• 2025 •

# ARCHITECTURAL SPECIFICATIONS

## GAMIFIED FINANCIAL VISUALIZER

For

**Demo 2**

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

**Architectural Strategy: Client-Server with Event-Driven Architecture (EDA)**
The Gamified Financial Visualizer adopts a client-server model enhanced by event-driven architecture (EDA) to deliver a secure, engaging, and scalable platform for financial learning and management. This approach supports both functional goals and long-term growth through modular, reactive design. Key architectural principles are grounded in the following business needs:

**Security**
All client-server interactions are secured using token-based authentication, TLS encryption, and role-based access control. Events such as account changes or transactions trigger background validations and audit logs to ensure compliance with GDPR and POPIA.

**Usability**
The client delivers an intuitive, accessible UI using real-time event streams (e.g., WebSocket) for dynamic feedback like XP gains or goal updates. Interfaces follow Material Design and Apple HIG, with AR and gamified features built for smooth, engaging interactions.

**Availability**
The backend uses cloud infrastructure, event queues, and load-balanced API gateways to ensure uptime and responsiveness. Services process events asynchronously to handle peak loads without degrading user experience.

**Scalability & Extensibility**
Services are containerized and loosely coupled, communicating via events (e.g., user-joined, goal-completed). This enables independent scaling and easy integration of new features like AI coaches or AR tools.

**Maintainability & Agility**
Modular codebases, clear event contracts, and automated CI/CD pipelines ensure fast iteration and minimal disruption. Services operate independently, allowing teams to deploy and evolve features quickly and safely.

# 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 product within academic timelines. The following elements define the scope of this system:

**Financial Tracking & Visualization**

- Users can upload their bank statements for automatic input and categorisation of their transactions.
- The system will process and visualize transactions using dynamic dashboards and 3D/AR environments.
- AR-based interaction allows users to view a financial "world" like "savings buildings" or "budget trees".

**Gamification Features**

- Reward systems for achieving 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 behavioural tips 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 (in order of priority)

## 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.
- **Event-driven pipeline architecture and modular filtering layers:** The system leverages an event-driven architecture coupled with a pipe-and-filter model to promote asynchronous processing, parallelism, and reduced system coupling
- **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

## 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.
- **Audit logger**: keep a record of user and system actions and their effects—to help trace the actions of, and to identify, an attacker.

| 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., Supabase, 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.

| 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

## 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.

- **Horizontal Scaling of Background Workers**: While the core app remains monolithic, workers and schedulers can scale independently to handle increased load (e.g., XP calculation, AR asset streaming, learning module progress updates).
- **Event-Driven Architecture (EDA):** Kafka 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

## 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) with ARCore/ARKit integration. Consistent design using Material Design and Apple HIG, with progressive disclosure to reduce complexity.
- **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

## 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

## 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 1 year:** 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 2 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.

## Client-Server Architetcure

It is structured around modular subsystems within a monolithic client-server architecture. Each functional domain is encapsulated as a cohesive module with clear interfaces and responsibilities.

**Key subsystems include**:

- Authentication & User Profile
- Finance Manager (Transactions, Budgets, Accounts)
- AI Classification & Insights Engine
- AR Visualization Layer
- Social & Community Interaction Layer
- Financial Goal Tracker
- Learning & Quiz Engine

**Benefits & NFR Alignment**

- **Security:** Security controls (like authentication and input validation) can be enforced at boundaries, limiting how threats propagate.
- **Scalability**: High-load operations (e.g., AI scoring, XP allocation) are processed asynchronously using an event-driven architecture, with background workers handling bursts of activity.
- **Reliability & Availability**: Modularization reduces cross-dependencies. Failures in one subsystem (e.g., Learning Engine) do not block critical flows (e.g., Transaction Processing).
- **Compatibility & Evolvability**: Modules can evolve independently within the monolith (e.g., you can upgrade the Goal Engine without modifying the Auth system).

## 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.

## 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

---

## 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. Typical cache hits achieve data retrieval within 10–30ms, significantly outperforming database queries, averaging 100–200ms.

