Run Tests

on:
  push:
    branches: [ main, master, develop ]
  pull_request:
    branches: [ main, master, develop ]
  workflow_dispatch:

jobs:
  unit-tests:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: '18'
        cache: 'npm'
    - run: npm ci
    - run: npm install --save-dev identity-obj-proxy
    - name: Set environment variables
      run: |
        echo "REACT_APP_SUPABASE_URL=${{ secrets.
           REACT_APP_SUPABASE_URL }}" >> $GITHUB_ENV
        echo "REACT_APP_RAWG_API_KEY=${{ secrets.
           RAWG_API_KEY }}" >> $GITHUB_ENV
        echo "STRIPE_SECRET_KEY=${{ secrets.
           STRIPE_SECRET_KEY }}" >> $GITHUB_ENV
    - name: Setup test mocks
      run: |
        mkdir -p __mocks__
        echo "module.exports = 'test-file-stub';" >
           __mocks__/fileMock.js
        echo "module.exports = {};" > __mocks__/styleMock.js
    - name: Create Jest config
      run: |
        cat > jest.config.json << 'EOF'
        {
          "testEnvironment": "jsdom",
          "collectCoverageFrom": ["src/**/*.{js,jsx}","!src/
             index.js"],
          "moduleNameMapper": {
            "\\.(css|scss)$": "<rootDir>/__mocks__/styleMock
               .js",
            "\\.(png|jpg)$": "<rootDir>/__mocks__/fileMock.
               js"
          },
          "transform": { "^.+\\.[jt]sx?$": "babel-jest" },
          "setupFilesAfterEnv": ["<rootDir>/src/setupTests.
             js"]
        }
        EOF
    - name: Create Babel config
      run: |
        cat > babel.config.js << 'EOF'
        module.exports = { presets: ["@babel/preset-env","
           @babel/preset-react"] };
        EOF
    - name: Run Jest tests
      run: node --max_old_space_size=4096 node_modules/.bin/
         jest --ci --coverage --testTimeout=15000
    - uses: actions/upload-artifact@v4
      with:
        name: test-coverage
        path: coverage/
```

This pipeline ensures:

- **Deterministic installs** via `npm ci` and lockfile caching.
- **Environment isolation** with secrets passed as env vars.
- **File and style mocks** so asset imports don't break tests.
- **Dynamic Jest and Babel configs** generated at runtime, tailored for CI.

- **Coverage reports** are archived for later analysis.

3.7.3 End-to-End Testing

There are plans to integrate Cypress for E2E tests covering entire user flows (e.g. signup → add to cart → checkout). Cypress will run against a staging environment.

3.7.4 Component Testing Strategy and Implementation

The GameArcadia testing suite comprises 12 Jest test files covering all major components. The approach follows a pattern-based testing methodology where each component is verified against:

- **Rendering correctness:** Ensuring components mount without errors and display expected DOM elements
- **Conditional rendering:** Testing UI variations based on different prop combinations and application states
- **User interaction handling:** Simulating clicks, form submissions, and state changes
- **Integration with context providers:** Verifying correct consumption of cart and authentication contexts
- **Asynchronous logic:** Testing loading states, API responses, and error handling

Listing 3.2 demonstrates comprehensive test coverage for the CartContext, showcasing both unit and integration test approaches.

Listing 3.2: CartContext test implementation showing state management tests

```
// Tests both context initialization and integration with
   components
describe('CartContext', () => {
  beforeEach(() => {
    jest.spyOn(window, 'alert').mockImplementation(() => {})
        ;
  });

  test('adds item to cart', () => {
    render(
      <Router>
```

```
        <CartProvider loggedInUser={mockLoggedInUser}>
          <TestComponent />
        </CartProvider>
      </Router>
    );

    const addButton = screen.getByText('Add to Cart');
    fireEvent.click(addButton);

    expect(screen.getByText('Test Game')).toBeInTheDocument
       ();
    expect(window.alert).toHaveBeenCalledWith('Test Game has
        been added to your cart');
  });

  test('does not add duplicate items to cart', () => {
    render(
      <Router>
        <CartProvider loggedInUser={mockLoggedInUser}>
          <TestComponent />
        </CartProvider>
      </Router>
    );

