Team info

Team Info

- America Pacheco: Front-End Developer: UI design, HTML, CSS, JavaScript components
- Bailey Bounnam: Back-End Developer: API integration, authentication
- Ross Henderson: Back-End Developer: Feature implementation
- Brian Fang: Database Manager: Schema design, MySQL optimization
- Bryan Partida: Front-End Developer: JavaScript logic, UI interactivity

Github Link

https://github.com/henderos/CS362-Class-Project

Communication Method

Our primary communication platform is Discord, which we use to create dedicated threads for tasks, announcements, and general Q&A. Team members strive to respond within 17 hours to keep collaboration efficient and transparent. If a question remains unanswered, a polite reminder is posted to prompt further discussion. We also hold weekly check-ins via voice calls to review progress and address any technical or scheduling issues. This structured communication framework helps us tackle problems early and maintain a consistent development pace.

Product description

Project Title

Dam Dollars

Abstract

Dam Dollars is a web-based personal finance platform designed to help users simplify and optimize their money management. It offers tools for budget-setting, subscription tracking, and real-time financial analytics, consolidating all banking data into one intuitive dashboard. By centralizing financial details from multiple accounts and providing advanced features without a paywall, Dam Dollars removes the hassle of manual tracking and helps users make informed spending and saving decisions. Moreover, it emphasizes user-friendly interfaces and actionable insights, aiming to motivate people to establish healthy financial habits. In essence, Dam Dollars acts as a one-stop solution for individuals seeking better control of their financial well-being.

Goal

This project will help users to manage and keep track of their personal finances, see where their money is going, and make decisions for future spending based on these facts.

Current Practice

In today's personal finance landscape, many individuals rely on spreadsheets, disjointed banking apps, or expensive third-party tools to track their money. These methods often require significant manual effort, from entering transactions to juggling multiple logins for different bank accounts, which increases the likelihood of errors. Furthermore, features such as advanced budgeting and subscription tracking are commonly locked behind paywalls, preventing widespread access. The lack of automation in these tools leaves users without a centralized, real-time snapshot of their overall spending, leading to frustration and missed opportunities for better budgeting. By contrast, Dam Dollars aims to eliminate these obstacles through a unified platform that automates transaction monitoring and provides all essential financial insights in one place.

Novelty

Most personal finance tools provide basic account tracking but still require users to jump between multiple apps for a complete overview, often charging extra for more sophisticated features. Dam Dollars bridges this gap by consolidating accounts, budgets, and subscriptions into a single platform while offering real-time spending insights—without locking key capabilities behind a paywall. Unlike existing solutions, which frequently rely on extensive manual input or premium tiers, Dam Dollars focuses on quickly identifying overlooked subscriptions and "problem areas" (e.g., high spending in specific categories) so users can take timely corrective action. This tailored approach emphasizes convenience and affordability, creating a unique blend of everyday usability and proactive guidance that stands apart from the fragmented, costly offerings many people encounter today.

Effects

The people who will be using our web app are people who want to see all their financial information with a user-friendly interface. If we are successful in our app, our app will show more transparency with how our users' money is being handled. This will help reduce unnecessary expenses by identifying underused subscriptions and help users allocate their income effectively by highlighting spending patterns.

Use Cases

Use Case 1: Subscription Tracking:
Actors: User, System
Triggers: User navigates to the "Subscriptions" section of the app.
Preconditions: User has linked financial accounts, and transaction data is available.
Postconditions: The system identifies recurring transactions and displays them as subscriptions.
List of Steps: User selects the "Subscriptions" tab. System retrieves recurring transactions from the linked accounts. System categorizes these transactions and flags underused subscriptions. User reviews and optionally adjusts subscription tracking preferences.
Extensions/Variations: User can manually add subscriptions. User can flag a subscription as "not recurring."
Exceptions: Errors in identifying recurring transactions. Incomplete transaction data.