For frequently accessed but rarely changed data, such as:

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

**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.

---

# Architectural Patterns cont.

## MVVM

Separates application logic into three interactive layers: the **Model** (data), **ViewModel** (state + logic), and **View** (UI). The ViewModel acts as a mediator, transforming raw data into a form the View can display while staying decoupled from backend services.
This structure promotes reusability and simplifies state-driven rendering. Each layer has a distinct role:

- **Model** handles data access and remote interactions (e.g., API calls)
- **ViewModel** encapsulates logic, data transformation, and user interactions
- **View** listens to state updates and renders declarative UI via React components

**Applied to:**

- React + Tailwind CSS frontend
- Custom hooks like as ViewModels
- Stateless UI components (View) render data passed from hooks
- Services and API clients (Model) provide persistent data

**NFR Alignment:**

- Performance: View updates only on relevant state changes
- Reliability: Logic and validation live in ViewModel, away from UI glitches
- Maintainability: Clear separation makes components reusable and testable
- Scalability: Feature growth is isolated to individual layers
- Accessibility: Declarative UI supports consistent ARIA and semantic tagging

# 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 behaviours for AR objects based on user tier

**Rationale:** Encapsulates interchangeable algorithms or behaviours, 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 View Model 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 behaviour of an object without modifying its structure.

**NFR Contributions:**

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

# Constraints

**Client 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.

**1. Client–Server Pattern**

Rationale: Enables clear separation of concerns, isolating the browser-based React frontend from a secured backend built in Node.js/Express..

*Constraints:*

Can't easily push updates to the client unless polling or sockets are added (no real time interactions out of the box).

Communication is HTTP-based with secure token verification; all mutations pass through backend validation.

**2. Event-Driven Architecture (EDA)**

Rationale: Enables loosely coupled services that can react to domain-specific events like goalCompleted, friendInvited, or transactionLogged.

*Constraints:*

Client requested social feedback features, so real-time event propagation was essential (e.g. notifying friends).

Harder to guarantee event ordering and consistency. Debugging across distributed event chains can be complex.

**3. MVVM (Model–View–ViewModel)**

Rationale: Organizes frontend logic using React hooks (ViewModels) to separate UI from business logic.

*Constraints:*

Dynamic UI features like progress tracking and challenges required layered state and logic separation.

Rapid frontend iteration and limited team size favored a predictable mental model for state/data flow.

# Constraints

**4. Pipe and Filter**
Rationale: Applied to analytics and progress insights where data flows through validation → transformation → formatting stages.

*Constraints:*
Difficult to maintain shared context or state across stages without breaking modularity. Not ideal for tightly coupled logic.
Derived metrics (e.g., “Goals completed this week”) require progressive, chained computation.
Needed a structure that could gracefully skip/modify steps based on data availability or user context.

**5. Cache-Aside Pattern**
Rationale: Improves performance for repeated reads (like dashboard summaries) using Redis as a read-through cache.

*Constraints:*
May introduce boilerplate in small apps; ViewModels can become overly complex if they absorb business logic.
Redis not always available during local dev, so fallback logic and environment-aware clients were essential.

---

# Technology Requirements

## Frontend Framework

Develop a cross-platform user interface that supports 3D/AR rendering, AI-driven features, and real-time feedback within an immersive and responsive experience.

**Technologies Considered**

1. *React (React.js / React Native)*
   - Overview: JavaScript library with a component-driven architecture and wide ecosystem support. React Native extends React for mobile platforms.
   - Pros:
     - Shared codebase between web and mobile (React Native).
     - Mature ecosystem for 3D/AR (React Three Fiber, ViroReact, WebXR).
     - Strong AI integration capabilities (TensorFlow.js, WebSockets).
     - Reusable components promote maintainability (MVVM alignment).
   - Cons:
     - Not as performant as fully native solutions for compute-heavy AR.
     - Some advanced AR features require native bridging.
2. *Flutter*
   - Overview: Google's UI toolkit for building natively compiled apps with a single Dart codebase.
   - Pros:
     - Excellent cross-platform mobile performance.
     - Beautiful UI rendering with customizability.
   - Cons:
     - Weaker browser support (Web is not production-optimized for complex AR).
     - Limited ecosystem for WebXR or full AI/ML integration.
     - Learning curve for Dart and less community support for AI/3D-specific tooling.
3. *Angular*
   - Overview: TypeScript-based full-stack frontend framework backed by Google.
   - Pros:
     - Built-in structure for enterprise-scale apps.
     - Strong tooling and testing framework.
   - Cons:
     - Heavier learning curve and boilerplate for smaller teams.
     - Less flexibility for integrating lightweight 3D or AR modules.
     - Slower development velocity due to rigidity in data binding and templating.