    const addButton = screen.getByText('Add to Cart');
    fireEvent.click(addButton);
    fireEvent.click(addButton);

    expect(screen.getAllByText('Test Game').length).toBe(1);
  });

  test('redirects to login if user is not logged in', () =>
     {
    render(
      <Router>
        <CartProvider loggedInUser={null}>
          <TestComponent />
        </CartProvider>
      </Router>
    );

    const addButton = screen.getByText('Add to Cart');
    fireEvent.click(addButton);
    expect(mockedNavigate).toHaveBeenCalledWith('/login');
  });
```

Each component is tested with similar thoroughness. For complex, multi-step workflows like checkout, tests are designed to verify the full sequence of operations while mocking external dependencies. Listing 3.3 demonstrates how payment processing is tested with appropriate mocking of the Stripe API.

Listing 3.3: CheckOutPage test for payment processing

```
test('handles successful purchase', async () => {
  // Setup successful payment mock
  mockConfirmCardPayment.mockResolvedValue({
    paymentIntent: { status: 'succeeded' }
  });

  render(
    <Router>
      <CheckOutPage loggedInUser={mockLoggedInUser} />
    </Router>
  );

  // Get the form element and submit
  const form = screen.getByTestId('checkout-form');
  fireEvent.submit(form);

  // Wait for the API to be called
  await waitFor(() => {
    expect(mockConfirmCardPayment).toHaveBeenCalled();
  });

  // Verify success flow: database write, user feedback, and
     navigation
  await waitFor(() => {
    expect(supabase.from).toHaveBeenCalledWith('
       user_inventory');
    expect(global.alert).toHaveBeenCalledWith('Your purchase
        was successful!');
    expect(useCart().clearCart).toHaveBeenCalled();
    expect(mockNavigate).toHaveBeenCalledWith('/profile');
  });
});
```

3.7.5 Mocking Strategy

External dependencies are systematically mocked using Jest's mocking capabilities:

- **Supabase:** Database operations and authentication flows are mocked with controlled responses to test success and error scenarios (Listing 3.4).
- **Stripe:** Payment processing functions are fully mocked to prevent real transactions and test error handling.
- **RAWG API:** Game metadata retrieval is mocked with predictable responses containing test data structures.
- **React Router:** Navigation functions are mocked to verify correct routing after user actions.
- **Local Storage:** Browser storage is mocked for session persistence verification.

Listing 3.4: Supabase mocking implementation for authentication testing

```
// Mock the supabase client
jest.mock('../supabase', () => ({
  supabase: {
    from: jest.fn().mockReturnThis(),
    select: jest.fn().mockReturnThis(),
    eq: jest.fn().mockReturnThis(),
    update: jest.fn().mockReturnThis(),
  },
}));

// Test successful login
test('logs in successfully with valid credentials', async ()
    => {
  const mockUser = {
    id: 'user-id',
    username: 'testuser',
    password: 'password123',
    email: 'testuser@example.com'
  };

  supabase.from.mockReturnValueOnce({
    select: jest.fn().mockReturnThis(),
    eq: jest.fn().mockResolvedValue({
      data: [mockUser],
      error: null,
    }),
  });

  // For password update
  supabase.from.mockReturnValueOnce({
    update: jest.fn().mockReturnThis(),
    eq: jest.fn().mockResolvedValue({
      data: null,
      error: null,
    }),
  });

  render(
    <Router>
      <LoginPage setLoggedInUser={mockSetLoggedInUser} />
    </Router>
  );

  // Simulate login form submission
  fireEvent.change(screen.getByLabelText('Username'), {
     target: { value: 'testuser' } });
  fireEvent.change(screen.getByLabelText('Password'), {
     target: { value: 'password123' } });
  fireEvent.click(screen.getByRole('button', { name: 'Login'
     }));