Use Case 2: Budget Setting:
Actors: User, System
Triggers: User navigates to the "Budgeting Tool" section of the app.
Preconditions: User has linked financial accounts and user has access to spending categories.
Postconditions: The system stores the user's budget for each selected category.
List of Steps: User selects "Budgeting Tool". User selects a spending category. User enters budget amount.
Extensions/Variations: If user is editing an existing budget
Exceptions: If the budget amount is too small/large

Use Case 3: Viewing Financial Data:
Actors: User, System
Triggers: User logs into the app
Preconditions: User has linked financial accounts.
Postconditions: The user sees an overview of recent transactions, monthly spending, and account balances
List of Steps: User logs in. System retrieves financial data. Dashboard shows financial information.
Extensions/Variations: User can change between light and dark mode
Exceptions: If the system can't connect to the bank account

Use Case 4: Receiving Budget Alerts:
Actors: User, System
Triggers: User is almost over budget for category
Preconditions: The user has linked financial accounts, the user has set budgets for categories and the user has set alert preference (email, through app, etc.)
Postconditions: The system notifies the user through an alert
List of Steps: The user spends money in tracked category. The system calculates total spending in category. The system identifies that user is close to their set budget. The system sends an alert to the user.
Extensions/Variations: The alert preference is changed
Exceptions: Alert fails to send

Use Case 5: Generating Spending Reports:
Actors: User, System
Triggers: User selects "Generate Spending Report" on app
Preconditions: The user has linked financial accounts and financial data is available for time period.
Postconditions: The system processes and generates a spending report for selected time period
List of Steps: The user selects "Generate Spending Report". The user selects a time period for report. The system processes and generates a spending report.
Extensions/Variations: The user wants to export report as a different file type
Exceptions: Insufficient data for selected time period

Non-functional Requirements

Scalability: The system should handle up to 10,000 concurrent users without performance degradation. This includes managing API calls to Plaid and retrieving data from the MySQL database efficiently.

Security and Privacy: User data must be encrypted in transit (HTTPS) and at rest (AES-256). Only authenticated and authorized users can access sensitive financial data.

Usability: The web application must be responsive and accessible on desktop devices and as a stretch goal, should also be accessible on mobile.

External Requirements

- The product must validate and handle user input errors gracefully.
- Deployment must include a public URL accessible to end-users.
- Comprehensive documentation must be provided, including installation instructions for developers and usage instructions for users.
- The project scope must align with the team's resources and timeline.

Technical Approach

Dam Dollars is designed as a unified web platform that securely centralizes all the key functions of personal finance management. At a high level, users create an account, link their financial institutions through a secure integration (e.g., Plaid API), and immediately gain a consolidated view of their finances. This consolidated data is stored in a MySQL database, where it is organized to support quick lookups, subscription tracking, and budgeting analytics. A Node.js/Express server handles the main business logic—managing user authentication, fetching and normalizing transaction data, and processing rules for features like budget alerts and subscription identification. On the front end, the user interacts with a streamlined interface built using modern JavaScript, which guides them through setting up budgets, reviewing subscriptions, and viewing real-time spending insights. By consolidating data sources in one place and pairing them with automated notifications, Dam Dollars significantly reduces manual overhead and the need to switch between multiple tools.

Major Features

- **Account Creation**: User has the possiblilty of cretaing an account using a valid email address and a password.
- **Financial Information**: Using Plaid, user can link different bank accounts to his Dam Dollars account easily, using banking credentials.
- **Dashboard Overview**: Display consolidated financial data, including recent transactions, and monthly spending summaries.
- **Subscription Tracking**: Identify and track recurring subscriptions, flagging underused services.
- **Spending Categories**: Provide visual breakdowns of expenses by category (e.g., groceries, rent, entertainment).

- **Budgeting Tool**: Allow users to set budgets for categories and receive alerts when nearing limits.

