RMIT University
EEET2582 | Software Engineering: Architecture and Design

Charitan Donation Platform Milestone 1 Report

Squad: Panther

Team: A

Student name & ID: Vo Duc Tan s3817693
Tran Thanh Tu s3957386
Tran Vinh Trong s3863973
Hur Hyeonbin s3740878
Bui Hong Thanh Thien s3878323

Lecturer: Dr. Tri Huynh

Contents

I. Introduction:

In our increasingly interconnected world, the influence of collective goodness can significantly affect communities and individuals in need. Charitan serves as a dynamic and comprehensive platform that connects funders, volunteers, and charitable projects globally. Charitan seeks to leverage technology to establish a cohesive and engaging experience for all users, promoting significant connections and encouraging a culture of generosity.

The platform is founded on two principal objectives. Initially, to aid donors and volunteers in identifying and supporting charitable efforts that align with their ideals, whether local or global. Charitan enables individuals to make informed decisions regarding the optimal allocation of their contributions for maximum impact. Secondly, Charitan equips organizations with the necessary tools to efficiently crowdfund initiatives across multiple sectors, including food, health, education, environment, religion, humanitarian endeavors, and housing. Charitan improves exposure and fosters collaboration and support from a varied array of resources by providing a single platform for these projects.

The following report delves deeper into the specifications of this innovative platform. We will explore the functionalities, user interfaces, and technical architecture that will enable Charitan to fulfill its mission of connecting hearts and hands in the service of global charity by.

II. Project Background:

The Charitan platform is based on the acknowledgment of the Internet's increasing significance in philanthropic contributions. The growing prevalence of online donations and the increasing number of charitable organizations using websites to attract potential donors highlight this trend. The realm of online charitable donations offers both obstacles and prospects. Despite the rise in online donations, a mere minority of charities have effectively utilized the Internet to secure a significant share of their funding. Furthermore, donor contentment with current charity websites is inadequate. [1]

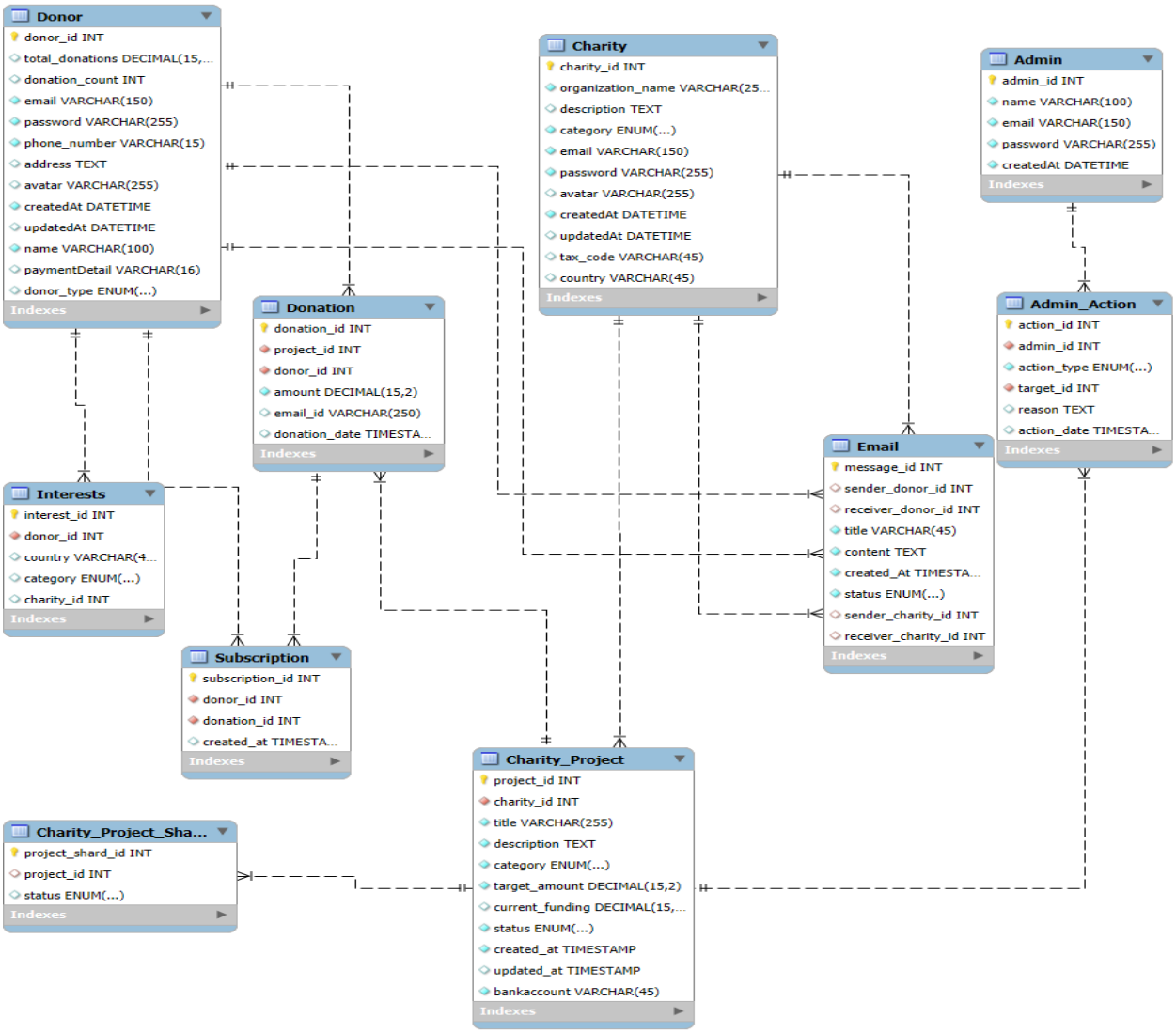
In addressing these problems, the Charitan platform seeks to offer an intuitive interface that streamlines the process of locating and supporting organizations that resonate with contributors' ideals. The software provides charities with comprehensive capabilities to efficiently manage their online fundraising initiatives. The Charitan platform's design and functions are based on a theoretical framework that incorporates the Elaboration Likelihood Model of persuasion, the

halo effect, and the principles of self-schema, congruity, and visual rhetoric. This theoretical framework aims to enhance the platform's efficacy in promoting online philanthropic contributions. [1]

The Charitan platform's key features encompass an intuitive interface for donors and charities, extensive charity search and discovery tools, integrated crowdfunding capabilities, and systems to facilitate collaboration and support from many resources. The platform is anticipated to provide multiple advantages, such as an increase in online philanthropic contributions, heightened donor satisfaction, augmented visibility for charities, and expanded collaboration and support among stakeholders. Charitan has the ability to significantly enhance communities globally by fostering relationships between donors and charities and offering a seamless online giving platform.

