You are working inside my full-stack Next.js + Supabase app called **BudgetU**, located at:

https://github.com/hishamkherbouch/BudgetU

This app already has:

• Auth
• Dashboard
• Income tracking
• Expense tracking
• Savings goals + contributions
• Debt tracking + payments
• AI assistant
• Landing page + onboarding
• Supabase migrations

Your job is **NOT** to rewrite the project.

Your job is to **upgrade BudgetU into a launch-ready student budgeting product** by making incremental, production-quality improvements.

You must:

• Read the existing repo structure first
• Reuse existing components when possible
• Do not delete features
• Make small PR-sized changes
• Explain each change before implementing
• Ensure no breaking changes

Do not create fake features. Only implement features listed below.

PROJECT GOAL

BudgetU should be usable by a real college student without confusion.

A user must be able to:

1. Sign up and complete onboarding
2. Add income, expenses, savings, debts
3. Edit and delete entries safely
4. Navigate between months

5. Import expenses via CSV
6. Export all data
7. See correct dashboard totals
8. Use the app on mobile
9. Trust their data will not be lost

IMPLEMENTATION ORDER

You must implement features in the following order.

Do not skip steps.

STEP 1 — Understand Repo

Before writing any code:

1. Analyze folder structure.
2. Identify:
    - Next.js app router layout
    - Supabase client setup
    - Existing tables and migrations
    - How dashboard totals are calculated
    - How expenses/income/savings/debts are stored
3. Produce a short summary of architecture.

Then continue.

STEP 2 — Core UX Improvements

Implement the following without changing design style.

2.1 Month Navigation

Add a global month selector component that:

• Shows current selected month
• Allows previous/next month
• Allows custom month selection
• Uses URL query like ?month=2026-02
• Updates Dashboard, Income, Expenses, Savings, Debt pages consistently

Acceptance criteria:
Switching months updates all totals and lists correctly.

2.2 Edit and Delete Everywhere

Ensure all entities support edit and delete:

• Income entries
• Expenses
• Savings contributions
• Savings goals
• Debts
• Debt payments

Requirements:

• Edit opens modal or inline form
• Delete requires confirmation
• Totals update immediately
• RLS respected

Acceptance criteria:
No entity is permanently stuck or uneditable.

2.3 Standard Empty States

Every list page must show a helpful message when empty.

Each empty state must include:

• What this section does
• Why it matters
• One clear button

Example:
“No expenses yet. Add your first expense to see your budget.”

2.4 Mobile Responsiveness

Audit all main pages:

• Dashboard
• Income
• Expenses
• Savings
• Debt
• Settings

Fix:

• Overlapping elements
• Modals off-screen
• Buttons too small
• Charts unreadable

Acceptance criteria:
App usable on 390px wide screen.

STEP 3 — Data Model Hardening

Without breaking existing data:

3.1 Add updated_at fields

All user-owned tables must include updated_at.

Update automatically on row change.

3.2 Add indexes

Add indexes for fast queries:

(user_id, date desc)

On income, expenses, savings contributions, debt payments.

3.3 Category System

Add a categories table if not present.

Requirements:

• Default categories seeded
• Users can add custom categories
• Expenses reference category_id

STEP 4 — CSV Import and Export

4.1 Export

Add “Export All Data” button in settings.

Export includes:

• Income
• Expenses
• Savings contributions
• Debts
• Debt payments

Single CSV or zip of CSVs.

4.2 Import Expenses

Add CSV uploader page.

Requirements:

• Upload CSV
• Map columns
• Preview before saving
• Auto-categorize using keyword rules
• Avoid duplicates

Duplicates detected using hash of date+amount+description.

Acceptance criteria:
Importing 50 rows works without errors.

STEP 5 — Budget Logic Verification

Create one central calculation function.

For selected month:

income_total = sum income
expense_total = sum expenses
savings_total = sum contributions
debt_paid_total = sum debt payments

budget_remaining = income_total − (expense_total + savings_total + debt_paid_total)

savings_rate = savings_total / income_total

All dashboard values must come from this function.

Add tooltip explaining formulas.

Acceptance criteria:
Dashboard numbers match lists exactly.

STEP 6 — Recurring Transactions

Add minimal recurring rules.

Users can create recurring income or expenses.

Fields:

• amount
• category
• frequency
• start date

Occurrences generated for selected month.

Acceptance criteria:
Recurring rent appears automatically each month.

STEP 7 — Subscription Detection

Detect recurring expenses automatically.

Logic:

If same merchant appears 3+ times monthly → subscription.

Create subscriptions page showing:

• merchant
• average amount
• estimated monthly total

Allow ignore or mark canceled.

STEP 8 — Debt Module Upgrade

Add debt detail page with payoff simulator.

Inputs:

• principal
• interest rate
• monthly payment

Outputs:

• payoff date
• total interest paid

Acceptance criteria:
Student can understand how long until loan is paid.

STEP 9 — Savings Goals Upgrade

Add templates:

• Emergency fund
• Study abroad
• Laptop purchase

Add goal detail page with contribution history.

Acceptance criteria:
Progress bar always matches contributions.

STEP 10 — Data Trust Features

Add in settings:

• Export data
• Delete account and data
• Privacy page

Acceptance criteria:
User can export or delete data in under 3 clicks.

STEP 11 — Quality and Testing

Add minimal tests:

• Budget math tests
• CSV import tests
• Category mapping tests

Add QA checklist page in README.

HOW YOU SHOULD WORK

For each step:

1. Explain what files will change.
2. Implement changes.
3. Ensure no TypeScript errors.
4. Run build.
5. Summarize changes.

Never implement multiple steps at once.

Wait for confirmation between steps.

IMPORTANT CONSTRAINTS

• Do not rewrite existing UI unless necessary
• Do not change auth logic
• Do not remove Supabase usage
• Do not create fake APIs
• Keep code clean and typed
• Use existing style system

OUTPUT FORMAT

When implementing a step:

1. Explanation
2. Files changed
3. Code diff
4. Testing instructions

This is a student project but must be production-quality.

Start with STEP 1: Analyze the repository architecture.