Wallee.

Personal Finance Application

Technical Design Document

Senior Design Project — University Capstone Presentation

Team Members

Emma Bahr | Kyle Gibson | Joshua Cajuste | Matteo Caruso

Table of Contents

1. Executive Summary

The landscape of personal finance management presents a persistent and largely unresolved challenge for a significant demographic of users: students, freelancers, gig-economy workers, and young professionals whose income streams are irregular, variable, or project-based. Traditional budgeting applications operate under the implicit assumption of a fixed, predictable monthly income — a model that systematically fails individuals whose financial reality is defined by biweekly paychecks of fluctuating value, seasonal income spikes, or multiple concurrent revenue sources. The resulting friction between tool design and user need produces disengagement, budgeting abandonment, and, ultimately, financial vulnerability.

1.1 Problem Statement

Existing personal finance tools — including industry incumbents such as Mint, YNAB (You Need A Budget), and Personal Capital — exhibit several critical design deficiencies when evaluated against the needs of variable-income users. First, their budgeting engines are fundamentally static: they require manual updates each pay cycle and do not automatically recalibrate spending limits following paycheck receipt. Second, their AI or recommendation features, where present, are implemented as thin wrappers over generic large language models (LLMs), producing advice that is plausible in tone but mathematically unverified and frequently inconsistent with the user's actual financial state. Third, they offer no mechanism for real-time financial health scoring or gamified engagement that would incentivize disciplined financial behavior. Fourth, their user interfaces prioritize data density over comprehension, overwhelming users with raw transaction data and poorly contextualized graphs.

The result is a market gap: no single application simultaneously offers automated paycheck-aware budgeting, rigorously verified AI-generated financial guidance, a dynamic health-scoring progression system, and a modern, cognitively accessible user experience tailored to the variable-income demographic.

1.2 Proposed Solution: Wallee

Wallee is a cross-platform personal finance mobile application engineered to address each of these deficiencies through a combination of purpose-built architectural decisions and novel AI design. The application integrates securely with users' bank accounts via the Plaid financial data API, ingesting real-time transaction data without ever exposing raw banking credentials to the application backend. A Python-based dynamic budgeting engine recalculates spending allowances automatically upon paycheck detection, distributing available funds across user-defined spending categories and long-term financial goals.

The centerpiece of Wallee's intelligence layer is a proprietary Two-Layer AI architecture. Rather than deploying a single general-purpose LLM against raw financial data — a design pattern prone to hallucination and mathematical error — Wallee separates the computational verification of

financial health (handled by a deterministic engine designated Wallee Zero) from the natural language communication of that health state (handled by a conversational AI agent designated Wallo). This separation of concerns eliminates an entire class of AI reliability failures: the conversational layer never performs arithmetic; it only translates pre-verified metrics into human-readable guidance.

1.3 Key Value Proposition

Capability	Description
Automated Budgeting	Bi-weekly budget recalculation triggered by paycheck detection; no manual user intervention required.
Verified AI Advice	Two-Layer AI architecture guarantees mathematical accuracy before any natural language output is generated.
Real-Time Bank Sync	Plaid OAuth integration provides live transaction ingestion without credential exposure.
Health Progression	Gamified four-axis financial health scoring system provides actionable, motivating feedback.
Accessible Design	Glassmorphism UI with haptic feedback reduces cognitive load and increases engagement.

2. Core Features & Functionality

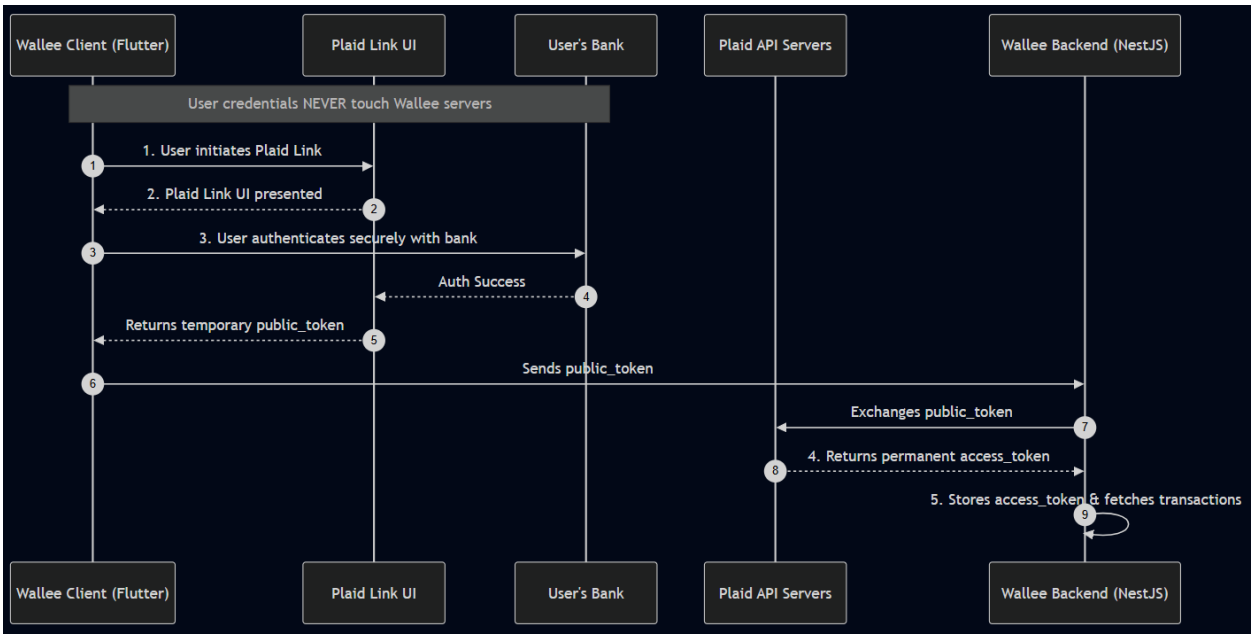
Wallee's feature set is designed around a single organizing principle: eliminating the cognitive and manual overhead of personal finance management so that users can make informed financial decisions with minimal friction. Each feature is the product of a deliberate engineering trade-off between automation depth, data accuracy, and user agency.

2.1 Bank Account Integration

Wallee establishes a secure, read-only connection to users' financial institutions through the Plaid API — the industry-standard financial data aggregation layer trusted by tens of thousands of applications and covering over 11,000 financial institutions in North America. The integration is initiated through a Plaid Link OAuth flow embedded within the Flutter client, which guides users through institution selection and credential entry entirely within Plaid's own secure environment. Wallee's backend receives only a non-sensitive access token upon successful authentication; raw credentials are never transmitted to or stored by Wallee's infrastructure.

Once linked, Plaid provides two categories of data critical to Wallee's operation: (1) real-time transaction webhooks, which notify the backend within seconds of a new transaction being posted to the user's account, and (2) balance data, which reflects current available funds. This dual data stream enables Wallee to maintain an accurate, live financial ledger without requiring any manual input from the user after initial account linkage.

Figure 1:



2.2 Automated Transaction Categorization

Every transaction ingested from Plaid is processed through a two-tier categorization pipeline. In the first tier, Plaid's native categorization taxonomy — which assigns each transaction to one of over 300 hierarchical categories (e.g., 'Food and Drink > Restaurants > Fast Food') — is mapped to Wallee's internal simplified category schema. This schema is designed for user comprehension rather than exhaustive taxonomy, grouping transactions into intuitive buckets such as Groceries, Dining Out, Rent & Utilities, Transportation, Entertainment, Healthcare, and Savings Transfers.

In the second tier, users are empowered to override or refine these assignments at the transaction level, creating custom categories or reclassifying transactions that Plaid's taxonomy miscategorized (a common occurrence with merchant names that encode no obvious categorical information). User-defined overrides are persisted to the PostgreSQL database and applied retroactively to past transactions from the same merchant, creating a continuously improving personalization layer that requires no machine learning infrastructure.

2.3 Dynamic Budgeting Engine

The Dynamic Budgeting Engine represents the most technically novel component of Wallee's feature set. Unlike static budgeting tools that reset to a fixed monthly allocation, Wallee's engine operates on a bi-weekly cadence synchronized with the user's actual paycheck receipt. When the Python FastAPI service detects an income transaction — identified by its Plaid category classification as 'Payroll', 'Direct Deposit', or a user-confirmed income source — it triggers a full budget recalculation cycle.

The recalculation algorithm operates as follows: the engine first computes the total amount of recurring fixed obligations (rent, subscription services, insurance premiums, loan payments) due within the current bi-weekly period, deriving these values from historical transaction patterns and user-confirmed bill schedules. It then subtracts these fixed obligations from the deposited income to produce a 'Discretionary Surplus' figure. This surplus is distributed across the user's variable spending categories according to predefined allocation percentages, with any remaining balance directed toward active savings goals.