III. Concept Data Model:

ER Model:



Description of Entities

Donor:

- Represents individual donors who contribute to charities.
- Attributes:
    - donor_id (PK)
    - total_donations
    - donation_count

- email
- password
- phone_number
- address
- avatar
- createdAt
- updatedAt
- name
- paymentDetail
- donor_type

Charity:

- Represents charitable organizations.
- Attributes:
  - charity_id (PK)
  - organization_name
  - description
  - category
  - avatar
  - createdAt
  - cpdatedAt
  - tax_code
  - country

Admin:

- Represents administrators who manage the platform.
- Attributes:
  - admin_id (PK)
  - name
  - email
  - password
  - createdAt
  - updatedAt

Donation:

- Represents donations made by donors to charities.
- Attributes:

- donation_id (PK)
- project_id
- donor_id
- amount
- email_id
- donation_date

Interests:

- Represents the interests of donors in specific charities or categories.
- Attributes:
  - interest_id (PK)
  - donor_id
  - country
  - category
  - charity_id

Subscription:

- Represents subscriptions to newsletters or updates from charities.
- Attributes:
  - subscription_id (PK)
  - donor_id
  - donation_id
  - created_at

Charity_Project:

- Represents projects undertaken by charities.
- Attributes:
  - project_id (PK)
  - charity_id
  - title
  - description
  - category
  - target_amount
  - current_funding
  - status
  - created_at
  - updated_at

- bankaccount

Charity_Project_Shard:

- Represents shards of the Charity_Project table for better scalability.
- Attributes:
  - project_shard_id (PK)
  - project_id

Email:

- Represents emails sent between donors, charities, and admins.
- Attributes:
  - message_id (PK)
  - sender_donor_id
  - receiver_donor_id
  - sender_charity_id
  - receiver_charity_id
  - title
  - content
  - created_at
  - status

Admin_Action:

- Represents actions performed by admins on the platform.
- Attributes:
  - action_id (PK)
  - admin_id
  - action_type
  - target_id
  - reason
  - content
  - action_date

Description of Relationships

- **Donor - Donation:** One-to-many relationship. A donor can make many donations.
- **Charity - Donation:** One-to-many relationship. A charity can receive many donations.
- **Donor - Interests:** One-to-many relationship. A donor can have many interests.
- **Charity - Interests:** Many-to-many relationships. A charity can be of interest to many donors, and a donor can be interested in many charities.

- **Donor - Subscription:** One-to-many relationship. A donor can subscribe to many charities.
- **Charity - Subscription:** Many-to-many relationship. A charity can have many subscribers, and a donor can subscribe to many charities.
- **Charity - Charity_Project:** One-to-many relationship. A charity can have many projects.
- **Charity_Project - Charity_Project_Shard:** One-to-many relationship. A project can have many shards.
- **Donor - Email:** Many-to-many relationship. A donor can send and receive many emails.
- **Charity - Email:** Many-to-many relationship. A charity can send and receive many emails.
- **Admin - Email:** Many-to-many relationship. An admin can send and receive many emails.
- **Admin - Admin_Action:** One-to-many relationship. An admin can perform many actions.

Attributes Details

- **PK:** Primary Key
- **FK**: Foreigner Key
- **INT:** Integer data type
- **DECIMAL:** Decimal data type
- **VARCHAR:** Variable-length character string data type
- **TEXT:** Text data type
- **ENUM:** Enumeration data type
- **TIMESTAMP:** Timestamp data type
- **Indexes:** Indexes for efficient data retrieval

Additional Notes

- The Charity_Project_Shard table is used for horizontal partitioning to improve scalability.
- The Email table may be used for several types of communication, such as notifications, fundraising appeals, and general correspondence.
- The Admin_Action table can be used to track and audit admin actions, such as creating new users, modifying settings, or deleting data.

Given the circumstances, this ERD is a thorough database design for a non-profit platform that covers admin actions, communication, project management, charity management, donor management, and donation tracking.

IV. System Architecture:

1. Overall System Architecture

- System Level Architecture Diagram & Description of Major Subsystems

1.1. Charitan SPA Application (Client-side)

The Charitan SPA Application is a Single Page Application (SPA) built using React.js and JavaScript. This application provides an interface that helps users intuitively access various features of Charitan Software. The expected user types are Charity, Donor, and Guest, and the Charitan SPA Application offers different pages and functionalities depending on the user type. For example, Charity users can use project management features, while Donor and Guest users can proceed with donations. Charitan SPA handles all interfaces on the client side and communicates with the backend via RESTful APIs.

1.2. Charity Backend Application (Server-side)

The Charity Backend Application is the core backend system that handles unique features for Charity users when they access the Charitan SPA Application. This application processes client-side requests and records the results of the actions performed by users in the database. The Charity Backend Application provides features used by Charity users to manage projects. These include functions for creating, updating, reading, and deleting projects. The backend executes business logic and processes data according to user requests. Moreover, it provides all general features related to the projects including project statistics and project views. The application is built using Express.js and handles API requests on the server side. The Charity Backend Application exchanges data with the Charitan SPA Application via REST APIs.

1.3. Donor Backend Application (Server-side)

The Donor Backend Application is the backend system used by Donor and Guest users when accessing the Charitan SPA Application. This application handles most donation-related functions, except for actual payment processing. It allows users to proceed with donations and view donation records. The Donor Backend Application interacts with the Charitan DB to store and retrieve donation-related data, and it plays a critical role in managing users' donation histories and related functionality.

1.4. Third Party Backend Application:

The Third-Party Backend Application acts as middleware, facilitating seamless integration between Charitan Software and external systems, such as email services, payment systems, Kafka, and Server-Sent Event (SSE) systems. It handles event-driven and request-based data exchanges triggered by Charitan Software. For example:

- When a user completes a donation, Kafka serves as a message broker to queue for the event, ensuring reliable and asynchronous communication between different services.
- An email notification is sent using integrated email services, and real-time updates are pushed to the user interface through SSE.
- The application ensures smooth integration by managing connections with third-party software, processing events through Kafka, and delivering updates via SSE, thereby maintaining the flow of data between Charitan Software and external systems.

