SMARTSPEND
THE EMOTION-AWARE CASH-FLOW COACH

Course Name: Capstone 2025
Team Name: Foxtrot
Date of Submission: December 12, 2025

CAPSTONE ARCHITECTURE DOCUMENT

Team Members:

Sarvesh Patil
811348253
spatil8@kent.edu
Master's in Computer Science

Sandeep Enamandala
811350293
senamand@kent.edu
Master's in Computer Science

Prajwal Devaraj
811351381
pdevaraj@kent.edu
Master's in Computer Science

Akshara Reddy Nookala
811330459
anookala@kent.edu
Master's in Computer Science

Meghamala Samala
811362078
msamala1@kent.edu
Master's in Artificial Intelligence

Abhijeet Yalamanchili
811354816
ayalaman@kent.edu
Master's in Computer Science

Prasadam Manoj Kumar
811374110
mprasada@kent.edu
Master's in Data Science

1. System Overview

SmartSpend is an ML-driven financial wellness and behavior-aware spending platform designed to help users understand how long their money will last, why they overspend, and how to make better daily financial decisions. The system predicts burn rate, runway (days left), and behavioral risks using a multi-tier machine learning pipeline, while providing users with a clean, intuitive dashboard that simplifies complex budgeting into actionable insights.

The application serves students, young professionals, and individuals with unpredictable income, offering clarity and emotional awareness rather than traditional, overwhelming budgeting tools. By combining financial data with behavioral patterns such as mood, late-night spending, and Needs/Wants/Guilt (NWG) classification, SmartSpend transforms raw transactions into personalized guidance.

1.1 Goals of the System

- Provide real-time financial stability indicators (Burn Rate, Days Left).
- Detect and highlight emotional or impulsive spending behaviors.
- Offer simplified visualizations such as NWG breakdowns and daily burn charts.
- Generate insights, warnings, and nudges to help users stay mindful about spending.
- Integrate a complete workflow across React frontend, Flask backend, MySQL database, and ML predictive models.
- Support recurring bills, scheduled expenses, runway goals, and notification alerts.

1.2 Target Users

SmartSpend is designed for:

- University students (18-28)
- Young professionals (22-35)
- Budgeting beginners
- Users prone to emotional or late-night spending

These groups commonly face rapid salary depletion, unplanned expenses, and lack of awareness about financial habits. SmartSpend addresses these gaps through real-time predictions and behavior-based insights.

1.3 High-Level System Context

At a high level, SmartSpend is a client-server architecture:

- The React frontend (hosted on GitHub Pages) provides dashboards, forms, graphs, and visual insights.
- The Flask backend handles REST API logic, authentication, and ML model execution.
- The MySQL database stores all persistent data including users, wallets, transactions, bills, goals, insights, and ML snapshots.
- The Machine Learning Engine processes historical data to predict burn rate, behavior flags, runway estimates, and emotional risk alerts.

1.4 Context Diagram

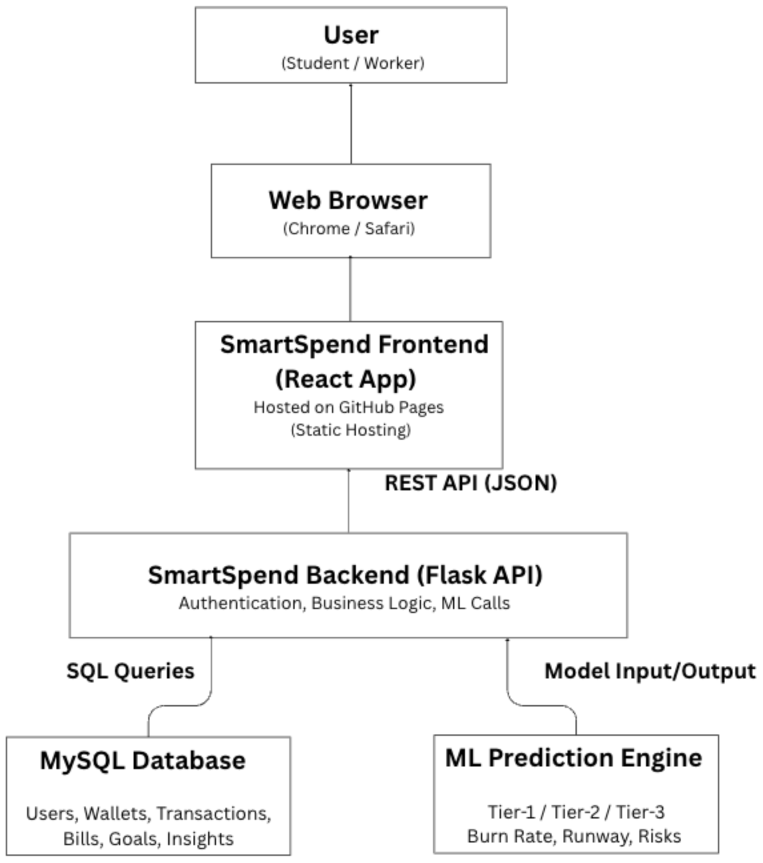

User
(Student / Worker)
Web Browser
(Chrome / Safari)
SmartSpend Frontend (React App)
Hosted on GitHub Pages (Static Hosting)
REST API (JSON)
SmartSpend Backend (Flask API)
Authentication, Business Logic, ML Calls
SQL Queries
Model Input/Output
MySQL Database
Users, Wallets, Transactions, Bills, Goals, Insights
ML Prediction Engine
Tier-1 / Tier-2 / Tier-3
Burn Rate, Runway, Risks

Fig 1.1. Context Diagram

1.5 Technology Stack

Layer	Technologies
Frontend	React (Vite), JavaScript, HTML/CSS
Backend	Flask (Python), SQL Alchemy, JWT
Database	MySQL
Machine Learning	MA7, Linear Trend, ARIMA/SARIMA, LSTM