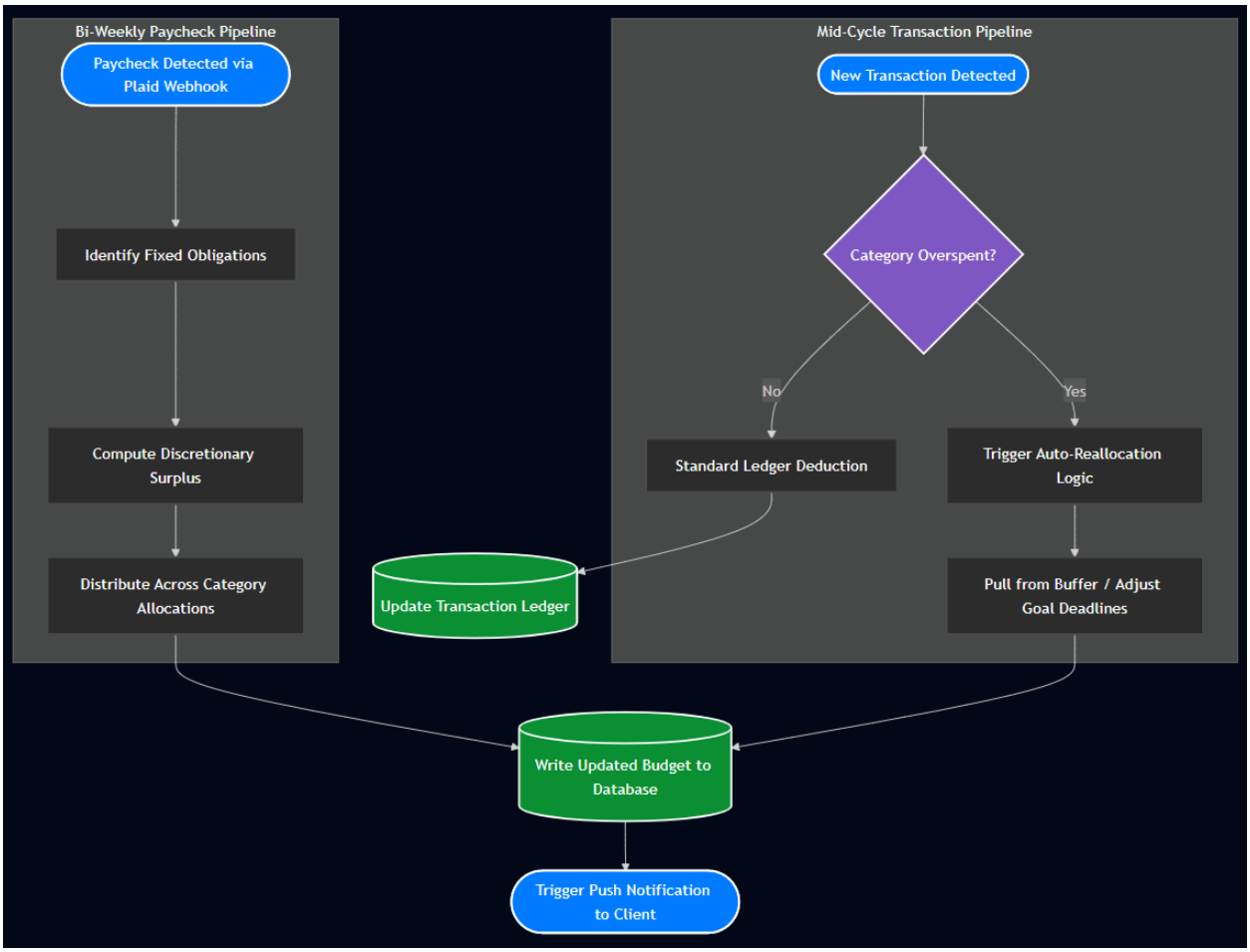
Dynamic Budget Recalculation: Worked Example

Scenario: User receives a $2,400 paycheck. Fixed obligations for the period total $1,100 (rent: $800, subscriptions: $120, insurance: $180). Discretionary Surplus = $1,300.

Allocation: Groceries 20% = $260 | Dining Out 10% = $130 | Transportation 15% = $195 | Entertainment 8% = $104 | Savings Goal 20% = $260 | Unallocated Buffer 27% = $351

Mid-period Overspend Event: User overspends on Dining Out by $50. Engine recalculates: reduces Entertainment allocation by $30 and extends active savings goal deadline by 4 days, maintaining budget integrity without user intervention.

Figure 2:



2.4 Financial Goals & Tracking

The financial goals module allows users to define both short-term objectives (e.g., saving $500 for a laptop within 60 days) and long-term objectives (e.g., accumulating a $10,000 emergency fund within 18 months). Each goal is represented as a database record with target amount, current progress, target date, and priority rank. The budgeting engine treats active goals as first-class participants in the allocation algorithm: goals with higher priority ranks receive preferential allocation from the discretionary surplus, and the engine dynamically adjusts contribution amounts when income is higher or lower than typical, accelerating or decelerating goal timelines accordingly.

Progress toward each goal is surfaced in the mobile client through animated circular progress indicators, percentage completion labels, and projected completion dates. When goal trajectories are endangered by spending patterns — for instance, when three consecutive over-budget weeks are detected — the Wallo AI agent proactively surfaces an advisory notification, quantifying the projected timeline extension and suggesting concrete corrective actions.

2.5 AI Financial Assistant (Wallo)

Wallo is the user-facing persona of Wallee's AI intelligence layer. Accessible through a conversational chat interface within the mobile application, Wallo responds to natural language queries about the user's financial state (e.g., 'How much can I spend on dinner this week?' or 'Am I on track to meet my savings goal?') with responses that are grounded exclusively in pre-verified financial data rather than the raw outputs of a general-purpose LLM.

The architectural mechanism that enables this verifiability is detailed in Section 4 (Two-Layer AI Architecture). From a user-experience perspective, Wallo presents as a friendly, non-judgmental financial advisor that communicates in plain language, avoids financial jargon, and calibrates the complexity of its explanations to the user's demonstrated level of financial literacy. Wallo also functions in a proactive capacity, generating unsolicited advisory notifications at financially significant moments: paycheck receipt, approach to a spending category limit, detection of an unusual or potentially fraudulent transaction, and proximity to a tax payment deadline for freelancer users.

2.6 Reports & Visualizations

Wallee generates automated financial reports on two schedules: bi-weekly summaries triggered by each paycheck cycle, and on-demand weekly and monthly summaries accessible through the Reports section of the mobile client. Each report presents a structured comparison of budgeted versus actual spending for each category, rendered as a horizontal bar chart with color-coded over/under indicators. Trend graphs display spending trajectories over rolling three-month windows, enabling users to identify seasonal patterns in their expenditure.

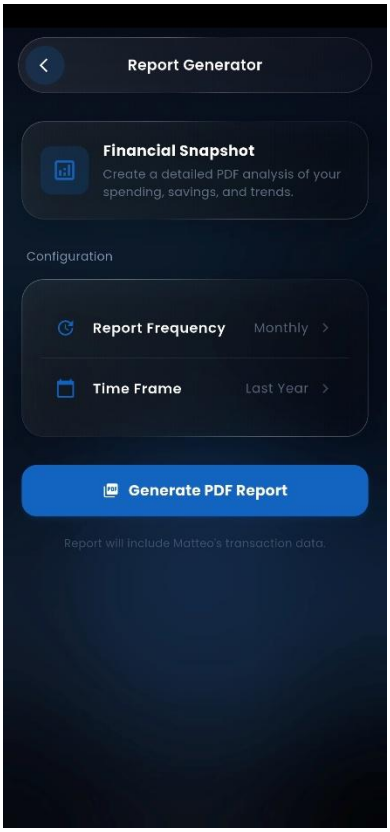


Figure 3: The Report Generator Configuration (Left)
The in-app Report Generator interface, allowing users to define the temporal scope of their financial snapshot. This portal serves as the trigger for the on-demand PDF generation, pulling live transactional data from the Supabase backend.

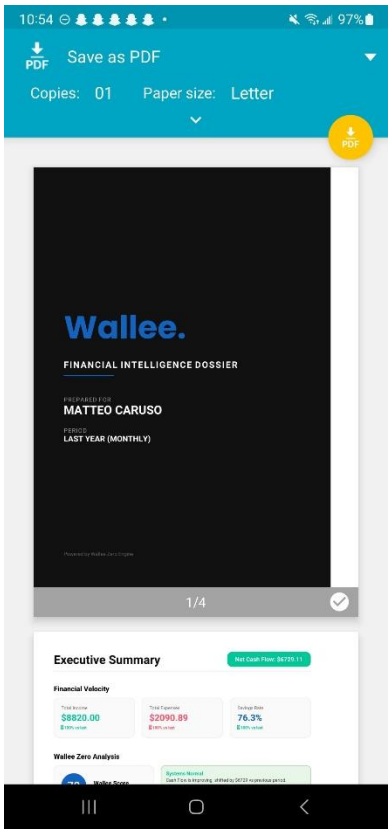


Figure 4: Automated Report Cover Page (Right)
The generated "Financial Intelligence Dossier." The PDF export utilizes a premium, high-contrast design language, dynamically inserting the user's name and the selected reporting period directly onto the cover.