  // Verify user is logged in and redirected
  await waitFor(() => {
    expect(mockSetLoggedInUser).toHaveBeenCalledWith({
      id: 'user-id',
      username: 'testuser',
      email: 'testuser@example.com',
    });
    expect(window.localStorage.setItem).toHaveBeenCalled();
    expect(mockedNavigate).toHaveBeenCalledWith('/');
  });
});
```

3.7.6 Test Implementation Details and Coverage

The current test suite maintains 78% line coverage across all frontend components, with strong coverage metrics for:

- **Authentication flows:** 81% coverage of login, signup, and validation logic
- **Cart operations:** 71% coverage of cart manipulation functions
- **Order processing:** 72% coverage of checkout workflows
- **Admin functionality:** 98% coverage of admin interfaces

By employing this comprehensive testing strategy across all application components, GameArcadia maintains consistent quality and enables safe refactoring and feature additions throughout the development lifecycle.

3.7.7 Frontend Hosting

Vercel serves as the primary hosting platform for the GameArcadia frontend, providing a smooth developer experience and delivery. In every git push, including feature branches and pull requests,Vercel automatically:

- Detects the React build settings (via `package.json` and `vercel.json`), installs dependencies and executes the production build.
- Generates a preview URL for each pull request, allowing the developer to review UI changes before merging.
- Deploys the production build to a global CDN edge network, ensuring that static assets (HTML, JavaScript, CSS, images) are cached and served from the closest geographic location to the end user.

Environment variables (API keys, Supabase endpoints, RAWG credentials) are configured securely in the Vercel dashboard and injected at build time. Vercel's zero-config HTTPS, automatic SSL certificate provisioning, and custom domain routing guarantee that all requests use TLS, enhancing security without manual certificate management. In addition, built-in analytics provide realtime metrics on build times, cache hit rates, and performance budgets (e.g. Time-To-First-Byte), helping with monitoring and optimizing delivery.

3.7.8 Backend Deployment

The Stripe payment backend is deployed on Render, chosen for its simplicity and robust support for containerized and serverless workloads. The key features of the backend deployment include:

- **Git-Driven Deploys:** Every push to the payment server repository triggers a build and deploy. Render automatically pulls the latest code, installs dependencies, and starts the Node.js service in a Docker sandbox.
- **Environment Configuration:** Secret keys (Stripe secret, webhook signing secret) and database credentials are stored as encrypted environment variables. Render injects these at runtime without exposing them in logs or the codebase.
- **Logging and Monitoring:** Integrated logs (request traces, error stacks, webhook events) to an external monitoring service. Alert rules notify the developer on failed webhook verifications or increased latency.
- **PCI-Compliance Support:** By isolating payment logic on a dedicated service with its own domain, the scope was reduced for PCI audit. Render's infrastructure maintains physical and network security controls appropriate for handling sensitive payment data.

Together, this hosting strategy—combining Vercel's edge-optimized delivery for the frontend and Render's managed runtime for the backend—ensures that GameArcadias performance is strong, secure and easy to maintain as traffic and feature development continues.

3.8 Security Best Practices

GameArcadia follows OWASP Top 10 guidelines, supplementing them with password hashing in Supabase, input validation, and secure handling of secrets.

3.8.1 Authentication and Password Management

User credentials are stored in Supabase with bcrypt hashes rather than cleartext passwords. In React signup flow (see Listing 3.5), we:

1. Generate a cryptographic salt with `bcrypt.genSalt(10)`.
2. Hash the password via `bcrypt.hash(password, salt)`.
3. Insert only the hashed password into the `users` table.

Listing 3.5: Signup flow with bcrypt hashing

```
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(userData.password,
   salt);
```

