• 2025 •

ARCHITECTURAL SPECIFICATIONS

GAMIFIED FINANCIAL VISUALIZER

For

Demo 1

Presented by :

CodeBlooded

Introduction

The Gamified Financial Visualizer is an interactive platform designed to help users manage and visualize their finances engagingly and educationally while promoting financial literacy. This document outlines the architectural design of the system, with a focus on key quality attributes including performance, usability, security, reliability, availability, scalability, maintainability, compatibility, and legal compliance. Additionally, it addresses the system's evolution from a traditional client-server model toward a scalable microservices architecture to support future enhancements.

Business Needs

The architectural strategy of the Gamified Financial Visualizer is aligned with the platform's core mission: to provide users with an engaging, educational, and secure environment for managing their finances. The system is designed not only to satisfy functional requirements but also to support long-term business objectives such as user trust, continuous access, and future growth. The following business needs serve as the foundation for all architectural decisions:

Security
The system safeguards financial and personal data through multi-factor authentication, encryption, and strict access control, while complying with regulations such as GDPR and POPIA.

Usability
Interfaces are designed to be intuitive and accessible for users with different levels of financial literacy. AR and gamified features follow Material Design and Apple HIG standards for a smooth user experience.

Availability
Using cloud infrastructure, load balancing, and fault-tolerant design, the system ensures high uptime and responsiveness, even during peak usage.

Scalability & Extensibility
A microservices and container-based architecture allows the platform to scale efficiently and integrate new features (e.g., AI modules or AR tools) without downtime.

Maintainability & Agility
Modular code, clear documentation, and CI/CD pipelines enable rapid development, updates, and reduced technical debt, allowing the system to evolve continuously.

Project Scope

This project is being developed as part of a university capstone course and is scoped to balance innovation with practicality, targeting a functional and demonstrable MVP (Minimum Viable Product) within academic timelines. The following elements define the scope of this system:

Financial Tracking & Visualization

- Users can link their financial data (via third-party aggregators like Stitch).
- The system will process and visualize transactions using dynamic dashboards and 3D/AR environments.
- AR-based interaction allows users to view financial "worlds" like savings islands or budget trees.

Gamification Features

- Reward systems for hitting financial goals.
- Points and achievements for completing educational modules and maintaining budget discipline.
- Community features such as leaderboards and challenges.

AI-Powered Financial Insights

- Transaction classification using NLP (DistilBERT).
- Personalized financial advice delivered through a conversational interface (local LLM such as LLaMA 3).
- Goal recommendations and behavioral nudges based on user profiles.

Educational Modules

- Interactive, gamified learning content tailored to financial literacy topics.
- Integrated quizzes with point-based rewards.

Augmented Reality Experience

- Visualization of financial status and progress through Unity-powered AR scenes.
- Real-time interaction with assets like savings vaults or debt mountains via mobile devices.

Cross-Platform Support

- Web access via React.js.
- Mobile support via React Native and Unity AR Foundation for Android and iOS.

Security & Compliance

- Implementation of secure authentication, encryption, and anonymization protocols.
- Compliance with data protection regulations (e.g., POPIA, GDPR).

Quality Requirements

Performance

The platform is designed for high performance, ensuring fast, responsive user experiences—even under heavy load or during intensive AR interactions. Key strategies include:

- **Load Balancing:** Distributes traffic across servers to prevent overload and maintain responsiveness.
- **Microservices and containers:** Enable horizontal scaling and efficient resource management for AI and AR services.
- **Cloud-based GPU-accelerated Computing:** Speeds up AR asset rendering and reduces client-side CPU load by offloading processing to cloud-based GPUs.
- **Caching:** Decreases latency by storing frequently accessed data at multiple points in the system.
- **Model Optimization:** Improves AI efficiency through model pruning and tuning, maintaining accuracy with lower computational cost.
- **Throttling:** Protects services from overload by controlling request rates during peak demand.

Stimulus Source	Stimulus	Response	Response Measure	Environment	Artifact
1000 users	Initiate 200 requests	Process all requests efficiently	Average latency of 4 seconds	Heavy load	System
Cloud rendering AR object	Create a large AR file	Load the display while rendering the AR object from the cloud	Average latency of 5 seconds	Heavy GPU usage	System

Quality Requirements

Usability

The platform prioritizes an intuitive, accessible, and responsive user experience across all interfaces, especially in AR interactions, ensuring smooth navigation, clear feedback, and inclusivity for all user types.

