Splitify

Project Name: Splitify
Team Number: 65
Team: Aparna Kudiyirikkal Anil (aparnak5), Ananya Anand (ananyaa9), Cindy Zou (cindyz3), Kaavya Vassa (kvassa2)

Pitch

Splitify is a web app that helps college students and new graduates manage their money more effectively. It scans receipts to fairly split shared expenses and provides personalized monthly budget recommendations based on spending habits.

Functionality

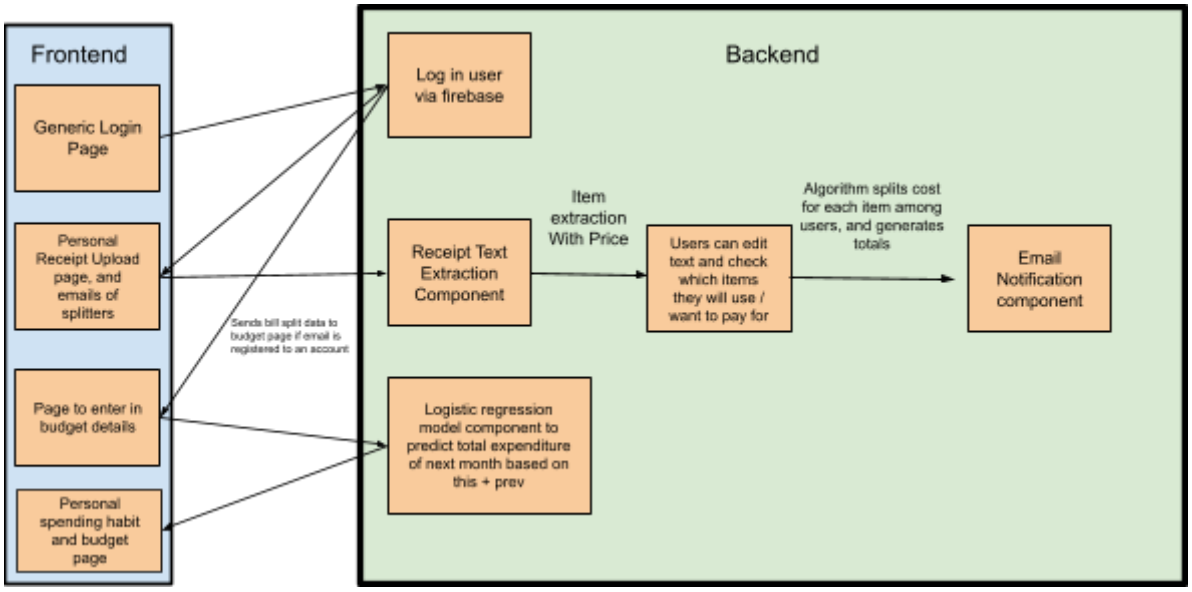
Part 1 - Bill splitting

1. Users can upload and scan a receipt image or PDF, where the system uses OCR to extract line-by-line items with names, quantities, unit prices, and totals.
2. Users can edit or confirm extracted items for accuracy, correcting names, prices, or quantities, and reconcile discrepancies with the detected subtotal and total.
3. Users can assign specific items to different people for bill-splitting, with support for equal division, custom shares, or multiple participants per item.
4. The app automatically splits totals including tax, tip, discounts, and fees across participants, allocating proportionally or evenly based on user preference.
5. App will send payment summaries via email that include itemized breakdowns, per-person totals.
6. Users can view their past spending history in the app, search and filter receipts by merchant or category, and upload receipts without splitting for personal tracking.

Part 2 - AI Budgeting

7. Users receive personalized monthly budget recommendations generated from historical spending, savings goals, and category trends, with adjustable savings sliders.
8. Users can track remaining money in each budget category such as food, rent, and utilities, with live counters showing budgeted, spent, and remaining amounts.
9. Users can export monthly summaries and receipts in CSV, XLSX, PDF, or OFX formats, including category totals, per-person ledgers, and links to original receipt images.
10. Users receive notifications about spending habits such as budget thresholds reached, recurring payments, unusual expenses, or reimbursement reminders, delivered in-app, via email, or optionally SMS.