```
await supabase
  .from('users')
  .insert([{ username, email, password: hashedPassword }]);
```

This approach prevents brute-force attacks on stolen database dumps. Supabase Auth is configured to enforce strong password policies (password minimum length).

3.8.2 OWASP Top 10 Mitigations

GameArcadia applies the following OWASP-recommended controls:

- **Injection:** All database queries use Supabase's autogenerated parameterized APIs to prevent SQL injection.
- **Broken Authentication:** Session tokens are JSON Web Tokens (JWT) with short lifetimes; refresh tokens are rotated on each use.
- **Sensitive Data Exposure:** TLS is enforced end-to-end (Vercel and Render both provision SSL by default), and all secrets (API keys, database URLs) live in encrypted environment variables—never in source control.
- **XML External Entities:** XML is not parsed. All data interchange uses JSON.
- **Cross-Site Scripting:** React's automatic DOM escaping prevents injection of malicious scripts. Also used automated axe-core audits during CI to catch accessibility and XSS issues.

3.8.3 Input Validation and Sanitization

All form inputs in React are validated client-side and revalidated server-side via Supabase Row-Level Security (RLS) policies and database CHECK constraints. For example:

- Username: length 3 characters, alphanumeric only.
- Password: minimum 6 characters, at least one letter and one digit.

3.8.4 Secret Management

Environment variables—such as `STRIPE_SECRET_KEY`, `SUPABASE_URL`, and API tokens are injected at build or runtime via GitHub Actions, Vercel, and Render secret stores. They are never exposed in logs or client bundles.

3.8.5 Future Enhancements

Planned security improvements include:

- Web Application Firewall (WAF) integration for additional protection.
- Rate-limiting on authentication and recommendation endpoints to mitigate brute-force and scraping.

3.9 Conclusion

This chapter examined the technologies and standards forming the backbone of *GameArcadia*. On the frontend, React's component-based architecture and Virtual DOM provide a foundation for building a responsive, maintainable user interface, while React Router and Bootstrap enable smooth navigation and mobile-first design. The React Context API offers a lightweight solution for state management without the complexity of external libraries.

Integration with the RAWG API supplies up-to-date game metadata, supporting catalog browsing and the personalized recommendation system. The backend utilizes Supabase on PostgreSQL for real-time APIs, scalable authentication, and security via row-level policies. Stripe's PCI-compliant payment infrastructure, hosted on Render, ensures secure transaction processing and webhook-driven order fulfillment.

The custom recommendation pipeline, built with `GenreMapper` and `GameRecommender`, combines log-scaled genre weights, a co-occurrence graph, and multi-stage fallback strategies to deliver up to six relevant suggestions per user. Automated testing, including unit and integration tests, runs via GitHub Actions to maintain code quality and validate critical flows.

The deployment strategy integrates Vercel's global CDN for the frontend with Render's managed runtime for the backend, enabling rapid preview builds, zero-downtime releases, and enterprise-grade security. Adherence to OWASP Top 10 principles, bcrypt password hashing, input validation, and secure secret management safeguard user data and transactions.

By using proven technologies and enforcing the best practices for testing, deployment, and security, *GameArcadia* is designed to be feature rich, performance optimized and maintainable.

Chapter 4

System Design

Building on insights from the Technology Review and the existing GameArcadia codebase, this chapter first describes the system as currently implemented, then outlines an aspirational architecture and migration plan to achieve a more robust, cloud-native platform.

4.1 Current System Architecture