- **UI/UX Framework:** Modular frontend (React.js/Flutter) with ARCore/ARKit integration. Consistent design using Material Design and Apple HIG, with progressive disclosure to reduce complexity.
- **AR Interaction Engine:** Built with Unity or Unreal Engine, targeting 60 FPS via LOD optimization. Supports intuitive gestures and haptic feedback for natural, responsive AR interactions.
- **Accessibility Layer:** Fully WCAG 2.1 AA-compliant UI, including semantic HTML, dynamic contrast, scalable fonts, and full screen reader support (VoiceOver/TalkBack).
- **Context-Aware AI:** AI responses adapt to user profiles, providing simplified, goal-relevant financial guidance using GPT-4 with jargon filtering.
- **Error Handling System:** Centralized monitoring (e.g., Sentry), localized and user-friendly error messages, input validation, and multilingual support.

Stimulus Source	Stimulus	Response	Response Measure	Environment	Artifact
100 new users	Complete AR onboarding	95% success rate < 5 minutes	95% success rate < 5 minutes	Normal operation	AR tutorial UI
Screen reader	Navigate dashboard	All elements read in logical order	100% WCAG 2.1 AA pass rate	Audit testing	UI components
Novice user	Ask, “How can I save more?	Receive plain-language advice (<100 words)	≥4.5/5 clarity rating	Production	AI advisor

Quality Requirements

Security

Given the sensitivity of financial data, the platform enforces strong security measures to protect user privacy and build trust.

- **Multi-factor Authentication:** Adds a second layer of protection to prevent unauthorized access.
- **Data Encryption:** All personal and financial data is encrypted at rest (AES-256) and in transit (TLS 1.3+), ensuring secure communication and storage.
- **Privacy via Anonymization:** User identities are anonymized when interacting with third-party services, complying with GDPR and POPIA.
- **Restricted Data Access:** External data providers have read-only access, preventing any modification of user account data and minimizing risk of misuse.

Stimulus Source	Stimulus	Response	Response Measure	Environment	Artifact
Authenticated user	Login request for financial dashboard	Prompt and validate MFA credentials	Grant access if MFA succeeds within 5 seconds	Normal operation	Auth & auth module
Unauthorized user	Login attempt with incorrect details	Validate credentials, deny access, prompt retry	Access denied, user prompted to retry	Normal operation	Auth & auth module
Third-party API	Sends user financial transactions	Encrypts & logs data with read-only access, anonymizes user info	Data stored securely with anonymization	Normal operation	System

Quality Requirements

Reliability & Availability

To ensure user trust and platform reliability, the architecture is designed for data integrity, resilience, and minimal downtime, especially during real-time financial tracking and AR rendering.

- **Event Sourcing + Write-Ahead Logging:** Ensures every user transaction is logged and recoverable even during partial failures.
- **Backup and Disaster Recovery:** Automated, encrypted backups stored across regions (e.g., AWS, GCP) for quick recovery.
- **Dedicated AR Rendering Service with CDN Edge Caching:** Offloads static AR assets to edge servers
- **Concurrency Throttling + Graceful Degradation:** If resources are strained, lower LOD (Level of Detail) assets are served.
- **Failover Clusters + Multi-Region Deployments:** Ensures services stay available even in case of regional outages.

Stimulus Source	Stimulus	Response	Response Measure	Environment	Artifact
User	Accesses AR gamified financial view	System allocates GPU-backed session and loads assets via CDN	Render response > 3 seconds	Normal operation	AR Rendering Engine + CDN
User	Attempts to access app or dashboard	Traffic routed to healthy instance by load balancer	Uptime maintained >= 99.9%	Normal usage	Load balancer + Service Cluster

Quality Requirements

Maintainability

To support ongoing enhancements and minimize rework, the system is designed for high maintainability, ensuring smooth evolution over time without compromising stability.