**Final Selection: React (React.js + React Native)**

- React was selected for both web and mobile development due to its alignment with MVVM, flexibility in building modular View components, and seamless integration with AI pipelines (e.g., WebSockets + TensorFlow.js). Its support for libraries like Three.js, React Three Fiber, and WebXR made it the most viable choice for implementing the Pipe and Filter rendering strategy used in the AR modules.
- In addition, React's shared-code strategy reduces complexity across platforms — a critical architectural constraint given limited resources and the need for rapid iteration. React's community-driven ecosystem also fit well with the team's skillset and CI/CD pipeline (GitHub Actions + Supabase deployment), making it ideal within the chosen Client–Server + Cache-Aside + EDA context.

# Technology Requirements

## Backend Framework & Service Layer

Develop a RESTful backend capable of handling authentication, financial logic, AR rendering support, and social interactions — with emphasis on modularity, security, and low-overhead integration with the frontend.

**Technologies Considered**

1. *Express.js + Node.js (Selected)*
   - Overview: Lightweight, unopinionated web framework for building REST APIs in TypeScript.
   - Pros:
     - Minimal setup, fast prototyping.
     - Full control over routing, middleware, and performance tuning.
     - Rich npm ecosystem and strong integration with frontend via shared JavaScript tooling.
     - Works seamlessly in containerized (Docker-based) CI/CD pipelines.
   - Cons:
     - Lacks an opinionated structure; discipline required to maintain MVC standards.
     - Requires manual integration of features like validation, ORM, or rate-limiting.
2. *NestJS*
   - Overview: Opinionated, TypeScript-first Node.js framework built around modules, decorators, and dependency injection.
   - Pros:
     - Out-of-the-box architecture aligned with clean code principles (Controllers, Providers, Guards).
     - Built-in support for Swagger, class-validation, and WebSockets.
   - Cons:
     - Heavier startup curve and boilerplate overhead.
     - Slower initial dev velocity, especially for small teams.
     - Less flexibility when deviating from standard patterns.
3. *Django (Python)*
   - Overview: Batteries-included web framework known for robustness and built-in admin tools.
   - Pros:
     - Excellent ORM, authentication, and admin out of the box.
     - Strong documentation and support for data-intensive apps.
   - Cons:
     - Python introduces an entirely separate stack, reducing dev efficiency.
     - Harder to integrate real-time features (WebSockets) and AR middleware.
     - Less frictionless with a Node/React frontend stack.

**Final Selection: Express.js + Node.js**

Express.js was selected for its minimal runtime footprint, tight integration with frontend tooling, and flexibility in mapping to MVC and Pipe-and-Filter architectures. It supports modular services (e.g., Auth, Financial Logic, AR data formatting) while remaining lightweight enough to scale with evolving requirements.

This decision aligns with architectural constraints such as:

- Event propagation in an Event-Driven Architecture (EDA)
- Separation of controller logic (aligns with MVVM frontend)
- External hosting limitations on resource-heavy runtime environments
- reinforcing performance, security, and maintainability goals.

# 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.

**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 Management System

Select a secure, scalable, and query-efficient data store capable of handling structured financial data, semi-structured preferences, and analytics-ready dashboards with strong access control.

