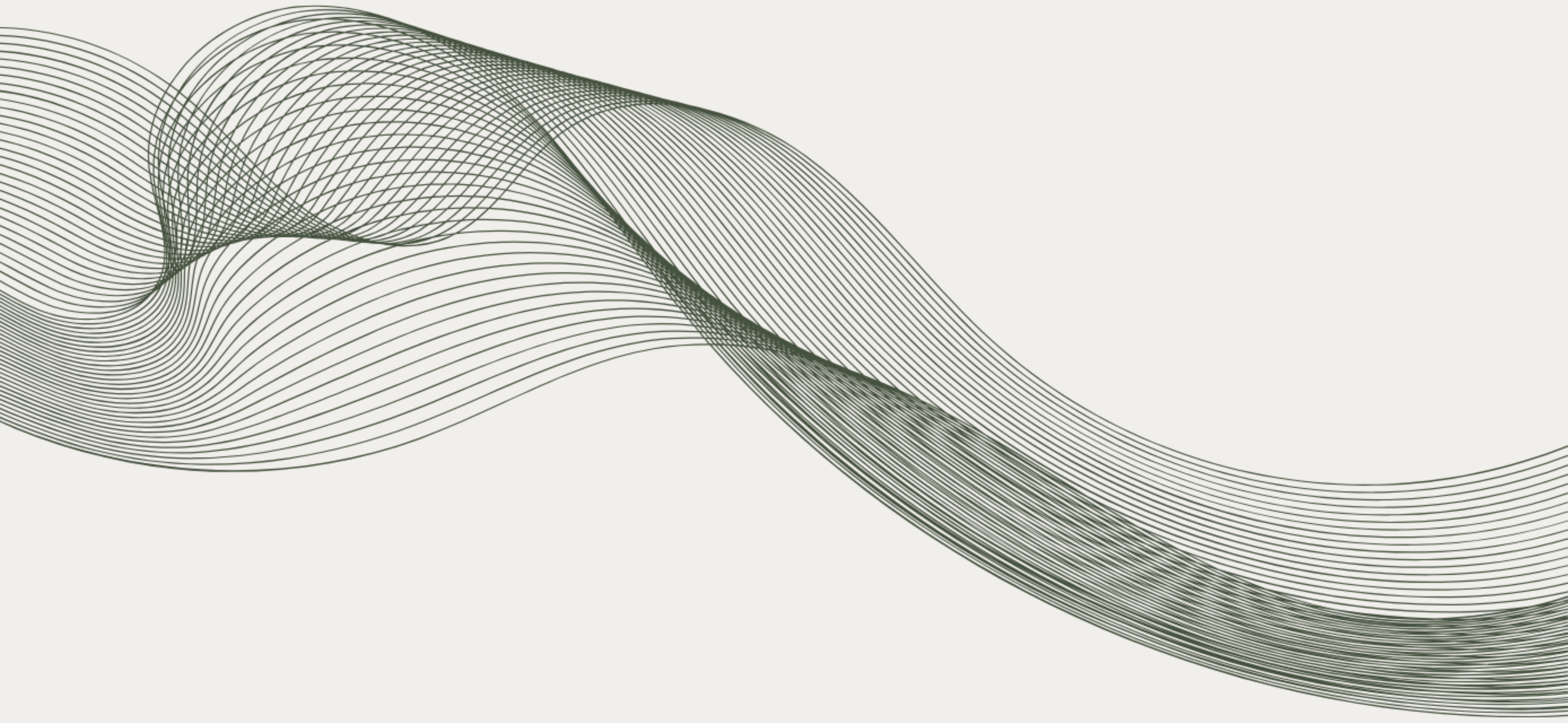
EC-360 NUST FALL'24

SOFTWARE ENGINEERING

Final Project Report



Iman Qaiser (36683)
Iraj Khan (365980)

Table of Contents

FEASIBILITY

Proposal

The proposed project is an e-commerce platform intended to enhance the classic café experience provided by Teaa Cafe. This platform seeks to revamp the ordering process by presenting customers with a user-friendly interface for browning the menu, placing orders and securely processing payments online. It also aims to optimize backend operations by incorporating a database to support funds management and order tracking, ensuring smooth operations for the cafe's management team.

The primary objectives of this project are:

- To improve customer experience by allowing them to browse and place orders remotely
- To streamline the ordering and funds management processes to increase operational efficiency
- To bring in new customers by targeting a broader audience who prefer online ordering options

The e-commerce platform will offer features like order customization, cart management, menu browsing, account creation and login, and secure payment processing as part of its aims and objectives. To guarantee that this digital extension of the cafe is a smooth, automated process that lessens the workload for the cafe management team, these features are to be completely integrated with the backend database.

In conclusion, this interactive online platform aims to provide a seamless online ordering experience while maintaining accurate and consistent order records. The following sections examine the feasibility of the fulfillment of the objectives within the defined financial constraints, resources and timeline of the proposed project.

Software Cost Estimation

Several cost estimation techniques are used to estimate the proposed project's cost in order to guarantee a realistic and balanced approach. Since each method has its strengths and weaknesses, it is important that our final cost estimation is made by combining insights from several approaches. The estimation techniques applied include those discussed in the lectures and are as follow:

1. **Expert Judgement**

By consulting our past database engineering professor and lab instructor, the team was able to gather insights into the most suitable database management server and resources for such an e-commerce platform, as well as an approximation on the cost, from both a financial and labor perspective. The recommended resource was **PostgreSQL Database**, due to its compatibility with **Django Frameworks** for backend functionality integration. Despite the fact that both resources are open source, which is advantageous financially, both faculty members felt that the integration was cumbersome and would thus be labor intensive.

2. **Estimation By Analogy**
   In addition to expert judgment, estimation by analogy techniques were employed by researching similar e-commerce database projects through online internet. This approach not only corroborated the knowledge provided by expert faculty members but offered additional insights into the required project resources. During this research, we discovered several projects that used PostgreSQL for database management and Django frameworks for web development and thus, were able to identify **Visual Studio** as an ideal IDE for effective Django implementation. Furthermore, we noted that **AJAX APIs** were frequently used with HTML web pages to create the smooth, dynamic user experience outlined as a key project objective. Fortunately, all the software and tools that have been identified—such as PostgreSQL, Django, Visual Studio, and AJAX—are free or open-source, which keeps the project cost for these resources minimal.

3. **Parkinson's Law**
   Considering that this project was undertaken as a student semester project, we were bound by a strict timeline and resource limitations, both of which Parkinson's Law considers when estimating costs. Deploying the platform on web services would have been ideal, but given the scope of the project, these high-cost resources would not have been feasible. Rather, we make use of the easily accessible resources via the Django framework, which is used to launch a prototype of our platform on a locally hosted, lightweight web server. This ultimately allows us to meet the outlined project objectives while considering the constraints of a semester project.