- **Modular Architecture:** The system will adapt to solid coding principles and implement loose coupling between components using dependency injections to make the code more flexible, testable, and maintainable.
- **Comprehensive Documentation:** All code will include JSDoc/Doxygen comments for functions and classes.
- **Automated testing frameworks:** The system will maintain code coverage through unit tests, integration tests, and end-to-end testing. A continuous integration pipeline will enforce test passes before code merges.
- **CI/CD practices:** Enables fast, reliable delivery of updates through continuous integration and deployment workflows.

Stimulus Source	Stimulus	Response	Response Measure	Environment	Artifact
User	Experiences bug in the UI interaction eg. dashboard fails to display on screen	Developer locates the issue, fixes code and deploys.	Bug identified and UI fixed without backend modifications	Production	Frontend components

Quality Requirements

Scalability

As user numbers grow, our architecture must scale effortlessly. The gamified and AR-heavy nature of the application demands responsive scaling without impacting system performance or user experience.

- **Microservices Architecture + Containerization:** Independent scaling of services using Docker + Kubernetes.
- **Event-Driven Architecture (EDA):** Kafka or RabbitMQ ensures asynchronous processing for high-throughput workloads.
- **Stateless Services with Shared Cache Layer:** Promotes scalability without session stickiness (using Redis/Memcached).
- **Indexed & Partitioned Tables:** PostgreSQL optimized using indexing and partitioning on frequently queried columns (e.g., date, user_id).
- Async Loading with Progressive Rendering: Render sections of the dashboard as data is fetched.

Stimulus Source	Stimulus	Response	Response Measure	Environment	Artifact
User	Opens dashboard	Cached or pre-aggregated data served instantly	<500ms query response time	High load	Dashboard Analytics Service
Backend Service	Generates new financial insights	Materialized views updated asynchronously	Query times not impacted	Normal operation	PostgreSQL + Redis

Quality Requirements

Compatibility

To ensure a seamless user experience across mobile, tablet, and desktop, the platform is built with cross-platform compatibility in mind, enabling consistent performance and visuals on all devices.

- **Blender with Three.js:** Renders immersive 3D financial models directly in browsers and apps, simulating AR without headsets.
- **React for web and React Native:** For mobile, enable a shared component model, allowing significant reuse of logic across platforms while providing native-like user experiences.
- **Responsive design:** principles ensure that UIs adapt dynamically to various screen sizes and orientations (mobile, tablet, desktop).
- **An API-first:** REST/GraphQL APIs allow consistent access to business logic across platforms.
- **Platform detection logic and conditional rendering:** Used to load optimized components depending on the user's device capabilities, ensuring performance and UI consistency.

Stimulus Source	Stimulus	Response	Response Measure	Environment	Artifact
User	Opens system on mobile	Loads responsive UI with 3D elements	Loads in <3 seconds	Mobile app	React Native UI + 3D models (3js)
User	Accesses via desktop browser	Displays full dashboard with 3D rendering	Desktop browser	Desktop browser	React UI + Three.js canvas
Developer	New platform introduced	System adapts with minimal changes	Support added within 1 week	Cross-platform dev	Shared React + 3D asset layer

Quality Requirements

Legal & Compliance

To ensure legal, ethical, and regulatory compliance, the system incorporates strict controls for financial advice, data accuracy, and long-term audit readiness.

- **Disclaimer injection logic:** AI-generated financial responses automatically include standard legal disclaimers.
- **A precision-focused computation layer:** Uses validated libraries (e.g., decimal.js, math.js) and testing to maintain <1% error margin in financial predictions.
- **Audit-ready logging mechanisms:** Structured, timestamped logs capture all AI/user actions, ensuring traceability.
- **Logs are retained for a minimum of 7 years:** Periodic validation of integrity and secure purging of older entries via scheduled jobs or cron-like functions in the backend infrastructure.

Stimulus Source	Stimulus	Response	Response Measure	Environment	Artifact
User	Receives AI-generated financial advice	System includes mandatory disclaimer at the end of the response	Disclaimer included 100% of the time	AI interaction context	Response rendering component
System Tester	Evaluates accuracy of financial prediction engine	System outputs results within acceptable error margins	<1% average error over test dataset	Backend model validation	Financial prediction module
Scheduled Job	Time-based data retention triggers	Logs older than 7 years are purged	No pre-2018 logs exist; system	Backend task scheduler	Retention enforce

Architectural Patterns

The architecture of the Gamified Financial Visualizer adopts a combination of modern, modular, and scalable design patterns to ensure the system is maintainable, high-performing, secure, and adaptable to future growth.