**Technologies Considered**

1. *PostgreSQL (Selected)*
   - Overview: An advanced open-source relational database with full ACID compliance, support for JSONB, custom indexing, and SQL views.
   - Pros:
     - Strong consistency and referential integrity using foreign keys and transactions.
     - Rich support for analytics (e.g. materialized views, CTEs).
     - JSONB fields offer flexibility for evolving preferences (like AR configuration).
     - Role-Based Access Control and Row-Level Security improve multitenancy and data isolation.
   - Cons:
     - Schema evolution requires careful planning and migration strategy.
     - Not as performant as NoSQL under highly dynamic, schema-less workloads.
2. *MongoDB*
   - Overview: A document-oriented NoSQL database designed for horizontal scaling and flexible schemas.
   - Pros:
     - Great for dynamic, evolving data with no rigid schema.
     - Rapid iteration on user-generated content or AR asset structures.
   - Cons:
     - No built-in relational integrity — joins are manual or application-level.
     - Requires custom implementations for analytics features like rollups or views.
     - Weaker transactional guarantees (multi-document ACID only recently introduced).
3. *Firebase Realtime Database / Firestore*
   - Overview: Backend-as-a-Service offerings with real-time data sync and client-first access patterns.
   - Pros:
     - Automatic scaling and real-time push for mobile/web clients.
     - Minimal ops overhead — fully managed by Google.
   - Cons:
     - Less control over query complexity and data modeling.
     - No out-of-the-box support for relational joins or analytics views.
     - Limited flexibility for advanced access controls and indexing.

**Final Selection: PostgreSQL**

PostgreSQL was selected as the primary database to support a unified schema for finance, community, and user-generated data, with reliable relational integrity and support for semi-structured fields (via JSONB). Its features align well with several architectural needs:

- Client–Server constraint: Secure backend queries expose only authorized rows via RLS.
- Cache-Aside strategy: High-read endpoints are optimized through indexes and materialized views, with Redis layered on top.
- MVVM alignment: Backend models map cleanly to ViewModel consumers via predictable REST endpoints.
- Pipe-and-Filter analytics: SQL views simplify filtering and aggregation for financial insights.

Its maturity, security posture, and ability to grow with complex schemas made PostgreSQL the ideal fit under both functional and architectural constraints.

# Technology Requirements

## Hosting and Backend Platform

Deploy a full-stack application with a scalable, secure, and developer-friendly platform supporting relational data, authentication, real-time updates, and serverless functions.

**Technologies Considered**

1. S*upabase (Selected)*

- Overview: Open-source Backend-as-a-Service (BaaS) built on PostgreSQL with integrated auth, storage, real-time subscriptions, and serverless functions.
- Pros:
  - Natively supports SQL, complex joins, and advanced relational queries.
  - Open-source and self-hostable, reducing vendor lock-in.
  - Built-in real-time support via PostgreSQL's replication layer.
  - Offers edge functions, object storage, and REST/GraphQL APIs out of the box.
  - Fine-grained access control through policies like Row-Level Security (RLS).
- Cons:
  - Smaller ecosystem and community than Firebase.
  - Real-time features still maturing compared to Firebase Realtime DB.
  - Some advanced analytics/extension capabilities may need manual configuration.

2. *Firebase*

- Overview: Google's BaaS platform offering real-time databases, storage, authentication, and cloud functions.
- Pros:
  - World-class real-time data syncing with offline-first support.
  - Seamlessly integrated services with minimal setup.
  - Vast community and plug-and-play analytics.
- Cons:
  - Uses NoSQL document stores, making relational queries complex or inefficient.
  - Locked into Google ecosystem; limited SQL expressiveness.
  - Less backend transparency and control, which hinders fine-tuned optimization.