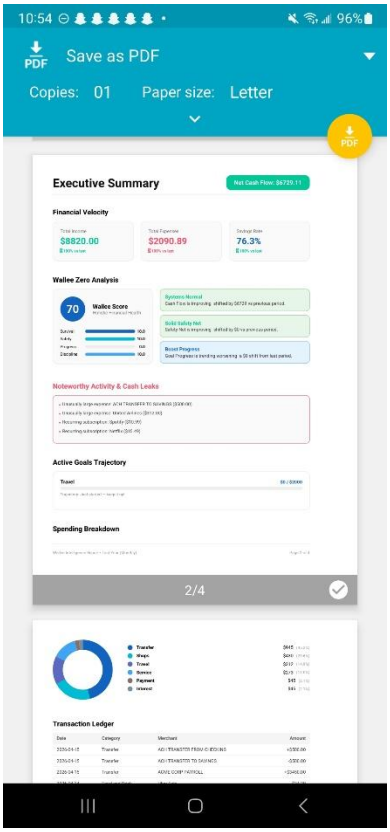


Figure 5: Executive Summary & Wallee Zero Analysis (Left)
Page two of the automated report features the Executive Summary. This includes the "Financial Velocity" metrics (Income, Expenses, Savings Rate) and the Wallee Zero Analysis, which breaks down the user's Wallee Score and automatically flags potential "Cash Leaks," such as unusually large expenses or recurring subscriptions.

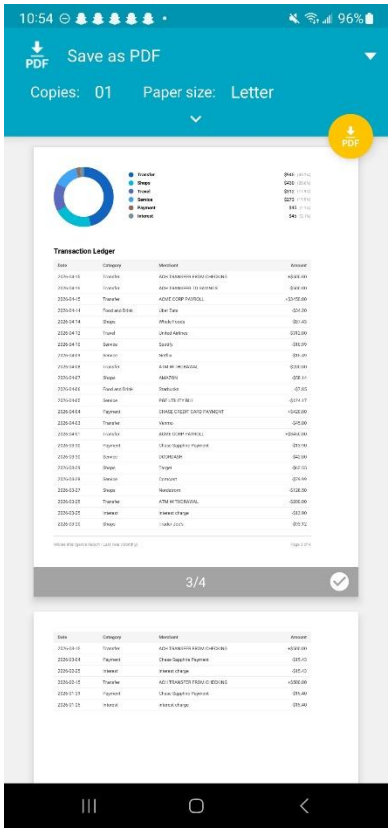
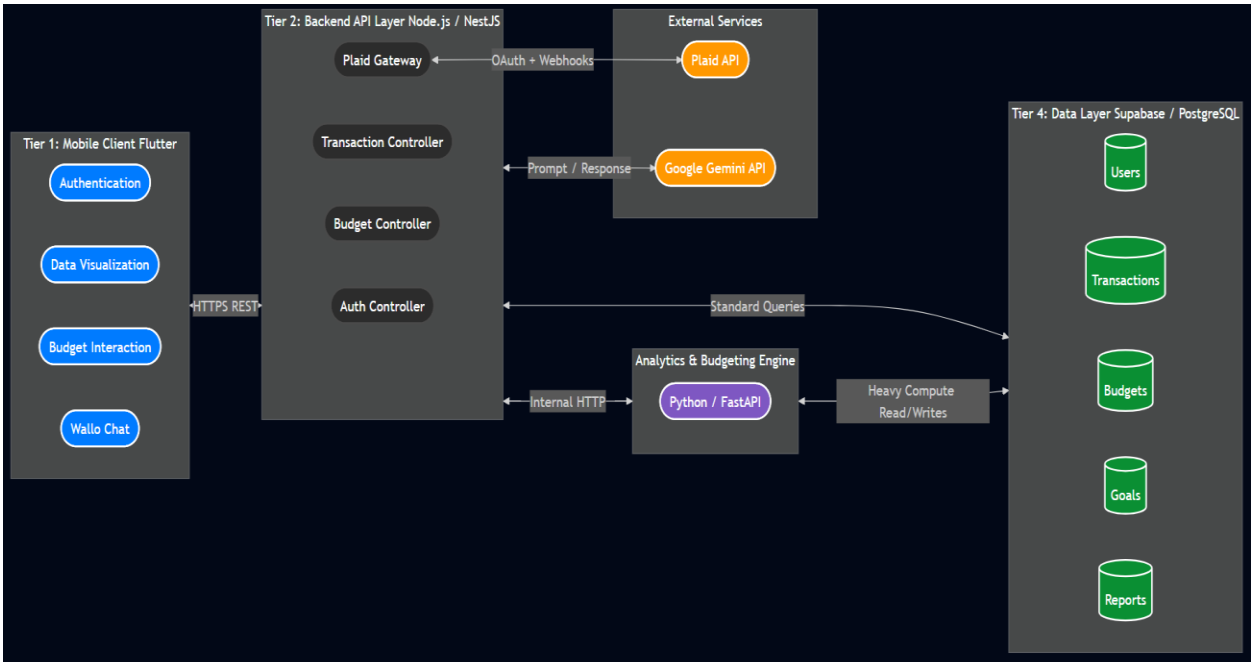


Figure 6: Spending Breakdown & Transaction Ledger (Right)
The final sections of the report provide a granular spending breakdown via a categorized donut chart, alongside a complete, itemized Transaction Ledger for the selected timeframe.

3. System Architecture

Wallee is constructed upon a modular, service-oriented architecture that deliberately partitions responsibilities across discrete layers. This design philosophy — often referred to in software engineering literature as a separation-of-concerns architecture — provides three primary benefits relevant to Wallee's technical requirements: (1) independent scalability, allowing the computationally intensive analytics service to be scaled without scaling the entire backend; (2) fault isolation, ensuring that a failure in the Python analytics service does not propagate to the authentication or transaction ingestion services; and (3) technology heterogeneity, permitting each layer to be implemented in the programming language and framework best suited to its specific responsibilities.

Figure 7:



3.1 Mobile Client (UI Layer)

The mobile client is implemented in Flutter, Google's open-source UI toolkit for building natively compiled, cross-platform applications from a single codebase. Flutter was selected over React Native and native Swift/Kotlin development for three reasons aligned with the project's constraints. First, Flutter's widget-based rendering engine draws directly to a hardware-accelerated canvas via the Skia graphics library, bypassing the native UI component bridge that introduces rendering inconsistencies in React Native. This property is particularly valuable for Wallee's Glassmorphism design aesthetic, which relies on precise blur and opacity rendering behaviors. Second, Flutter's single codebase produces identical behavior on both iOS and Android, eliminating the platform-specific debugging overhead that would constrain a four-person development team. Third, Flutter's mature state management ecosystem — specifically the

Riverpod or BLoC patterns — supports the reactive data flows required for real-time budget and transaction updates.

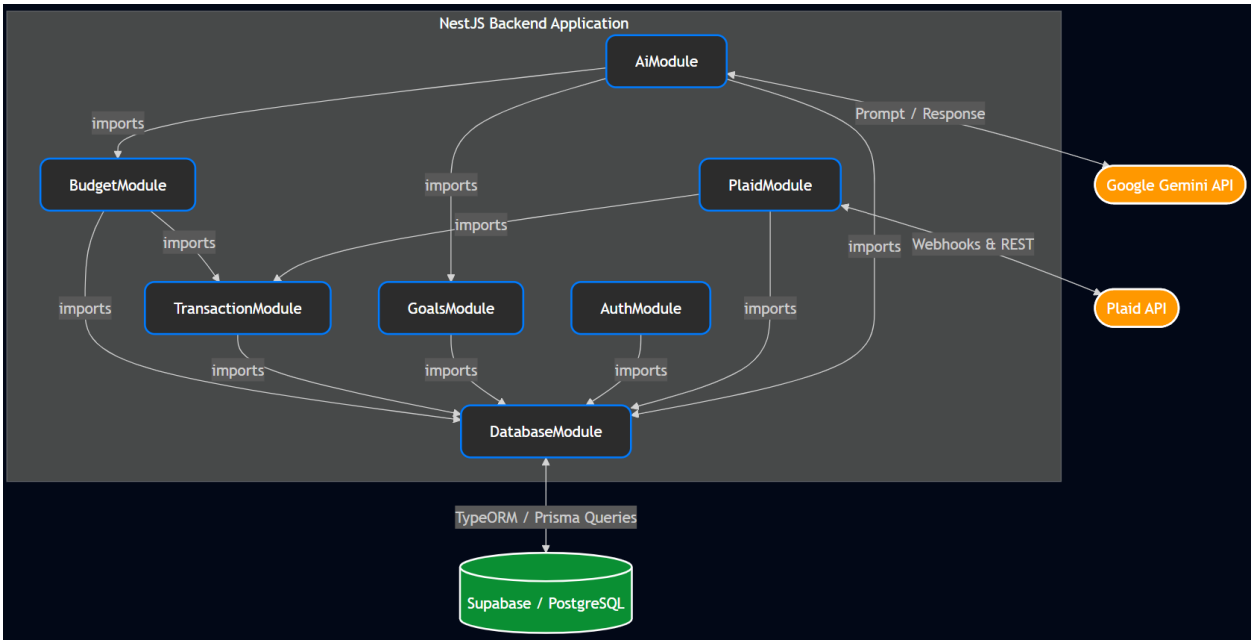
The client layer implements the following responsibilities: user authentication via Supabase Auth JWT tokens; secure storage of the Plaid access token in the device's encrypted keychain; rendering of all data visualization components (charts, progress indicators, goal trackers); the Wallo conversational chat interface; push notification handling for proactive AI alerts; and haptic feedback triggering synchronized with financial events. All communication with the backend occurs exclusively over HTTPS REST APIs, with request payloads authenticated via JWT bearer tokens validated on the NestJS API layer.