Microservices

Each functional subsystem is encapsulated as a self-contained service. Key services include:
Authentication & User Profile

- Finance Manager
- AI Classification & Insights
- AR Visualization Engine
- Social Layer
- Financial Goal Engine
- Learning Engine

Benefits & NFR Alignment:

- **Maintainability:** Clear separation of concerns simplifies debugging and updates.
- **Scalability:** Services scale independently based on user demand.
- **Reliability & Availability:** Service isolation prevents cascading failures.
- **Security:** Fine-grained access control and surface area isolation.
- **Compatibility:** Services can evolve without breaking dependencies.

Model View-View Model

The MVVM pattern helps cleanly separate an application's business and presentation logic from its user interface (UI). Maintaining a clean separation between application logic and the UI helps address numerous development issues and makes an application easier to test, maintain, and evolve.

Used primarily for the React-based frontend, MVVM separates the UI logic (View), business/data logic (ViewModel), and raw data (Model). This is also adaptable in Unity via ViewModel scripts that handle AR interactions.

Benefits & NFR Alignment:

- **Maintainability:** Easier to test and update logic independently of UI.
- **Usability:** More responsive and dynamic UIs by cleanly binding logic to views.
- **Compatibility:** Modular UI logic works across different devices and resolutions.

Architectural Patterns cont.

Pipe and Filter

In system design, the Pipe and Filter architecture structures processing tasks into a sequence of stages known as "pipes," where each stage filters and transforms data incrementally. This modular framework enables filters to operate independently, improving scalability and reusability. Each filter is tasked with specific operations, such as validating or formatting data, and passes its output to the next stage via interconnected pipes.

Applied to:

- **AI Processing Pipeline** (classification, scoring, RAG)
- **AR Rendering Pipeline** (data transformation → 3D object generation)

Each stage is modular and responsible for one transformation.

Benefits & NFR Alignment:

- **Performance:** Each step can be optimized individually.
- **Reliability:** Filter stages include validation and fallback logic.
- **Maintainability:** Easily test, log, and swap components within pipelines.
- **Legal & Compliance:** Allows filters like disclaimer injection to be consistently applied

---

Event-Driven Architecture

With event-driven architecture (EDA), various system components communicate with one another by generating, identifying, and reacting to events. These events can be important happenings, like user actions or changes in the system's state. In EDA, components are independent, meaning they can function without being tightly linked to one another.

System behavior is driven by events ("goal achieved", "expense logged", "quiz completed") broadcast to subscribed services like notifications, AI insights, and AR updates.

Benefits & NFR Alignment:

- **Scalability:** Loose coupling makes it easier to add new reactive services.
- **Reliability:** Non-blocking event queues improve fault tolerance and retry capabilities.
- **Usability & Real-Time Feedback:** Users receive timely updates and responses.
- **Maintainability:** Adding new reactions (reward logic) doesn't require modifying the event source.

Architectural Patterns cont.

Cache Aside

The Cache-Aside Pattern improves performance by checking the cache before querying the database. On a cache hit, data is used directly; on a cache miss, data is fetched from the database, stored in the cache, and then returned. This approach reduces database load, speeds up access to frequently used data, and enhances scalability.

For frequently accessed but rarely changed data, such as:

- AI scoring results
- Transaction category lists
- Financial goals
- AR asset metadata

Workflow:

1. App checks cache (e.g., Redis).
2. If hit, return data.
3. If miss, query DB → store result in cache → return to user.

NFR Contributions:

- **Performance:** Sub-50ms access for cached reads.
- **Scalability:** Offloads read pressure from the database.
- **Reliability:** Reduced latency failures under heavy load.
- **Maintainability:** Transparent to service interfaces and controllers.

Design Patterns

1. Singleton Pattern

Used for:

- Database connection pools (PostgreSQL)
- Caching clients (Redis)
- Logger instances (Winston/Morgan)

Rationale: Ensures that only one instance of a resource-heavy object (e.g., DB connection) is created and reused throughout a service.

NFR Contributions:

- **Performance:** Reduces overhead from repeatedly initializing resources.
- **Maintainability:** Centralized configuration and usage.
- **Reliability:** Prevents resource leaks or connection saturation.