Components



Short List

Backend:

- Login mechanism to separate accounts (firebase)
- Receipt Processing Component
- Bill Splitting Component
- Email Notification Component
- Budgeting & Recommendation Component
- Database Component

Frontend:

- Web Interface Component

Longer List

Backend Components

1. **Login Component**
    a. **Functionality:** Handles authentication for users. Uses Firebase Authentication for login and integrates with Supabase to manage session persistence and link user accounts to stored receipts, splits, and budgets.
    b. **Language:** TypeScript/Node.js (for Firebase SDK integration) or Python (if handled in backend services).
    c. **Libraries:** Firebase Auth SDK, Supabase Auth client.
    d. **Testing:** Unit tests using mocked Firebase responses; integration tests ensure login sessions sync correctly with Supabase database entries.
    e. **Interactions:** Communicates with Firebase for authentication and Supabase for storing and retrieving user session data. Provides authenticated context to the Receipt Processing, Bill Splitting, Budgeting, and Email Notification components.
2. **Receipt Processing Component**

a. **Functionality:** Handles image upload, performs OCR to extract text, parses line items, and prepares editable data for users. Saves results to Supabase
b. **Language:** Python (strong image-processing ecosystem & AI/ML support).
c. **Libraries:** OpenCV (image preprocessing), Tesseract or Google Vision API (OCR).
d. **Testing:** Unit tests with a sample dataset of receipts to measure OCR accuracy and parsing correctness. Integration tests confirm correct database writes.
e. **Interactions:** Communicates with Supabase to store parsed receipt data; sends structured line items to the Bill Splitting Component.

3. **Bill Splitting Component**
    a. **Functionality:** Manages assignment of items to participants, computes fair totals including tax/tip, handles rounding, and generates payment requests.
    b. **Language:** Python (business logic in backend).
    c. **Libraries:** None beyond standard math/logic libraries.
    d. **Testing:** Unit tests for split calculations (edge cases like rounding, uneven splits). Integration tests validate reconciliation with totals.
    e. **Interactions:** Reads parsed items from the Receipt Processing Component; writes per-person totals into Supabase; passes results to the Email Notification Component.

4. **Email Notification Component**
    a. **Functionality:** Sends itemized breakdowns of what each participant owes, with links to pay.
    b. **Language:** Python or Node.js (for async backend integration).
    c. **Libraries:** SendGrid API
    d. **Testing:** Integration tests with sandbox/test accounts to verify formatting and delivery; mocked APIs for unit testing.
    e. **Interactions:** Pulls split results from Supabase; communicates directly with users via email.

5. **Budgeting & Recommendation Component**
    a. **Functionality:** Analyzes past spending history and predicts next month's expenditures using a simple **logistic/linear regression** model.
    b. **Language:** Python (for ML/analytics).
    c. **Libraries:** Pandas (data analysis), Scikit-learn (linear/logistic regression), Prophet (optional forecasting).
    d. **Testing:** Unit tests on budget calculations; synthetic data simulations to validate predictions.
    e. **Interactions:** Reads historical data from Supabase; sends budget recommendations to the Frontend.

6. **Database Component (Supabase/PostgreSQL)**
    a. **Functionality:** Stores user accounts, receipts, line items, participants, splits, budgets, and predictions. Provides authentication via Firebase integration.
    b. **Language:** SQL (PostgreSQL).

c. **Libraries:** Supabase SDK: Prisma or SQLAlchemy (ORM).
d. **Testing:** Schema validation, migration testing, and integration tests with backend components.
e. **Interactions:** Central store accessed by all backend components; synchronizes with Firebase for auth sessions.

Frontend Components

1. **Generic Welcome + Login Page Component**
    a. **Functionality:** Provides the initial landing page where users are greeted and prompted to log in. Displays a simple welcome message, branding, and a login button. The login button integrates with Firebase Authentication to allow sign-in via email, Google, or other providers. After successful login, the component redirects the user to their personal receipt upload and budget pages.
    b. **Language:** TypeScript with Next.js (leverages server-side rendering for faster page loads and smooth user experience).
    c. **Libraries: React for UI, Firebase Auth SDK for login, Tailwind CSS and shadcn/ui for styling and components.**
    d. **Testing:** End-to-end tests with Cypress to validate login flows, mocked Firebase responses to simulate authentication states, and Jest snapshot tests for consistent UI rendering.
    e. **Interactions:** Communicates with Firebase Authentication for login; upon success, passes the user's session to Supabase for persistence and grants access to backend APIs.
2. **Receipt Upload & Splitter UI Component**
    a. **Functionality:** Allows users to upload receipts, review extracted items, and assign them to participants. Displays split results.
    b. **Language:** TypeScript with Next.js.
    c. **Libraries:** React, Tailwind CSS, shadcn/ui.
    d. **Testing:** End-to-end UI testing with Cypress/Playwright (upload → parse → confirm).
    e. **Interactions:** Calls backend APIs for receipt OCR and bill splitting; updates Supabase session state.

3. **Budget & Spending Page Component**
    a. **Functionality:** Lets users enter budget details, view spending summaries, and predictive spending for the next month.
    b. **Language:** TypeScript with Next.js.
    c. **Libraries:** React, Chart.js/Recharts for visualization.
    d. **Testing:** UI snapshot testing and integration tests with simulated backend responses.
    e. **Interactions:** Pulls aggregated data and predictions from backend (Budgeting Component); writes user budget goals into Supabase.

