Virtusa Corp.
2000 West Park Drive
Westboro, MA 01581
Phone : (508) 389-7300
Fax : (508) 366 9901
E-mail : info@virtusa.com
URL : www.virtusa.com

virtusa
Accelerating Business Outcomes

Use Case: Dynamic Pricing of Financial Products

Architecture Document for

ApexWolf – AI Powered Dynamic Pricing System

Aashik T S

KavyaSree MG

Mohammed Imran S

Sharmili S

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

This document defines the system architecture, functional behavior, and technical specifications of the comprehensive Loan and Investment Management Platform. It is intended for use by development teams, system architects, quality assurance engineers, and stakeholders involved in the deployment, evaluation, and maintenance of the platform.

The primary objective of this document is to establish a clear and verifiable understanding of system components, interfaces, data flows, user roles, and underlying computational logic that supports business objectives, particularly in the domains of digital loan processing, investment forecasting, and customer engagement.

1.2 Scope

This architecture encompasses all user-facing, administrative, and backend components of the system, including but not limited to:

- Loan interest rate prediction and optimization
- User-specific loan eligibility computation
- Investment prediction and performance tracking
- Document-based loan processing and agreement workflows
- Expense monitoring and financial analytics
- Role-specific administrative configuration and control panels
- Accessibility, voice control, and interactive guidance systems
- Notification, feedback, and customer support subsystems

The system is designed for deployment as a web-based platform, with MySQL as the backend database and modular integration of machine learning models, graphical analytics, and natural language processing. It includes multi-role access (user and admin) with a strict role-based feature set and granular data logging for compliance and auditability.

1.3 Definitions, Acronyms, and Abbreviations

Term	Definition
AI	Artificial Intelligence
NLP	Natural Language Processing
EMI	Equated Monthly Instalment
SIP	Systematic Investment Plan
NAV	Net Asset Value
RBI	Reserve Bank of India
POC	Proof of Concept
API	Application Programming Interface
ML	Machine Learning
UX	User Experience
UI	User Interface
DB	Database
ROI	Return on Investment
OTP	One-Time Password

1.4 References

- Web Content Accessibility Guidelines (WCAG) 2.1
- Reserve Bank of India Official Data Sources
- MySQL Documentation v8.0
- FastAPI Framework Documentation
- React TypeScript Documentation v18
- CatBoost, LightGBM, XGBoost ML Framework Documentation

2. ARCHITECTURAL GOALS AND CONSTRAINTS

2.1 Reusability

The architecture is designed with a strict focus on component reusability across modules and future project phases. Core elements such as the eligibility engine, EMI simulator, interest optimization logic, and investment predictor are encapsulated in service-oriented components that expose defined interfaces. The user interface layer is modularized into role-specific views and widgets, enabling seamless reuse during the back-end data server enhancement phase without reengineering core UI logic.

2.2 Scalability

The system must support horizontal and vertical scalability to accommodate increasing user loads and transaction volumes. Application services are designed to be stateless and independently deployable under orchestration frameworks. However, the scalability of the current flat file I/O subsystem remains a limitation. Subsequent phases will transition to fully managed RDBMS clusters to eliminate I/O bottlenecks and improve concurrency under scale.

2.3 Customizability

ApexWolf is designed to support functional and visual customization via configuration rather than code alteration. UI customization is abstracted through layout schemas and feature toggles. On the backend, configurable eligibility weights, role-based permissions, and AI model parameters allow for business rule adjustments without code modification. This design enables rapid policy reconfiguration by authorized administrators.

2.4 Extendibility

The system architecture is modular and service-oriented, allowing new modules and business logic to be integrated without disrupting existing components. APIs conform to version-controlled contracts, ensuring that extensions can be introduced incrementally. Use

of dependency injection and message brokers further decouples services, reducing regression risks during functional extensions.

2.5 Use of Existing Business Logic

Existing business rules and algorithms from the legacy application are refactored into discrete, reusable libraries packaged as shared services or dynamically linked components. This approach ensures continuity in domain logic while enabling clean separation from presentation and storage concerns. Legacy loan and eligibility logic is preserved as independently callable modules within the modern architecture.

2.6 Time to Market

Accelerated delivery timelines constrain the selection of technologies. Frameworks and tools have been chosen based on maturity, ease of integration, and availability of in-house expertise. Reuse of pre-validated components and libraries, such as CatBoost and TextBlob, reduces implementation time while maintaining accuracy and compliance. Dependencies on experimental or under-documented technologies are explicitly avoided.

2.7 Portability

The architecture ensures operational portability across environments with minimal configuration overhead. Application services are containerized and orchestrated to support deployment across Linux-based systems. Database interactions are abstracted using ORM frameworks, allowing portability between MySQL, PostgreSQL, and equivalent RDBMS platforms. No OS- or hardware-specific dependencies are introduced at runtime.

2.8 Availability

The system must achieve high availability under normal and peak conditions. Critical services, including authentication, prediction APIs, and processing engines, are deployed in fault-tolerant clusters with automated failover. Scheduled maintenance windows are

designed to be non-disruptive. Session persistence and retry mechanisms are implemented to prevent loss of transactional state.