---

2. Strategy Pattern

Used for:

- AI transaction categorization strategies (manual vs. automated)
- Goal evaluation strategies (milestone-based, percentage-based)
- Rendering behaviors for AR objects based on user tier
-

Rationale: Encapsulates interchangeable algorithms or behaviors, allowing dynamic switching without changing client code.

NFR Contributions:

- **Maintainability:** New strategies can be added without touching core logic.
- **Usability:** Personalized behavior (e.g., AI or AR) via interchangeable modules.

---

3. Observer Pattern

Used for:

- Notification systems (subscribing to changes in user goals or activity)
- Real-time dashboard updates via WebSockets or EventEmitters
-

Rationale: Allows components to subscribe to and react to events from observable subjects (used heavily in EDA context).

NFR Contributions:

- **Usability:** Immediate user feedback on events (e.g., “Goal Achieved!”).
- **Reliability:** Decoupled reaction chains with fallback logic.
- **Scalability:** Reactive model supports many consumers asynchronously

Design Patterns

4. Factory Pattern

Used for:

- Dynamic generation of AR assets based on transaction types or goals.
- Creation of ViewModel instances based on user context (e.g., financial literacy, preferences)
-

Rationale: Abstracts the instantiation process to return different object types based on input configuration or state.

NFR Contributions:

- **Maintainability:** Easily adapt object creation logic as requirements change.
- **Compatibility:** Supports creation of platform-specific AR models or UI elements.

---

5. Adapter Pattern

Used for:

- Integrating external APIs (e.g., bank data parsers, PDF upload tools, third-party vector databases)
- Harmonizing data formats from AI responses into UI-compatible schemas

Rationale: Converts interfaces from third-party or legacy systems into application-compatible interfaces.

NFR Contributions:

- **Compatibility:** Seamless integration with external APIs.
- **Maintainability:** Isolates changes needed for external format updates.

---

6. Command Pattern

Used for:

- Executing user actions such as “log transaction,” “redeem reward,” or “reset goal”
- Undoable or repeatable operations (e.g., edit/delete financial entries)

Rationale: Encapsulates an operation as an object, allowing logging, undo, and queuing of user actions.

NFR Contributions:

- **Reliability:** Actions can be logged and replayed, or rolled back.
- **Maintainability:** Centralized command execution logic.

Design Patterns

7. Decorator Pattern

Used for:

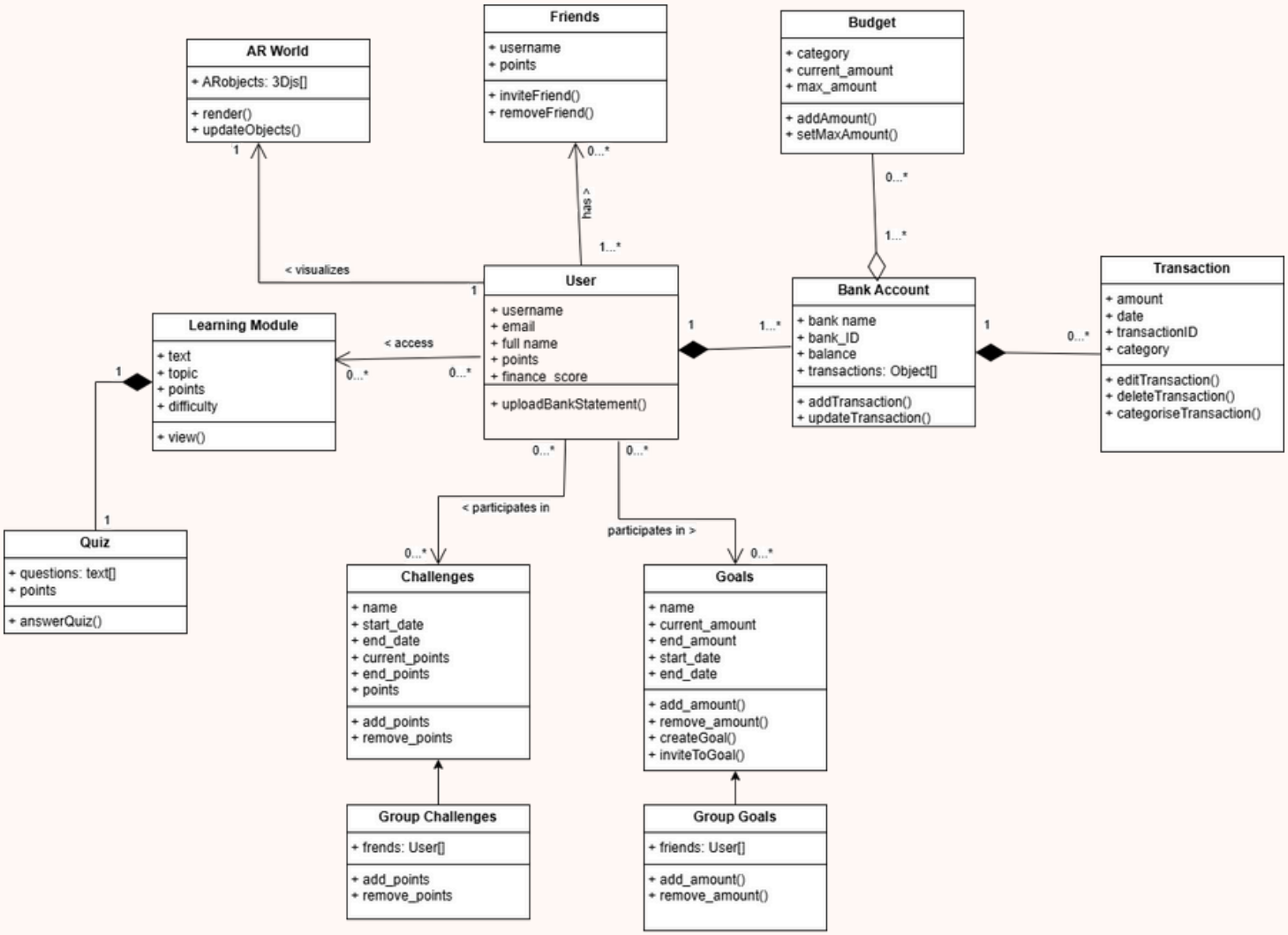
- Enhancing AI responses with post-processing (e.g., disclaimer injection, tone adaptation)
- Adding AR effects based on user score or tier
-

Rationale: Dynamically extends the behavior of an object without modifying its structure.

NFR Contributions:

- **Compliance:** Guarantees mandatory enhancements (e.g., legal disclaimers).
- **Performance:** On-demand decorations minimize resource usage.

Domain Model V1



Constraints

The system should not process or store financial data in ways that violate relevant data protection regulations such as POPIA as the architecture must include mechanisms for data protection and integrity.

The system should not present financial insights without appropriate disclaimers, all AI generated financial guidance must include clear disclaimers to comply with legal and ethical guidelines.

The system should not rely heavily on vertical scaling. Each service must support horizontal scaling through containerization and stateless design for efficient resource management.

The system's front end and backend should not be tightly coupled. The architecture must enforce separation of concerns to prevent changes in one layer from negatively affecting the rest of the system.

AR components should not depend on continuous high speed internet connections, and must provide basic functionality through local caching when connectivity is limited.

Technology Requirements

Frontend: React Framework

React was selected for both web (React.js) and mobile (React Native) due to its strong cross-platform support, 3D/AR/VR capabilities, and AI integration.

Implementation Overview

- Tech Stack: JavaScript/TypeScript with component-based design.
- Web: React.js + Three.js, React Three Fiber, WebXR.
- Mobile: React Native + ViroReact, ARKit.
- AI Integration: TensorFlow.js, WebSockets, chatbot APIs.

Pros

- Shared codebase across platforms.
- Strong 3D/AR/VR support.
- Real-time AI features using JavaScript tools.
- Scalable and reusable UI components.
- Backed by a large, active community.

Cons

- Slightly slower than native apps for heavy animations.
- Some AR/VR features require native bridging.

Key Considerations

- Ideal for immersive, AI-driven cross-platform apps.
- Scalable for future AR/VR devices (e.g., Oculus, Vision Pro).
- Chosen over Angular, Vue, Flutter, and native for flexibility and ecosystem fit.

Backend Architecture – Express.js + PostgreSQL

The backend uses Express.js to create a RESTful service layer that handles authentication, transactions, goal tracking, and AR asset management.