3.2 Backend API Layer (Node.js / NestJS)

The central API layer is built with Node.js and the NestJS framework. NestJS was chosen for its opinionated, modular architecture — which enforces a clean separation between controllers, services, and data access layers through its dependency injection system — and for its TypeScript-first design, which provides compile-time type safety across a codebase that handles sensitive financial data. NestJS's decorator-based routing and guard system simplifies the implementation of the authentication and authorization middleware required at every endpoint.

The NestJS application is structured into the following modules: AuthModule (handles JWT issuance and validation, delegates credential verification to Supabase Auth); PlaidModule (manages Plaid API communication, access token storage, and webhook receipt); TransactionModule (receives, persists, and serves transaction data); BudgetModule (exposes budget state to the client and proxies recalculation requests to the FastAPI service); GoalsModule (manages CRUD operations for user financial goals); and AiModule (constructs the verified financial context payload and manages requests to the Google Gemini API). The modular boundary enforced by NestJS ensures that a developer modifying the PlaidModule cannot inadvertently introduce regressions in the BudgetModule.

Figure 8:



3.3 Analytics & Budgeting Engine (Python / FastAPI)

The deliberate decision to implement the analytics and budgeting logic as a separate Python microservice, rather than co-locating it within the NestJS backend, warrants technical justification. Python's scientific computing ecosystem — specifically the NumPy, Pandas, and SciPy libraries — provides native, vectorized operations for the statistical analyses underlying Wallee's budget calculations (e.g., rolling averages for recurring expense detection, linear regression for spending trend projection, percentile computations for anomaly detection). Implementing equivalent functionality in TypeScript/Node.js would require either third-party libraries of inferior numerical performance or the duplication of algorithms that Python's ecosystem has already solved optimally.

The separation also provides a critical operational advantage: the Node.js event loop is non-blocking and single-threaded by design, making it well-suited for high-throughput I/O operations (API request handling, database queries) but poorly suited for CPU-bound numerical computations. If budget recalculation were performed synchronously within the NestJS process, a complex multi-goal recalculation could block the event loop for hundreds of milliseconds, degrading responsiveness for all concurrent API requests. The Python service receives recalculation requests asynchronously over an internal HTTP interface, performs the computation in a separate process, and returns the result without ever interrupting the main API's request-handling capacity.

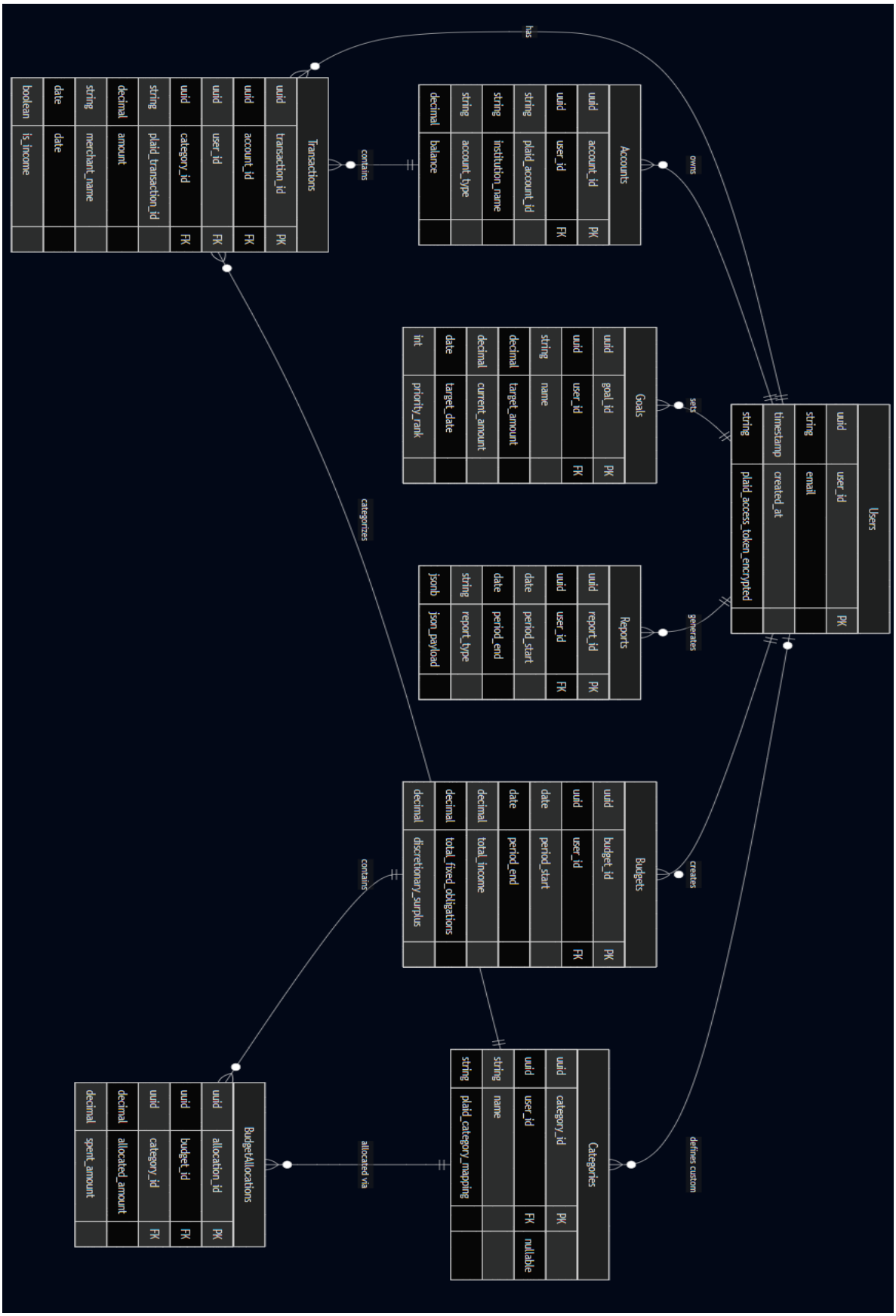
FastAPI was selected as the Python web framework over Flask and Django REST Framework for its native asynchronous I/O support (via Python's asyncio), automatic OpenAPI documentation generation, and Pydantic-based request/response validation, which provides type safety equivalent to TypeScript's compile-time guarantees at the Python service boundary.

3.4 Database Layer (Supabase / PostgreSQL)

Wallee's persistent data store is a PostgreSQL relational database hosted and managed through the Supabase platform. PostgreSQL was selected over NoSQL alternatives (e.g., MongoDB, Firebase Firestore) for its superior suitability for financial data: relational integrity constraints enforce the consistency rules required of a financial ledger (e.g., every transaction must reference a valid user and category; every budget allocation must sum to no more than the period's available income). ACID-compliant transactions ensure that budget recalculation writes are atomic, preventing inconsistent budget states from being committed during a partial write failure.

Supabase augments raw PostgreSQL with three capabilities directly relevant to Wallee's requirements: Row-Level Security (RLS) policies, which are enforced at the database engine level and guarantee that a user's query can never return data belonging to a different user regardless of application-layer bugs; real-time database subscriptions, which enable the Flutter client to receive live updates to transaction and budget records without polling; and Supabase Auth, which manages JWT-based authentication and integrates natively with RLS policies, tying query authorization directly to the authenticated user's identity.

Figure 9:


Users
uuid
user_id
PK
string
email
timestamp
created_at
string
plaid_access_token_encrypted
Accounts
uuid
account_id
PK
uuid
user_id
FK
string
plaid_account_id
string
institution_name
string
account_type
decimal
balance
Transactions
uuid
transaction_id
PK
uuid
account_id
FK
uuid
user_id
FK
uuid
category_id
FK
string
plaid_transaction_id
decimal
amount
string
merchant_name
date
date
boolean
is_income
Goals
uuid
goal_id
PK
uuid
user_id
FK
string
name
decimal
target_amount
decimal
current_amount
date
target_date
int
priority_rank
Reports
uuid
report_id
PK
uuid
user_id
FK
date
period_start
date
period_end
string
report_type
jsonb
json_payload
Budgets
uuid
budget_id
PK
uuid
user_id
FK
date
period_start
date
period_end
decimal
total_income
decimal
total_fixed_obligations
decimal
discretionary_surplus
Categories
uuid
category_id
PK
uuid
user_id
FK
nullable
string
name
string
plaid_category_mapping
BudgetAllocations
uuid
allocation_id
PK
uuid
budget_id
FK
uuid
category_id
FK
decimal
allocated_amount
decimal
spent_amount
has
owns
contains
sets
categorizes
generates
creates
contains
defines custom
allocated via

3.5 External Service Integrations

Service	Integration Role & Technical Detail
Plaid API	Provides secure bank connectivity via OAuth 2.0. Wallee uses the Transactions product for historical and real-time data, the Balance product for available fund queries, and Plaid's webhook system for real-time transaction event delivery. Integration is managed by the NestJS PlaidModule using the official plaid-node SDK.
Google Gemini	Powers the Wallo conversational AI agent. Gemini receives a structured prompt containing pre-verified financial context (health scores, category summaries, goal progress) from the AiModule and returns natural language responses. Gemini was selected over OpenAI GPT-4 for its superior context window length and cost-per-token ratio at the anticipated query volume.
Supabase Auth	Manages user registration, login, password reset, and JWT issuance. The NestJS backend validates Supabase-issued JWTs on every authenticated endpoint using the Supabase Admin SDK.
Supabase Realtime	Provides WebSocket-based change data capture (CDC) streams to the Flutter client, enabling live updates to transaction lists and budget displays without polling.