This system ensures reliability and scalability by leveraging Kafka for message queuing and SSE for real-time communication, enabling efficient and timely notifications.

1.5. Charitan DB (Database)

Charitan DB serves as the central repository for storing and managing all system data. It permanently stores data generated by the Charitan Backend Application and the Donor Backend Application, including user information, donation records, project data, and other essential information. Charitan DB is designed as a MySQL database deployed on AWS RDS, and it utilizes MySQL to manage data through the Express.js server. The database is a relational database designed for transaction management and data integrity assurance. By using AWS RDS, Charitan DB ensures high availability and scalability, with features like automatic backups and management.

- Integration Points with External Systems

Third Party Email Service: Charitan Software is integrated with the external email service through the Third-Party Backend Application. This integration enables automatic sending and receiving of emails when a user takes an action that triggers the sending or receiving of an email. Data transmission with the email service is handled through the Third-Party Backend Application, and email sending requests are processed via API.

Third Party Payment Service: Charitan Software is integrated with the payment system to enable donors to make donations. It also verifies the payment details provided by the donor or Charitan. When

the donor completes the donation, the Third-Party Payment Service processes the payment, and the result is recorded in the Charitan DB. This integration is carried out through the Third-Party Backend Application, ensuring secure transmission of payment-related data.

Kafka: Kafka serves as an event streaming platform to manage critical events generated by Charitan Software. For instance, when a user completes a donation or a new notification is created, Kafka handles these events as a message queue. The Charitan Backend Application acts as a **Producer**, sending messages to Kafka, while components like the Third-Party Backend Application or SSE function as Consumers, processing these messages. This enables asynchronous handling of tasks such as real-time notifications and email dispatch.

SSE: SSE is used to deliver real-time notifications to users. The Charitan Backend Application leverages SSE to push events, such as notification updates, directly to clients in real time. This connection operates as a one-way stream where the client remains continuously connected to the backend server. As a result, users can immediately view notifications as they occur.

AWS RDS (MySQL): The Charitan DB utilizes a MySQL database deployed on AWS RDS for data storage and management. The Charity Backend Application, Donor Backend Application, and Third-Party Backend Application connect to this database to read and store user information, donation data, and project data.

- Communication Patterns

Client-Server Communication (Charitan SPA - Backend)

Communication between the Charitan SPA Application and the Charitan Backend Application occurs through RESTful APIs. Requests initiated by the client are sent as HTTP requests to the server, which processes the requests and returns the results as HTTP responses. This method follows the standard client-server communication model, where data is exchanged in JSON format.

Backend-Database Communication (Backend - Database)

Communication between the Charity Backend Application, Donor Backend Application, Third Party Backend Application, and Charitan DB is carried out using MySQL queries. When retrieving, updating, deleting, or creating data in the database, the Express.js server executes the MySQL queries and returns the results. This communication is synchronous, and REST API endpoints and JSON are used for data exchange.

Third Party Integration (Backend – third party software)

The Third-Party Backend Application serves as an intermediary between Charitan Software and external systems, such as email and payment services. For example, after a donation is completed, it invokes external email services to send notification emails or processes payments through payment

APIs. This communication occurs via HTTP-based API requests and responses, enabling real-time data exchange between systems.

Additionally, communication with Kafka is conducted over TCP, ensuring reliable and efficient message transmission. Kafka handles donation data and notification-related messages, supporting Charitan Software with high scalability and real-time processing capabilities.

2. Backend Architecture:

Back-end component diagram

DONOR Backend Application

Description of container:

Donor System Application:

This is the main application for back-end of Charitan system for Donor and Guest users. A Guest user can browse, search, view and donate to charity projects, same goes for Donor user. Moreover, Donor user can review the total amount of their donation to projects and receive the notification regarding to successful transactions of a donation.

Components:

Router Controller and Route Component:

Using NodeJS, the Route can handle HTTP requests from Donors or Guests through Route Logic Execution for available components such as **Project Management Controller**, **Donation Payment Controller, Donor Statistic Controller.**

Project Management Controller:

The controller will handle browsing requests such as an overview of available projects and each of their details. Moreover, searching requests are used on **Project Search Service,** and profile information reviewing is based on **Donor Repository** and **Transaction Repository**.

Donation Payment Controller:

Lists out donations of both Donors and Guests' transactions which are process of **Payment Processing Service.**

Donor Statistic Controller:

For Donors only, the controller will provide statistics of donation from using **Donor Statistics Service.**

Project Search Service:

Utilizing **Donor Repository** and **Transaction Repository,** the service will implement the logic to search and filter projects, each project has total amount donation and has record of whoever fund it.

Payment Processing Service:

Integrating the **External Payment,** the service will handle transaction requests with the bank that the donors use in which will be recorded in **Transaction Repository.**

Donor Statistics Service:

Utilizing **Transaction Repository,** the service will statistically list out information regarding the donation that each donor has been contributing.

Donor Repository and Transaction Repository:

Managing CRUD Operations, both Repository send data information. such as donor data and transaction record individually, to the **Database** of the whole Donor System Application.

External Components:

External Payment:

The component is only for external bank system.

CHARITY BackEnd Diagram

Description of container:

Donor System Application:

This is the main application for back-end of Charitan system for Charity users. A Charity user can perform CRUD operations on the projects with necessary information such as title, description, funding goals, and duration. Donation progress of each project from Donor/Guest users can be tracked and update the details of that project. The donation progress can be recorded statistically.

Email API System:

The system provides components for Charity users to receive email confirmation of successful projects.

Components:

Charity Project Controller:

Utilizing the logic of **Charity Project Service** and **Charity Statistics Service,** the controller handles CRUD operations for charity projects.

Charity Statistics Service:

With **Redis Cache,** the logic of the service can perform storing charity project details smoothly with frequently accessed data to calculate total donations and project statistics and send those details to be stored in the **Database** after CRUD operations.