2.9 Performance

The system is expected to support sub-second response times for prediction queries, eligibility checks, and investment simulations under typical user loads. Backend models are optimized for inference latency, and input pre-processing is conducted asynchronously where feasible. Caching is employed for static datasets such as NAV values and historical repo rates. Load testing is performed on every deployment milestone.

2.10 Any Other Critical Goals Applicable

- **Security Compliance**: All components must comply with enterprise-grade security policies, including encrypted data transmission (TLS 1.2+), secure password hashing (bcrypt or higher), and input sanitization to mitigate injection threats.
- **Auditability**: All user actions and prediction outcomes must be recorded with timestamped logs to support audit trails and regulatory reporting.
- **Accessibility**: The frontend interface must conform to WCAG 2.1 AA standards, with accessibility toggles to support users with visual and cognitive impairments.
- **Maintainability**: Codebases must adhere to defined coding standards and support automated linting, testing, and documentation. Logs and system alerts must be structured for monitoring.

3. PRODUCTIZATION ASSESSMENT

3.1 Re-Usable Components

The architectural strategy emphasizes modularization to maximize reuse of functional components across platforms and deployments. Key reusable components include:

- **Prediction Engine Services**: Encapsulates ensemble model inference (CatBoost, LightGBM, XGBoost) with role-specific parameter tuning. Reusable across multiple scoring workflows.
- **Eligibility Scoring Module**: Implements polynomial regression logic configurable via dynamic weight inputs. Integrated as an independent rule-based microservice.
- **Interest Rate Optimizer**: NLP-based rate revision justification engine, with shared tagging and classification logic.
- **Investment Forecast Engine**: Generic SIP calculator supporting both prediction and tracking use cases.
- **Document Processing Utility**: Parses input PDFs and generates final summaries. Shared between user uploads and admin audit pipelines.
- **Amortization Generator**: EMI breakdown and amortization table renderer with export capabilities.

All components are encapsulated behind service APIs, designed for direct reuse across user roles (user/admin), modules (loan/investment), and future enterprise systems.

3.2 Analyse Architectural Frameworks in Repository

The internal architectural repository was analysed for pre-existing frameworks with relevance to the project's requirements. The following assets were identified:

- **Legacy Eligibility API (v1.2)**: Refactored to support modular weight injection, improving configurability for multiple loan products.
- **Admin Dashboard Template (v2.1)**: Adapted for current platform with role-based rendering and permission-based route isolation.
- **Microservice Bootstrapping Layer**: Common base for containerized services, integrated with observability and log routing tools.
- **User Session Manager**: Extracted from prior deployment and integrated for secure session token lifecycle management.

Each framework was reviewed for licensing, performance under load, and compatibility with the containerized environment.

3.3 Identify and Analyse Open Source and COTS Products

A set of open-source and commercial off-the-shelf (COTS) products were shortlisted and validated for integration. Each was selected based on maturity, documentation quality, security posture, and licensing permissibility. Legal and compliance approvals were obtained for all non-native dependencies.

Component	Type	Purpose
CatBoost / LightGBM / XGBoost	Open Source	Machine learning models for determining interest rate, discount rate, tenure and risk scoring
TextBlob	Open Source	NLP-based justification analysis
Recharts	Open Source	Data visualization for investment simulations
MySQL	Open Source	Primary RDBMS for data storage
Docker	Open Source	Containerization and service isolation
GraphQL / REST Toolkit	Open Source	API layer between UI and backend services
PDFKit	Open Source	Generation of downloadable reports (PDF format)

All dependencies are tracked via internal SBOM (Software Bill of Materials) and updated in accordance with patching policies.

4. SYSTEM ARCHITECTURE

4.1 Overview

The ApexWolf architecture is tiered and modular. It consists of discrete, loosely coupled components organized into three principal layers:

- **User Interface Services**: Manages all client-facing interactions, tailored dynamically to user roles (user/admin) using role-based rendering and routing.
- **Business Services**: Encapsulates the core logic, including prediction, eligibility, optimization, and investment engines. Exposed via secured APIs.
- **Data Services**: Handles persistent data operations using structured RDBMS tables, with support for configuration versioning and audit logging.

Each tier operates independently and interacts only through defined interfaces to ensure encapsulation, scalability, and testability.

4.2 Logical / Functional View

At a high level, the system is composed of functional components distributed across user-centric and admin-centric domains. Each domain integrates with shared microservices responsible for prediction, scoring, and document management. Below is a structural abstraction of key logical groupings.

User-Facing Components

- Dashboard Overview
- Role-Based Prediction
- Loan Eligibility
- Interest Rate Optimization
- Smart Investment (Predict + Track)
- Loan Simulator
- Loan Processing
- Emergency Loan
- Repo Rate
- Market Trends (Bank Rates + Global Indicators)
- Expense Tracker
- Tasks & Rewards

- Repayment & Badges
- Announcements Banner
- Notification Centre
- Static AI Chatbot
- Privacy Settings
- Biometric Profile Upload
- Feedback and Support
- Recovery Management

Admin-Facing Components

- Admin Overview
- Role-Based Analytics
- Eligibility Weight Configuration
- Eligibility Prediction Logs
- Simulator Rate Updater
- Loan Processing Control
- Rate Optimization Interface
- Emergency Request Review
- Investment Management
- Market Trends Monitor
- Gamification Configuration
- Broadcast Notification Panel
- Feedback Review and Export
- System Logs Viewer
- Platform Settings

The application maintains clear logical boundaries between modules, ensuring that shared services are abstracted from specific front-end components and reusable across workflows.