Implementation Highlights:

- Tech Stack: Node.js, Express.js, PostgreSQL (via pg or Knex.js)
- Security: JWT-based auth, bcrypt, Helmet, rate-limiting, and express-validator
- Architecture: MVC structure with Docker-based deployment
- Docs: Swagger/OpenAPI for route documentation

Modular Services:

1. Authentication Module: Stateless JWT auth, secure password hashing
2. Financial Logic: Endpoints for transactions/goals with query optimization
3. Visualization Support: Converts data for frontend AR/3D rendering
4. Social Layer: Leaderboards, group tracking, comparison features

Pros: Lightweight, fast to develop, integrates easily with the frontend and AR.
Cons: Lacks built-in ORM, requires architectural discipline.

Technology Requirements

AI/ML – Transaction Categorization & Personalization

The platform uses DistilBERT (via Hugging Face + TensorFlow) for NLP-based automatic transaction classification and LLMs for conversational financial advice.

Core Components:

- DistilBERT: Fast, transformer-based classifier trained on Kaggle datasets
- Local Inference: Prioritized to preserve privacy
- Knowledge Embedding: Transaction data vectorized (FAISS/Qdrant) for semantic understanding
- Advice Engine: Local LLM (e.g., LLaMA 3 70B) for context-aware, private financial guidance

Pros: Privacy-preserving, scalable, high contextual accuracy
Cons: LLMs require GPU/server support; embedding needs tuning for grounding

---

Data Science – Trend Analysis & Forecasting

A data analytics engine provides personalized and community-level financial insights.
Functions & Tools:

- Forecasting: ARIMA, Prophet for spending/goal predictions
- Clustering: K-Means for behavior segmentation
- Pattern Discovery: mlxtend, Orange3
- Visualization: Plotly, Matplotlib, optional Streamlit dashboards

Pros: Enables gamified insights, financial alerts, and adaptive coaching
Cons: Needs anonymized aggregation logic, tuning for evolving user behavior

---

API Development Framework: Express.js

Express.js is used to build internal APIs and handle integration with Stitch.

Key Benefits:

- Lightweight, flexible, and fast
- Fits well with full-stack JavaScript development (React frontend)
- Supports real-time features via WebSockets
- Easily integrates with testing tools (e.g., Jest)

Comparison to Alternatives:

- Django: Heavier, Python-based — less aligned with team skills
- Spring Boot: Higher learning curve
- FastAPI: High performance but less synergy with JavaScript stack

Challenges:

- Manual setup for error handling and middleware
- No built-in ORM—requires additional tooling

Technology Requirements

Database: PostgreSQL

The platform uses PostgreSQL as a unified, secure, and scalable relational data store to manage all core domains—users, transactions, goals, assets, and community metrics.

Implementation Highlights:

- Normalized schema with foreign keys for data integrity
- JSONB fields for semi-structured user preferences and AR settings
- Indexes on high-read fields (e.g., user_id, date)
- PostgreSQL Views for analytics and dashboards
- Row-Level Security (RLS) to restrict access to user-owned data
- Backup and failover configuration for high availability

Data Model Overview:

- Core Financial: Users, Accounts, Transactions, Categories
- Visualization: VisualAssets, Goals, AssetLinks
- Community: Groups, GroupMembership, Milestones
- Preferences: UserPreferences (in JSONB)

Pros:

- Strong ACID compliance and SQL support
- High integrity and performance for structured data
- Flexible handling of user-specific configs with JSONB
- Built-in security and analytics features

Cons:

- Less flexible for highly dynamic data
- Requires careful query tuning at scale
- Schema changes may need migration planning

Hosting

After evaluating major cloud providers, Render was chosen as the hosting platform for the Gamified Finance Visualizer due to its simplicity, cost-effectiveness, and alignment with the project's university context.

- Pros:
  - Simplified deployment and transparent pricing
  - Built-in CI/CD pipeline and GitHub integration
  - Native support for React and Node.js
  - Free tier suitable for development and testing
- Cons:
  - Limited services compared to AWS
  - Fewer compliance certifications

Final Decision:

Render was selected for its developer-friendly environment, making it ideal for rapid development and streamlined deployment in a university project setting.

Service Contracts

In Collaboration With