Charity Project Service:

With **Redis Cache,** the service can constantly fetch/update cached project data if there is a change in charity business logic in **Charity Project Repository.** As a new project event published, **Kafka Message Broker** receives and sends notifications to **Email Utility API.**

Charity Project Repository and Router:

Using repository, the logic component can manage charity project data by routing database queries to appropriate shard based on a sharding key in CRUD Operations. There are 6 database shards which are **Health, Education, Environment, Religion, Humanitarian** and **Housing.**

Email API System:

With **Email Service** and **Email Template Manager** handling the logic, in **Email Notification Controller,** the controller will handle the requests to send emails notifications if there is new update on **Charity Project Service.** Moreover, the notifications will be used on the provided template.

External Components:

Redish Cache:

The component is intended to handle the flow of frequently accessed data for optimal performance.

THIRD-PARTY BACKEND APPLICATION

Description of container:

Third-Party Backend Application:

This is the main application for back-end of Charitan system for Donor and Guest users. A Guest user can browse, search, view and donate to charity projects, same goes for Donor user.

External System:

A container contains 2 databases storing Payment and Email and a Stripe Software System.

Components:

Route Controller and Third-Party Controller:

These components are intended for receiving the HTTP requests to manage Third-Party applications such as payment, email, and real-time notification system.

Payment Controller and Payment Service:

These components are intended for processing the incoming payment requests to execute transactions and oversee cryptographic protocols.

Email Controller and Email Notification Controller:

These components are intended for invoking functions from **Email Service** and **Email Template Manager** to dispatch emails of the notifications for the upcoming donation and the updates on the projects by handling requests.

Email Service and Email Template Manager:

These components are intended for managing emails through logging and fetching the templates then recording to the database of the email.

Kafka Controller and SEE Controller:

These components are intended for handling real-time notifications of any upcoming requests from **the Third-Party Controller.**

Transaction Logger:

The component is intended to record any past transactions then store them in the Payment Database.

Payment Gateway Intergration:

The component is intended to process external Stripe banking system.

External Components:

External Payment:

The component is only for external bank system.

3. Frontend Architecture:

Pages/Views

- Page Based Component Diagram
- Project View Component Diagram

- Charity View Component Diagram
- Donor (Personal) Component Diagram
- Charity (Personal) Component Diagram

- **Main Page:**

  Displays the top 10 individual and organization donors. To show the top 10 donors, it interacts with the 'Donor Hook' to retrieve donor information. Additionally, it includes all links and provides global headers, which are part of the 'Main Nav Bar Component' that allows users to navigate to other pages.

- **Project List Page:**

  Displays project information with its general details in a list format. It includes the "Interesting Setting Modal," "Search Input Component," "Project List Component," "Filter Component," and "Pagination Component" to allow users to search, filter, and load a limited number of projects for lazy loading. It also enables users to set their interests, such as country, project category, and charity. To achieve this, the search, filter, and pagination components modify the query string, and the page fetches data based on the updated query string using the "Project Hook." The filtered data received by the project hook is displayed on the screen with lazy loading, showing only the currently fetched data.

- **Project Detail Page:**

  The page can be routed from the "**Project Card Component.**" When the user clicks the card, they can navigate to the page that displays detailed project information such as project description, goals, and funding status. Also, they can also navigate to a donation page to make one from this page.

- **Donation Page:**

  The user can choose whether to subscribe to donations and make recurring payments every month or just make a one-time donation. If the user already has payment details, the system will retrieve the user's payment details from the server. If not, the user will enter the details manually and validate them through a payment hook before making the donation. If the user is a guest, they can proceed with the donation after validating their name and payment details. Once the validation is completed, the user can include a message of up to 250 characters in the donation. After entering the information and the message, payment is completed through the third-party payment software and server communication via the payment hook. Upon

successful completion, the donation information is stored in the database through the donation hook. The project hook updates the donation amount for the project, checks the goal progress, and changes the status if the goal is achieved. Then, the project is moved to the "**Charity Project Shard**," and an email is sent to the charity notifying them of the goal achievement. The donor's message and the donation summary are also emailed to the charity. If the user has subscribed to a donation, the donation hook saves the information about the subscription, along with the email content, via the "Subscription Hook" (guests cannot subscribe). The donation will be processed on the 15th of each month, and the message registered during the initial subscription will be sent to the charity with each monthly payment.

- **Charity List Page:**

  Displays charity information with its general details in a list format, and it uses "Search Input Component", "Project List Component", "Filter Component", and "Pagination Component" which are the same component that are used in Project List page, and the way to search, filter, and pagination is same as "Project List Page**".**

- **Charity Detail Page:**

  By clicking on a charity card loaded from the charity list page, users can navigate to the charity detail page. This page displays all public details of the charity, including its logo, name, description, and projects. Users can also explore all the projects associated with the charity, and by selecting a project, they can navigate to the project detail page, where they can continue with the donation process.

- **Donor Page:**

  This page is accessible only to verified users who have the donor role. On this page, donors can view all their personal information, donation history, and emails. They can see notifications for new emails as well as previous ones, all displayed in a list format, and the user can check the email notification as well. By clicking on a specific email in the list, they can check the content of that email. Additionally, donors can update their personal information and payment details. However, changing payment details will update the payment information for any active subscriptions, and the new payment details must be verified through the payment hook. Donors can view all the projects they have subscribed to and have the option to cancel their subscriptions. They can also view a list of all their donations, along with a comprehensive donation summary. Furthermore, statistics for the projects associated with their donations are also provided.

To avoid prop drilling, its subpages access to the donor information through the **Donor Context API**.

- **Charity Page:**

  The charity can modify its personal information and payment account details. When modifying the payment account, it must be verified through third-party payment software via the Payment Hook. The charity can review and modify information for all projects it has created, but cannot change the name, category, or country. If the name, category, or country is changed, the project will be considered a different one, making it meaningless to previous donors. If changes are made, the project's modification date will be recorded, and it will be marked as updated. Additionally, the charity can check the current donation progress, including the total amount donated for each project, and can navigate to the page to create a new project.

- **Donation Statistic Page:**

  On this page, the donor can view their contribution and the current total donation amount for the project they donated to.

- **Project Statistic Page:**

  Charities can check the progress of their project based on the target amount and the current donation amount. They can also view the list of users who have donated to the project and identify who made the donations. Also, they can modify the project's status (such as in-active, active, hidden) and other related details. Charities can also delete the project. When a project is deleted, it is moved to the Project Shard through the Project Hook.