4. Two-Layer AI Architecture

The AI architecture underlying Wallee's financial intelligence capabilities represents the most architecturally novel component of the system and the primary technical differentiator from competing applications. This section documents the design rationale, component responsibilities, and data flow of the Two-Layer AI architecture, which comprises two distinct systems: Wallee Zero (the Logic Core) and Wallo (the AI Agent).

4.1 Design Rationale: The Hallucination Problem

The fundamental motivation for the Two-Layer architecture is the elimination of a class of failure known in AI safety literature as 'hallucination' — the tendency of large language models to generate confident, syntactically fluent outputs that are factually incorrect. In the domain of personal finance, this failure mode is not merely an annoyance; it is a material safety risk. An LLM responding to 'Can I afford a $400 car repair this week?' with a confidently incorrect 'Yes' based on a misread of the user's available balance could precipitate an overdraft, a missed bill payment, or a defaulted loan.

The standard mitigation strategy — providing the LLM with raw financial data in the prompt and instructing it to perform calculations — is insufficient. LLMs are not calculators; their arithmetic capabilities are fundamentally probabilistic rather than deterministic, and they exhibit documented failure patterns in multi-step financial arithmetic involving decimals, negative values, and percentage allocations. Any architecture that asks an LLM to compute the answer and then report it conflates two responsibilities that should be strictly separated: mathematical verification (a deterministic, testable computation) and natural language generation (a probabilistic, non-deterministic process).

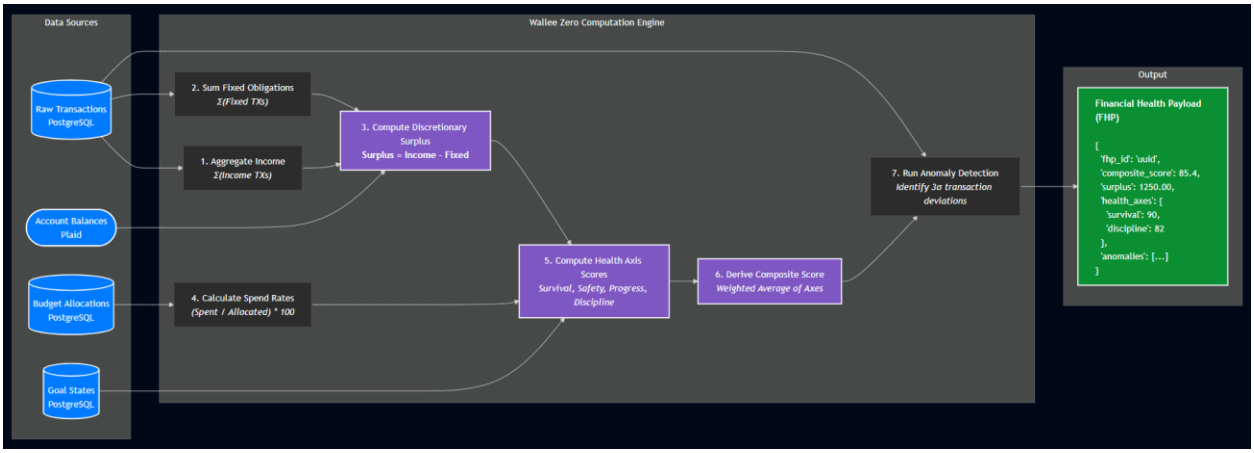
4.2 Layer One: Wallee Zero (Logic Core)

Wallee Zero is a deterministic, mathematically auditable computational engine implemented within the Python FastAPI service. It receives raw financial data from the PostgreSQL database — specifically, transaction records, account balances, budget allocations, and goal states — and produces a structured output object designated the Financial Health Payload (FHP). The FHP contains no free-form text; it is a precisely typed JSON document containing only numerically verified values and categorical classifications derived from those values.

Wallee Zero computes the following fields within the FHP: current account balance per linked account; total income for the current bi-weekly period; total fixed obligation expenditure; total variable expenditure per category, expressed both in absolute dollar terms and as a percentage of the category's allocation; discretionary surplus remaining; individual health scores across four axes (Survival, Safety, Progress, Discipline — detailed in Section 5); a composite Financial Health Score (FHS) normalized to a 0–100 scale; and anomaly flags for transactions exceeding a statistically significant deviation from historical merchant averages. Each computed value in the

FHP is accompanied by the precise input values and formula used to derive it, enabling full audit traceability.

Figure 10:



4.3 Layer Two: Wallo AI (Conversational Agent)

Wallo is the natural language interface layer that translates the verified numerical outputs of Wallee Zero into human-readable financial guidance. When a user submits a query to Wallo — either through the chat interface or implicitly through a proactive alert trigger — the NestJS AiModule performs the following sequence: it requests a fresh FHP from the Python service, constructs a structured prompt for the Google Gemini API, submits the prompt, and returns the response to the Flutter client.

The critical architectural constraint governing the Wallo prompt construction protocol is this: the prompt supplies Wallo exclusively with pre-verified values from the FHP. It does not supply raw transaction data, account balances, or any other numerical input that Wallo might be tempted to re-derive arithmetically. The prompt instructs Wallo explicitly: 'Do not perform any calculations. All numerical values provided are pre-computed and verified. Your task is to explain, contextualize, and advise based exclusively on the values provided.' This constraint is enforced at the application layer, not at the model layer, providing a deterministic guarantee rather than a probabilistic one.

Wallo Prompt Construction Protocol — Example

System Context supplied to Gemini:

'You are Wallo, a friendly and non-judgmental personal finance assistant for the Wallee app. You help users understand their financial health. Do not perform any calculations — all values below are pre-verified. Explain, contextualize, and advise only.'

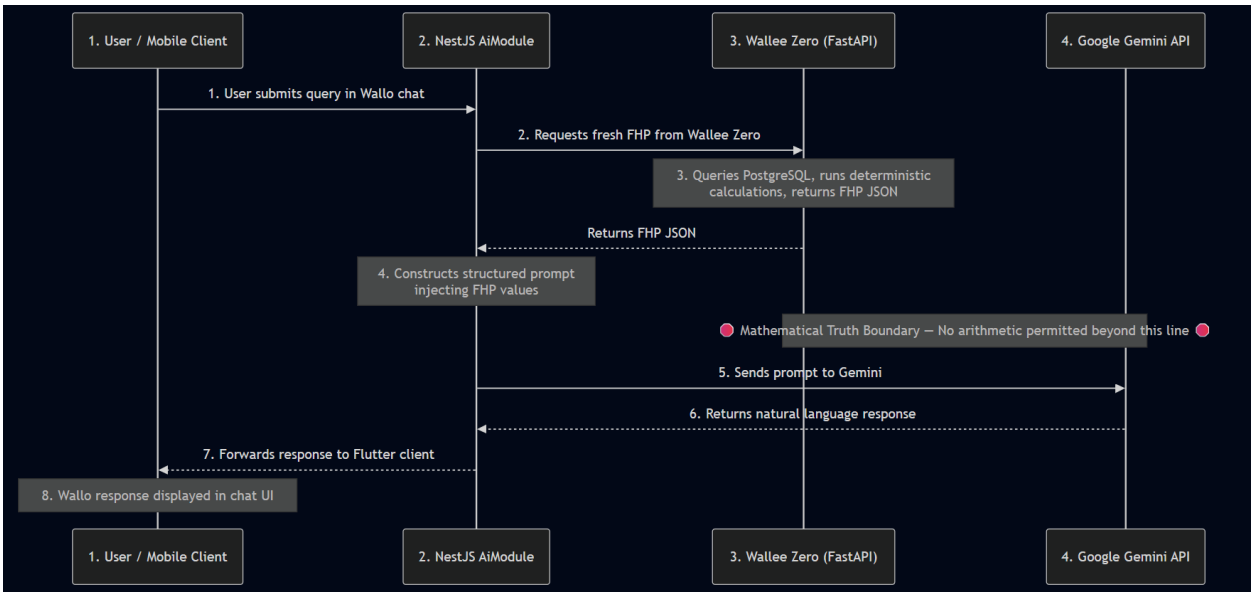
Pre-Verified Financial Context Block (from FHP):

Current Period: June 1–14. Income: $2,400.00. Fixed Obligations Paid: $1,100.00. Discretionary Surplus: $1,300.00. Dining Out Spent: $180.00 of $130.00 allocation (138% — OVER BUDGET). Health Score: 71/100 (Safety axis: 82, Discipline axis: 55). Active Goal: Emergency Fund — $1,240 of $5,000 (24.8%).

User Query:

'Why is my score lower this week?'

Figure 11:



4.4 Privacy Architecture of the AI Layer

A secondary benefit of the Two-Layer architecture is its privacy-preserving property. Because the Gemini API receives only a structured numerical summary (the FHP) rather than raw transaction data, no personally identifiable financial information — merchant names, specific transaction amounts, or account numbers — is ever transmitted to a third-party AI provider. The FHP contains aggregated category totals and normalized scores: information that is analytically meaningful to Wallo but unintelligible as a record of any individual's financial behavior. This design choice aligns with GDPR data minimization principles and provides a defensible data handling posture for user disclosure purposes.