Stretch Goals

- **AI**: Implement AI suggestions and advice for users.
- **Mobile Support**: Display content to be seen and used effectively on mobile devices.

Timeline

Week 1: Project Initialization: Set up project environment, configure version control, establish coding standards, and set up MySQL databased.
Week 2: Front-End & Database Design: Design wireframes, create front-end skeleton, and finalize database schema. (Requires project setup from Week 1)
Week 3: Plaid API Setup: Research, register, and integrate basic Plaid API functionality. (Requires database setup from Week 1)
Week 4: Data Retrieval & Display: Fetch and display financial data from Plaid API on the front-end. (Requires Plaid API setup and front-end structure from Week 2)
Week 5: Budgeting Tool Development: Implement budget categories, user input fields, and real-time data storage. (Requires database schema and financial data retrieval)
Week 6: Subscription Tracking Feature: Identify recurring transactions and display user subscriptions. (Requires financial data retrieval from Week 4)
Week 7: Spending Report Generation: Implement data visualization for spending trends and financial reports. (Requires data retrieval and categorization from previous weeks)
Week 8: Usability Testing & UI Refinements: Conduct user testing, fix UI issues, improve accessibility, and refine navigation. (Requires major features to be implemented)
Week 9: End-to-End Testing & Debugging: Test entire workflow, fix integration issues, and optimize performance. (Requires all major features completed)
Week 10: Final Deployment & Documentation: Write final project documentation, deploy the application, and conduct final review. (Requires complete system testing)

Software Architecture

Overview

The Personal Finance Analyzer is a web application designed to help users track and analyze their spending habits, manage subscriptions, and set budgets. The application follows a layered architecture with a Front-End (UI/UX), Back-End (APIs and business logic), and a Database layer for secure financial data storage. We rely on the Plaid API for retrieving financial transactions from various financial institutions.

Major Software Components

Front-End (Client)
- **Technologies**: HTML, CSS, JavaScript
- **Functionality**:
- Displays user dashboards, budget tools

- Sends requests to the Back-End for data
- Offers interactive visualization

Back-End (Server)
- **Technologies**: Node.js, Express
- **Functionality**:
- Receives requests from Front-End and processes business logic
- Integrates with Plaid API to retrieve and normalize financial data
- Interacts with the database layer for secure data retrieval and updates

Database Layer:
- **Technologies**: MySQL
- **Functionality**:
- Stores users, account information, transaction history, budget info
- Ensures data consistency, security, and integrity through relational schema constraints

Plaid API
- **Functionality**:
- Provides secure access to a user's bank account and transactions

Interfaces Between Components

- **Front-End and Back-End**: RESTful API calls.
- **Back-End and Database**: MySQL queries.
- **Back-End and Plaid API**: Secure API calls for transaction retrieval.

Data Storage Details

- **User Data**: UserID, authentication credentials (hashed), account settings.
- **Transactions**: Date, amount, category, merchant.
- **Budgets**: Category, allocated amount, actual spending.
- **Subscriptions**: Recurring payments identified via transaction patterns.

Assumptions

- Users will link their bank accounts via Plaid.
- MySQL is scalable to compensate for user growth.
- All sensitive data will be encrypted.

Alternative Architectural Decisions

While Dam Dollars employs a layered design with a relational database (MySQL) and a REST-based API, we also evaluated other architectural choices to determine the best fit for

our goals, timeline, and team expertise. Below are three potential alternatives we considered:

Alternative 1: NoSQL Instead of Relational Databases
- **Context**: A NoSQL database (e.g., MongoDB) could simplify storage of rapidly changing or varied data structures. - **Trade-Offs**: This flexibility might accelerate early development but can weaken strict data consistency—crucial in financial contexts where transactions and budgets must adhere to clear relational constraints. We ultimately chose MySQL for its robust integrity checks and well-defined relational schema.