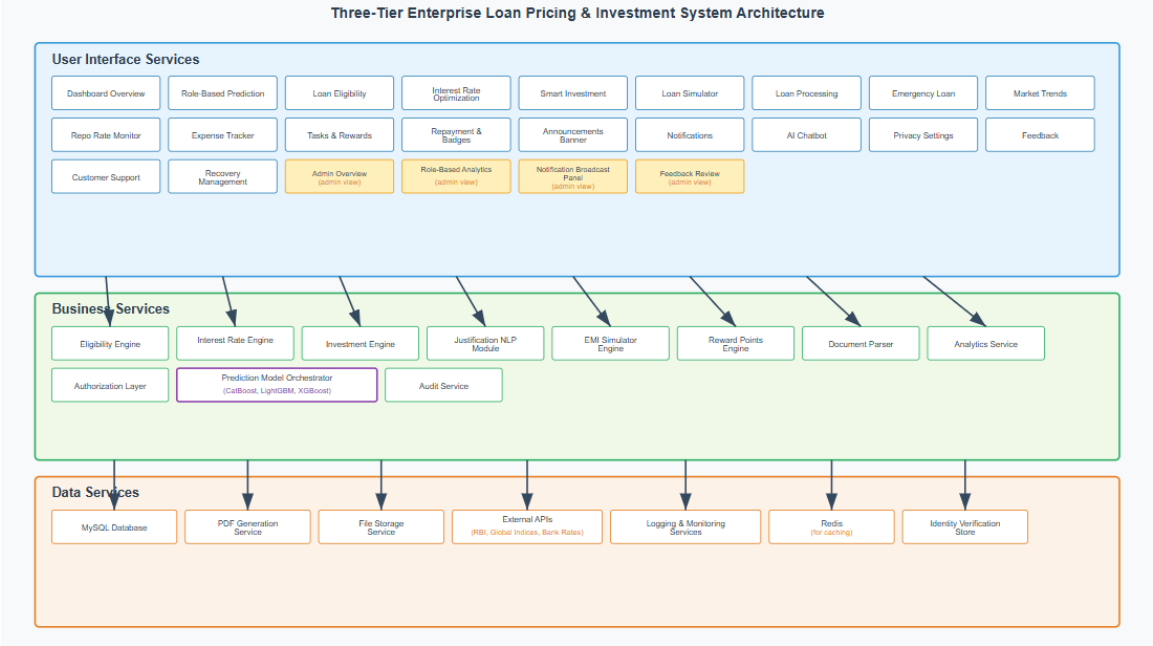


FIGURE 1: Application Architecture Block Diagram

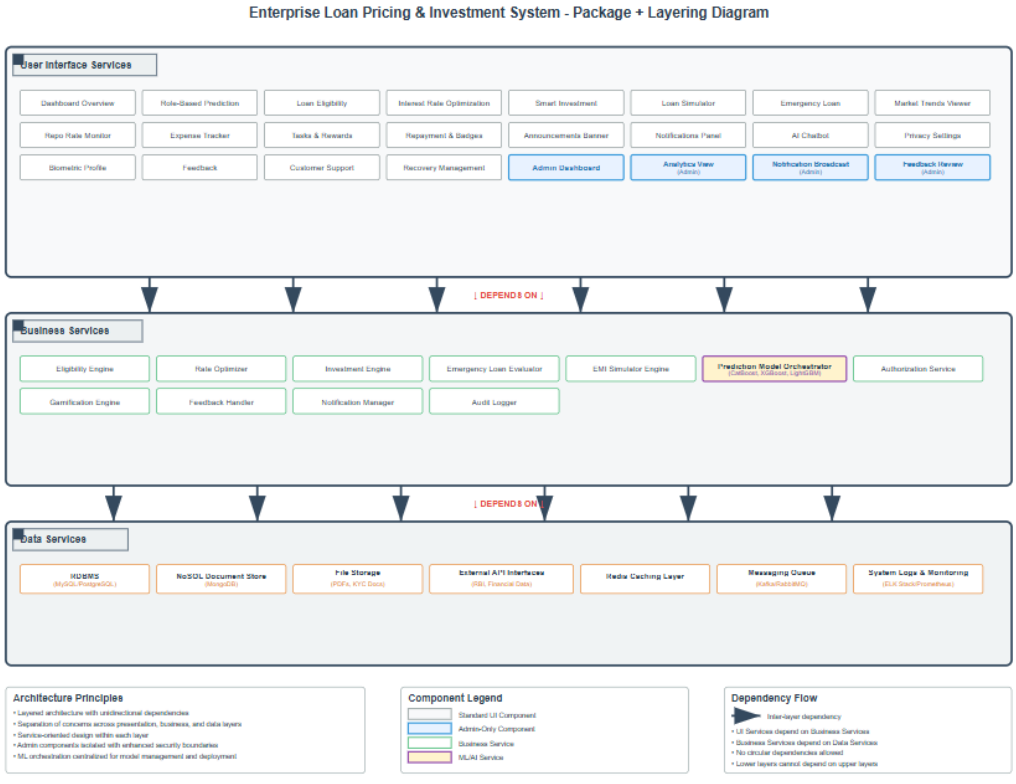
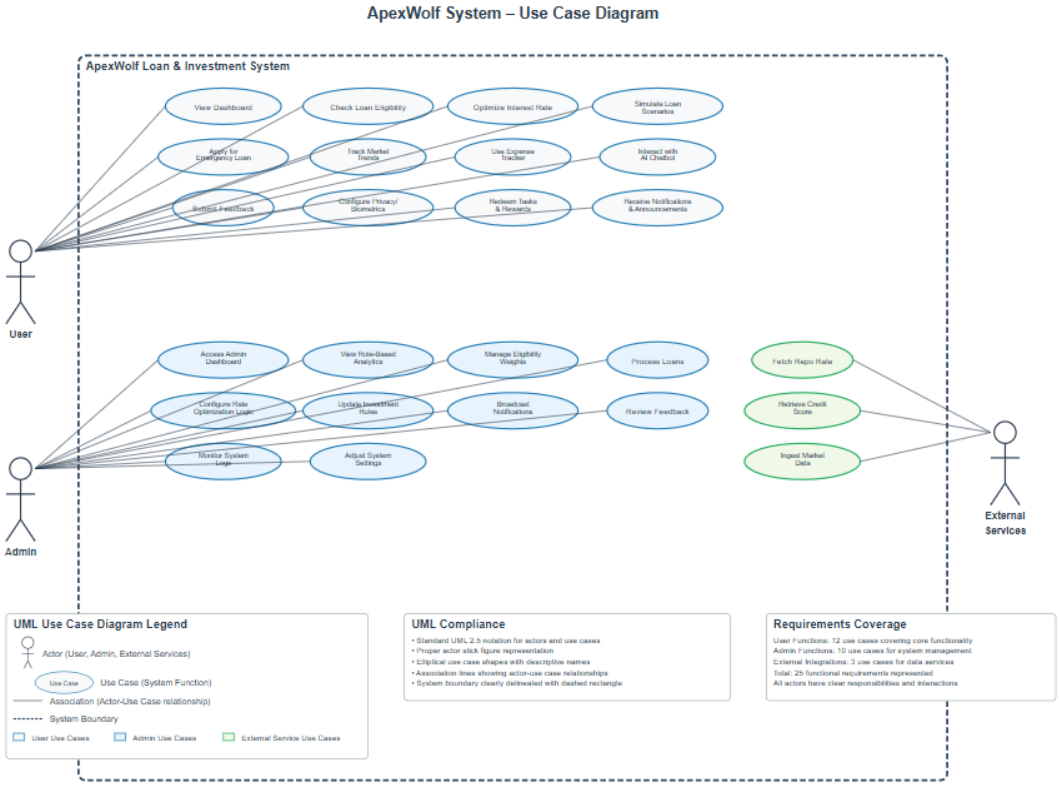


FIGURE 2: Application Architecture Package and Layering Diagram

4.3 Use Case View