Continuous Integration

1. **Testing Library**
    a. **Backend (Python):** Pytest will be used for unit and integration testing. This includes receipt OCR parsing tests, bill-splitting calculations, ML model predictions, and Supabase database interactions. Pytest is chosen for its simplicity, rich ecosystem of plugins, and familiarity from prior coursework.
    b. **Frontend (TypeScript/Next.js):** Jest will be used for component-level testing, along with React Testing Library for user interaction tests. We will also use Cypress/Playwright for end-to-end testing to simulate workflows (e.g., uploading a receipt and receiving a split summary)
2. **Style Guide**
    a. **Python:** PEP 8 will be the style guide for backend code. We will enforce this automatically with Flake8 and Black for formatting.
    b. **TypeScript:** ESLint and Prettier will be used to enforce consistent style in the frontend codebase. These tools integrate easily with VS Code and CI pipelines, ensuring formatting and linting checks are automated.

3. **Test Coverage**
    a. **Backend:** Coverage.py will be used to measure which parts of our Python code (receipt processing, bill splitting, ML budgeting, Supabase queries) are exercised by tests.
    b. **Frontend:** Jest's built-in coverage reporter will track which components, hooks, and utility functions are tested. End-to-end coverage will be approximated through Cypress runs.
    c. Our target is at least **80% coverage** across backend and frontend codebases to ensure all major workflows are validated.

4. **Pull Request Workflow**
    a. Each feature branch must open a pull request (PR) within one week of development starting.
    b. Every PR must receive at least **one peer review** before merging. Reviews will focus on correctness, adherence to style guides, and adequacy of test coverage.
    c. Reviewer assignments will **rotate weekly** during team meetings to balance review responsibilities. If a reviewer is unavailable, the alternate will be assigned during the meeting.
    d. To prevent merge conflicts, contributors will rebase their feature branches onto the latest main before merging. Small, frequent merges are encouraged to minimize conflicts with parallel development.

Schedule (10 Weeks)

Week 1

- [x] ~~Set up GitHub repository~~
- [ ] Design initial Supabase database schema and integrate Firebase authentication.

Week 2

1. Build the generic welcome + login page in the frontend.
2. Connect login flow with backend session persistence (Firebase → Supabase).

Week 3

1. Create receipt upload UI in the frontend.
2. Implement basic backend API to accept receipt files and store metadata in Supabase.

Week 4

1. Integrate OCR pipeline in the backend (OpenCV + Tesseract/Google Vision).
2. Display parsed line items in an editable preview in the frontend.

Week 5

1. Implement bill splitting logic in the backend (tax, tip, rounding).
2. Build frontend UI for assigning items to participants and viewing split results.

Week 6

1. Add email notification system using SendGrid/AWS SES.
2. Build backend integration to send per-person breakdowns and update delivery logs in Supabase.

Week 7

1. Enable CSV upload of past transactions for budgeting analysis.
2. Store imported transactions in Supabase and make them viewable in the frontend.

Week 8

1. Implement simple ML-based budgeting recommendations in backend (linear regression / moving average).
2. Create frontend dashboard to display budget summaries and recommendations.

Week 9

1. Add notification system for spending thresholds and unusual transactions.
2. Build UI components for showing alerts and reminders within the dashboard.

Week 10

1. Conduct final end-to-end testing across frontend and backend (receipt upload, split, notifications, budget).
2. Polish UI, resolve bugs, and keep buffer time for last-minute issues.

Risks

1. **OCR Accuracy Too Low**
    - **Plan:** Allow manual editing of extracted items.
    - **Impact:** Slows users slightly but keeps the feature usable, prioritizes accuracy.
    - **Schedule Adjustment:** No major changes needed.
2. **Budget Recommendation AI Too Complex (?)**
    - **Plan:** Start with simple 50/30/20 rules and spending averages.
    - **Impact:** Delays advanced personalization but keeps the main functions useful.
    - **Schedule Adjustment:** Leave AI forecasting for a later version.
3. **Email Delivery Failures**
    - **Plan:** Use SendGrid's sandbox/testing mode early and monitor delivery metrics with bounce/complaint alerts
    - **Impact:** Some users might not receive payment summaries, reducing trust in the system.
    - **Schedule Adjustment:** If unresolved, pivot to in-app notifications as the primary channel and keep email optional.

Teamwork

- **Reducing Friction:** Use Docker Dev Containers or VSCode Remote to standardize environments, ensuring consistent builds across all team laptops.
- **Work Division:**
    - Rotate roles in lead tech, backend, UI/UX, and testing developer.
    - Catch each other up on the sections that we've progressed in.