- **Project Create Page:**

  Charities can create a new project through this page, including the title, description, category, target amount, and image.

o Components

- **Main Navbar Component**: Provides various routing and functions, including project view, charity view, personal view, and the opening of authentication-related modals for the currently logged-in user.
- **Auth Modal:** A modal that can be opened from the main navigation bar, allowing unauthenticated users to sign up or log in through the Sign Up or Sign In process.
- **Signin Component:** A component accessible through the Auth Modal that performs user authentication using the user's email and password.

- **Signup Component:** A component accessible through the Auth Modal that collects user information and facilitates the sign-up process.
- **SignOut Component:** A component accessible to authenticated users that logs the current user out of the web application.
- **Auth Form Component:** A form that collects user information for signing in or signing up, designed to be reusable across both components.
- **Location Change Component:** Helps users change their location and language, interacting with the Context API to manage the user's location and language globally within the web application.
- **Interest Setting Modal:** A modal that allows users to set preferences for notifications (such as country, category, charity, etc.), which can be synchronized with the backend for receiving relevant alerts.
- **Project List Component:** Displays project data retrieved from the project view page in a list layout.
- **Project Card Component:** Displays basic project information in a card format and provides a clickable link to the project's detailed page.
- Search Input Component: A component that helps users search for projects by name or charity name.
- **Pagination Component:** Improves user experience by rendering limited number of projects loaded at once, reducing network load, and providing pagination functionality.

- **Filter Component:** Allows users to filter search results based on various criteria, such as category, country, etc.
- Monthly Subscription Component: A component that allows users to choose whether to make donations on a monthly subscription basis.
- **Payment Detail Form Verification Component:** A component that verifies the validity of the user's existing or newly entered payment information.
- **Donation Message Form Component:** A component that helps donors write messages to charities when making donations, with a character limit of 250.
- **Guest User Payment Form Component:** A component that allows unauthenticated guest users to make donations by entering their name and payment details.
- **Donation Confirm Component:** A component that processes the actual payment using the entered payment information, creates a donation record, and sends notifications to the charity.
- **Charity List Component:** A component that displays all charities in a list on the charity view page.

- **Charity Card Component:** Displays basic charity information in a card format, with a clickable link to the charity's detailed page.
- **Charity Information Component:** Provides detailed information about a charity, including its logo, name, and description.
- **Email Notification Component:** Displays real-time email notifications for the user and triggers a modal to open the user's email inbox.
- **Email Inbox Modal:** A modal where users can view all received emails in a list format.
- **Email Card Component:** Displays basic email information and provides a clickable link to view the email content on an email viewer component.
- **Email Viewer Component:** A component for viewing the detailed content of an email.
- **Donor Subscription List Component:** Displays a list of all projects currently subscribed to by the donor.
- **Donor Subscription Card Component:** Displays basic information about each subscribed project in a card format, with an option to cancel the subscription.
- **Donor Info Component:** A component that provides detailed information about the donor.
- **Donor Info Form Component:** A form that allows donors to update their personal information, including payment details, with validation checks on new payment information.
- **Donation Info Component:** Displays a list of all donations made by the donor and summary information.
- **Donation Summary Component:** Provides a summary of all donations made by the donor.
- **Donation List Component:** Displays all donation records made by the donor in a list format. **Donation Card Component:** Displays basic information about each donation record in a card format.
- **Charity Info Component:** A component that allows users to view all detailed information about a charity.
- **Charity Info Edit Component:** A component that allows a charity to edit its own information, including changes to donation accounts and other personal information.
- **Project Filter Component:** Provides a filtering feature for charities to filter projects based on their status, such as deleted, halt, or completed.
- **Charity Project List Component:** Displays a list of all projects managed by the charity.
- **Charity Project Card Component:** Displays basic information about each project managed by the charity, with a clickable link to the project's detailed page.

- **Project Statistic Component:** Displays project-specific statistics, such as goal, amount donated, and progress.
- **Project Update Component:** A component for updating project-related information, including status, but not allowing changes to the project name, country, or category.
- **Project Form Component:** A form used by charities to create or update projects, allowing them to enter details such as project name, goal, duration, country, and category.

o Services

- **Auth Hook:** Provides functionality for user login by verifying the user's email and password. Additionally, it collects email, password, and personal information to register the user via a REST API and provides functionality for logging the user out.
- **Charity Hook:** Handles interaction with the charity (donation) database, allowing for reading and modifying charity data via a REST API. It helps implement lazy loading by returning the correct charity data based on query strings for pagination, search, and filtering.
- **Email Hook:** Receives the email recipient and sender and sends the email to the recipient. It also allows checking the emails received by the user from the email database based on the current user.
- **Project Hook:** When the project's goal amount is reached, it changes the project's status to "completed" and sends an email to the user. It deletes the project from the Charity_Project table and moves it to the Charity Project Shard table. Additionally, when reading the project list, it returns only projects that belong to the specified country. By using query strings for search, filtering, and pagination, it sends requests to the REST API and returns only the necessary information, implementing lazy loading. Pagination, filtering, and search input trigger lazy loading.
- **Donor Hook:** Provides functionality to return the top 10 individual and corporate donors from the database. It reads information, including the donor's payment details. It also allows access to the donor table to read and modify donor information.
- **Payment Hook:** Handles interaction with third-party software to verify the user's payment details and process the actual order via a REST API.
- **Donation Hook:** Saves donation records in the database, including donor information (who donated, how much was donated) and the donated project.
- **Interest Hook:** Provides functionality to interact with the Interest Table via a REST API, allowing for reading, creating, updating, and deleting donor interests. This is used for sending notifications to donors when projects are added.
- **Subscription Hook:** Provides functionality to interact with a REST API to manage donor subscription data, enabling users to subscribe to or unsubscribe from projects.

- **Notification Hook:** Interacts with the Context API for notifications and connects with Kafka and SSE software to provide real-time notifications to users.

o Utilities

These are general-purpose functions designed for reusability across multiple components or files. They aim to reduce repetitive code and enhance readability by employing clear and descriptive function names.

1. **resizeImageHandler:**

   Reduces the size and quality of uploaded images before sending them to the server. This process occurs during the creation or deletion of charity projects to optimize storage in the database.