Alternative 2: Using GraphQL Instead of RESTfil APIs
- **Context**: GraphQL can reduce data over-fetching and empower clients to request precisely the data they need. - **Trade-Offs**: It generally requires more advanced server-side setup and a steeper learning curve for the team. Given our relatively straightforward endpoints and desire for a quicker initial build, REST offered simplicity and widespread familiarity without sacrificing essential functionality.

Alternative 3: Microservices Instead of a Monolithic Structure
- **Context**: Splitting features—such as subscription management, budgeting, and user profiles—into separate services can enhance scalability and fault tolerance. - **Trade-Offs**: A microservices approach demands more elaborate DevOps pipelines and communication strategies (e.g., message queues or service discovery). For our current scope and team size, a monolithic structure simplified development and testing. Should our user base or features expand significantly, transitioning to microservices remains a future option.

Software Design

Our software design focuses on creating a user-centric environment that is both intuitive to navigate and robust in handling financial data. Below is an overview of the key design considerations and structural elements that guide Dam Dollars.

User Interface Structure

- **Entry Points**: Users begin at the sign-up or login pages, each designed for clarity and minimal friction.
- **Dashboard**: After authentication, users are presented with a consolidated overview of account balances, recent transactions, and quick links to key features like subscription management or budget creation. The dashboard uses prominent visual indicators (e.g., charts, alerts) to highlight spending trends.
- **Budget Management**: A dedicated section allows users to create and modify budget limits across various categories (groceries, entertainment, etc.). Real-time alerts and progress bars offer immediate feedback on spending status.
- **Suscription Management**: This area displays recurring charges across linked accounts. Users can review each subscription's cost, frequency, and usage, making it easier to identify unnecessary or overlapping services.

Data Flow

- **Client-Side Interactions**: When a user performs an action (e.g., adding a budget, marking a subscription as canceled), the front end (built with modern JavaScript frameworks) sends a request to the server.
- **Server-Side Processing**: The Node.js/Express server receives and validates these requests, executes relevant business logic (such as budget recalculations or subscription updates), and retrieves or updates data in the database.
- **Database Layer**: MySQL houses user accounts, transaction records, and subscription details. By using relational constraints, the system ensures data consistency (e.g., a transaction must be linked to an existing user_id).
- **Third-Party Integration**: Where supported, Dam Dollars connects to external APIs (e.g., Plaid) for secure retrieval of transaction data, which is then normalized and stored.

Database Schema

- **Users**: Stores user profiles, authentication credentials (securely hashed), and any personalization settings (e.g., default currency or notification preferences).
- **Accounts**: Lists bank or credit card accounts linked to each user, capturing details like account type and balance.
- **Transactions**: Contains the timestamp, amount, merchant/category, and references the appropriate account_id and user_id.
- **Budgets**: Tracks spending limits across different categories (e.g., groceries), along with timestamps to support monthly or custom budget cycles.
- **Subscriptions**: Records recurring charges (e.g., Netflix, gym memberships) and frequency details, helping users spot underutilized services.
- **Auxilary Tables**: May include categories or merchant lookups to assist with consistency and reporting.

Design Philosophy And Constraits

- **Clarity Over Complexity**: Each page is organized around a primary user goal (e.g., reviewing budgets), minimizing clutter.
- **Role-Based Access and Security**: While many features are end-user facing, administrative tools (like analyzing aggregated usage metrics) are restricted to authorized team members via role-based access control.
- **Scalability and Felixibility**: If Dam Dollars needs to integrate with additional third-party providers or introduce new financial categories, the database schema can accommodate these expansions through well-defined relationships and table structures.