Deployment	GitHub Pages + GitHub Repo
Version Control	GitHub
Security	JWT authentication, hashed passwords, role-based access

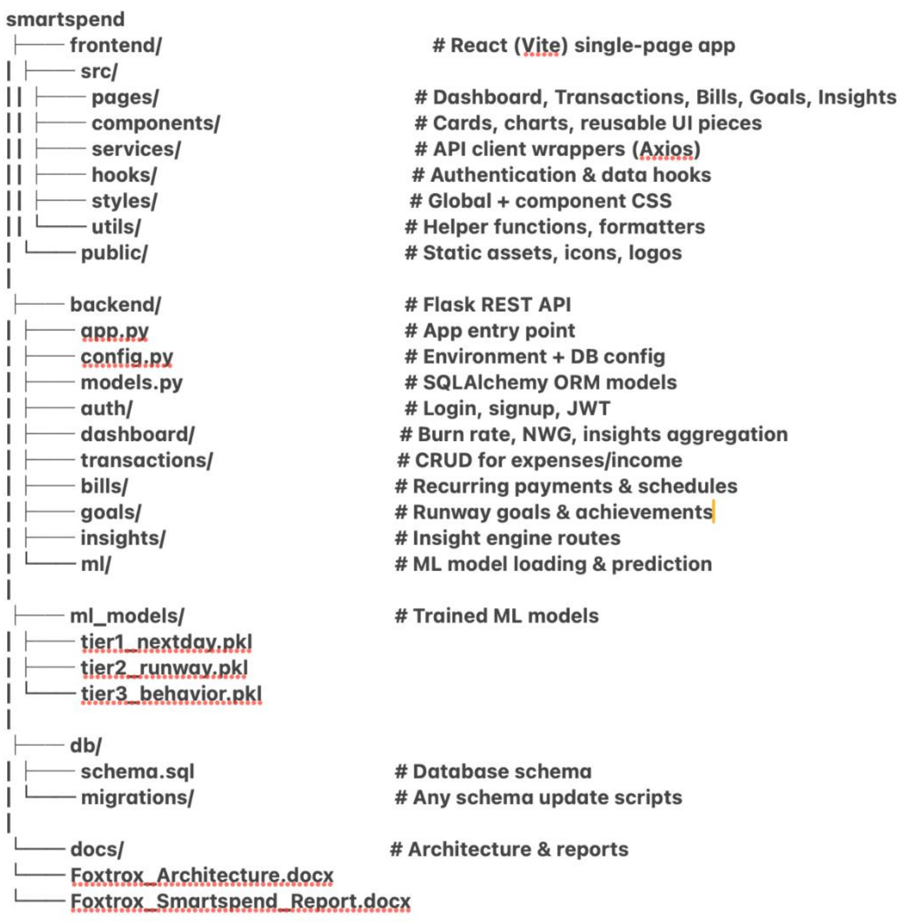
Table 1.1 Technology

2. Architecture Overview

SmartSpend follows a client-server architecture with a React frontend, a Flask REST API backend, a MySQL database layer, and a separate Machine Learning (ML) engine. This section describes how the codebase is organized, how major modules interact, and how assets and data are structured across the system.

2.1 Application Folder and File Layout

The SmartSpend project is organized as a full-stack web application with a React frontend, Flask backend, MySQL database schema, and separate ML artifacts.



```
smartspend
 ├── frontend/                    # React (Vite) single-page app
 │ ├── src/
 │ │ ├── pages/                   # Dashboard, Transactions, Bills, Goals, Insights
 │ │ ├── components/              # Cards, charts, reusable UI pieces
 │ │ ├── services/                # API client wrappers (Axios)
 │ │ ├── hooks/                   # Authentication & data hooks
 │ │ ├── styles/                  # Global + component CSS
 │ │ └── utils/                   # Helper functions, formatters
 │ └── public/                    # Static assets, icons, logos
 │
 ├── backend/                     # Flask REST API
 │ ├── app.py                     # App entry point
 │ ├── config.py                  # Environment + DB config
 │ ├── models.py                  # SQLAlchemy ORM models
 │ ├── auth/                      # Login, signup, JWT
 │ ├── dashboard/                 # Burn rate, NWG, insights aggregation
 │ ├── transactions/              # CRUD for expenses/income
 │ ├── bills/                     # Recurring payments & schedules
 │ ├── goals/                     # Runway goals & achievements
 │ ├── insights/                  # Insight engine routes
 │ └── ml/                        # ML model loading & prediction
 │
 ├── ml_models/                   # Trained ML models
 │ ├── tier1_nextday.pkl
 │ ├── tier2_runway.pkl
 │ └── tier3_behavior.pkl
 │
 ├── db/
 │ ├── schema.sql                 # Database schema
 │ └── migrations/                # Any schema update scripts
 │
 └── docs/                        # Architecture & reports
 └── Foxtrox_Architecture.docx
 └── Foxtrox_Smartspend_Report.docx
```

2.2 Major Scripts or Modules and Their Functions

Frontend (React) modules

- App.tsx/main.tsx: Mounts the React app, sets up routing, and global providers (e.g., auth context).
- src/pages/DashboardPage.tsx: Displays current balance, burn rate, days left, NWG charts, upcoming bills, and key insights.
- src/pages/TransactionsPage.tsx: Lists transactions, supports filters, and opens dialogs to add / edit / delete income and expenses with NWG + mood tagging.
- src/pages/BillsPage.tsx: Manages recurring bills (rent, subscriptions, etc.) with status and next-due dates.
- src/pages/GoalsPage.tsx: Shows the Runway Goal Slider, progress toward target days, and achievements.
- src/pages/InsightsPage.tsx: Renders ML-driven insights: late-night overspend, guilt spikes, bill risk, and behavior trends.
- src/components/Charts/*: Reusable chart components (line chart for daily burn, donut for NWG, bar chart for mood vs spend).
- src/services/apiClient.tsx: Centralized HTTP client; wraps Axios/fetch and attaches JWT for calls to /auth, /dashboard, /transactions, /bills, /goals, /insights, /ml.