Primary User Scenarios:

- Loan application and eligibility assessment
- Investment planning and performance tracking
- Financial dashboard and analytics review
- Document upload and agreement processing

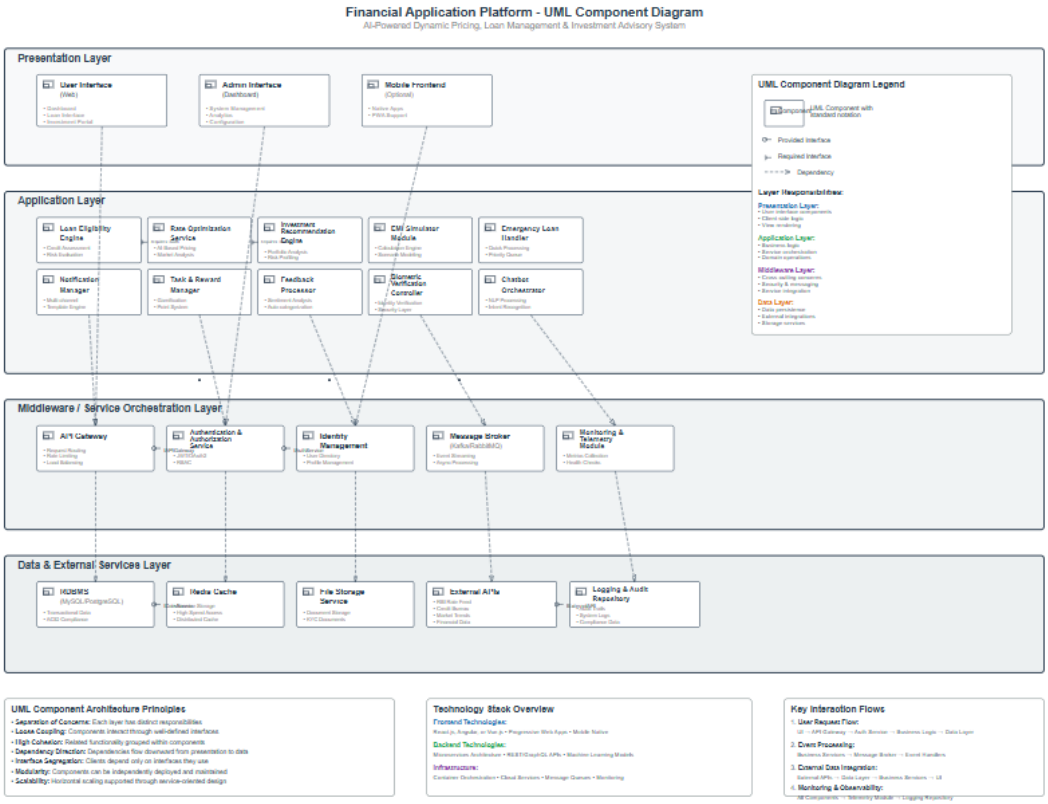
Administrative Scenarios:

- Loan application review and approval
- System configuration and parameter management
- User support and issue resolution
- Reporting and analytics generation

System Integration Scenarios:

- External data source synchronization
- Automated notification delivery
- Regulatory compliance reporting
- Performance monitoring and alerting

4.4 Implementation/System View



Frontend Implementation:

- React TypeScript single-page applications
- Component-based architecture with reusable UI elements
- State management through React hooks and context
- Client-side routing and navigation

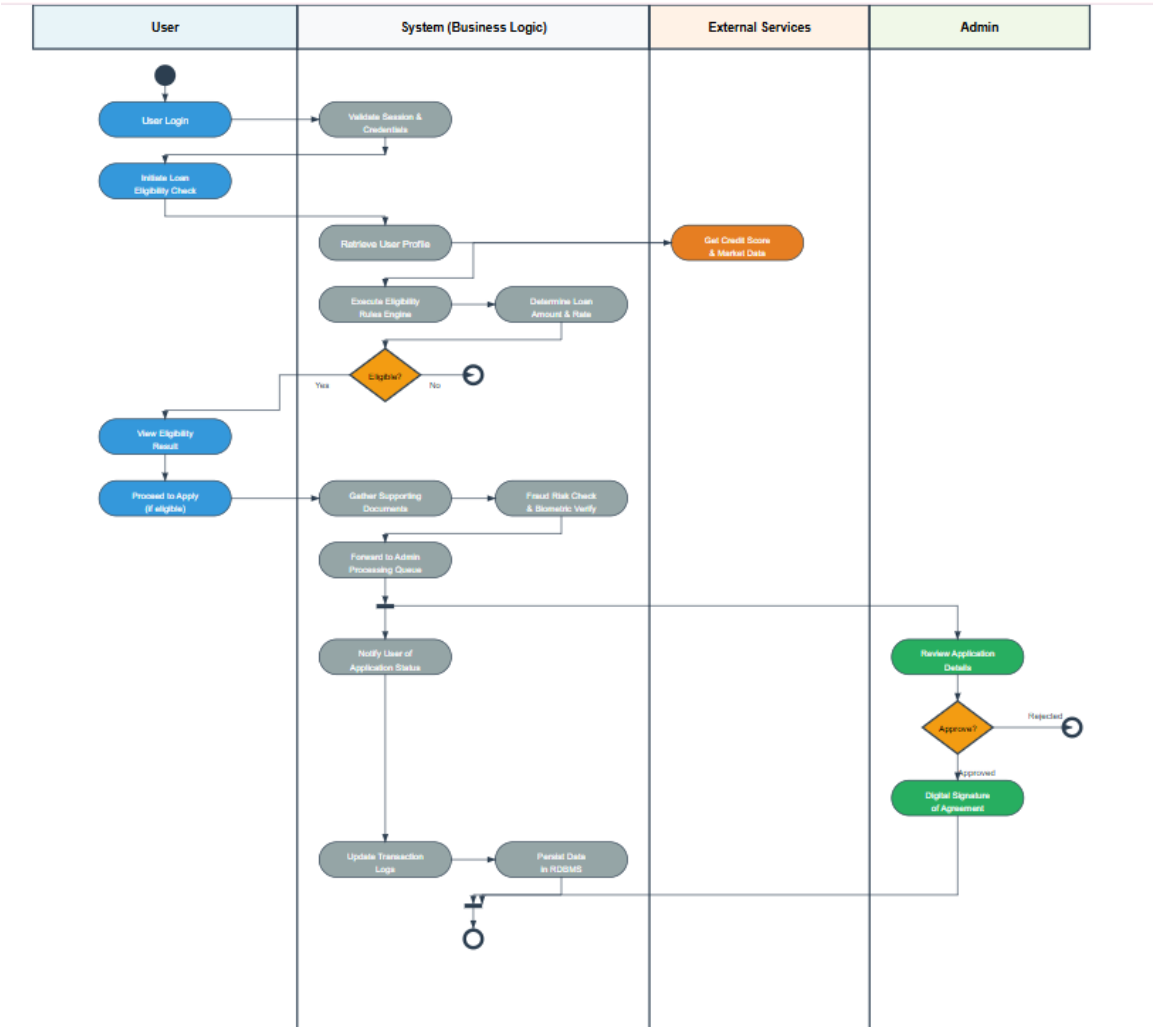
Backend Implementation:

- FastAPI microservices architecture
- Asynchronous request processing
- RESTful API design with OpenAPI documentation
- Background task processing for ML predictions

Database Implementation:

- Normalized relational schema design
- Stored procedures for complex business logic
- Transaction management for data consistency