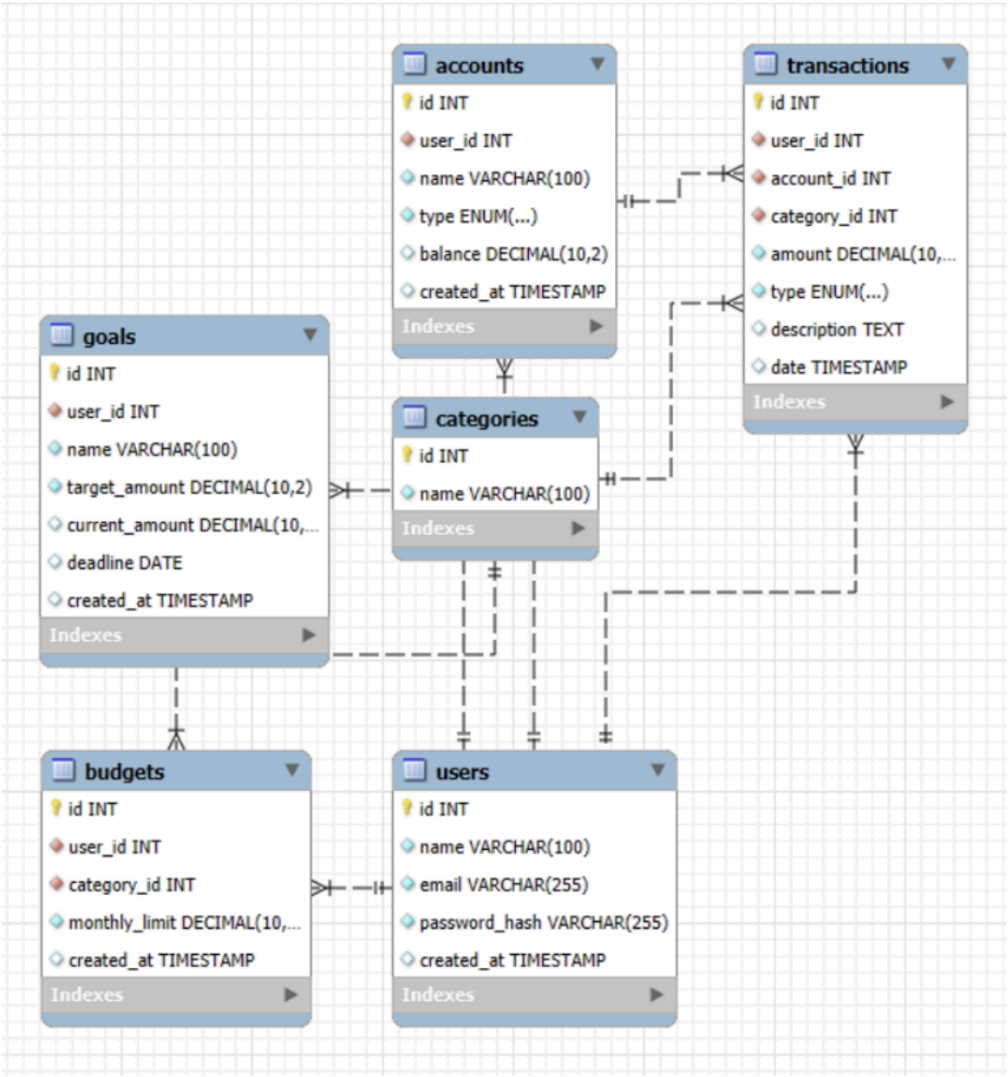
Adaptability for Future Features

- **Modular UI**: Each major feature (budgets, subscriptions, reports) is a distinct component, allowing for straightforward updates or the addition of new modules (e.g., AI-driven financial advice) without overhauling existing code.

- **Extendable Schema**: Extra fields or tables can be introduced as data needs grow, such as for more detailed transaction analytics or advanced forecasting features.

Database Tables

- **users (id, email, passwordHash, settings)**
- **transactions (id, userID, date, amount, category, merchant)**
- **budgets (id, userID, category, limit, spent)**



Coding Guidelines

- **JavaScript (Node.js, Front-End)**: Airbnb JavaScript Style Guide
- **SQL (MySQL)**: SQL Style Guide

We chose these guidelines because they are widely accepted industry standards, ensuring code readability, maintainability, and consistency across the project. To enforce them, we will do continuous checks each week on the new pushes to the GitHub repo and make sure they follow the guidelines.