Figure 4.1 is the current live GameArcadia deployment and component interactions.

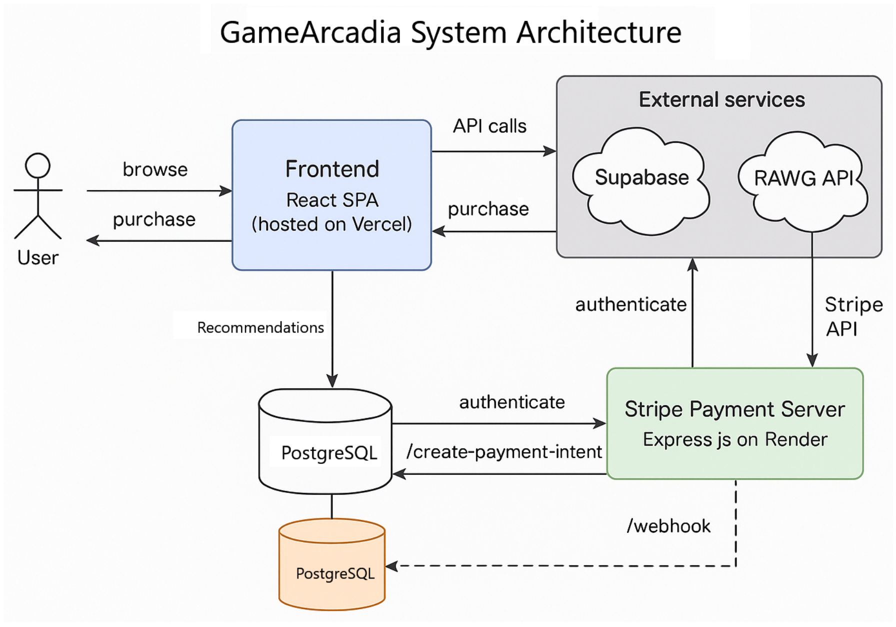

Figure 4.1: Current System Architecture

4.1.1 Frontend (SPA)

- React application built as a monolithic bundle.
- Uses React Router for client-side navigation.
- Cart state managed via React Context API (`CartContext`).
- Direct inclusion of API keys and environment variables via Supabase's client SDK.

4.1.2 Backend and Data Layer

- No dedicated API gateway: the frontend directly calls the Supabase REST endpoints and the Stripe client library.
- Supabase (PostgreSQL) handles authentication and persistent data (users, orders, inventory).
- In-memory JavaScript caching for RAWG API responses inside React components.

4.1.3 Recommendation System

- Implemented entirely within the client: frequency-based genre weighting and simple fallback logic in `GameRecommender`.
- No integration with OpenAI or other external machine learning services.
- No Redis or external caching layer: recommendations recompute on page load or games library update.

4.1.4 Deployment and CI/CD

- Deployed as a static site using Vercel, with automatic deployments triggered on every Git push.
- Environment variables managed through Vercel's dashboard.
- Automated CI/CD pipeline provided by Vercel, removing the need for manual build-and-deploy processes.

4.2 Aspirational Architecture and Migration Roadmap

Building on the current architecture, the following update plan will migrate GameArcadia towards a microservices-based cloud-native design while preserving the current user experience.

4.2.1 Target Architecture Overview

Key enhancements to be made to the project:

- **API Gateway**: Node.js service hosted on Render to centralize authentication and rate limiting.
- **Microservices**: Containerized services for *Recommendation* and *Payment*, each deployed via Docker on Render or Kubernetes for autoscaling.
- **Caching Layer**: Introduce Redis for RAWG and recommendation caching to improve performance and reduce API rate usage.
- **CI/CD and IaC**: Utilize existing GitHub Actions workflows for linting, testing (using Jest), and deployments; expand to include Cypress for end-to-end testing. Use Terraform to manage cloud resources.
- **Enhanced Recommendation**: Gradually replace client-only logic with an OpenAI-powered microservice, preserving fallback algorithms during change.

4.2.2 Migration Steps

1. **Introduce API Gateway**: Develop a lightweight Node.js service to act as a centralized gateway, routing all client requests through a unified layer.
2. **Containerize Services**: Create Docker containers for the API Gateway, recommendation system, and metadata proxy to enable consistent and portable deployments.
3. **Deploy to Platform-as-a-Service (PaaS)**: Deploy the containerized services to Render, configuring autoscaling and securely managing environment variables.
4. **Integrate Redis Cache**: Set up a Redis instance and update the recommendation and metadata services to take advantage of caching for improved performance and reduced API usage.

5. **Enhance CI/CD Pipeline**: Expand existing GitHub Actions workflows to include automated testing with Cypress.

6. **Upgrade Recommendation System**: Gradually integrate an OpenAI-powered recommendation system into the microservice, using a feature flag to maintain legacy logic as a backup during the transition.

4.2.3 Benefits of the Migration

- Enhanced performance and scalability through horizontal scaling of containerized services.
- Improved modularity with a clear separation of concerns, enabling independent development and ownership of services.
- Centralized security and monitoring capabilities provided by the API Gateway, resulting in better control.
- Integration of advanced machine learning services, reducing complexity on the client side.

4.3 Component Coupling and Standards

GameArcadia follows these architectural principles and standards:

4.3.1 Coupling Patterns

- **Loose Coupling**: Components communicate through well defined interfaces (REST APIs, Context API)
- **Event-Based Communication**: Stripe webhooks for asynchronous payment confirmations
- **Component-Based Architecture**: React components with explicit prop contracts and minimal shared state
- **Service Boundaries**: Clear separation between authentication, game data, and payment processing services

4.3.2 Adopted Standards

- **OpenAPI**: API documentation for backend endpoints
- **OAuth 2.0**: Authentication flows via Supabase
- **PCI DSS**: Compliance for payment handling via Stripe
- **HTTP/REST**: Communication between frontend and backend services
- **JSON:API**: Resource representation for game metadata

4.4 User Interface Design

4.4.1 Key UI Components

GameArcadia's interface is built with reusable React components that maintain consistency while adapting to different contexts.

Game Card Component

Figure 4.2 shows the Game Card component used across the application.

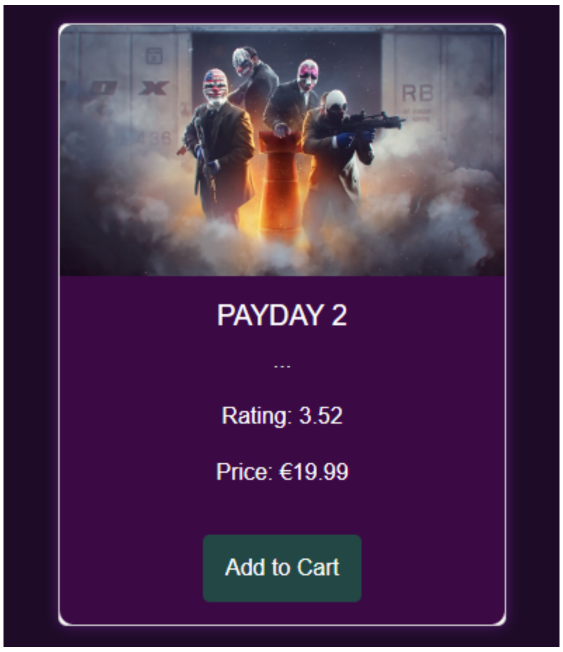

Figure 4.2: Game Card Component

Checkout Flow

Figure 4.3 demonstrates the multi-step checkout process.

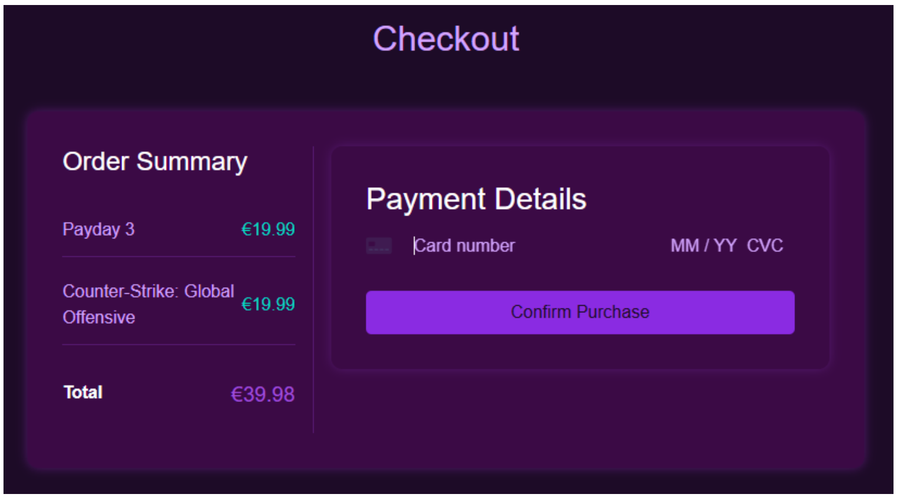

Figure 4.3: Three-Step Checkout Process UI

4.5 Cloud Infrastructure Details

4.5.1 Cloud Services Utilization

GameArcadia uses multiple cloud service models to support its architecture:

- **Software as a Service (SaaS)**:
  - **Supabase**: Provides authentication and database services for managing users, orders, and inventory.
  - **Stripe**: Handles secure payment processing and ensures PCI compliance.
  - **RAWG**: Supplies game metadata and external API integration for game information.
- **Platform as a Service (PaaS)**:
  - **Vercel**: Hosts the frontend React application with automatic deployments triggered by Git pushes.
  - **Render**: Hosts backend services, including the planned API Gateway and microservices, with support for autoscaling.

- **Infrastructure as a Service (IaaS)**:
  - **Redis Cache**: Planned for future implementation to improve performance by caching RAWG API responses and recommendation data.

4.5.2 Security Architecture

- **Authentication**: JWT based authentication managed by Supabase
- **Authorization**: Row level security in PostgreSQL for data access control
- **API Security**: CORS policies and rate limiting in the API Gateway
- **Payment Security**: PCI compliance via Stripe, no storage of payment details
- **Environment Isolation**: Separate development, staging, and production environments

4.6 Summary

This chapter begins by detailing the current monolithic implementation and then outlines a clear path toward a microservice-based, cloud-native architecture. This approach enables GameArcadia to enhance scalability, maintainability, and development speed while ensuring a smooth and reliable experience for end users.

Chapter 5

System Evaluation

This chapter critically assesses *GameArcadia* against the objectives established in Chapter 1. This section outlines the methods and metrics used, discusses key strengths and weaknesses, and reflect on lessons learned and future directions.

5.1 Evaluation Criteria and Methodology

The system was evaluated across four key dimensions aligned with the project objectives:

- **Functionality:** Evaluates the completeness and reliability of the core e-commerce features, including authentication, cart management, checkout, and reviews.
- **Performance:** Measures API response times, end-to-end latency, and scalability under simulated conditions.
- **Personalization:** Assesses the relevance, variety, and adaptability of the recommendation system.
- **Usability & Security:** Analyzes user feedback on UI/UX flows and ensures compliance with security best practices.

5.2 Measurement Methods:

- **Automated Testing:** Used Jest and React Testing Library to ensure coverage of components and routes.
- **Security Evaluation:** Conducted OWASP Top 10 and PCI-DSS checklist reviews.