4.5 Process/Thread View



Request Processing Flow:

1. Client request authentication and validation
2. Business logic service invocation
3. Database transaction processing
4. Response formatting and delivery

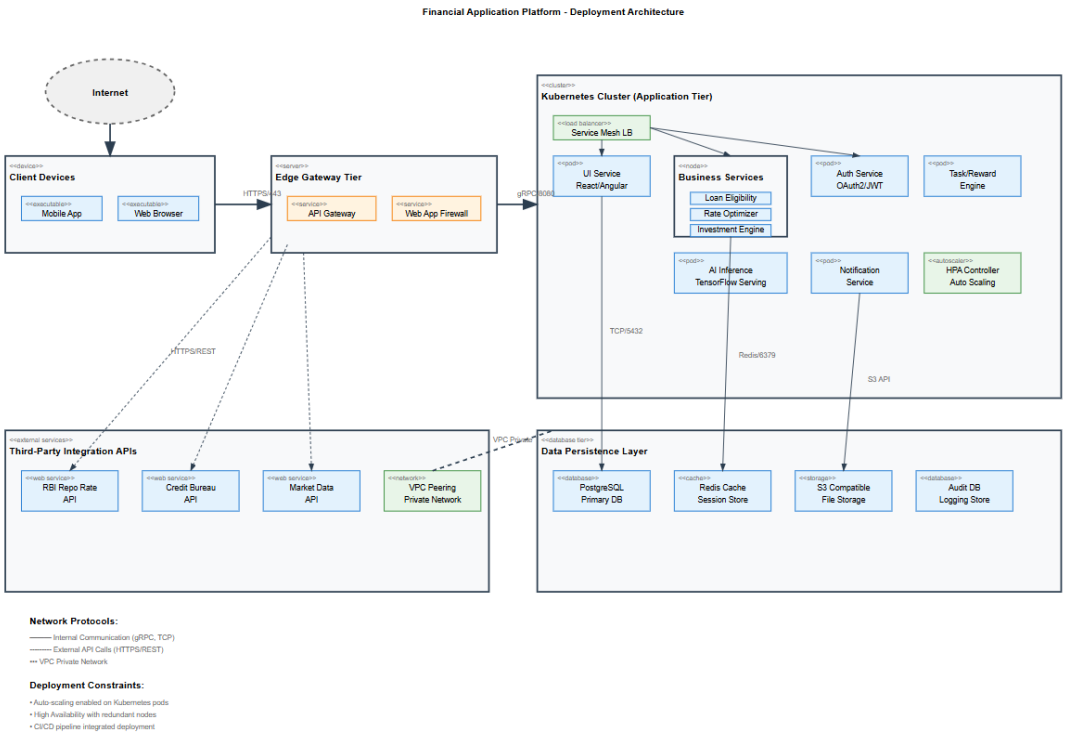
ML Prediction Processing:

1. Input data validation and preprocessing
2. Ensemble model prediction execution
3. Result aggregation and confidence scoring
4. Prediction logging and audit trail

Background Processing:

- Scheduled data synchronization tasks
- Report generation and distribution
- System maintenance and optimization
- Backup and archival operations

4.6 Deployment View



Production Environment:

- Load-balanced web servers for frontend delivery
- Application servers for business logic processing
- Database cluster with replication for high availability
- File storage system for document management

Development Environment:

- Local development servers with hot-reload capability
- Testing databases with sample data
- Continuous integration and deployment pipeline
- Code quality and security scanning tools

5. GENERAL ARCHITECTURE FOR CORE TECHNICAL SERVICES

5.1 Persistence

The system employs a layered persistence strategy to ensure data integrity, scalability, and modularity. The primary relational datastore is MySQL 8.0, selected for its ACID compliance, broad driver support, and operational maturity. Redis is used as a secondary cache layer for frequently accessed key-value data such as user sessions and computed eligibility scores.

Trade-offs:

- MySQL offers transactional integrity but exhibits performance limitations for high-throughput analytics workloads.
- NoSQL solutions were excluded due to schema enforcement requirements and referential integrity needs.

Build, Buy, or Reuse Analysis:

- **Buy**: Commercial OLAP systems were evaluated and rejected due to licensing overhead.
- **Reuse**: ORM capabilities are reused where applicable.
- **Build**: Custom DAO layers are built to encapsulate business rules within data access logic, maintaining separation of concerns.

5.2 Inter-Process Communication

The architecture uses a hybrid communication approach combining REST (synchronous) and gRPC (asynchronous and high-performance) for service-to-service interactions. Kafka is adopted for eventual consistency use cases such as audit logging, analytics, and broadcast messaging.

Trade-offs:

- REST simplifies debugging and external integration but adds latency.
- gRPC supports high-throughput internal service calls with strict schema enforcement.
- Kafka ensures resilience and decoupling but adds operational complexity.

Build, Buy, or Reuse Analysis:

- **Buy**: Managed Kafka service via Confluent Cloud is under evaluation.
- **Reuse**: Protocol Buffers for gRPC communication are shared across services.

- **Build**: Custom retry and circuit-breaker middleware are built to standardize fault tolerance across IPC channels.

5.3 Authentication and Authorization

Authentication follows OAuth 2.0 with JWT-based token issuance for stateless sessions. Authorization is role-based (RBAC) with policy enforcement centralized via an AuthService.