Process Description

Risk Assessment

1. **API Downtime**
   - **Likelihood**: Medium
   - **Impact**: High
   - **Evidence**: Plaid API has occasional service outages
   - **Steps to Reduce Likelihood/Impact**: Implement error handling and retry mechanisms. Monitor Plaid's status page and API response logs.
   - **Plan for Detecting the Problem**: Set up API health checks to monitor response failures. Implement logging for API calls to detect failures early.
   - **Mitigation**: Implement caching, fallback strategies
2. **Security Breach**
   - **Likelihood**: Low
   - **Impact**: High
   - **Evidence**: Financial data is a high-value target for attackers, and API keys or database vulnerabilities could be exploited.
   - **Steps to Reduce Likelihood/Impact**: Use AES-256 encryption for sensitive data. Implement authentication and authorization best practices.
   - **Plan for Detecting the Problem**: Set up monitoring for unauthorized access attempts.
   - **Mitigation**: Immediately revoke compromised credentials and notify affected users and follow a data breach response plan.
3. **Database Performance Issues**
   - **Likelihood**: Medium
   - **Impact**: Medium
   - **Evidence**: The project aims to scale up to 10,000 users, and MySQL performance may degrade with large datasets.
   - **Steps to Reduce Likelihood/Impact**: Optimize queries and indexing strategies.

- **Plan for Detecting the Problem**: Monitor query performance and server load.
- **Mitigation**: Indexing, query optimization

Since we submitted the Requirements document, we realized that combining different parts of the project might be harder than we first thought. This is because multiple people are working on different features at the same time. To help with this, we are focusing on better communication and doing regular tests to catch problems early. We are also making a clearer schedule to make sure everything fits together smoothly.

Test Plan & Bug Tracking

- **Unit Testing**: Jest for JavaScript, Mocha for Node.js.
- **Integration Testing**: Postman for API validation.
- **Usability Testing**: User feedback sessions.
- **Bug Tracking**: GitHub Issues.

Documentation Plan (User Guide)

1. **Introduction**
   - Overview of the Personal Finance Analyzer.
   - Key features: Dashboard, Budgeting, Subscription Tracking, Spending Reports.
   - How it helps users manage their finances.
2. **Getting Started**
   - System Requirements: Browser compatibility, internet connection.
   - Account Creation: Signing up, logging in, password recovery.
   - Linking Bank Accounts: Using Plaid API for secure integration.
3. **Using the Dashboard**
   - Overview of financial summary.
   - Navigating different sections (transactions, budgets, reports).
   - Customizing the dashboard.
4. **Managing Transactions**
   - Viewing and filtering transactions.
   - Categorizing expenses.

- Identifying recurring subscriptions.

5. **Budgeting Tools**
   - Setting up budget categories.
   - Tracking spending vs. budget.
   - Adjusting budget limits.
6. **Generating Reports**
   - Viewing spending trends.
   - Exporting financial summaries.
   - Customizing reports.

Testing and Continuous Integration Plan

Test Automation and CI Integration

Test-Automation Infrastructure

Our project utilizes the following test automation tools: - **Mocha & Chai**: For unit and integration testing on the back-end (Node.js/Express). - **Jest**: For front-end testing of JavaScript logic and React components. - **Postman/Newman**: For API validation and integration testing. - **MySQL Test Containers**: To facilitate database integration testing in an isolated environment.