5. Financial Health Progression System

The Financial Health Progression System is the motivational and diagnostic framework through which Wallee communicates a user's overall financial status. Rather than presenting users with an undifferentiated financial summary that requires expert interpretation, the system decomposes financial health into four independently scored axes — Survival, Safety, Progress, and Discipline — each of which captures a distinct and equally important dimension of financial wellbeing. This multi-dimensional approach is grounded in behavioral finance research, which demonstrates that users are more likely to engage with and act upon financial feedback when it is specific, attributable, and incremental.

Figure 12: The Wallee Score Dashboard (Left)
The primary Wallee Score interface. It aggregates overall financial health into a central metric (69) supported by four foundational axes. The Detailed Breakdown then translates raw transactional data into highly specific, 10-point success rates (e.g., Survival and Safety) to provide immediate, targeted diagnostics.

Figure 13: Diagnostic Axis Breakdown (Right)
A continuation of the Detailed Breakdown isolating specific behavioral metrics. By distinctly separating Progress from Discipline, the UI delivers the incremental, attributable feedback required by our behavioral finance framework. This allows users to quickly pinpoint areas needing attention—like the 0/10 Progress score—without the cognitive load of an undifferentiated summary.

5.1 Survival Axis

The Survival axis evaluates the most fundamental question of financial health: are the user's essential fixed obligations being met within the current period? This axis computes the ratio of fixed obligation expenditure to available income for the period. A score of 100 on the Survival axis indicates that all recurring fixed obligations (rent, utilities, loan payments, insurance premiums, and minimum credit card payments) have been or are projected to be paid in full within the current period with no shortfall. The score degrades proportionally as the projected fixed obligation total approaches or exceeds available income, reaching 0 when obligations exceed income by 20% or more.

The Survival axis is intentionally given the heaviest weighting (35%) in the composite Financial Health Score, reflecting the consensus view in financial planning literature that obligation fulfillment is a precondition for all other financial health dimensions. A user whose Survival score is critically low receives an immediate Wallo alert regardless of scores on other axes, as no amount of savings discipline compensates for an inability to meet fixed obligations.

5.2 Safety Axis

The Safety axis measures the adequacy of the user's financial buffer — the unallocated discretionary surplus remaining after all budget category allocations and goal contributions have been assigned. A robust safety buffer provides resilience against unexpected expenses (medical bills, car repairs, equipment failures) without requiring the user to incur debt or liquidate savings goals. The Safety axis score is computed as a logistic function of the unallocated surplus as a percentage of total income: a surplus of 15% or more of period income yields a score approaching 100, while a surplus below 5% yields a score below 30.

This logistic rather than linear scoring function is intentional: the marginal safety benefit of increasing a surplus from 5% to 10% of income is substantially greater than the benefit of increasing it from 20% to 25%, and the scoring curve reflects this diminishing return. The 25% weighting of the Safety axis in the composite score reflects its role as the primary buffer against financial emergencies.

5.3 Progress Axis

The Progress axis evaluates the trajectory of the user's goal contributions relative to the target contribution rate required to meet each active goal on schedule. For each active goal, Wallee Zero computes the required periodic contribution rate (derived from the goal's target amount, current balance, and target date) and compares it to the user's actual average contribution rate over the trailing three periods. The Progress axis score is the weighted average of per-goal on-track ratios, where goals with nearer target dates receive higher weights — reflecting the greater urgency of near-term commitments.

The Progress axis carries a 20% weighting in the composite score. It is the axis most responsive to the bi-weekly budgeting cycle: a user who receives an unusually large paycheck and directs

the surplus toward goals will see an immediate and visible improvement in their Progress score, providing a positive reinforcement mechanism that encourages goal-directed saving behavior.

5.4 Discipline Axis

The Discipline axis measures adherence to the user's self-defined variable spending category limits. For each variable spending category, Wallee Zero computes the ratio of actual spending to allocated budget for the current period, then aggregates these ratios into a composite adherence score. Perfect adherence across all categories yields a Discipline score of 100; each category overspend reduces the score in proportion to both the magnitude of the overspend and the category's share of total variable allocation.

The Discipline axis is assigned a 20% weighting, acknowledging that perfect category adherence, while desirable, is less critical to overall financial health than obligation fulfillment and safety buffering. The Discipline axis also provides the most granular diagnostic signal for Wallo's advisory function: when the Discipline score is low, Wallo can identify the specific category or categories responsible and quantify the corrective action required, providing actionable rather than generic guidance.

Health Axis	Weight	Score Input	Critical Threshold
Survival	35%	Fixed Obligations / Income	Score < 40 triggers alert
Safety	25%	Unallocated Surplus % of Income	Score < 30 triggers alert
Progress	20%	Actual vs. Required Goal Contributions	Score < 50 shown in Wallo advisory
Discipline	20%	Actual Spend vs. Category Allocation	Score < 60 shown in Wallo advisory

6. User Interface Design

Wallee's user interface design is governed by a single overarching principle: financial clarity without cognitive overload. The observation that motivates this principle is well-documented in the human-computer interaction literature: users who find financial applications visually overwhelming or cognitively demanding disengage from them, defeating the application's purpose regardless of the quality of its underlying data. Every design decision in Wallee's UI layer is therefore evaluated against this principle before implementation.

6.1 Design Philosophy: Context-Aware Glassmorphism

Wallee's visual aesthetic is based on a design system designated 'Context-Aware Glassmorphism' — a spatial design language adapted from the contemporary glassmorphism trend (characterized by translucent, frosted-glass UI elements) and extended with contextual color signaling that communicates financial state through visual appearance rather than explicit text labels.

The foundational glassmorphism elements — semi-transparent cards with backdrop-blur effects, soft-shadow depth cues, and gradient-infused borders — reduce the visual 'weight' of data-dense screens by creating a sense of layered depth that guides the user's eye to the most important information. The context-aware extension applies a dynamic color shift to these translucent surfaces: card backgrounds shift toward green-tinted translucency when the displayed metric is healthy, toward amber-tinted translucency when it is approaching a limit, and toward red-tinted translucency when a limit has been exceeded. This ambient financial signaling allows experienced users to assess their financial state at a glance without reading a single number.

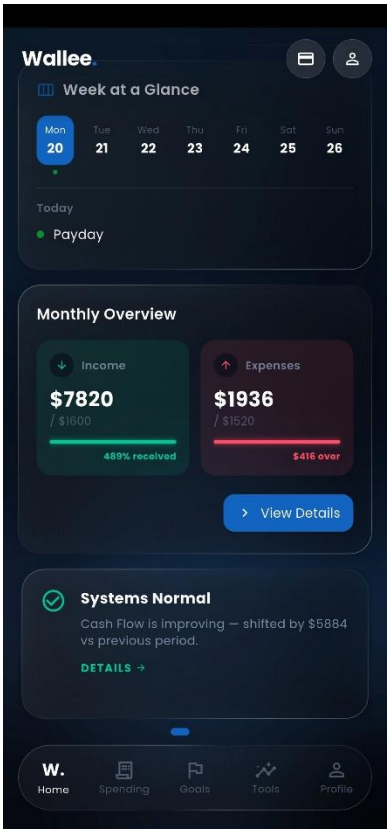


Figure 14: Context-Aware Glassmorphism Foundation
The Spending dashboard illustrates the core visual architecture. Semi-transparent cards with backdrop-blur effects create layered depth, reducing the visual weight of the data-dense interface. The top Income and Expenses cards utilize subtle green and red structural elements to establish the baseline contextual signaling.

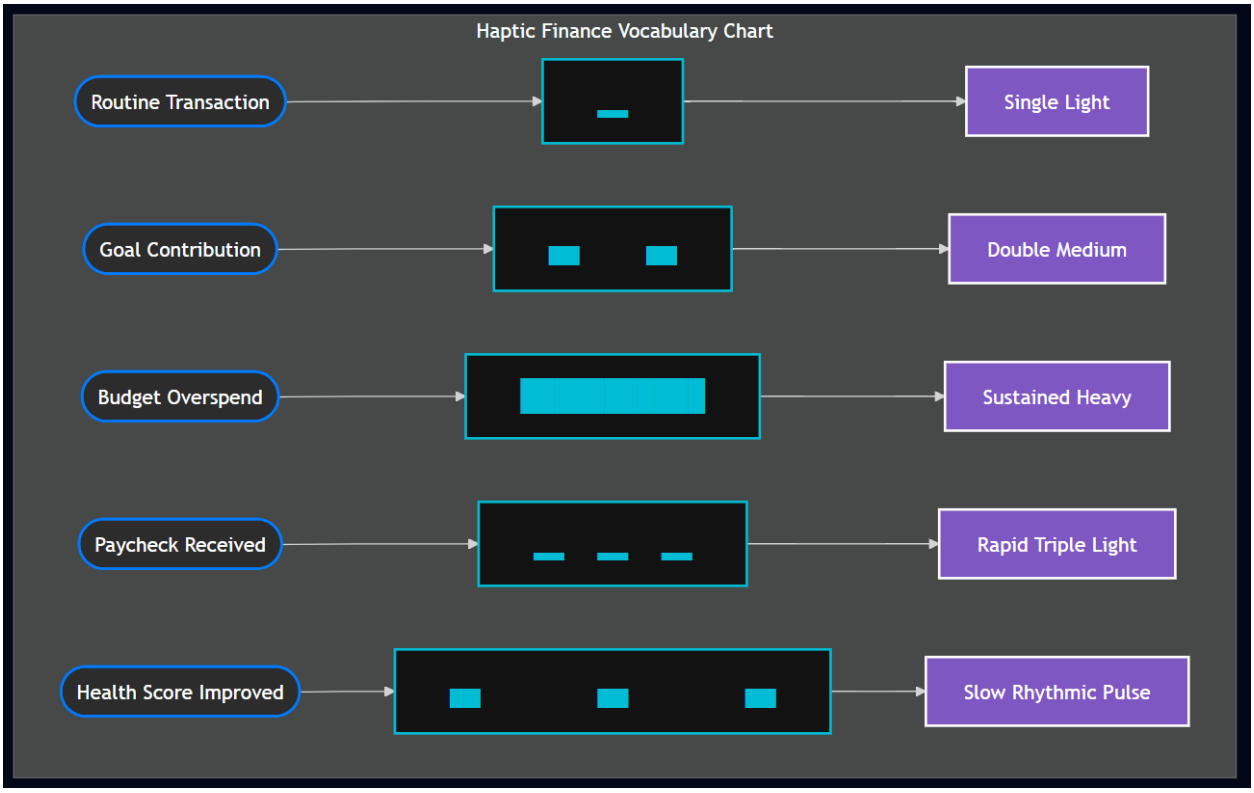
Figure 15: Ambient Financial Signaling
The Monthly Overview demonstrates dynamic color shifting in active use. The Income card adopts a green-tinted translucency to reflect a healthy surplus, while the Expenses card shifts to a red-tinted translucency to immediately communicate an exceeded budget limit ($416 over), enabling rapid, text-free state assessment.