Backend (Flask) modules

- app: Creates the Flask app, registers blueprints, configures CORS, and loads ML models at startup.
- Models: SQLAlchemy models for entities such as Users, Wallets, Categories, Transactions, Recurrence_Rules, Goals, Runway_Snapshots, Mood_Events, Pattern_Flags, Insights, Notifications, Achievements, burn_Models.

Entity-Relationship (ER) Table

- auth/routes.py: Handles signup, login, password reset, and issues JWTs.
- dashboard/routes.py: Aggregates burn rate, days left, NWG percentages, upcoming bills, and quick stats for the main dashboard.
- transactions/routes.py: Validates and stores transactions, computes running balances, and triggers ML updates where needed.
- bills/routes.py: Manages recurring bills: create, update, pause, mark as paid, and computes the next_due_date.
- goals/routes.py: Stores runway target days, retrieves history, and calculates progress percentages.

- insights/routes.py: Fetches and persists insights and notifications generated by the insight engine.
- ml/routes.py: Wraps the ML pipeline: extracts features from DB, calls Tier-1/2/3 models, and returns predictions to the frontend.

2.3 Asset Organization (Images, Data, Models)

- UI Assets: Logos, icons, and static images live in frontend/public/ or frontend/src/assets/. They are referenced by React components and bundled by Vite.
- Styling Assets: Global theme, responsive layouts, and component-level styles are in frontend/src/styles/ (CSS / SCSS modules).
- ML & Data Assets
    - Training data and engineered daily summaries (e.g., Daily_Summary_USD.csv) are stored under ml_models/data/.
    - Trained model artifacts (tier1_nextday.pkl, tier2_runway.pkl, tier3_behavior.pkl) live in ml_models/.
    - Any configuration (feature lists, scaler parameters) is saved as JSON or pickled objects alongside the models.
- Database Assets: Schema definitions and sample seed data are stored in db/schema.sql and db/seed.sql and reflected by SQLAlchemy models.

2.4 Module Diagram

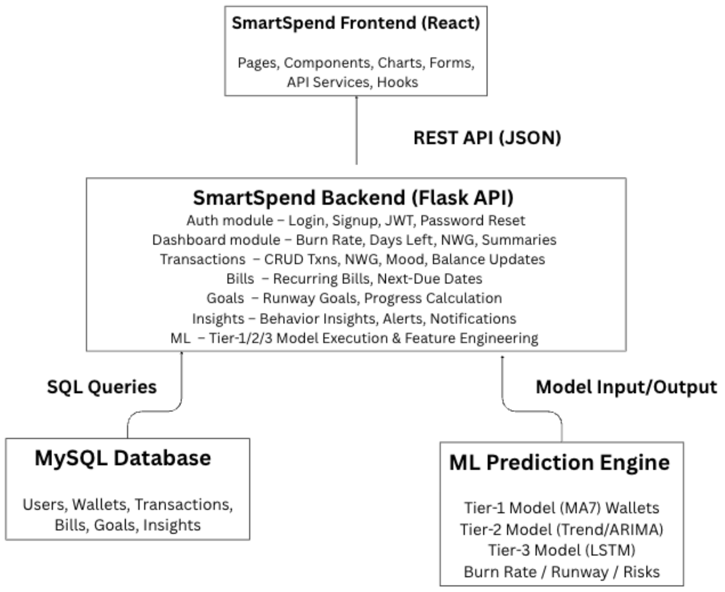

SmartSpend Frontend (React)
Pages, Components, Charts, Forms, API Services, Hooks
REST API (JSON)
SmartSpend Backend (Flask API)
Auth module – Login, Signup, JWT, Password Reset
Dashboard module – Burn Rate, Days Left, NWG, Summaries
Transactions – CRUD Txns, NWG, Mood, Balance Updates
Bills – Recurring Bills, Next-Due Dates
Goals – Runway Goals, Progress Calculation
Insights – Behavior Insights, Alerts, Notifications
ML – Tier-1/2/3 Model Execution & Feature Engineering
SQL Queries
Model Input/Output
MySQL Database
Users, Wallets, Transactions, Bills, Goals, Insights
ML Prediction Engine
Tier-1 Model (MA7) Wallets
Tier-2 Model (Trend/ARIMA)
Tier-3 Model (LSTM)
Burn Rate / Runway / Risks

Fig. 2.1 Module Diagram

2.5 Libraries and Frameworks Included

- Frontend
  - React - core UI library for building SPA components.
  - Vite - build tool and dev server for fast bundling.
  - React Router - client-side routing between pages.
  - Axios or fetch - HTTP client for REST API calls.
  - Charting library (e.g., Chart.js / Recharts / similar) - visualizes burn rate, NWG distribution, trends, and mood vs spend.
  - CSS/SCSS Modules - layout, theming, responsive design.
- Backend
  - Flask - lightweight Python web framework for REST APIs.
  - Flask-JWT / PyJWT - JWT-based authentication.
  - SQLAlchemy - ORM for interacting with MySQL.
  - Marshmallow / pydantic (if used) - request/response validation.
- Machine Learning
  - pandas, NumPy - data manipulation and feature engineering.
  - scikit-learn / statsmodels - Tier-1 and Tier-2 models (moving average, linear trend).
  - PyTorch - Tier-3 LSTM models for deep time-series forecasting.