Trade-offs:

- OAuth 2.0 ensures compatibility with modern identity providers but increases implementation complexity.
- JWT enables scalability but requires secure token expiration and refresh mechanisms.

Build, Buy, or Reuse Analysis:

- **Buy**: Keycloak and Auth0 were evaluated; Keycloak is currently integrated for enterprise-grade user federation.
- **Reuse**: OpenID Connect protocol support from Keycloak is reused.
- **Build**: Custom RBAC policy enforcer is implemented for fine-grained admin-level permissions.

5.4 Error Handling

A centralized error handling framework captures and categorizes errors at the controller, service, and repository layers. All errors are normalized into application-level error codes, with severity tags assigned for downstream processing.

Trade-offs:

- Centralized error reporting improves traceability but increases coupling with the logging framework.
- Decentralized approaches reduce coupling but limit observability.

Build, Buy, or Reuse Analysis:

- **Reuse**: Application-wide error schema and HTTP status standards are reused across modules.
- **Build**: Custom middleware for exception interception and user-facing sanitization is developed.
- **Buy**: No external solutions are procured.

5.5 Logging

Structured logging is implemented using Fluentd for log shipping and aggregation, ElasticSearch as the indexing backend, and Kibana for visualization. All services log events in JSON format.

Trade-offs:

- Structured logs improve machine readability but increase storage costs.
- Log aggregation pipelines increase reliability but introduce latency for real-time queries.

Build, Buy, or Reuse Analysis:

- **Buy**: ELK Stack is deployed via managed service provider.
- **Reuse**: Common log schema is shared across services.
- **Build**: Contextual trace ID injection and log masking layers are developed in-house.

5.6 Transaction Management

Transactional integrity is ensured using ACID-compliant transactions via MySQL's InnoDB engine. Distributed transactions are minimized by adhering to domain-driven boundaries and utilizing eventual consistency where cross-service coordination is required.

Trade-offs:

- Strong consistency ensures data accuracy but may block processing in failure scenarios.
- Eventual consistency improves availability but introduces complexity in reconciliation.

Build, Buy, or Reuse Analysis:

- **Reuse**: Existing MySQL transaction mechanisms are utilized.
- **Build**: Compensation logic is implemented for rollbacks in distributed flows.
- **Buy**: No third-party transaction coordinator tools are used.

5.7 Other Applicable Technical Services

Caching:

Redis is used for short-lived, frequently accessed data. Caching patterns follow the cache-aside model. TTL policies are enforced to prevent stale data exposure.

Internationalization:

Message bundles are externalized using ISO 639 locale tags. Admin dashboard supports future translation overlays via JSON metadata.

Validation Framework:

Server-side validation leverages decorators (e.g., Joi for Node.js services) for schema enforcement. Input filtering and sanitization layers prevent injection vulnerabilities.

Fault Tolerance:

Custom circuit breakers with fallback logic are embedded in all external service invocations. Retry policies follow exponential backoff patterns.

Client and Server Initialization:

Service bootstrap routines include dependency injection resolution, health check registration, and dynamic configuration loading from Vault.

Installation Mechanism:

Deployment is orchestrated using Helm charts. A centralized CI/CD pipeline triggers blue-green deployments with automated rollback support.

6. RISKS / LIMITATIONS

The architecture is designed for modularity, scalability, and extensibility; however, several risks and limitations have been identified that may impact the system's operational and strategic objectives.

6.1 Technical Debt Accumulation

Modularization introduces risk of inconsistent design standards across independently developed services. Without stringent governance and design reviews, duplicated logic or redundant data access patterns may emerge, increasing long-term maintenance overhead.

6.2 External Dependency Vulnerability

The system leverages multiple third-party components, including open-source frameworks, authentication providers, and cloud-native infrastructure services. Any deprecation, licensing change, or supply chain vulnerability in these components poses a significant risk to system continuity and security posture.

6.3 Data Consistency Across Services

Adoption of eventual consistency and event-driven design for performance and decoupling introduces latency in data propagation. In scenarios involving transaction reversals, concurrent updates, or multi-service orchestration, the risk of temporary data divergence must be actively managed.

6.4 Single-Region Deployment Limitation

Initial deployment is confined to a single cloud region. This architecture lacks geo-redundancy and may suffer from regional outage risks, adversely affecting availability commitments.

6.5 Resource Contention in Peak Load Conditions

Under high concurrency scenarios, contention on shared resources such as database connections, in-memory caches, and message queues may degrade system performance. Without predictive autoscaling and rate-limiting controls, critical service SLAs may be violated.

6.6 Limited Real-Time Analytical Support

The system prioritizes OLTP workloads and does not currently integrate real-time analytical processing. Reporting and forecasting modules rely on batch-based pipelines, introducing latency in metrics visibility.

6.7 Complex Cross-Domain Authorization Logic

Due to the coexistence of diverse user roles (e.g., customers, agents, administrators), authorization logic becomes non-trivial. Maintaining a centralized RBAC system may result in rigid policy structures and delayed onboarding of new role definitions.

6.8 Security Surface Expansion

Microservices, while modular, increase the attack surface across network boundaries. Improperly secured endpoints, misconfigured service meshes, or exposed metadata endpoints may be exploited without rigorous API gateway and perimeter defense strategies.