2. **IsEmpty:**

   Checks whether input fields are empty. Useful for validating required input fields like titles, descriptions, and user registration fields.

3. **IsOverWordCount:**

   Verifies if text content exceeds the maximum allowed character count in various contexts, such as authentication, project creation/updates, and email content.

4. **IsEmail:**

   Validates whether the provided input matches the format of a valid email address.

5. **IsPassword:**

   Checks if the given password adheres to the required format and strength criteria.

6. **formdate:**

   Returns a properly formatted date string for account creation, project creation, and updates.

7. **toBase64:**

   Converts an image file into a Base64-encoded string, useful for tasks like embedding images into JSON or sending them via APIs.

8. **toFile:**

   Converts a Base64-encoded string back into a file object, enabling it to be processed as a downloadable file.

- **Component Interactions:**

  In React, component interaction can be achieved in numerous ways. For this project, the most common approach is used: passing props from parent components to child components. Additionally, the Context API is employed to manage global variables, avoiding the need for prop drilling when props must be passed down multiple layers. In the future, if the need arises

to control global variables more extensively, Redux will be integrated alongside the Context API.

Moreover, composite components are utilized to make effective use of common components, significantly reducing the amount of repetitive code.

- **Composition Component:**

  This is a method for ensuring code reusability and design consistency. Commonly used elements such as <form> and <button> tags are standardized with a consistent design and structure, allowing them to be reused across various components.

- **Context API for language:**

  When a user mounts the application, their current location is determined using GPS. Based on the detected location, the Accepted Language Header is updated accordingly. The language is then stored as a global variable and applied across all components to ensure consistency. The Accepted Language Header and related global variable can be changed by dispatch function.

- **Context API for Notification:**

  Upon user authentication, the application tracks the user's email address and enables real-time notifications. It connects to external services like Kafka and SSE to send accurate notifications to the user.

- **Charity Context API:**

  The Charity Page involves multiple components and pages. To avoid prop drilling, charity-related data is made globally accessible to all child components and pages within the Charity Page.

- **Donor Context API:**

  Similarly, the Donor Page spans multiple components and pages. To eliminate prop drilling, donor-related data is globally accessible to all child components and pages within the Donor Page.

Integration with Backend

Hooks: The application connects to the backend through hooks that directly use the HTTP request REST API endpoints supported by the backend.

HttpUtilities: It manages the base endpoints and finalizes the requests and responses used by the hooks, ensuring that consistent requests and responses are received.

V. Architecture Analysis:

1. Maintainability:

Maintainability is essential for the sustained success of the Charitan platform. To guarantee the platform's robustness, scalability, and adaptability for future requirements, the following maintainability factors are considered:

- Design choices for maintainability:
    - Modularity: The platform will feature modular architecture, deconstructing functionalities into autonomous, reusable components. This method enhances code reusability, facilitates upgrades, and isolates potential problems, so rendering maintenance more feasible.
    - Loose Coupling: Components will exhibit minimal interdependencies, hence reducing their coupling. This mitigates the ripple effect of modifications, facilitating the alteration of one component without impacting others.
    - Abstraction will conceal intricate implementation details behind transparent interfaces. This enhances comprehension of the code and mitigates the danger of unforeseen repercussions during modifications.
    - Compliance with Standards: Coding standards and style guides will be rigorously implemented to guarantee uniformity and clarity throughout the codebase. This facilitates comprehension and maintenance of the code for developers, regardless of whether they were the original authors.
    - Documentation: Thorough documentation will be developed and upheld, encompassing design documents, API specifications, and code annotations. This assists developers in comprehending the system and implementing informed modifications.

2. Extensibility:

Backend of charitan system is designed with a high extensibility due to its modular architecture and standardized APIs. Each service, such as Charity Project Service and Payment Processing Service, was built independently to handle specific business functions. This approach allows developers to introduce new features or update existing ones without impacting unrelated

components. For example, the use of standardized API like Email Utility API ensures communication among the internal and external system, allowing smooth integration with third-party services like Stripe or PayPal.

A significant advantage of this approach is the support for future enhancements. Whenever a user needs change, the system can adapt by adding new services. Loose connection between services, enabled by Router, Redis Cache and Kafka minimize dependencies. This makes it easier to replace, upgrade or extend modules without extensive refactoring of the entire system. Moreover, the system microservice-ready design allows for independent deployment of services, further enhancing flexibility.

However, extensibility comes with challenges. Ensuring all components comply with standard API protocols and are well-documented increases initial development and maintenance effort. Poor documentation or inconsistent use of interfaces could complicate future integrations or modifications. Additionally, as more services are added, the complexity of managing inter-service communication can grow, potentially introducing integration issues.]

Finally, there is the risk of over-modularization, which may happen when we break the app with too many small parts [3]. While modularity promotes flexibility, it may lead to overhead performance from excessive API calls between services.

3. Resilience:

The Charitan System is designed to be resilient, ensuring consistent operation even under difficult circumstances. A key strength is the use of Kafka Message Broker, which handles asynchronous event processing. This allows the system to recover from temporary failures by replaying queued events ensuring no critical data is lost [4]. Moreover, Redis Cache provides an ability to maintained with two instances, minimizing the risk of downtime caused by cache fail. The database design also enhances resilience through sharding. By dividing data into multiple small shards, the platform isolates failures to specific subsets of data, preventing system-wide disruptions. This approach is especially useful for large-scale operations, as it ensures continued availability of unaffected shards during outages.

However, these resilience mechanisms also have certain trade-offs. For example, reliance on Kafka and Redis requires expertise for proper configuration and monitoring. Improperly configured message brokers or coaching systems could themselves become points of failure.

Another limitation is the complexity of database sharding [5]. As it required careful design to ensure data consistency across shards. Any errors in shard management could lead to data integrity issues or hinder recovery efforts. Finally, even though redundancy reduces the risk of data loss, it

also increases system cost and resource usage. For example, the cost of running two Redis doubles the infrastructure for caching.

4. Scalability:

The Charitan system is built to scale effectively accommodating increasing user data. An outstanding feature is the modular architecture, which allows independent scaling of services like Donation Payment Controller and Project Search Service. This ensures that high demand components can be scaled without overloading the entire system. Kafka Message Broker enhances scalability by distributing event processing across multiple consumers. Redis Cache plays a crucial role in optimizing database by caching frequently accessed data, so that reducing load on the primary database. This setup ensures that the system can handle the increase in traffic, such as real time notification without performance degradation. Moreover, database sharding partitions data into smaller, manageable subsets, enabling horizontal scaling. So that ensuring consistent performance even with large datasets.