These libraries collectively support an architecture that is modular, testable, and extensible while meeting the project's goals of prediction, behavioral analytics, and financial insights.

3. Data and State Management

SmartSpend uses a relational MySQL database for persistent storage and a React state layer for client-side UI state. All critical financial and behavioral data is stored server-side, while the browser keeps only temporary UI preferences and JWT tokens for the current session. The design closely follows the ER tables and schema defined for entities such as Users, Wallets, Transactions, Bills, Goals, Runway_Snapshots, Mood_Events, Pattern_Flags, Insights, Notifications, Achievements, and burn_Models.

3.1 Persistent Data Model (Database Side)

All long-term state is stored in MySQL and accessed through SQLAlchemy models in the Flask backend.

Key entities:

- Users - login identity and profile; one user can own multiple wallets.

- Wallets - logical account for a user; aggregates transactions, goals, snapshots, insights.
- Categories - Need/Want/Guilt mapping and custom labels per user.
- Transactions - income, expenses, and balance adjustments; link to wallet and category; include amount, NWG cache, merchant, time of day, mood source, and note.
- Recurrence_Rules - templates for recurring bills or incomes (cadence, amount, next occurrence).
- Goals - runway targets (target_days, set_at) to track how long users want their money to last.
- Runway_Snapshots - computed ML outputs (balance, days_left_current, days_left_power_save, quantile bands, model version).
- Mood_Events - explicit or inferred mood labels associated with users and optionally with specific transactions.
- Pattern_Flags - off-pattern event labels such as late-night spends, high-spend spikes, vendor streaks, or guilt spikes.
- Insights - coaching messages (goal_gap, bill_risk, food_overshare, late_night_streak, high_spend) and their metadata.
- Notifications - delivery channel records for urgent insights (in-app, push, email).
- Achievements - badges like RUNWAY_GOAL_7D or NO_LATE_NIGHT_5D.
- burn_Models - metadata about trained burn-rate models (algorithm, features, coefficients, training_days, model_version).

This schema ensures that financial numbers, behavior signals, and ML outputs are all traceable over time and reproducible for auditing or model retraining.

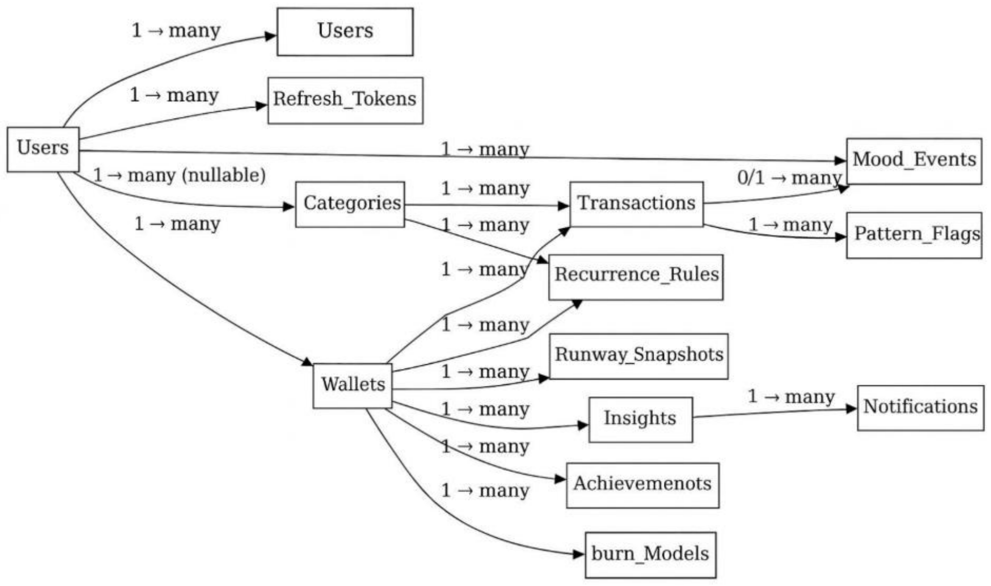


Fig. 3.1. ER Diagram

3.2 Frontend State Management

On the client side, React manages view state and short-lived interaction data:

- Authentication State
    - JWT token + user id stored in localStorage/sessionStorage.
    - In-memory React context tracks whether the user is logged in and exposes the current user to all pages.
- Dashboard State
    - Currently selected date range (e.g., last 7 / 30 days).
    - Active filters for category, mood, and NWG.
    - Cached API responses for KPIs (burn rate, days left, balance, charts).
- Page-specific State
    - Transactions page: form inputs, modal open/close, selected transaction for edit.
    - Bills page: list of recurring bills, editing flags, current month view.
    - Goals page: slider position for target_days, previewed vs saved goal.
    - Insights page: which insight is expanded, dismissed items in the current session.

Any authoritative data (balances, days left, ML outputs) always comes from backend APIs. Frontend state is considered Derived or Temporary and is refreshed on important actions (login, add transaction, change month, etc.).

3.3 Data Loading and Update Patterns

Dashboard Load

1. User opens Dashboard.
2. Frontend sends GET /dashboard with JWT.
3. Backend aggregates from:
    - Transactions (recent spends, NWG ratios)
    - Runway_Snapshots (latest days_left_current / power_save)
    - Bills / Recurrence_Rules (upcoming bills)
    - Insights (top 3 active insights)
4. Response includes KPIs + chart data; React stores this in local state and renders cards and graphs.

Add / Edit Transaction

1. User submits an expense/income form on Transactions page.
2. Frontend sends POST /transactions/add with amount, type, category, mood, time, and note.
3. Backend:
    - Inserts a row into Transactions for that wallet.

- Updates running balance and any relevant summaries.
- Optionally logs Mood_Events and Pattern_Flags (late night, high spend, vendor streak).
- Triggers the ML pipeline or schedules it to update Runway_Snapshots and Insights.