6.9 Biometric and AI Module Compliance Risk

Biometric authentication and AI-driven decision components are subject to evolving regulatory standards (e.g., GDPR, RBI guidelines). Non-compliance may result in legal liability or forced architectural rework.

6.10 Client-Side State Management Risk

Session continuity relies partially on client-side storage of JWTs. If improperly handled in the client, tokens may be vulnerable to theft or misuse, especially in shared or unmanaged devices.

7. ALTERNATIVE SOLUTIONS CONSIDERED

Multiple architectural alternatives were evaluated during the design phase to address core system requirements such as modularity, scalability, real-time prediction, and operational maintainability. Each alternative was assessed on technical feasibility, alignment with business objectives, time-to-market impact, and integration complexity.

7.1 Monolithic Architecture

A single-tiered monolithic design was initially evaluated due to its lower initial setup and simplified deployment process.

Rationale for Rejection:

- Limited scalability and maintainability for high-volume or feature-rich systems.
- Increased risk of service-wide failure from localized issues.
- Inefficient CI/CD and testing pipelines due to tightly coupled components.

7.2 Serverless Event-Driven Architecture

A fully serverless model based on managed functions (e.g., AWS Lambda or Azure Functions) was considered for its elasticity and low infrastructure overhead.

Rationale for Rejection:

- Cold start latency inconsistent with real-time loan prediction requirements.
- Increased architectural complexity in orchestrating distributed business logic.
- Vendor lock-in concerns and limited portability across cloud platforms.

7.3 Graph-Based Knowledge Architecture for Decision Engines

A graph-based approach was explored for modelling dependencies between eligibility parameters, user behaviour, and financial outcomes.

Rationale for Rejection:

- Over-engineered for current scope; unsuitable for deterministic business rules.
- Lack of mature enterprise tooling for hybrid graph and relational operations.
- High implementation effort with uncertain performance benefits at current scale.

7.4 Centralized API Gateway with Modular Microservices

A gateway-based microservices architecture was selected as the final solution.

Justification:

- Clear separation of concerns across user interface, business logic, and data layers.

- High reusability and modularization enabling parallel development and deployment.
- Supports stateless service design, facilitating container-based orchestration.
- Enables fine-grained security control, rate-limiting, and observability.
- Ensures alignment with future cloud-native scalability requirements.

This architecture also allows incremental adoption of AI, gamification, and analytics capabilities without requiring major reengineering of the core platform.

8. APPENDIX

8.1 Expected Software Response

The software is expected to deliver consistent, deterministic responses across identical inputs under defined load conditions. All core modules—including loan eligibility, simulator, dynamic pricing, and investment advisory—must return valid outputs within the SLA boundaries. Security, data integrity, and compliance checks must not degrade functional accuracy under high concurrency. AI-driven predictions should remain explainable and bounded within pre-calibrated confidence levels.

8.2 Performance Bounds

The following performance bounds are defined for the system:

- **User Response Time**: All UI-facing endpoints must respond within 400 milliseconds (P95) under normal load, and within 750 milliseconds under peak concurrency (≤ 2,000 concurrent sessions).
- **Prediction Engine Latency**: Maximum end-to-end latency of 800 milliseconds per model inference, including pre-processing and post-processing.
- **Loan Processing Throughput**: The system must support ≥ 100 loan applications per minute with linear horizontal scalability.
- **Notification Broadcast**: Latency must not exceed 3 seconds across user cohorts ≥ 50,000 records.

8.3 Identification of Critical Components

The following components are deemed critical due to their business impact, complexity, or cross-module dependencies:

- **Eligibility Engine**: Core to decision-making workflows; errors impact creditworthiness assessment.
- **AI Recommendation Models**: Used in pricing, investment, and default risk prediction; directly influence customer outcomes and compliance.
- **Authentication Service**: Controls access, session validity, and role-based view segregation.
- **Data Synchronization Layer**: Ensures consistency between transactional and analytical stores; failure may lead to reporting inaccuracies.

- **Gamification Engine**: Drives engagement loops; tightly coupled with rewards and behavioural incentives.

8.4 Review Comments on Architectural POC

A limited-scope Proof of Concept (POC) was conducted on the following architectural components: microservice orchestration, eligibility computation engine, and notification delivery via message queue. Review outcomes:

- **Scalability Validated**: Services performed well under simulated concurrent loads.
- **Resilience Observed**: Graceful degradation and fallback logic were observed during induced service failures.
- **Integration Gaps Identified**: Minor latency overheads in chaining the prediction engine with the eligibility module; resolved via asynchronous decoupling.
- **Security Auditing**: Static code analysis flagged excessive privilege allocation in API Gateway policies; remediated before production hardening.

8.5 Justification of Changes to Existing Architecture

Changes introduced relative to the baseline architecture include:

- **Transition to Stateless Services**: Previously session-dependent modules restructured for container orchestration compatibility.
- **Introduction of Async Processing Pipelines**: To offload long-running loan simulation and scoring workflows and reduce API response times.
- **Incorporation of AI-Driven Modules**: Introduced machine learning inference layers with side-channel feedback to improve explainability and accuracy.
- **RBAC Framework Extension**: Legacy role systems were replaced by a scalable policy-based access framework integrated with service-level scope checks.