4. **Pricing To Win**
   In this approach, the project's cost is determined by the customer's budget. For this project, the development team acted as both the creators and the funders, covering all expenses out of pocket. Given these circumstances, the optimal scenario was to minimize financial expenditure as much as possible, especially since the project was already labor intensive. Therefore, we proceeded with the development process, using the open-source frameworks identified through the earlier estimation methods.

Since all our cost estimation techniques pointed to the same approximate development cost of creating the platform for free, we created a Resource-Cost Matrix to summarize our findings.

This matrix highlights each planned resource, its associated cost, and a contingency cost allocated for emergencies related to that specific resource.

Resource Name	Description	Cost	Remarks
PostgreSQL Database	Database Management System	Free	Open-source
Django Framework	Backend Development Framework and Local Server Provider for Prototype Hosting	Free	Open-source
Visual Studio Code	IDE for Frontend and Backend coding	Free	Open-source
Figma	UI/UX Design tool	Free	Free student license
AJAX APIs	APIs for Dynamic Web Functionality	Free	Open-source APIs
Google Docs	Collaborative Document Creation Software	Free	Open-source
Canva	Collaborative Design and Presentation Software	Free	Free student license
Printing	Printed Documentation and Reports	Rs. 500-750	Requires funding
Contingency	Emergency Expenses	Rs. 1000 - 2000	For unforeseen issues with resources

Table 1: Resource-Cost Matrix

Software Development Scheduling

To ensure that all tasks are completed within the defined project timeline, a detailed software development schedule was created in the form of a Gantt Chart. This chart outlines the sequence of activities, activity relationships and dependencies, project member responsibilities as well as project milestones. With the help of the clear structure provided by the software development schedule, the team was able to meet the project deadline.

The project development begins with a planning phase, consisting of a feasibility study to outline the project objectives, estimate costs and create a schedule outlining the different phases of the development lifecycle. A requirement analysis is conducted after this study to determine the precise features that the platform must have. The database schema is created with the requirements in mind, and concurrently, the platform's web pages are modeled and prototyped using Figma. An overall system level diagram is also created to define workflow. The web pages are then created using HTML and CSS on the front end and Django on the back end to code and implement this design. To guarantee a seamless user experience, the two components are integrated using AJAX APIs after being tested independently. The integrated platform undergoes thorough testing before the project transitions into the documentation phase, marking the final stage of the development lifecycle.

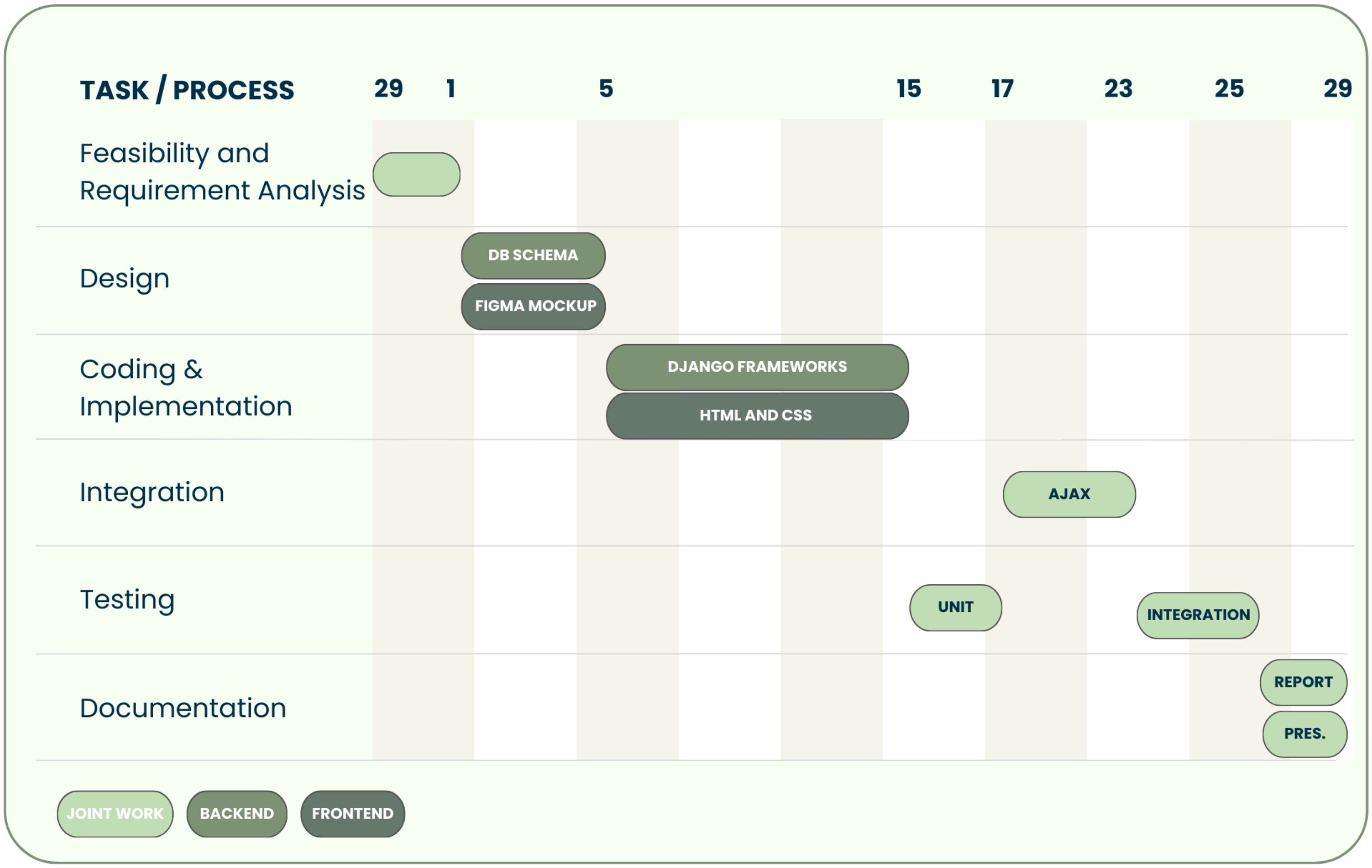


Figure 2: Software Development Gantt Chart

REQUIREMENTS

Introduction

This Software Requirements Specification (SRS) document aims to provide a comprehensive outline of the system's requirements for successful development of the proposed e-commerce platform. This document details both functional and non-functional requirements, ensuring the platform meets the needs of its users as well as the Teaa Cafe's operational objectives.

The SRS serves as a foundation for planning and managing the project, providing the development team with the necessary information to establish deadlines, allocate resources effectively, and set milestones. By clearly specifying requirements and proposed functionalities, this document supports the team in developing an effective user-friendly platform.

1. **Product Scope**
   As established in the feasibility study, the proposed platform aims to enhance the customer experience at Teaa Cafe by allowing users to browse the menu, customize their orders, and securely process payments online. The project focuses on streamlining backend operations like funds management and order processing through database integration while also providing functionalities, like account creation and order customization, through a user-friendly interface.