4. Updated KPIs and insights are returned or fetched on next dashboard refresh.

Bills and Recurring Expenses

- Bill definitions are stored in Recurrence_Rules or a Bills table with cadence, amount, and next_due_date.
- A daily job or on-login check expands due rules into upcoming payments and raises bill_risk insights if runway is threatened.

ML Data Pipeline

For model training and advanced analytics, transaction-level data is summarized into daily and weekly features:

1. Raw Transactions, Runway_Snapshots, and behavioral tags are aggregated into a daily summary file/table (e.g., Daily_Summary_USD), including features like need/want/guilt percentages, burn_rate, weekend, late-night flag, and mood indicators.
2. Tier-1, Tier-2, and Tier-3 models are trained and saved to ml_models/ as versioned .pkl artifacts.
3. At prediction time, the backend loads the latest model, pulls recent history from the DB, builds feature vectors, and writes results back into Runway_Snapshots, Pattern_Flags, and Insights.

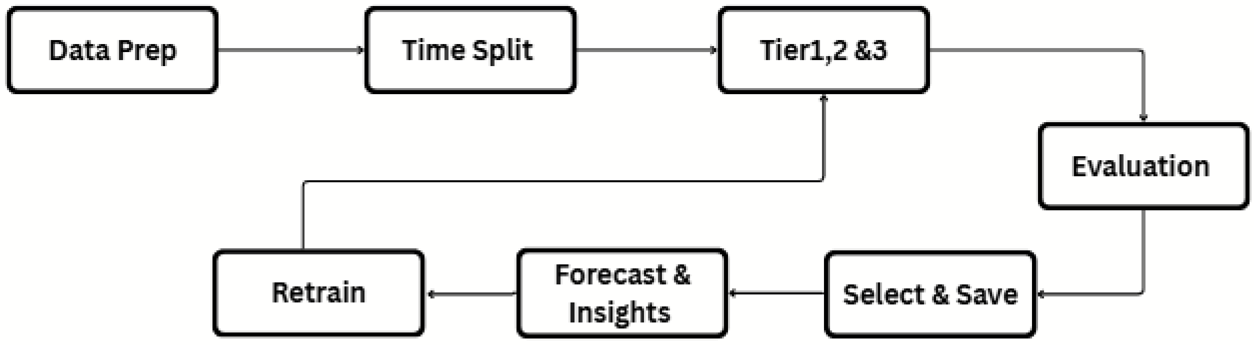


Fig. 3.2 ML Pipeline

3.4 Data Flow Diagram - "Add Expense and Update Dashboard"

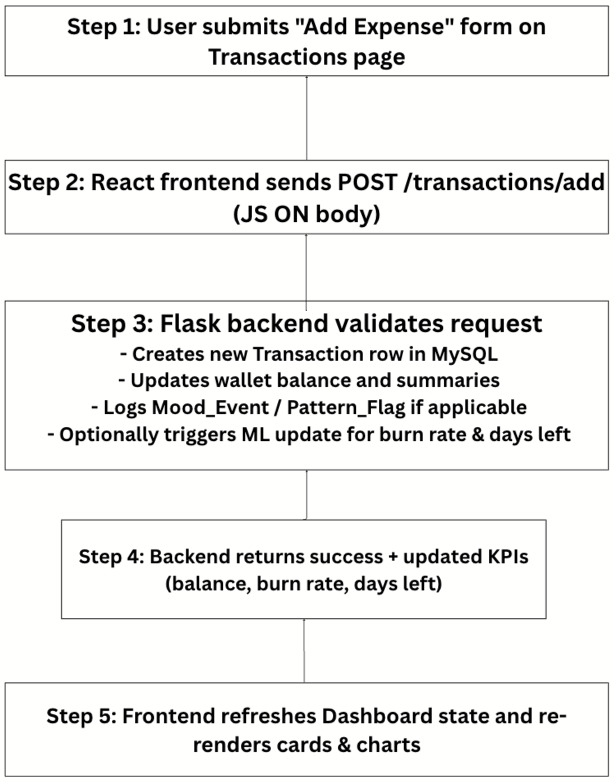

Step 1: User submits "Add Expense" form on Transactions page
Step 2: React frontend sends POST /transactions/add (JS ON body)
Step 3: Flask backend validates request
- Creates new Transaction row in MySQL
- Updates wallet balance and summaries
- Logs Mood_Event / Pattern_Flag if applicable
- Optionally triggers ML update for burn rate & days left
Step 4: Backend returns success + updated KPIs (balance, burn rate, days left)
Step 5: Frontend refreshes Dashboard state and re-renders cards & charts

Fig. 3.3 Data Flow Diagram

4. Deployment Pipeline (CI/CD)

SmartSpend uses a lightweight deployment process centered around GitHub, GitHub Pages, and a manually deployed Flask backend. The system does not rely on Cassini; instead, the frontend is fully hosted on GitHub Pages, while the backend and ML engine are deployed on a dedicated runtime environment (local server, VM, or cloud instance).

The deployment workflow ensures that updates to the UI, API, or ML models can be rolled out in a controlled manner with versioning, predictable builds, and rollback mechanisms.

4.1 Development Workflow

The development lifecycle follows standard Git-based iteration:

1. Feature Branch Creation: Each developer clones the repository and creates a named feature branch (e.g., feature/transactions-api, ui/dashboard-cards).

2. Local Development
    - Frontend runs locally using npm run dev.
    - Backend runs locally using flask run.
    - MySQL runs via local DB instance or Docker.
    - ML models are loaded from ml_models/ during local development.
3. Testing Before Commit - Developers manually test:
    - Key API endpoints (Postman / Thunder Client)
    - UI flows (transactions, bills, dashboard refresh)
    - ML prediction API results
    - Database migrations (if any)
4. Pull Request (PR): Changes are pushed to GitHub and reviewed by another team member before merging into main.