6.2 Haptic Finance

A distinctive and novel aspect of Wallee's user experience is its systematic use of device haptic feedback — tactile vibrations generated by the device's linear resonant actuator (LRA) — to communicate financial events. This design pattern, designated 'Haptic Finance' within the project, extends the standard visual and auditory feedback channels with a proprietary haptic vocabulary that allows users to 'feel' the financial significance of their actions.

The haptic feedback system is implemented using Flutter's HapticFeedback API, which exposes iOS's UIImpactFeedbackGenerator and Android's VibrationEffect APIs through a unified interface. The system defines a vocabulary of five distinct haptic patterns, each mapped to a specific category of financial event: a single light impact for a routine transaction notification; a double medium impact for a successful savings goal contribution; a sustained heavy impact for a budget category overspend alert; a rapid triple light impact for paycheck receipt; and a slow, rhythmic pulse for a positive financial health score improvement.

Figure 16:



6.3 Visual Hierarchy & Information Architecture

Wallee's information architecture is structured around a primary bottom navigation bar with five destinations: Home (financial health dashboard), Transactions (full transaction list with category filters), Budget (bi-weekly budget allocation view), Goals (goal tracking and contribution history), and Wallo (AI chat interface). This five-destination structure was validated through card-sorting exercises with representative users in the design phase and represents the minimum navigation depth required to surface Wallee's core feature set without multi-level menu traversal.

Within each destination, information is organized according to a consistent two-tier hierarchy: an upper interactive visualization layer (large-format charts, score gauges, and goal progress indicators) provides immediate trend comprehension, while a lower scrollable list layer provides access to granular transaction and allocation records. This architecture supports both the 'quick check' use case (a user who opens the app to verify they are on budget) and the 'deep dive' use case (a user who wants to audit specific transactions), without forcing either type of user to navigate through content irrelevant to their intent.

Figure 17: Two-Tier Information Hierarchy
An at-a-glance financial overview featuring live cashflow, balance trends, active subscriptions, and a recent transaction ledger.

Figure 18: Interactive Visualization Layer
Tracks planned versus actual income and expenses through interactive visualizations and fully customizable spending categories.

Figure 19: Goals Architecture
A dedicated interface for users to fund and visually track custom lifestyle aspirations and non-essential purchases.

Figure 20: Tools Ecosystem
The central hub for advanced utilities, housing the Wallee Score diagnostic, Wallo AI assistant, automated reports, and tax tracking.

Figure 21: Profile & Configurations
Manages user settings, personal data, and secure API connections for linked external financial accounts.

7. Security & Privacy

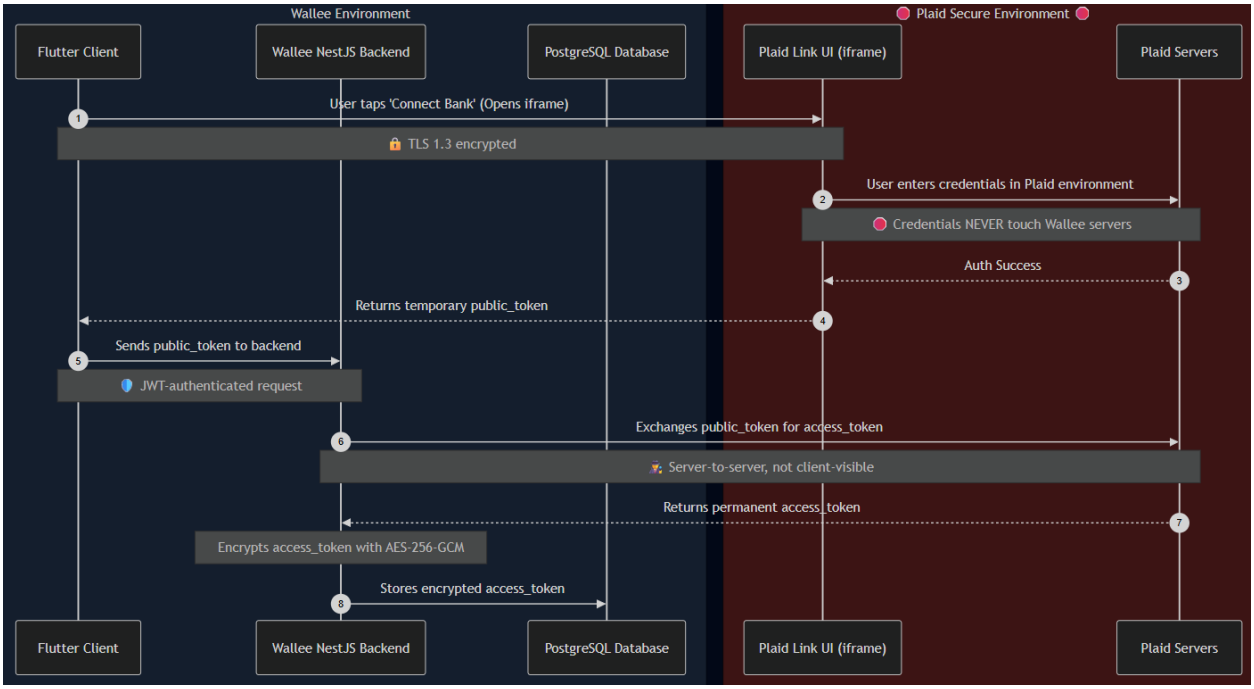
The security architecture of a personal finance application demands a standard of rigor commensurate with the sensitivity of the data being processed. A breach of Wallee's data store would expose not only personally identifiable information but also detailed records of users' income, spending behavior, and financial obligations — information that could be exploited for identity theft, social engineering, or financial fraud. Wallee's security design addresses this risk surface through a layered defense strategy that applies cryptographic, architectural, and policy controls at every tier of the system.

7.1 Zero-Knowledge Banking Protocol

Wallee's most fundamental security property is its Zero-Knowledge Banking Protocol: the guarantee that Wallee's infrastructure never possesses, processes, or stores a user's banking credentials (username, password, or one-time codes). This guarantee is achieved through the architectural decision to delegate all credential handling to Plaid. When a user connects a bank account, they authenticate exclusively within Plaid Link — a secure iframe-based UI element served from Plaid's domain — and Wallee's servers are never in the credential transmission path.

Upon successful authentication, Plaid issues a one-time public_token to the Flutter client, which immediately forwards it to the NestJS backend. The backend exchanges this public_token for a permanent access_token via a server-to-server API call to Plaid — a call that never passes through the mobile client and is therefore invulnerable to interception by mobile network eavesdropping. The access_token, which grants read-only access to the user's transaction data, is encrypted using AES-256-GCM before being stored in the PostgreSQL database, with the encryption key stored separately in an environment variable rather than in the database itself.

Figure 22:



7.2 Row-Level Security (RLS) via Supabase

All financial data in Wallee's PostgreSQL database is protected by Supabase's Row-Level Security (RLS) system — a PostgreSQL feature that attaches access control policies directly to database tables, evaluated by the PostgreSQL query planner before any row is returned to the application layer. RLS policies are defined as SQL expressions that compare the row's user_id foreign key against the JWT sub claim of the authenticated request, granting access only when they match.

The critical security property of RLS is that it is enforced at the database engine level, beneath the application layer. This means that even if a bug in the NestJS application logic inadvertently constructs a query without a WHERE user_id = ? clause, the RLS policy will still prevent the query from returning any rows belonging to other users. This defense-in-depth approach eliminates an entire class of application-layer security failures — horizontal privilege escalation through insecure direct object references — that represents one of the most common vulnerability patterns in financial applications.