   This platform is designed as a prototype, hosted on a locally managed server, to demonstrate functionality before a full-scale launch. The project scope includes both frontend development, with HTML and CSS for website design, and backend development, utilizing Django and PostgreSQL for database integration. It will also include integration testing to ensure smooth operation of the system's features.The product's primary objective is to improve customer convenience, reduce operational inefficiencies, and ultimately increase customer reach for Teaa Cafe.

2. **Product Value**
   The e-commerce platform will provide significant value to both customers and Teaa Cafe management. Customers will benefit from the convenience of browsing the menu, placing orders, and securely processing payments from the comfort of their homes or on the go. The platform simplifies operations for the cafe’s management by automating the order tracking process. The integration of the backend database will not only ensure accurate record-keeping but reduce manual labor. Expanding the cafe's presence to

online customers allows the platform to draw in a wider audience, thereby enhancing sales and increasing customer engagement.

3. **Intended Audience**
   The intended audience for this e-commerce platform includes both existing and potential customers of Teaa Café, as well as the café's management team. It caters to individuals who value convenience, prefer online ordering, and appreciate the flexibility of customizing their orders.

4. **Intended Use**
   Customers will use the platform to browse the menu, place and customize orders, and securely process payments online. The café's management team will use the platform to efficiently receive orders and manage finances, streamlining operational tasks and enhancing overall efficiency.

5. **General Description**
   The e-commerce platform for Teaa Café will perform essential functions to enhance both customer experience and operational efficiency.

   Key features to be included:

   1. **User Accounts:** Allows users to create profiles and manage accounts.
   2. **Menu Browsing:** Provides a visually appealing and easy-to-navigate menu display.
   3. **Order Customization:** Enables users to customize orders and add items to their shopping cart.
   4. **Cart Management:** Facilitates adding, removing, and reviewing items in the shopping cart.
   5. **Secure Payment Gateway:** Ensures secure login after order placement.
   6. **Order Tracking:** Includes a backend system to track and update the status of placed orders.
   7. **Funds Management**: Calculates the total order invoice prices for each transaction and makes it accessible in the backend system to be used for financial and accounting purposes

   This software will integrate these features to create an interactive platform for customers while automating backend processes for the café's management team.

Functional Requirements

1. **User Authentication**
   - **Feature 1.1: User Account Creation**
     Definition: The system must allow users to create an account by providing their email, username, and password.

Inputs: The user provides their full name, their desired username, a secure password, their telephone number, their email address and their address.
Outputs: Success or error messages are displayed on the sign-up page to inform the user about the status of their registration. Error messages may include warnings about usernames or emails already in use.
Other Required Entities: The user database is required to store user credentials as well as perform the different registration status checks.
Pre-Conditions: The user has not already registered with the provided username and email address, has provided valid input and filled all the the fields in the sign-up form.
Post-Conditions: A new user record is created in the User Database, storing their email, username, and encrypted password. The user is shown different success or error messages and in case of success, the order is processed and the user is redirected to the invoice page to view their completed order.
Side-Effects: Consecutive new entries in the user database could provide the café management with insights into user demand, potentially encouraging them to expand their in-person facilities to meet increased customer interest.

- **Feature 1.2: User Login**
  Definition: The system must authenticate users by verifying their credentials during login. If the credentials are valid, users are allowed to complete the checkout process.
  Inputs: The user inputs their username and password into the login form.
  Outputs: If the credentials are invalid, an error message is displayed, indicating that the login attempt was unsuccessful.
  Other Required Entities: The user database is required to authenticate user credentials.
  Pre-Conditions: The user must already have a registered account with valid credentials. The credentials entered must be correctly formatted when entered to match those stored in the database.
  Post-Conditions: If the credentials are valid, the user is granted access and redirected to the invoice page. In the case of incorrectly entered usernames or passwords, error messages are displayed.
  Side-Effects:

2. Ordering System

- **Feature 2.1: Menu Browsing**
  Definition: The system must allow users to browse the available menu items and view their details such as name, description, and price.
  Inputs: The user selects the "Menu" section on the website.
  Outputs: The system displays a list of available menu items with relevant details for each item, including the name, description, and price.
  Other Required Entities: The system requires access to a static HTML file where menu items are defined.

Pre-Conditions: The web page must be designed to be user-friendly using Figma, implemented in HTML, and stylized with CSS.
Post-Conditions: Displayed menu items can be added to the user's shopping cart to customize their order.
Side-Effects: Changes to the menu may cause the HTML file to be updated so that users can browse the updated menu.

- **Feature 2.2: Order Customization**
  Definition: Users may add the displayed menu items to their cart to comprise their order.
  Inputs: The "Add to Cart" button is clicked by the user to add a particular item to their order.
  Outputs: Upon adding an item to their order, a brief success message is displayed at the bottom of the screen.
  Other Required Entities: The system requires access to a static HTML file where menu items are defined as well as the JavaScript function with the 'addToCart' functionality.
  Pre-Conditions: The web page must be designed to be user-friendly using Figma, implemented in HTML, and stylized with CSS and the "Add to Cart" button must be linked to an 'addToCart' functionality.
  Post-Conditions: The chosen items are saved in the user's cart.
  Side-Effects: Frequently added and popular added items give the cafe management an idea about how inventory is to be managed.

- **Feature 2.3: Shopping Cart**
  Definition: The system must allow users to add, remove, and review items in their shopping cart. The cart must display an updated total price based on the selected items.
  Inputs: Users can remove items by clicking the "Remove" button next to each product, and adjust the quantity by modifying the number in the input field for each item.
  Outputs: The cart dynamically updates by adding new rows for each item and removing rows for deleted items, with the total price and individual item prices calculated and updated automatically.
  Other Required Entities: The system requires access to the cartItems data structure defined in JavaScript and the Product table in the database to retrieve item prices for calculating the total cart price.
  Pre-Conditions: User has added at least one item to the cart.
  Post-Conditions: The shopping cart displays an updated list of items, including the names, prices, quantities, and total price.
  Side-Effects: Frequent updates to the cart could give insights into popular items, helping the management plan promotions or inventory adjustments.

- **Feature 2.4: Checkout Process**

Definition: Users must be able to proceed with checkout and the purchase of their completed cart.
Inputs: Clicking the 'Purchase' button initiates the checkout process.
Outputs: A popup displays the completed cart details, including items, quantities, and total price.
Other Required Entities: Access to the cartItems data structure and the database are required to process the order
Pre-Conditions: The cart must be filled with items, and the user must be logged in to proceed with the checkout.
Post-Conditions: The user is redirected to the invoice page, and the order is added to the database.
Side-Effects: The company receives payment for the order.

3. **Order Management**
   Definition: Once an order has been placed, the details of the order must be stored in the backend database.
   Inputs: The finalized cart with completed items and total price.
   Outputs: The order details are stored in the backend database
   Other Required Entities: Access to the cart structure and the database is required.
   Pre-Conditions: The cart is not empty, and the user has successfully completed the purchase process.
   Post-Conditions: Entries are added to both the order and order item tables in the database.
   Side-Effects: The system automatically generates a unique order number for each placed order, which can be used by the café management for tracking and managing orders.