Justification for Choosing These Tools

- **Mocha & Chai**:
  - Widely used for Node.js applications.
  - Supports asynchronous testing.
  - Provides flexible and readable reporting.
  - Requires additional setup for assertions.
- **Jest**:
  - Optimized for JavaScript and front-end testing.
  - Includes built-in mocking capabilities.
  - Fast execution and easy configuration.
  - Can be slower for larger test suites.
- **Postman/Newman**:
  - Facilitates API testing with minimal setup.
  - Provides automation for integration tests.
  - Useful for validating endpoints before deployment.
  - Requires GUI-dependent setup, which can be complex.

- **MySQL Test Containers**:
  - Enables database testing without external dependencies.
  - Ensures data integrity during test execution.
  - Requires Docker for setup and execution.

These tools ensure robust testing coverage for both front-end and back-end components while being easy to integrate with CI.

Adding a New Test

1. **Back-end Test (Mocha & Chai):**
   - Navigate to `server/tests` directory.
   - Create a new test file, e.g., `transactions.test.js`.
   - Use Mocha syntax for writing tests:

```
const chai = require('chai');
const chaiHttp = require('chai-http');
const app = require('../server');

chai.use(chaiHttp);
describe('Transactions API Tests', () => {
  it('should retrieve all transactions', (done) => {
    chai.request(app)
      .get('/api/transactions')
      .end((err, res) => {
        chai.expect(res).to.have.status(200);
        done();
      });
  });
});
```

2. **Front-end Test (Jest):**
   - Navigate to `client/src/__tests__`.
   - Create a new test file, e.g., `BudgetTool.test.js`.
   - Write Jest-based test:

```
import { render, screen } from '@testing-library/react';
import BudgetTool from '../components/BudgetTool';

test('renders budget tool component', () => {
  render(<BudgetTool />);
  expect(screen.getByText(/Set Your
Budget/)).toBeInTheDocument();
});
```

CI Service and Repository Linkage

We have chosen **GitHub Actions** as our CI service.

Justification for Choosing GitHub Actions

- Fully integrated with GitHub, making configuration straightforward.
- Free for open-source projects and provides cost-effective CI/CD.
- Supports extensive custom workflows with YAML configuration.
- Runs CI/CD pipelines directly within GitHub, reducing external dependencies.
- Can be slower compared to dedicated CI tools like CircleCI.

CI Service Comparison

CI Service	Pros	Cons
GitHub Actions	- Deep integration with GitHub - Free for open-source projects - Flexible YAML workflow definitions	- May have slower performance for large projects - Free tier has limited parallelism and minutes
Travis CI	- Long-standing reputation and proven integration with GitHub - Simple configuration with a straightforward `.travis.yml` file	- Build queues can be slow during peak times - Limited free build minutes for private repositories
CircleCI	- Excellent parallelism and customizable workflows - Good performance with advanced caching options - Detailed build insights and reporting	- Steeper learning curve for configuration - Pricing may be less competitive for larger teams

CI Service Configuration

1. **Creating the CI Workflow**
   - Add a `.github/workflows/ci.yml` file to the repository:

```yaml
name: CI Pipeline
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
jobs:
  build:
    runs-on: ubuntu-latest
```

```yaml
steps:
  - name: Checkout repository
    uses: actions/checkout@v3
  - name: Set up Node.js
    uses: actions/setup-node@v3
    with:
      node-version: '16'
  - name: Install dependencies
    run: npm install
  - name: Run tests
    run: npm test
```

2. **Repository Linkage**
   - GitHub Actions automatically integrates with our repository.
   - We ensure that every push or pull request triggers the CI pipeline.

Tests Executed in CI Builds

- **Unit Tests**: Run using Mocha/Chai and Jest.
- **Integration Tests**: API endpoint validation with Postman/Newman.
- **Database Tests**: Run MySQL test containers to validate queries.
- **End-to-End Tests**: (Future Implementation)

CI Triggers

- **On Push**: Runs the full test suite whenever code is pushed to the `main` branch.
- **On Pull Request**: Runs the test suite before merging PRs to ensure stability.
- **Scheduled Runs**: (Future Implementation) Nightly builds to detect regressions.