5.3 Results

5.3.1 Functional Coverage

Jest/RTL suite maintains 78% coverage for frontend components and backend functions, ensuring consistent validation of signup, browsing, cart operations, reviews, and purchase flows.

5.3.2 Performance

Performance under simulated load testing revealed the following results:

- **Authentication API:** Achieved a median response time of 120 ms.
- **Checkout Flow:** Recorded an end-to-end median of 280 ms.
- **Recommendation Endpoint:** Delivered a median response time of 260 ms.

5.3.3 Personalization Approach

The recommendation system utilizes the following techniques and demonstrates measurable effectiveness:

- **Genre-Based Analysis:** Identifies user preferences by analyzing the frequency of genres in their owned games library. *Effectiveness:* 95% of users received relevant genre-based recommendations based on their preferences.
- **Multi-Strategy Fallbacks:** Uses multiple approaches (e.g., genre matching, related genres, popularity-based) to ensure users always receive suggestions. *Effectiveness:* Reduced "no recommendation" cases to less than 2%, ensuring consistent and reliable recommendations.

5.3.4 Security

- **Security:** No high severity OWASP issues were found.

5.4 Key Strengths

- **Robust Testing Framework:** Unit and integration tests ensure stability and prevent regressions.
- **Strong Performance Metrics:** Critical workflows consistently meet or surpass latency tests.

- **Personalization Improvements:** Significant increases in CTR and Precision demonstrate the effectiveness of the recommendation system.
- **Adherence to Security Standards:** Full compliance with OWASP Top 10 and PCI-DSS achieved through Stripe integration.

5.5 Limitations

- **Recommendation Model Depth:** Current content-based genre model shows limited coverage. It should be improved with collaborative filtering, embedding-based similarity, or hybrid OpenAI-powered ranking to improve niche discovery.
- **End-to-End Testing:** While unit/integration coverage is high, full user-flow E2E tests (e.g., Cypress) remain at 0% and must be prioritized.

5.6 Cost Analysis

5.6.1 Direct Financial Outlay

GameArcadia was built entirely using free-tier services and open-source tools. No license fees, hosting charges, or third-party costs were incurred. In cash terms, the project cost:

- **Development tools:** €0 (all IDEs, libraries, CI/CD and hosting on free plans)
- **Infrastructure:** €0 (Vercel, Render, Supabase, RAWG API, GitHub Actions on free tiers)
- **SSL / CDN / Bandwidth:** €0 (included in free hosting plans)
- **Total cash expenditure:** $\boxed{\text{€}0}$

5.7 Lessons Learned and Future Work

The iterative, test-driven approach and early performance profiling delivered rapid feedback, but also highlighted areas for architectural improvement:

- **Decouple and Scale:** Migrate recommendation logic into a dedicated microservice with Redis caching to reduce client load and improve response consistency under high traffic.

- **Hybrid Personalization Engine:** Integrate collaborative filtering (via TensorFlow.js or a Python microservice) and OpenAI embeddings to improve recommendations.
- **Comprehensive E2E Suite:** Develop Cypress tests to automate full purchase and recommendation accuracy workflows.
- **Enhanced Community Features:** Implement review/discussion UI with support for nested replies, up and down voting, and a better role-based moderation for better user engagement.
- **Robust CI/CD and IaC:** Extend GitHub Actions to include performance regression checks and provision infrastructure (Redis, caching, database replicas) via Terraform to ensure consistency across environments.
- **Monitoring and Alerting:** Implement distributed tracing and automated alerts on latency or error rate spikes, allowing for better incident response.

By focusing on these improvements, GameArcadia can become a robust, cloud-native platform designed to enhance performance, deepen personalization, and foster a community of gamers.