4. **Invoice Generation**
   Definition: The system must automatically generate an invoice after an order is placed, based on the finalized cart details and product pricing from the database.
   Inputs: The finalized order, including cart details and product information.
   Outputs: An invoice is generated and stored in the Invoice table and its details are displayed to the user via the Invoice page.
   Other Required Entities: The product table for pricing, the order table for order details, the order item table for product quantities and the Invoice web page to display details on.
   Pre-Conditions: An order must be placed before proceeding with invoice generation.
   Post-Conditions: A new entry is created in the Invoice table, capturing all relevant details including customer information, product data, quantities, and total price.
   Side-Effects: The generated invoices may assist in tracking revenue, facilitating financial reporting and managing cafe funds.

5. **Location-Based Cafe Search**
   Definition: The system must allow users to search for nearby cafes in their location.
   Inputs: The user's inputted search location.

Outputs: If the location is found, the platform redirects the user to the part of the web page displaying branch details. If no match is found, an appropriate error message is displayed.
Other Required Entities: The location table in the database for storing cafe locations, and a locations web page to display branch details when a match is found.
Pre-Conditions: The location table must be populated with valid location entries corresponding to cafe branches.
Post-Conditions: The user is redirected to if a location match is found, or an error message is displayed if the search location is does not exist in the database.
Side-Effects: This location functionality will potentially draw in new customers and increase business.

Non-Functional Requirements

1. **Availability**
   TAG: Availability.FunctionalityDuringDemonstration
   AMBITION: Ensure the platform remains operational during testing and presentations
   SCALE: The percentage of uninterrupted functionality during a 1-hour demonstration or testing session
   METER: Logs of system behavior during 10 testing sessions
   MUST: At least 90% uptime during testing sessions, with no more than one restart
   PLAN: At least 95% uninterrupted performance during demonstrations
   WISH: 100% uninterrupted functionality throughout testing and presentations

2. **Efficiency**
   TAG: Efficiency.ResponseTime
   AMBITION: Ensure the platform processes user actions and database queries efficiently in order to provide a smooth user experience
   SCALE: The elapsed time in seconds for processing user actions, such as adding an item to the cart, updating quantities, or checking out
   METER: Timestamp-based testing for 50 user interaction test cases
   MUST: An average completion time of 5 seconds
   PLAN: An average completion time of 3 seconds
   WISH: An average completion time of 1 second

3. **Flexibility**
   TAG: Flexibility.MenuUpdates
   AMBITION: The system must allow for updates to the menu, including adding, removing, or modifying product listings.
   SCALE: Time and effort required to update the menu by adding new products to both the HTML file and the database.
   METER: Testing with five product updates to ensure changes are reflected both in the HTML file and the database, without errors.

MUST: Menu updates must involve adding new products to both the HTML file for front-end display and the product database.
PLAN: Product updates should take no more than 5 minutes for a user familiar with HTML and database management.
WISH: Updates should be easy enough that team members with minimal technical knowledge can add products to both the HTML and database.

4. **Integrity**
TAG: Integrity.UserDataSecurity
AMBITION: Ensure user data is accurate, consistent, and secure throughout the system.
SCALE: Percentage of user data integrity, measured by audit and validation checks.
METER: Perform routine audits and validation checks on stored user data to ensure accuracy and security.
MUST: At least 95% of user data must be accurate, consistent, and secure, with no unauthorized access or data corruption.
PLAN: 98% of user data must be accurate, consistent, and secure, with no unauthorized access or data corruption.
WISH: 100% of user data should be accurate, consistent and secure, being validated in real-time with automated checks to ensure complete integrity.

5. **Interoperability**
TAG: Interoperability.DatabaseIntegration
AMBITION: Ensure smooth integration between the eCommerce platform and PostgreSQL
SCALE: Time taken to retrieve or update data
METER: Stopwatch testing on 100 queries like placing an order, signing up users or validating user passwords
MUST: Queries must complete within 5 seconds
PLAN: Queries should ideally finish in 2 seconds
WISH: Queries should complete in under 1 second for near-instantaneous database access.

6. **Reliability**
TAG: Reliability.OrderProcessing
AMBITION: Ensure the system can process and store orders in the database reliably without any errors
SCALE: Number of orders successfully placed without errors or inconsistent data storage
METER: Measure the success rate of order placements over 10 test orders
MUST: The system must allow at least 95% of orders to be processed without errors or data inconsistencies.
PLAN: The system should aim for processing at least 98% of orders without errors or data inconsistencies.

WISH: The system should handle 99% of orders successfully.

7. **Robustness**
   TAG: Robustness.UserSignupValidation
   AMBITION: Handle invalid user signup inputs (e.g., incomplete forms, invalid usernames, or used emails) without errors.
   SCALE: Percentage of invalid inputs correctly flagged with clear error messages.
   METER: Measure system handling of 10 signup attempts
   MUST: Prevent submissions of incomplete and invalid forms.
   PLAN: Display an error message for 100% of duplicate entries.
   WISH: Display custom, targeted error messages for 100% of duplicate entries.

8. **Usability**
   TAG: Usability.UserInterface
   AMBITION: Ensure the platform is easy to use and navigate for users
   SCALE: User satisfaction rate based on feedback regarding platform functionalities
   METER: Conduct user testing with 5 participants and receive feedback based on their experience
   MUST: Allow users to complete key tasks after receiving initial guidance.
   PLAN: Ensure 90% of users can complete core actions independently after guidance.
   WISH: Ensure 100% of users can complete core actions without guidance or frustration.

Domain Requirements

1. **Security**
   TAG: Security.UserCredentials
   AMBITION: Ensure user credentials are securely stored and comply with privacy policies.
   SCALE: Number of security incidents or breaches related to user credentials.
   METER: Track unauthorized login attempts and breaches.
   MUST: User passwords must be encrypted and securely stored, with less than 5% unauthorized login attempts.
   PLAN: Use CSRF tokens for encryption and protection against cross-site request forgery, aiming for less than 3% unauthorized login attempts.
   WISH: Enable two-factor or biometric authentication, with less than 1% unauthorized login attempts.

2. **Taxes**
   TAG: Taxes.ProductPricing
   AMBITION: Ensure that all product prices are inclusive of applicable taxes.
   SCALE: Time taken to update prices to reflect tax rate changes.
   METER: Track the number of hours taken to update product prices after a tax rate change.
   MUST: Prices must be updated within 24 hours of a tax rate change.

PLAN: Prices must be updated within 12 hours of a tax rate change.
WISH: Prices must be updated within 6 hours of a tax rate change.