Despite these strengths, scalability introduces challenges. Managing sharded databases requires complex partitioning strategies and query optimization. Poorly designed shards can lead to uneven data distribution, causing some shards to become overburdened while others remain underutilized. Similarly, scaling Kafka consumers requires careful monitoring to avoid bottlenecks or message processing delays. Another potential drawback is cost. Horizontal scaling often involves adding more infrastructure, such as additional database instances or caching servers, which can increase operational expenses. Furthermore, ensuring that scaled services maintain consistent communication and data integrity can be technically demanding, especially in a distributed architecture. Lastly, the reliance on caching for scalability introduces the risk of serving outdated data if synchronization with the primary database is delayed. This could lead to inconsistencies in donor or project information.

5. Security:

The Charitan platform emphasizes security as a core component of its system architecture, employing extensive methods to safeguard user data, financial transactions, and overall system integrity. Authentication and authorization constitute the primary security mechanism, employing a complex multi-tiered user access control system that differentiates various user roles, including Donors, Charities, Guests, and Administrators.

User authentication is conducted via a secure process necessitating email and password verification. The platform's Auth Modal serves as a regulated access point, employing validation measures to guarantee that only authorized users obtain admission. Each user role is assigned distinct

rights, thereby prohibiting unwanted access to confidential information. Donors may access solely their personal information and donation history, whereas charities can oversee their projects, and administrators possess extensive system management authority.

Financial security is emphasized through effective payment protection techniques. The platform collaborates with external payment services, guaranteeing that sensitive financial data is handled via safe, third-party channels. Payment information is subjected to stringent verification by specialized Payment Hooks, and all transactions are executed with many layers of security protocols. This method mitigates the danger of financial fraud and safeguards both donors and nonprofit entities.

Data protection encompasses not just financial transactions but also extensive communication security. The platform's email communication system incorporates encryption and regulated notification protocols. An Email Hook overseas exchanges, guaranteeing that user contacts are confidential and safe. The technology grants user's authority over their notification settings, enhancing personal data security.

The platform utilizes the enhanced security features of AWS RDS at the infrastructure level, capitalizing on MySQL's intrinsic security characteristics. The database sharding technique enhances security by segregating data subsets, hence mitigating the risk of extensive data breaches. RESTful API communications are secured by established protocols, potentially incorporating rate limiting and request validation to avert illegal access or system attacks.

Although the existing security architecture is strong, the platform recognizes the persistent difficulty of ensuring security across various linked services. Potential vulnerabilities may arise from third-party service integrations, requiring ongoing monitoring and periodic security assessments. The development team is advised to incorporate more security improvements, including multi-factor authentication, routine security patch upgrades, and thorough logging of administrative activities.[6]

Adherence to data protection regulations is a paramount concern. The platform guarantees transparent user consent protocols and secure management of personal and financial data. Charitan seeks to establish and sustain user trust in its contribution platform by using best practices in data protection and adopting a proactive security strategy.

The security plan embodies a comprehensive approach that reconciles user accessibility with stringent protective measures. The platform establishes a secure environment for charity donations through the implementation of numerous security levels, encompassing user authentication and infrastructure protection. Ongoing enhancement, frequent vulnerability evaluations, and remaining informed about the latest security technologies will be essential for preserving the platform's integrity and user trust.

6. Performance:

The performance architecture of the Charitan system is engineered to provide efficient, responsive, and scalable user experiences across various service engagements. The platform fundamentally employs advanced caching and distributed computing techniques to enhance resource efficiency and reduce response times.

Redis Cache is essential for performance enhancement, offering an in-memory data storage solution that markedly alleviates database strain. By caching frequently requested data, the system may retrieve user information, project details, and donation histories with reduced latency. This method guarantees rapid responses to repeated questions, significantly enhancing the overall user experience, especially during peak traffic times.

The modular architecture improves performance by enabling the independent scalability of individual services. Services such as the Donation Payment Controller and Project Search Service can be dynamically adjusted to accommodate fluctuating load intensities. This detailed approach mitigates system-wide bottlenecks and guarantees that high-demand components can be optimized without affecting the overall performance of the platform. The Kafka Message Broker enhances this method by dividing event processing among several customers, so avoiding single points of congestion and facilitating effective asynchronous communication. [6]

Database sharding constitutes a significant performance optimization strategy. By segmenting data into smaller, more manageable chunks, the system can execute queries more efficiently and allocate computational resources effectively. This method enhances query response times and facilitates horizontal scaling, guaranteeing that the platform can accommodate growing data quantities without compromising performance. This strategy necessitates advanced segmentation techniques and ongoing oversight to avert imbalanced data distribution.

Notwithstanding these sophisticated performance strategies, the system encounters obstacles. The intricacy of sustaining uniform communication across expanded services presents certain latency problems. Furthermore, sophisticated caching systems, although enhancing performance, inherently risk delivering outdated data. The infrastructure's dependence on several technologies such as Redis, Kafka, and sharded databases heightens operational complexity and may introduce performance overhead.

Resource utilization is meticulously governed by astute design decisions. The system's hooks and context APIs are designed to reduce superfluous data transfers by employing lazy loading approaches that get only essential information. The Project Hook retrieves projects according to precise query parameters, hence minimizing superfluous data transmission and processing burden.[6]