3. *AWS Amplify*

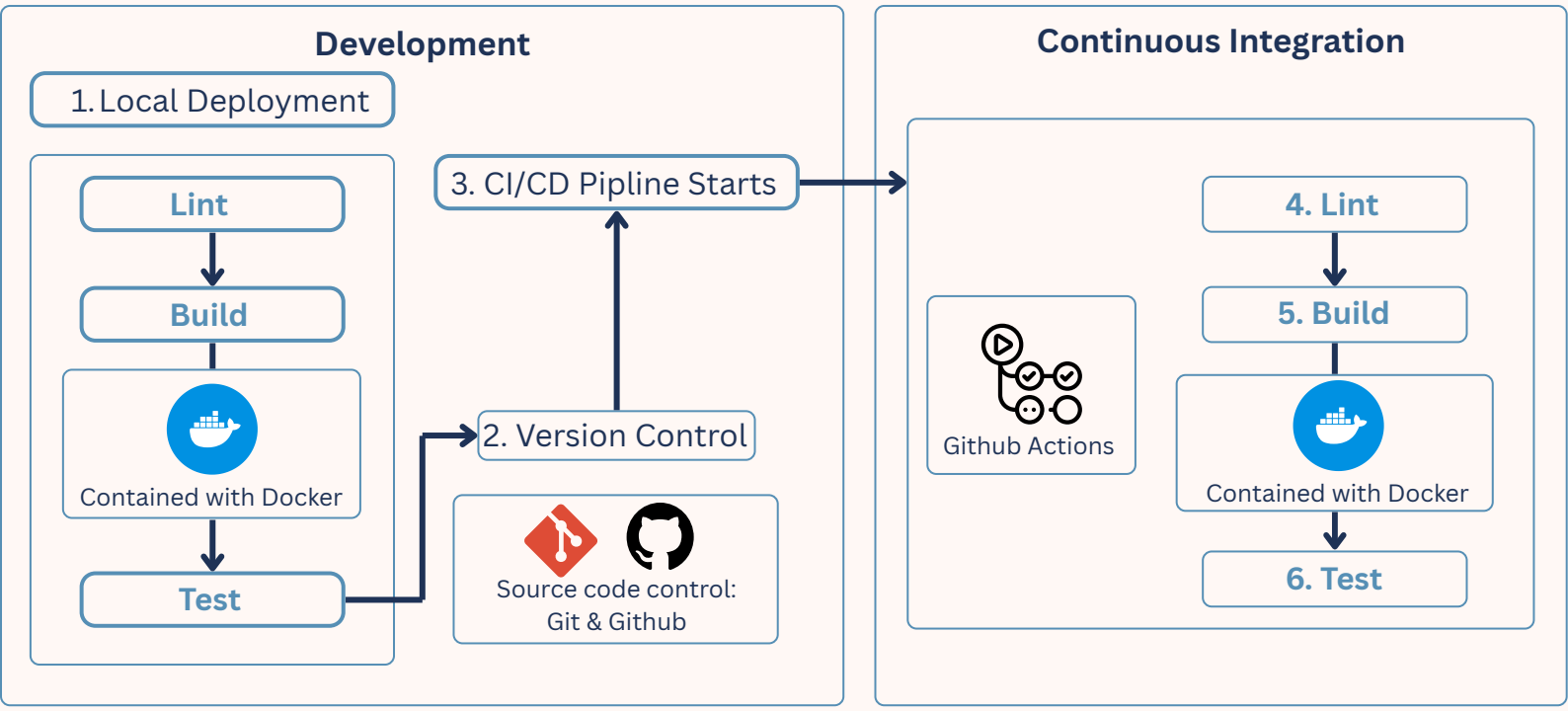
- Overview: Serverless application backend that integrates with AWS services like Cognito, AppSync, S3, and Lambda.
- Pros:
  - High scalability and reliability.
  - Granular service configuration and access control.
- Cons:
  - Complex to configure and debug without AWS experience.
  - Longer development time; poor DX (developer experience) for small teams.
  - GraphQL-first model isn't ideal for all use cases.

**Final Selection: Supabase**

Supabase was selected for its strong alignment with the project's relational data needs, modular architecture, and open-source transparency. Built on PostgreSQL, it complements existing database decisions while supporting advanced queries and enforcing strict access controls through RLS — critical in a Client–Server setup where user isolation and structured analytics are required.

- Its support for Edge Functions and real-time channels integrates well with Event-Driven Architecture (EDA) for goal tracking, notifications, and collaborative experiences. Compared to Firebase, Supabase offers more flexibility for server-side logic, schema evolution, and backend customization — a priority given the need for custom analytics, AR data flows, and secure financial transactions.
- This choice also respects platform constraints like GitHub Actions-based CI/CD and Docker compatibility for backend deployment.

# Deployment model



Gamified Finance

In Collaboration With