3. **Audit**
   TAG: Security.OrderStorage
   AMBITION: Ensure that each order is stored in the backend with a unique identifier and timestamp for audit and reporting purposes.
   SCALE: Number of orders stored with a unique order number and timestamp.
   METER: Monitor the number of orders successfully stored with a unique order number and timestamp, and track the time taken for each order to be saved.
   MUST: Each order must be stored with a unique order number and timestamp within 5 minutes of being placed.
   PLAN: 90% of orders should be stored with a unique order number and timestamp within 3 minutes of being placed.
   WISH: 100% of orders should be stored with a unique order number and timestamp within 2 minutes of being placed.

4. **Out of Stock Item Management**
   TAG: MenuManagement.ItemAvailability
   AMBITION: Ensure that out-of-stock items are promptly removed from the menu.
   SCALE: Time taken to update the menu when an item goes out of stock.
   METER: Track the number of hours taken to remove out-of-stock items from the menu after the stock status changes.
   MUST: Out-of-stock items must be removed from the menu within 3 hours of being flagged as unavailable.
   PLAN: Out-of-stock items must be removed from the menu within 1 hour of being flagged as unavailable.
   WISH: Out-of-stock items must be removed from the menu within 10 minutes of being flagged as unavailable.

5. **Pricing Reviews**
   TAG: PricingReview.PriceUpdate
   AMBITION: Ensure that product prices are actively reviewed and updated based on economic conditions.
   SCALE: Time taken to update product pricing after a pricing review meeting.
   METER: Track the number of hours taken to update product prices after a pricing review meeting.
   MUST: Prices must be reviewed and updated within 72 hours of a pricing review meeting.
   PLAN: Prices must be reviewed and updated within 48 hours of a pricing review meeting.
   WISH: Prices must be reviewed and updated within 24 hours of a pricing review meeting.

DESIGN

Introduction

The design phase of this project focuses on converting the requirements outlined in the SRS document, into a structured, organized design format that is suitable for the implementation and coding of the desired platform. The design focuses on converting the various entities and components of the e-commerce platform, along with their relationships and roles in different functionalities, into a format that can be easily implemented.

Data Flow Diagram

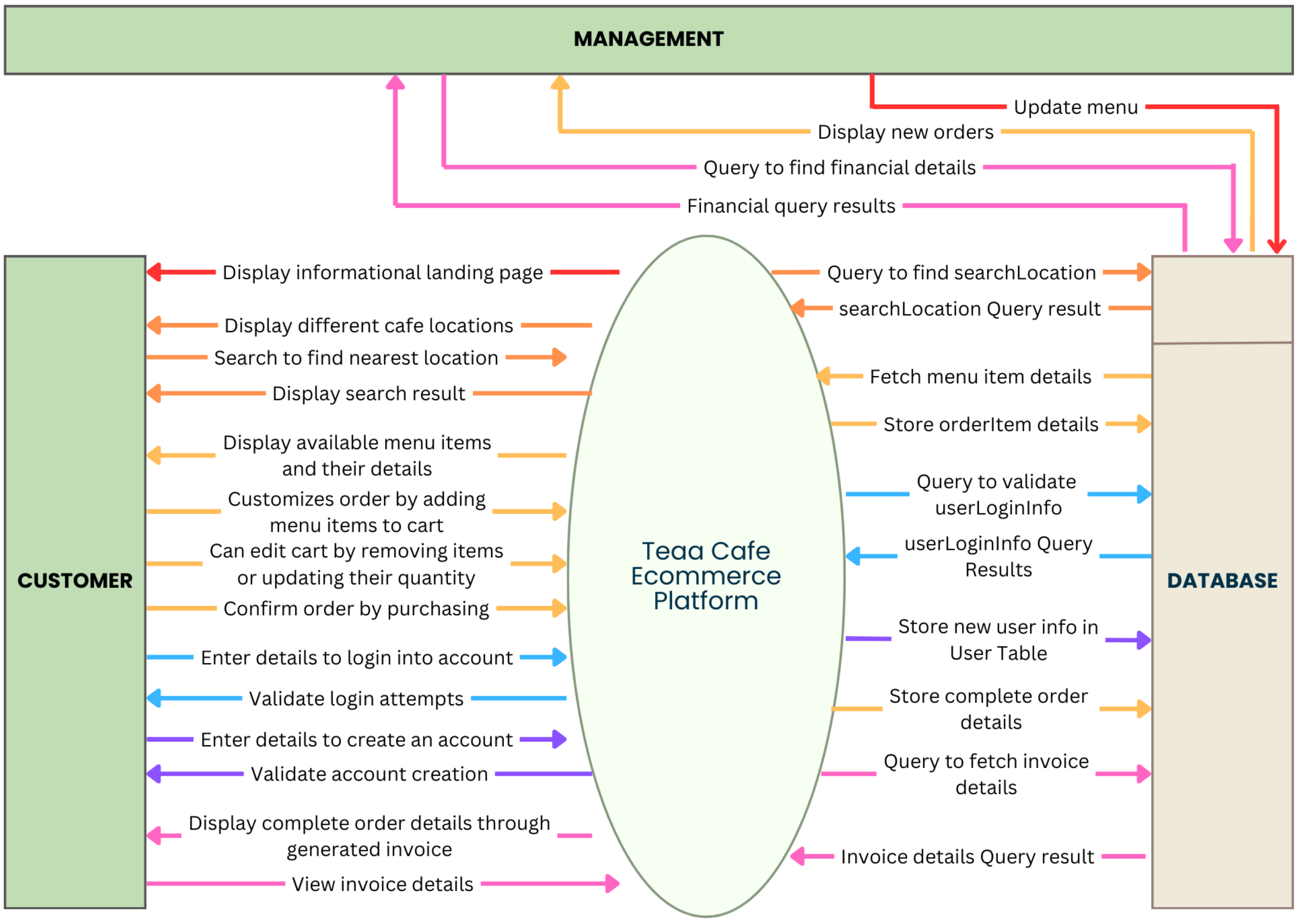

MANAGEMENT
Update menu
Display new orders
Query to find financial details
Financial query results
CUSTOMER
Display informational landing page
Display different cafe locations
Search to find nearest location
Display search result
Display available menu items and their details
Customizes order by adding menu items to cart
Can edit cart by removing items or updating their quantity
Confirm order by purchasing
Enter details to login into account
Validate login attempts
Enter details to create an account
Validate account creation
Display complete order details through generated invoice
View invoice details
Teaa Cafe Ecommerce Platform
Query to find searchLocation
searchLocation Query result
Fetch menu item details
Store orderItem details
Query to validate userLoginInfo
userLoginInfo Query Results
Store new user info in User Table
Store complete order details
Query to fetch invoice details
Invoice details Query result
DATABASE