4.2 Frontend Deployment (GitHub Pages)

The frontend is deployed automatically through GitHub Pages.

Deployment Steps

1. Developer merges changes to main.
2. GitHub Actions (or a Pages build) runs:
    - Installs dependencies
    - Builds React app with npm run build
    - Publishes static files to the gh-pages branch (or Pages output directory).

Advantages

- Zero-cost hosting
- Instant cache-based updates
- Simple rollbacks (reset gh-pages branch)

Rollback Procedure

If a deployment breaks:

1. Revert last commit on main.
2. Re-run GitHub Pages build.
3. Deployment instantly rolls back to a stable build.

4.3 Backend Deployment (Flask API)

The Flask backend is deployed using a manual deployment workflow.

Deployment Steps

1. Pull the latest backend code from GitHub:

    git pull origin main

2. Install required Python packages:

    pip install -r requirements.txt

3. Apply schema updates to MySQL (if needed).
4. Start Flask using a production WSGI server (e.g., gunicorn) or a managed runtime.

Environment Configuration

Backend reads its environment variables from a .env file:

- DB_HOST, DB_USER, DB_PASS, DB_NAME
- JWT_SECRET
- ML_MODEL_PATHS

Rollback Procedure

If the backend fails:

- Checkout previous tag or commit:

    git checkout v1.0.3

- Restart server
- System reverts to prior API behavior and prior ML model versions

4.4 ML Pipeline Deployment

The ML engine relies on pre-trained models stored in:

```
ml_models/
  ├── tier1_nextday.pkl
  ├── tier2_runway.pkl
  └── tier3_behavior.pkl
```

Updating ML Models

1. Retrain model locally or in notebook.
2. Export .pkl file into the ml_models directory.
3. Update version metadata in the database's burn_Models table.
4. Restart backend so the new model loads at boot.

ML Rollback

- Restore previous model files from version control
- Update model version table
- Restart backend

This allows safe experimentation without breaking predictions.

4.5 CI/CD Diagram

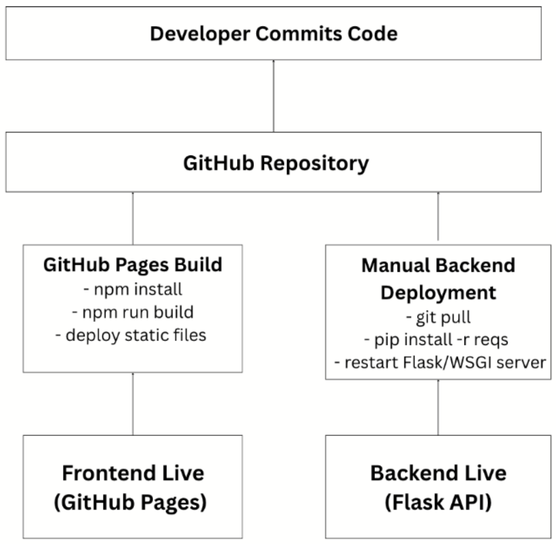


Fig. 4.1. CI/CD Diagram

5. Security & Configuration

SmartSpend manages personal financial records and behavioral insights, so the system applies multiple security layers across authentication, API access, database operations, and ML model handling. These measures ensure the confidentiality, integrity, and safe processing of user data.

5.1 Authentication & User Identity

- SmartSpend uses JWT (JSON Web Tokens) for secure, stateless login.
- Passwords are hashed using industry-standard algorithms before storage.
- Tokens include:
    - user_id
    - expiration timestamp
    - server-issued signature (prevents tampering)
- All protected endpoints (dashboard, transactions, bills, goals, insights) require a valid JWT in the request header:
- Authorization: Bearer <token>
- Expired or invalid tokens block access and return 401 Unauthorized.

5.2 API Security

- Communication between frontend and backend uses HTTPS to prevent interception.
- CORS policies restrict API usage to the SmartSpend frontend domain.
- Every incoming request undergoes input validation:
    - numeric limits (amount, balance)
    - valid date/time formats
    - valid NWG category and mood labels
- Error messages are generic to avoid exposing internal logic.
- SQLAlchemy ORM ensures automatic parameterization, preventing SQL injection.

5.3 Database Security

- MySQL is only accessible through the backend; users cannot query it directly.
- Backend uses a restricted database user account with minimal permissions.
- Schema constraints, foreign keys, and cascades maintain integrity across:
    - Users
    - Wallets
    - Transactions
    - Bills / Recurrence Rules
    - Goals
    - Insights
    - ML Snapshots
- Sensitive data (JWT secrets, DB passwords) is never stored in source code.

5.4 ML Model Security

- ML model files (tier1_nextday.pkl, tier2_runway.pkl, tier3_behavior.pkl) are stored on the server and loaded only by the backend.
- No API gives direct access to models or raw training data.

- All ML predictions use sanitized feature vectors to prevent malformed or harmful inputs.
- Model versions are tracked so predictions can be reproduced or rolled back safely.

5.5 Configuration Management

- SmartSpend uses a centralized .env file for:
    - DB credentials
    - JWT secret key
    - ML model paths
    - API origin settings
- The .env file is ignored by Git and never uploaded to GitHub.
- Development and production have separate configurations to prevent accidental exposure.

5.6 Client-Side (Frontend) Security

- JWT is stored only in localStorage or sessionStorage; no financial data is persisted locally.
- The React app sanitizes dynamic content to prevent injection attacks.
- API client automatically attaches JWT and redirects users to login when unauthorized.

SmartSpend's layered security model ensures:

- strong identity management (JWT)
- safe API access
- isolated database access
- protected machine learning artifacts
- environment-based configuration
- secure client-side behavior

This allows the system to operate safely while handling sensitive financial and behavioral data.