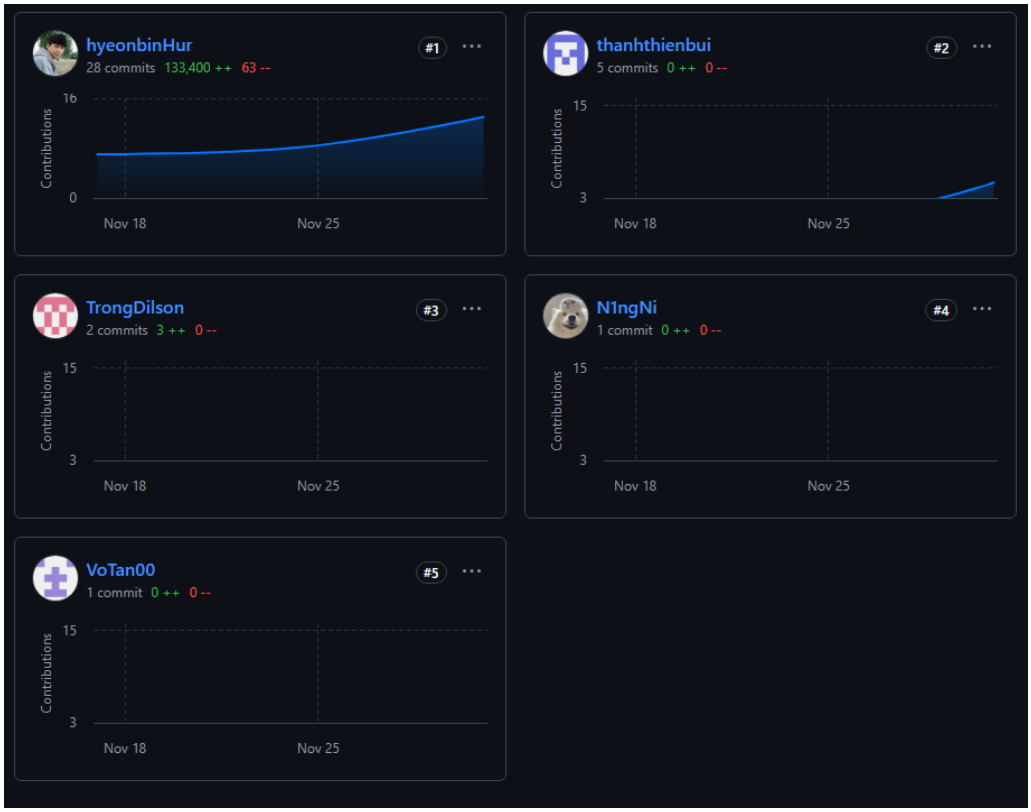
Performance evaluation and ongoing enhancement are essential. The development team must consistently evaluate system metrics, detect potential bottlenecks, and optimize the infrastructure. This may entail modifying caching algorithms, enhancing database searches, or readjusting service scalability procedures.

The performance architecture of the Charitan platform exemplifies a refined equilibrium between technological intricacy and user experience. The system utilizes contemporary distributed computing methods, intelligent caching, and modular architecture to deliver a rapid, dependable, and scalable contribution platform that can expand alongside its user base while upholding elevated performance standards.

VI. GitHub contribution:

GitHub repository: https://github.com/hyeonbinHur/Charitan.git

GitHub ID	Real Name	Student Number
hyeonbinHur	Hur Hyeonbin	s3740878
VoTan00	Vo Duc Tan	s3817693
thanhthienbui	Bui Hong Thanh Thien	s3878323
TrongDilson	Tran Vinh Trong	s3863973
N1ngNi	Tran Thanh Tu	S3957386


hyeonbinHur
28 commits 133,400 ++ 63 --
#1
Contributions
16
0
Nov 18
Nov 25
thanhthienbui
5 commits 0 ++ 0 --
#2
Contributions
15
3
Nov 18
Nov 25
TrongDilson
2 commits 3 ++ 0 --
#3
Contributions
15
3
Nov 18
Nov 25
N1ngNi
1 commit 0 ++ 0 --
#4
Contributions
15
3
Nov 18
Nov 25
VoTan00
1 commit 0 ++ 0 --
#5
Contributions
15
3
Nov 18
Nov 25

VII. Conclusion:

The architecture of the Charitan platform is engineered to deliver a secure, high-performance, scalable, and resilient solution for the management of charitable donations. Its modular and extensible architecture enables adaptation to future requirements and problems, while its emphasis on security and performance guarantees an optimal user experience. By integrating optimal practices and perpetually enhancing its architecture, Charitan may proficiently advance its purpose of linking donors and organizations globally.

The platform's architecture offers a robust basis for present operations while also accommodating future expansion and transformation. The modular architecture, together with standardized APIs, enables developers to seamlessly incorporate new features or alter old ones without compromising the platform's fundamental operation. This adaptability is essential in the ever-changing realm of online philanthropy, where user requirements and technology innovations are always advancing.

Furthermore, the Charitan platform is designed to foster a culture of trust and transparency. Its security protocols safeguard sensitive user information and financial activities, while its performance enhancements guarantee a seamless and efficient user experience. By prioritizing these aspects, Charitan aims to build confidence among both donors and charities, encouraging greater participation and engagement with the platform.

The Charitan platform can be improved by integrating advanced analytics and reporting tools for charities, personalized donor recommendations, and social networking features to cultivate a stronger community among users. Through persistent innovation and adaptation to the evolving dynamics of online philanthropy, Charitan can reinforce its status as a premier platform for linking donors and charities globally.

VIII. References:

[1] “Three research essays on the effects of charity website design on online donations - ProQuest,” *Proquest.com*, 2014. https://www.proquest.com/openview/2df8f28791552f9c5074e67ab029ec6d/1?pq-origsite=gscholar&cbl=18750 (accessed Nov. 27, 2024).

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. [Online]. Available: https://silab.fon.bg.ac.rs/wp-content/uploads/2016/10/Refactoring-Improving-the-Design-of-Existing-Code-Addison-Wesley-Professional-1999.pdf. [Accessed: Dec. 3, 2024].

[3] A. Has, "The Over-Modularization in Mobile App Development," *Medium*, Aug. 5, 2020. [Online]. Available: https://adityahas.medium.com/the-over-modularization-in-mobile-app-development-6e29de5f64be#:~:text=Over%2Dmodularization%20happens%20when%20you,more%20problems%20than%20it%20solves. [Accessed: Dec. 3, 2024].

[4] IBM, "What is a Message Broker?", [Online]. Available: https://www.ibm.com/topics/message-brokers#:~:text=Message%20brokers%20are%20often%20used,the%20components%20of%20an%20application. [Accessed: Dec. 3, 2024].

[5] Design Gurus, "What are the disadvantages of sharding?", [Online]. Available: https://www.designgurus.io/answers/detail/what-are-the-disadvantages-of-sharding. [Accessed: Dec. 3, 2024].

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 1999. [Accessed: Dec. 3, 2024] (for architectural analysis part).