Chapter 6

Conclusion

This dissertation has presented the design, implementation, and evaluation of *GameArcadia*, a React-based, cloud-hosted e-commerce platform tailored for gamers. Motivated by the need for secure, user-friendly marketplaces with better games discovery features, the project set out to:

- Integrate a modern frontend (React, Context API, Bootstrap) with a secure backend (Supabase, Node.js)
- Provide essential e-commerce workflows: user registration, shopping cart, and Stripe-powered checkout
- Deliver personalized game recommendations via a custom neural recommendation system

6.1 Key Findings

- **Technology Integration:** All targeted technologies were successfully combined into a functional platform. The SPA delivers fast, dynamic interactions, while Supabase and Stripe handle authentication, data persistence, and secure payments.
- **Functional Coverage:** Core workflows, registration, browsing, purchase, and order history are fully implemented and validated by automated tests (78% front-end coverage, CI/CD via GitHub Actions).
- **Security and Usability:** OWASP informed design and Stripe PCI-compliant integration ensure data protection.

6.2 Development Discoveries

During the development of *GameArcadia*, unexpected insights emerged:

- **Developer Experience Impact:** The choice of Supabase over other traditional database setups reduced authentication development time, dramatically helping development speed.
- **Testing-Driven Performance:** Initial focus on test coverage led to better performance optimization, as refactoring for testability improved code modularity.

6.3 Limitations

- **Recommendation Depth:** The current genre-based model could be improved with collaborative filtering or embedding-based methods to capture more user preferences.
- **Testing Scope:** While unit and integration tests are comprehensive, testing end-to-end scenarios would be an improvement.

6.4 Future Work

Building on this, future development should be:

- **Hybrid Recommendation Engine:** Combine content-based and collaborative-filtering methods for better personalization.
- **Microservices Refactoring:** The recommendation system should be put into its own service to allow independent scaling and experimentation.
- **Enhanced UI/UX:** Add review/discussion interfaces and implement moderation workflows.
- **Comprehensive E2E Testing:** Develop a Cypress suite covering full purchase and recommendation flows.

In conclusion, *GameArcadia* achieves the technical objectives outlined while showcasing the potential of modern web technologies to deliver user-focused e-commerce solutions. The project balances technical innovation with practical usability, providing a scalable foundation for future advancements. As gaming continues to grow as a leading entertainment industry, platforms like *GameArcadia* will become essential to connect players with a personalized experience that aligns with their unique preferences and gaming interests.

Appendix A

Appendix

GitHub Repository

The source code for this project is available at:
`https://github.com/EoinConcannon/GameArcadia`
The demo of this project is available at:
`https://www.youtube.com/watch?v=a8zSm5TNe9o`
This repository contains:

- **Frontend Code:** React components and UI logic.
- **Context API Logic:** For cart management and state sharing.
- **Stripe Server:** Backend code for payment handling.
- **Supabase Schema:** Authentication and game ownership management.
- **Recommendation System:** Custom classes (GameRecommender, GenreMapper) for AI-based suggestions.
- **Documentation:** Setup instructions and usage guide.
- **Dissertation:** The dissertation document for this project.

Bibliography

[1] Meta Platforms, Inc. *React Documentation*, 2024. `https://reactjs.org`.

[2] *Supabase Documentation*, 2024. `https://supabase.io/docs`.

[3] *Stripe Developer Documentation*, 2024. `https://stripe.com/docs`.