6. Testing & Quality Assurance

To ensure SmartSpend performs reliably, produces accurate predictions, and maintains data integrity, several layers of testing are applied across the frontend, backend, database, and ML components. The goal is to verify functionality, prevent regressions, and maintain system stability throughout development and deployment.

6.1 Frontend Testing

Manual UI Testing

The React interface is tested manually during development to verify:

- Page navigation and routing (Dashboard, Transactions, Bills, Goals, Insights)
- Form validation for adding/editing transactions and bills
- Chart rendering (burn rate, NWG %, mood vs spend, runway goals)
- Edge cases like empty wallets, zero transactions, or overdue bills
- Visual responsiveness across laptop and mobile sizes

State & Error Handling Tests

- Authentication guard redirects unauthenticated users to login
- Error messages appear for invalid or incomplete inputs
- Dashboard updates immediately after adding a transaction

6.2 Backend Testing

API Endpoint Testing

Tools such as Postman / Thunder Client are used to test:

- /auth/login, /auth/signup
- /dashboard aggregation results
- /transactions/add, /transactions/update, /transactions/delete
- /bills/* for recurring bill logic
- /goals/* for runway target updates
- /insights/* for behavior-based insights

Each endpoint is tested for:

- Valid inputs
- Invalid inputs (negative amounts, invalid dates)
- Missing or incorrect JWT tokens
- SQL validation via ORM

Error Handling

- Requests with invalid data return 400 Bad Request
- Unauthorized requests return 401
- Internal failures return safe error messages without exposing system details

6.3 Database Quality Assurance

Schema Validation

- Foreign keys ensure every Transaction belongs to a valid Wallet
- Cascading behavior prevents orphan entries
- ENUMs/NWG/Mood fields validate behavioral attributes

Data Integrity Tests

- Adding a transaction correctly updates wallet balance
- Modifying a bill updates the next_due_date correctly
- Goal progress matches runway calculations

6.4 ML Model Testing

Prediction Accuracy Checks

Before deploying updated ML models:

- Tier-1 (MA7) predictions are compared with recent spend trends
- Tier-2 (Trend/ARIMA) predictions are checked for stable long-term runway estimation
- Tier-3 (LSTM) behavior detection is checked for unusual spikes, late-night pattern accuracy

Input Validation

- Feature vectors are tested with real and edge case data (e.g., zero spend days)
- Models reject malformed inputs gracefully

Version Control

- Every ML update is tagged with model_version
- Previous models can be rolled back if anomalies are detected

6.5 Integration Testing

Integration testing ensures that components work together smoothly:

- Adding a transaction updates:
    - Wallet balance
    - Dashboard burn rate
    - Days-left prediction
    - NWG chart percentage
- Updating a recurring bill reflects in upcoming bill schedules
- Setting a runway goal updates progress bars and achievements

These scenarios verify end-to-end workflow correctness across frontend, backend, database, and ML.

6.6 User Acceptance Testing (UAT)

Volunteers and teammates tested the system for:

- Ease of navigation
- Clarity of dashboard KPIs
- Accuracy of insights and warnings
- Realistic runway predictions based on spending habits

Feedback was used to refine UI flows, improve insight wording, and adjust prediction smoothing.

6.7 Quality Assurance Summary

SmartSpend's testing strategy ensures:

- The UI renders correctly and handles errors gracefully
- API endpoints behave predictably across all scenarios
- Data remains consistent and relationally sound
- ML predictions are stable, realistic, and reproducible
- Key user workflows perform reliably end-to-end

This layered approach provides confidence that the platform is functional, secure, and ready for real-world use.

7. Performance Considerations

SmartSpend is designed to load quickly, respond fast to user actions, and keep ML predictions efficient. Performance is maintained across the frontend, backend, database, and ML layers.

7.1 Frontend Performance

- React renders only the components that change, improving dashboard responsiveness.
- Backend sends pre-aggregated values (burn rate, days left, NWG %), reducing browser computation.
- Vite build + GitHub Pages provide fast static delivery and browser caching.
- Pagination and filtered views prevent loading large datasets at once.

7.2 Backend Performance

- Lightweight Flask endpoints send compact JSON responses.
- Frequently used values (burn rate, days left) are stored in Runway_Snapshots to avoid recalculating every time.
- Insights are generated only when transactions change.
- Running behind a WSGI server allows handling multiple requests smoothly.

7.3 Database Performance

- Indexes on user_id, wallet_id, timestamp, and category speed up queries.
- Queries use JOINs efficiently thanks to a normalized schema.
- Dashboard requests use summary data, not full transaction history.

7.4 ML Performance

- Tier-1 and Tier-2 models are extremely fast; Tier-3 LSTM is loaded once at startup.
- Feature engineering uses daily summaries to reduce computation time.
- Predictions can be cached and updated only when new spending occurs.

7.5 Scalability

- GitHub Pages scales naturally for frontend hosting.
- Backend is stateless and can be horizontally scaled if needed.
- MySQL can scale vertically for higher workloads.

8. Operational Concerns & Monitoring

SmartSpend is designed to run reliably once deployed, with simple operational practices that ensure stability, quick recovery from issues, and predictable updates. This section outlines how the system is monitored and what operational steps are required to keep it running smoothly.

8.1 Reliability & Uptime

- The frontend is hosted on GitHub Pages, which provides highly reliable static hosting with automatic CDN distribution.
- The backend runs on a controlled environment (local server/VM/cloud instance) and must be restarted after updates or ML model changes.
- The application is stateless, allowing future scaling without major redesign.