Figure 3: Data Flow Diagram

Entity Relationship Diagram

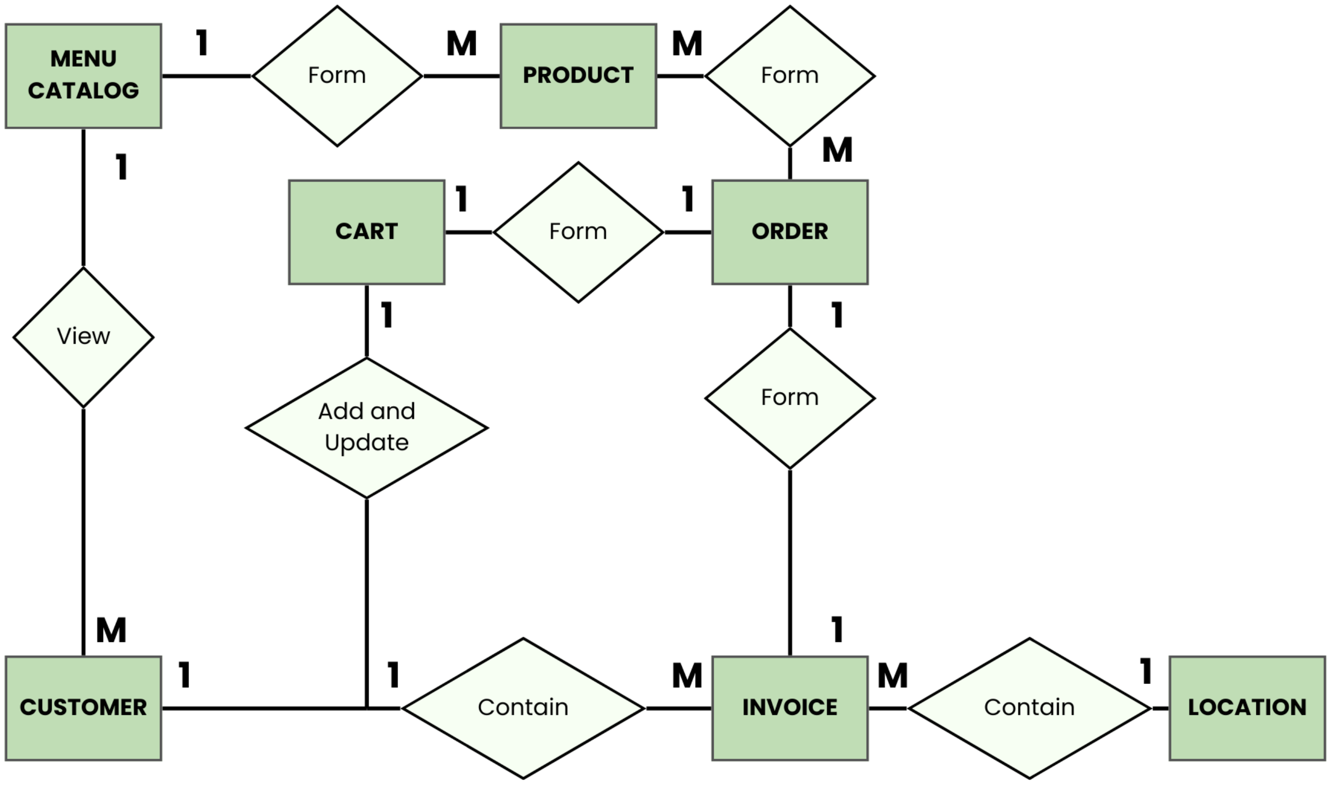


Figure 4: Entity Relationship Diagram

Language Matrix

Category	Language	Purpose
Frontend	HTML	Structures the content and layout of the webpage.
	CSS	Styles the webpage, defining fonts, colors, and layout.
	Bootstrap	Frontend framework for responsive design
	JavaScript	Implements frontend functionality, such as form handling, taking inputs, animating buttons
	AJAX	Allows asynchronous communication between the browser and the server for dynamic page updates without reloading.

Backend	Python (Django)	Backend business logic, implementing the Model-View-Controller (MVC) pattern, processing user requests, and interacting with the database using Django's Object-Relational Mapper (ORM)
	SQL	Language for managing and querying data in the PostgreSQL database

Environment Matrix

Category	Environment	Purpose
Frontend	Pinterest	Used for inspiration and gathering ideas for website design and graphics
	Figma	Used for designing web pages
	Visual Studio	IDE used for coding in HTML, CSS, JavaScript, implementing Bootstrap for responsive design and implementing AJAX for dynamic features
Backend	Visual Studio	IDE used for coding Django (Python) framework for backend logic and database interactions
	PostgreSQL	Database management system used to store and manage project data
Learning & Debugging	YouTube Tutorials	Used as a learning resource for complex backend functionalities and integration
	ChatGPT	Used for debugging, troubleshooting, and fixing errors in code.
Documentation	Google Docs	Used for collaborating with the team on the project report
Diagrams	Canva	Used to create design diagrams, slides and visual content for the project.

TESTING

Introduction

The testing phase of this project focuses on ensuring that the functionalities of the different web pages perform as expected. To achieve this, we apply both white-box and black-box testing techniques. These methods help test not only the logical flow of the web pages but also ensure that all possible input values, including edge cases and corner cases, are thoroughly examined. This approach ensures a smooth and reliable user experience.

Home Page

The main purpose of the platform's home page is to act as a landing page and allow users to navigate between different sections of the site. However, it also has a hidden backend function: upon successful login, the page services a HTTP POST request and links order details (such as product information and quantities) to the logged-in user. This link is used to save necessary order information in the database, against the correct username. While this order-saving functionality is not immediately visible to users, it is crucial for enabling smooth transitions in the checkout process.

- Black Box Testing
    a. Equivalence Class Partitioning
        i. Valid Username and Purchase Details
            - Equivalence Class:
                - A valid username existing in the database
                - Purchase details that include positive quantities and prices, and non-empty data.
            - Expected Behavior:
                - The order should be successfully saved by discreetly redirecting to the homepage.

        ii. Invalid Username or Purchase Details
            - Equivalence Class:
                - Invalid username that does not exist in the database.
                - Invalid purchase details, that may include negative quantities, negative pricing, or non-numeric values in quantity or price.
            - Expected Behavior:
                - The cart should not be processed or saved if any invalid input is detected.
                - There should be no redirections to the homepage as the order shouldn't be saved.
                - Relevant error messages are shown.

iii. Empty Username or Purchase Details
- Equivalence Class:
  - Missing username or purchase details
- Expected Behavior:
  - The system prevents the user from submitting empty data and prompts them to re-enter details.
  - The order should not be saved, and an appropriate validation message should be shown

b. Boundary Value Analysis

i. Username Boundary Tests:
- Empty Username: This checks the lower boundary where the username field is completely empty.
- Valid Username: This checks the range where usernames are valid.
- Maximum Length Username: This tests the upper boundary where the username is at its maximum allowed length of 150 characters.

ii. Quantity and Price Boundary Tests:
- Zero Quantity: This tests the lower boundary of quantity.
- Negative Quantity: This tests values below the valid range.
- Maximum Quantity: This tests the upper boundary where the quantity approaches its maximum integer value.(2147483647)
- Minimum Price: This tests the lowest valid price (e.g. Rs 000000000.01).
- Maximum Price: This tests the upper boundary for prices(99999999.99).
- Zero Price: This tests below the valid range for price.