7.3 Data Encryption

Wallee applies encryption controls at three levels. First, all data in transit between the Flutter client and the NestJS backend is protected by TLS 1.3, the current cryptographic standard for transport-layer security, enforced at the infrastructure level with HTTP Strict Transport Security (HSTS) headers preventing protocol downgrade attacks. Second, Plaid access tokens at rest are encrypted with AES-256-GCM as described in Section 7.1. Third, Supabase's underlying cloud infrastructure (AWS) applies transparent AES-256 encryption to all database volumes at rest, providing a storage-layer encryption baseline that protects against physical media compromise.

7.4 Authentication & Authorization

User authentication is managed through Supabase Auth, which implements the OAuth 2.0 authorization framework with JSON Web Tokens (JWTs) as access credentials. Each JWT is signed with a 256-bit HMAC-SHA256 key and carries a standard set of claims including the user's unique identifier (sub), token expiry time (exp), and issuer (iss). JWTs are issued with a 1-hour expiry and refreshed automatically by the Flutter client using Supabase's refresh token mechanism, limiting the window of vulnerability for a compromised token.

The NestJS backend validates JWT signatures on every authenticated request using the Supabase Admin SDK, verifying the token's cryptographic signature, expiry time, and issuer claims before processing any request. All API endpoints that access financial data require a valid JWT; the backend exposes no unauthenticated data endpoints. Role-based access control (RBAC) distinguishes between standard user roles and future administrative roles, with the access control logic implemented as NestJS Guards that evaluate role claims embedded in the JWT payload.

Security Control	Implementation Detail
Credential Isolation	Plaid OAuth — banking credentials never reach Wallee servers.
Transport Security	TLS 1.3 + HSTS on all client-server communication channels.
Token Encryption at Rest	AES-256-GCM with environment-variable key management.
Database Access Control	PostgreSQL Row-Level Security enforced at engine level.
Auth Token Management	Supabase Auth JWTs, 1-hour expiry, HMAC-SHA256 signed.
Volume Encryption	AWS AES-256 transparent disk encryption via Supabase infrastructure.
AI Data Minimization	Only aggregated FHP scores (no raw transactions) sent to Gemini API.

8. Evaluation Metrics

The systematic evaluation of Wallee's performance against defined success criteria is essential both for validating the correctness of the system's core computations and for assessing its effectiveness as a user-facing product. This section defines the evaluation framework across four dimensions: Accuracy, Reliability, Performance, and User Experience, specifying for each dimension the measurement methodology, success threshold, and the evaluation context (automated testing, load testing, or user survey).

8.1 Accuracy Metrics

Accuracy evaluation targets two distinct components of Wallee's computation pipeline: transaction categorization and budgeting arithmetic. Transaction categorization accuracy is measured by constructing a labeled test dataset of 500 transactions (drawn from anonymized real-world transaction histories and manually annotated with ground-truth categories), running the full categorization pipeline against this dataset, and computing precision, recall, and F1-score for each category. A system-level F1-score of 0.85 or above constitutes a passing evaluation; individual categories falling below an F1-score of 0.70 are flagged for prompt engineering refinement.

Budgeting arithmetic accuracy is evaluated through a suite of deterministic unit tests implemented using the pytest framework within the Python FastAPI service. Each test case defines an input state (a set of transactions, balances, and goal records) and an expected output state (the correct FHP), and asserts that Wallee Zero's computed output matches the expected output to within a $0.01 floating-point tolerance. A passing evaluation requires 100% of arithmetic unit tests to pass with zero tolerance violations, reflecting the zero-error requirement for financial calculations.

8.2 Reliability Metrics

Reliability is evaluated across the budget recalculation and AI advisory pipelines. For budget recalculation, 1,000 synthetic paycheck events are generated with randomized income amounts, transaction histories, and goal configurations, and the system is required to produce a correctly structured FHP for each event within a defined time window. A reliability score of 99.9% (one failure per 1,000 events) constitutes a passing threshold. Failures are categorized as either computational errors (incorrect output values) or system errors (service unavailability, unhandled exceptions).

For the AI advisory pipeline, reliability is evaluated through a set of adversarial prompt injections designed to test whether Wallo can be induced to perform its own arithmetic — a failure mode that would violate the Two-Layer architecture's mathematical integrity guarantee. A panel of 50 adversarial prompts is submitted through the standard Wallo interface, and each response is reviewed for evidence of arithmetic output. A passing evaluation requires zero instances of Wallo generating numerical outputs not present in the supplied FHP.

8.3 Performance Metrics

System performance is evaluated against three latency benchmarks that define the responsiveness thresholds for Wallee's primary user-facing operations. These benchmarks are measured under a simulated concurrent load of 50 simultaneous users using the Locust load testing framework.

Operation	P50 Target	P95 Target	P99 Target
Transaction Categorization	< 200 ms	< 500 ms	< 1,000 ms
Budget Recalculation (Full Cycle)	< 1,500 ms	< 3,000 ms	< 5,000 ms
Wallo AI Response (first token)	< 800 ms	< 2,000 ms	< 4,000 ms
Plaid Webhook Processing	< 500 ms	< 1,200 ms	< 2,500 ms

8.4 User Experience Evaluation

User experience evaluation is conducted through a structured beta testing protocol with a target cohort of 20 users drawn from the variable-income demographic (university students with part-time employment, freelancers, and gig workers). Each beta participant uses Wallee as their primary personal finance application for a two-week period spanning at least one full paycheck cycle, after which they complete a structured usability survey.

The survey instrument evaluates five dimensions on a 1–5 Likert scale: (1) Ease of Initial Setup (bank account connection and budget configuration), (2) Clarity of Financial Health Scores (do users understand what the scores mean and how to improve them?), (3) Perceived Usefulness of Wallo's Advice (did Wallo's guidance influence any financial decision during the two-week period?), (4) Visual Clarity of Reports and Visualizations (can users identify spending trends and over-budget categories without instruction?), and (5) Overall Satisfaction (would users recommend Wallee to a peer with similar financial circumstances?).

Passing thresholds are defined as a mean score of 4.0 or above on all five dimensions, with no individual participant score below 2.0 on any dimension. Qualitative feedback collected through open-ended survey questions will inform prioritized UX refinements for the production release following the Senior Design presentation.

Dataset 1: Feature Value Distribution

We asked our 52 testers to select the top two features that provided the most immediate value to their daily financial tracking.

Feature	% of Users Selecting in Top 2	Raw Votes (Out of 52)
Visual Cashflow (Sankey / Money Map)	79%	41
Wallee Discipline Score	65%	34
Wallo AI Assistant	39%	20
Automated PDF Reports	14%	7
Tax Hub / Resources	4%	2

Dataset 2: Context-Aware Glassmorphism Effectiveness

Wallee uses ambient color signaling (green/red translucency) rather than just text. We tested how quickly users could assess their financial state without reading the exact numbers.

Comprehension Level	% of Users	Raw Votes
Instantly clear at a glance	67.30%	35
Understood after a few interactions	25.00%	13
Relied mostly on the text/numbers	5.80%	3
Found the color shifts confusing	1.90%	1

Dataset 3: Navigation Architecture (The Bottom Pill)

We evaluated our flat, 5-destination navigation pill structure to see if users could find specific transactions or budget limits without getting lost in nested menus.

Usability Rating (1-5)	Meaning	% of Users
5-May	Highly Intuitive (Zero friction)	59.60%
5-Apr	Intuitive (Found it quickly)	28.80%
5-Mar	Neutral (Expected behavior)	7.70%
5-Feb	Slight Friction	3.90%
5-Jan	Confusing / Got Lost	0.00%

Appendix A: Technology Stack Summary

Component	Technology / Version & Justification
Mobile Client	Flutter 3.x (Dart) — Cross-platform, hardware-accelerated rendering, single codebase for iOS and Android.
Backend API	Node.js 20 LTS + NestJS 10 — TypeScript-first, modular DI architecture, proven scalability for API gateways.
Analytics Engine	Python 3.11 + FastAPI 0.110 — Native NumPy/Pandas support for numerical computation; async I/O via asyncio.
Database	PostgreSQL 15 via Supabase — ACID compliance, relational integrity, native RLS, real-time subscriptions.
Authentication	Supabase Auth (JWT/OAuth 2.0) — Native RLS integration, refresh token management, social login support.
Bank Connectivity	Plaid API (Transactions, Balance, Webhooks) — Industry-standard, 11,000+ institutions, read-only OAuth tokens.
Conversational AI	Google Gemini 1.5 Pro API — Superior context window (1M tokens), cost efficiency, multilingual support.
State Management	Flutter Riverpod 2.x — Compile-time safe, testable, reactive data flow for complex financial state.
Testing (Backend)	Jest (NestJS unit/integration) + pytest (FastAPI unit) + Locust (load testing).
CI/CD	GitHub Actions — Automated test, build, and deployment pipelines on pull request and main branch merges.
Hosting	Supabase (DB + Auth), Railway or Fly.io (NestJS + FastAPI containers), Cloudflare (CDN + DDoS protection).

Document Information

Field	Value
Document Title	Technical Design Document: Wallee – Personal Finance Application
Version	1.0
Project Type	University Senior Design Capstone
Prepared By	Emma Bahr, Kyle Gibson, Joshua Cajuste, Matteo Caruso
Intended Audience	University Faculty Panel, Client Stakeholders, Technical Reviewers
Classification	Academic Project Documentation — Public Presentation