8.2 Error Logging & Monitoring

- The backend logs:
  - API errors
  - Token/authentication failures
  - Database connection issues
  - ML model loading or prediction errors
- Logs help identify:
  - Incorrect inputs from the frontend
  - Server misconfiguration
  - Failing queries or missing schema elements

Monitoring can be done manually via logs or connected to simple tools like PM2 or a Linux system journal.

8.3 Backup & Data Recovery

- MySQL database snapshots should be backed up periodically (daily or weekly).
- Backups include:
  - Users
  - Wallet data
  - Transactions
  - Bills
  - Goals
  - Insights and ML snapshots
- Recovery involves restoring the SQL dump and restarting the backend.

8.4 Deployment Updates

When deploying new features:

1. Pull the latest code from GitHub.
2. Install new dependencies if required.
3. Apply schema updates if the database changed.
4. Restart the backend server so new changes and ML files load correctly.

For frontend updates, GitHub Pages automatically redeploys after each push.

8.5 Monitoring ML Behavior

- Monitor predictions for unusual spikes in burn rate or runway.
- If a new model version behaves unexpectedly, revert to the previous .pkl file.
- Check Runway_Snapshots table to ensure predictions are updating normally.

8.6 Incident Handling

Typical issues and responses:

- Backend not responding: Restart the API server and check environment variables.
- Frontend outdated: Trigger a new GitHub Pages deployment.
- Model predictions failing: Check model load paths and restore previous version.
- Database errors: Validate schema, fix corrupt rows, or restore from backup.

9. Architectural Decisions (ADR Summary)

This section highlights the key technical decisions made during the development of SmartSpend and the reasons behind them.

9.1 React + Flask Architecture

Decision: Use React for the frontend and Flask for the backend.

Reason: Clear separation of UI and API logic, fast development, easy debugging, and strong support for REST-based apps.

9.2 Frontend Hosted on GitHub Pages

Decision: Deploy the UI as static files on GitHub Pages.

Reason: Zero cost, automatic scaling, fast loading through CDN, simple deployment.

9.3 MySQL for Persistent Storage

Decision: Use MySQL with SQLAlchemy.

Reason: Handles structured financial data well, supports indexing, foreign keys, and efficient queries.

9.4 JWT Authentication

Decision: Implement stateless JWT-based login.

Reason: Works well with single-page apps, easy to protect routes, secure token-based identity.

9.5 Tiered ML Model Design

Decision: Use Tier-1 (MA7), Tier-2 (Trend/ARIMA), and Tier-3 (LSTM).

Reason: Combines fast predictions with deeper behavioral analysis, flexible fallback options.

9.6 Precomputed Snapshots

Decision: Store burn rate and days-left calculations in Runway_Snapshots.

Reason: Improves dashboard performance and avoids repeated computation.

9.7 Modular Backend Structure

Decision: Split backend into blueprints (auth, transactions, bills, goals, insights, ML).

Reason: Keeps the code organized and easier to maintain.

9.8 Environment-Based Configuration

Decision: Use .env for DB credentials, JWT secrets, and model paths.

Reason: Protects sensitive data, supports separate dev/production setups.

9.9 Manual Backend Deployment

Decision: Deploy backend manually rather than using Cassini or full CI/CD.

Reason: Simpler for the project's scope, easier to control model loading and updates.

10. Risks and Future Work

SmartSpend is functional and stable, but like any growing system, it carries certain risks and opportunities for expansion.

10.1 Key Risks

1. Model Accuracy Variability

ML predictions may fluctuate with limited historical data or sudden spending changes.

Risk: Incorrect runway estimates or behavior alerts.

2. Manual Backend Deployment

Updates depend on manual steps.

Risk: Human error or inconsistent environments.

3. Increasing Data Volume

As transactions grow, queries may slow down without additional optimization.

Risk: Reduced dashboard performance.

4. Security Reliance on Proper Configurations

Misconfigured .env files or weak environment setups can affect security.

Risk: Unauthorized access or exposure of secrets.

5. User Behavior Complexity

Emotional and pattern-based spending models are approximate.

Risk: False positives/negatives in insights.

10.2 Future Work

1. Automated Deployment (CI/CD)

Add GitHub Actions or similar pipelines to automate backend deployment.

2. Mobile App Version

Create native iOS/Android apps to improve accessibility and real-time notifications.

3. Advanced Behavioral Models

Integrate richer sentiment analysis, pattern detection, or personalized LSTM models.

4. Budget Templates & Recommendations

Provide AI-driven budgeting plans tailored to lifestyle and student profiles.

5. Notifications & Reminders

Add push/email alerts for bill due dates, overspending, or sudden runway drops.

6. Shared Wallets & Multi-User Support

Allow family or roommates to manage shared expenses.

7. Analytics Dashboard for Trends

Add deeper financial analytics: weekly patterns, vendor analysis, lifestyle scoring.

11. Conclusion

SmartSpend brings together modern web technologies, structured data modeling, and tiered machine learning to create an intelligent, behavior-aware financial management tool. By combining a React-based frontend, a modular Flask backend, a relational MySQL database, and predictive ML models, the system provides users with real-time insights into spending habits, burn rate, and financial runway.

The architecture is designed to be scalable, maintainable, and easy to extend. Clear separation of components, secure authentication, optimized performance, and consistent data flow allow the system to remain reliable while handling sensitive financial information.

While there are areas for future improvement such as automated deployment, deeper behavioral analytics, and mobile application support the current version successfully demonstrates a functional, user-centered financial coach capable of predicting and guiding smarter money decisions.

REFERENCES

1. Kim, J. (2021). Machine Learning Approaches for Personal Finance Forecasting. Journal of Financial Data Science.
2. Thaler, R. H., & Sunstein, C. R. (2008). Nudge: Improving Decisions About Health, Wealth, and Happiness. Yale University Press.
3. Kahneman, D. (2011). Thinking, Fast and Slow. Farrar, Straus and Giroux.