- White Box Testing

```
def home(request):

    if request.method == 'POST':
        username = request.POST.get('username')
        purchase_details = request.POST.getlist ('purchaseDetails')

        user = CustomUser.objects.get(username=username)
        # Create an order for the user
        #django handles fk relatioship itself, userid not needed, date ordered is to current time when
create is called
        order = Order.objects.create(customer=user, total_price=0)

        # Process and save the purchase details here
        for index, detail_str in enumerate(purchase_details):
            # Now you have the purchase detail as a dictionary, you can access its individual properties
            detail_dict = json.loads(detail_str)
            #jason load converts json string to dictionary
            product_name = detail_dict['productName']
            quantity = detail_dict['quantity']
            price = detail_dict['price']
            total = detail_dict['total']

            # Retrieve the Product object corresponding to the product name
            product = Product.objects.get(name=product_name)

            # Determine if this is the last item
            is_last_item = index == len(purchase_details) - 1
            # Create an order item for the current product
            order_item = OrderItem.objects.create(order=order, product=product,
quantity=quantity, price=total, last_item=is_last_item)

            # Update the total price of the order
            order.total_price += order_item.price * order_item.quantity
            order.save()

        # Return a JSON response indicating success
        return JsonResponse({'message': 'Order successfully placed!', 'order_id': order.id})

    else:
        return render(request, 'home.html')
```

a. Statement Coverage
   Statement coverage ensures that every line of code is executed at least once. It considers if-else as one statement so to cover both the if and else blocks, we run two separate test cases.

   Test Case = (Post Request with Valid Data) OR (GET Request)

   i. POST Request with Valid Data
      - Input: A valid username and correctly formatted purchase details.
      - Expected Behavior: The function will enter the if block, process the purchase details, save the order, and return a success message.

   ii. GET Request
      - Input: A non-POST request.
      - Expected Behavior: The function will enter the else block and render the home.html page.

b. Branch Coverage
   Branch coverage ensures that all branch points are tested for both true and false outcomes. For this, we evaluate both the if and else conditions so that both conditions are tested.

   Test Case = (Post Request with Valid Data) AND (GET Request)

   i. POST Request with Valid Data
      - Input: A valid username and correctly formatted purchase details.
      - Expected Behavior: The function will enter the if block, process the purchase details, save the order, and return a success message.

   ii. GET Request
      - Input: A non-POST request.
      - Expected Behavior: The function will enter the else block and render the home.html page.

c. Path Coverage
   Path coverage ensures that all possible paths through the code are tested, considering all combinations of decisions and loops.

   Test Case = (Post Request with Valid Data) AND (GET Request)

i. POST Request with Valid Data
   - Input: A valid username and correctly formatted purchase details.
   - Expected Behavior: The function will enter the if block, process the purchase details, save the order, and return a success message.

ii. GET Request
   - Input: A non-POST request.
   - Expected Behavior: The function will enter the else block and render the home.html page.

About Page

The About Page provides basic information about Teaa Cafe. It is a simple page that displays static content and is rendered when accessed, without any complex backend functionality. Therefore, black box testing was not necessary as there were no boundaries or equivalence classes to analyze. For white box testing and testing the different coverages, we simply verify that 'about.html' is being rendered correctly.

```
def about(request):
    return render(request, 'about.html')
```

Order Page

The order view in Django retrieves and displays a list of products on the order.html page. It allows users to add products to their shopping cart, update quantities, remove items, and view the updated total price. Once a user is done customizing their order, they may choose to purchase their cart.

- Black Box Testing
  a. Equivalence Class Partitioning
     i. Valid Purchase Details
        - Equivalence Class:
          - Purchase details that include positive quantities and prices, and non-empty data.
        - Expected Behavior:
          - The user will be redirected to the login page to complete their purchase.

     ii. Invalid Purchase Details
        - Equivalence Class:
          - Invalid purchase details, that may include negative quantities, negative pricing, or non-numeric values in quantity or price.
        - Expected Behavior:
          - The cart purchase should not be processed if any invalid input is detected.
          - Relevant error messages are shown.
          - The user will not be redirected to the login page.

iii. Empty Purchase Details
- Equivalence Class:
  - Missing purchase details
- Expected Behavior:
  - The cart purchase should not be processed if it is currently empty.
  - Relevant error messages are shown.

b. Boundary Value Analysis

i. Quantity Boundary Tests:
- Zero Quantity: This tests the lower boundary of quantity.
- Negative Quantity: This tests values below the valid range.
- Maximum Quantity: This tests the upper boundary where the quantity approaches its maximum integer value.(2147483647)
- Valid Quantity: This tests the valid quantity range.

ii. Price Boundary Tests:
- Minimum Price: This tests the lowest valid price (e.g. Rs 000000000.01).
- Maximum Price: This tests the upper boundary for prices(99999999.99).
- Zero Price: This tests below the valid range for price.
- Valid Price: This tests the valid price range.

- White Box Testing

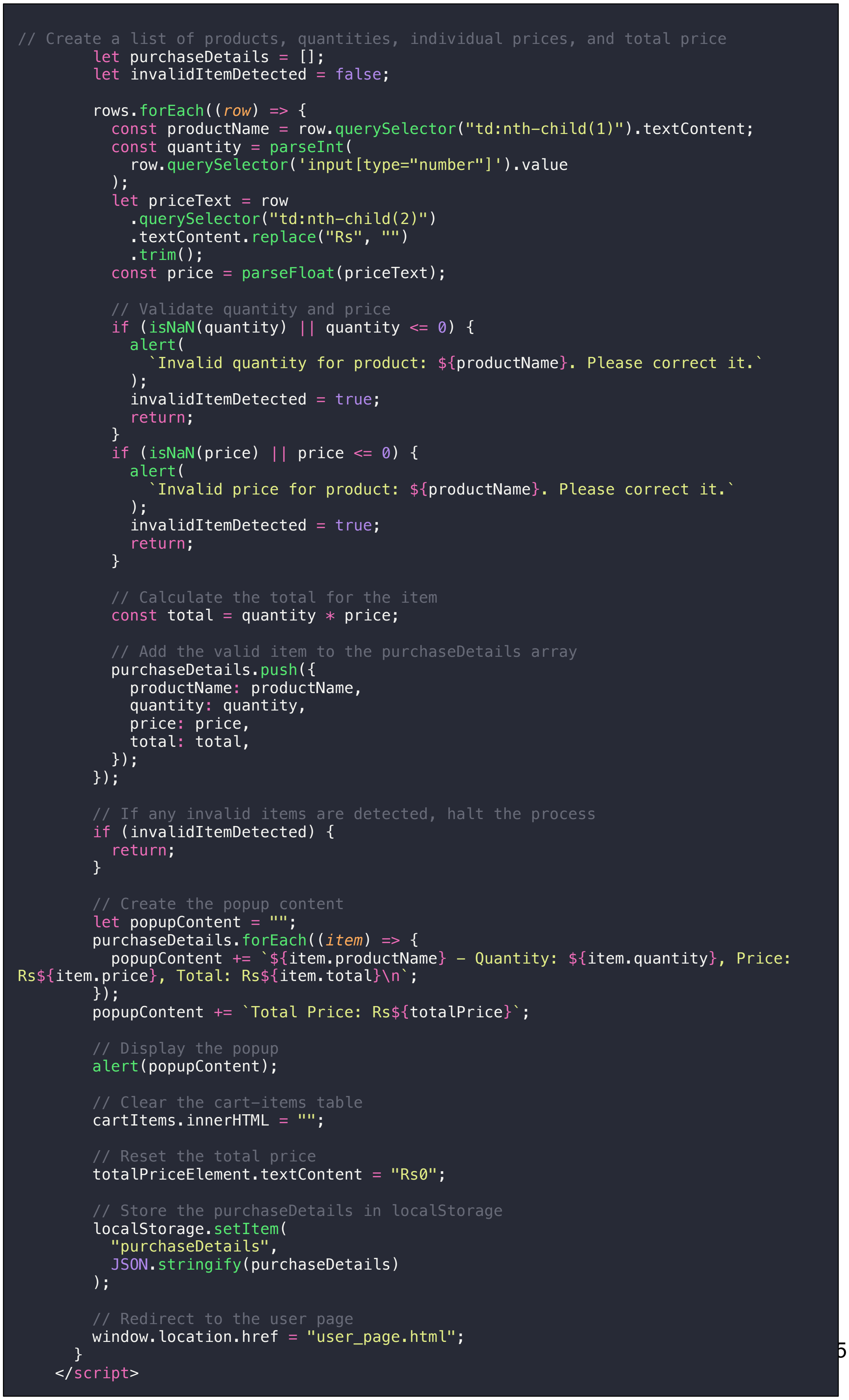
  For white box testing, we simply need to verify that ‘order.html’ is being rendered correctly with the product information fetched from the Product table in the platform’s database. Each instance of the invalid purchase details and empty purchase details classes have their own if blocks, catering to these anomalies. Therefore, by black box testing we inadvertently have already performed statement, path and branch coverage.

```
def order(request):
        products = Product.objects.all().values('name',
'price').order_by('id')
        # Handle the case when the form is not submitted via POST
        return render(request, 'order.html', {'products': products})
```

```
      function showPurchasePopup() {
        // Get the cart items
        const cartItems = document.querySelector(".cart-items");

        // Get the total price
        const totalPriceElement = document.querySelector(".cart-total-
price");
        let totalPrice = totalPriceElement.textContent.replace("Rs",
"").trim();
        totalPrice = parseFloat(totalPrice) || 0;

        // Check if the cart is empty
        const rows = cartItems.querySelectorAll("tr");
        if (!rows.length) {
          alert("Cart is empty. Please add items before proceeding.");
          return;
        }
```



```javascript
// Create a list of products, quantities, individual prices, and total price
        let purchaseDetails = [];
        let invalidItemDetected = false;

        rows.forEach((row) => {
          const productName = row.querySelector("td:nth-child(1)").textContent;
          const quantity = parseInt(
            row.querySelector('input[type="number"]').value
          );
          let priceText = row
            .querySelector("td:nth-child(2)")
            .textContent.replace("Rs", "")
            .trim();
          const price = parseFloat(priceText);

          // Validate quantity and price
          if (isNaN(quantity) || quantity <= 0) {
            alert(
              `Invalid quantity for product: ${productName}. Please correct it.`
            );
            invalidItemDetected = true;
            return;
          }
          if (isNaN(price) || price <= 0) {
            alert(
              `Invalid price for product: ${productName}. Please correct it.`
            );
            invalidItemDetected = true;
            return;
          }

          // Calculate the total for the item
          const total = quantity * price;

          // Add the valid item to the purchaseDetails array
          purchaseDetails.push({
            productName: productName,
            quantity: quantity,
            price: price,
            total: total,
          });
        });

        // If any invalid items are detected, halt the process
        if (invalidItemDetected) {
          return;
        }

        // Create the popup content
        let popupContent = "";
        purchaseDetails.forEach((item) => {
          popupContent += `${item.productName} - Quantity: ${item.quantity}, Price:
Rs${item.price}, Total: Rs${item.total}\n`;
        });
        popupContent += `Total Price: Rs${totalPrice}`;

        // Display the popup
        alert(popupContent);

        // Clear the cart-items table
        cartItems.innerHTML = "";

        // Reset the total price
        totalPriceElement.textContent = "Rs0";

        // Store the purchaseDetails in localStorage
        localStorage.setItem(
          "purchaseDetails",
          JSON.stringify(purchaseDetails)
        );

        // Redirect to the user page
        window.location.href = "user_page.html";
      }
    </script>
```

Location Page

The location page allows users to search for the nearest cafe location. If the location is found, the page will automatically scroll to cafe branches in that location. However, if the location is not found, an error message will briefly appear.

- Black Box Testing
  a. Equivalence Class Partitioning
     i. Valid Location
        - Equivalence Class:
          - A location corresponding to an existing Teaa Cafe branch.
        - Expected Behavior:
          - The page will automatically scroll to the web page section corresponding to the inputted location's branch.
     ii. Invalid Location
        - Equivalence Class:
          - Locations that do not exist in the platform's database.
        - Expected Behavior:
          - A temporary popup is displayed informing users that a branch in that location does not yet exist.
  b. Boundary Value Analysis
        - Proper Case Format: This tests the valid location.
        - Mixed Case Format: This tests a corner case of the valid locations, ensuring that both lowercase and uppercase characters are being considered.
        - Maximum Length: This tests the upper boundary of the input length i.e
        - Above Maximum Length: This tests outside the upper boundary of the input length i.e: 100 characters.
        - Empty Input: This tests outside the lower boundary of the input length i.e: 100 characters.
        - Numerical Input: This tests for invalid location inputs.
- White Box Testing

```
def location(request):
    if request.method == 'POST':
        user_loc = request.POST.get('location').lower()


        if Location.objects.filter(name = user_loc).exists():
            loc = Location.objects.get (name = user_loc)
            return JsonResponse({'message': 'location found', 'loc_id':
loc.id})

        else:
            return JsonResponse({'message': 'location not found',
'loc_id': -1})

    else:
        return render (request, 'location.html')
```

a. Statement Coverage
   Statement coverage ensures that every line of code is executed at least once. It considers the different cases within the select statement as one statement. Therefore, we use test cases; one for valid locations. and the other for invalid locations.

   Test Case = POST[(“islAMAbaD”) OR (“Atlanta”)]

   i. Valid Location
      - Input: “islAMAbaD”
      - Expected Behavior: The function will convert the input location into lowercase, find a matching location in the platform database and scroll to the Islamabad branch section.
   ii. Invalid Location
      - Input: “Atlanta”
      - Expected Behavior: The function will convert the input location into lowercase, not find a matching location in the platform database and display the error message popup.

b. Branch Coverage
   Branch coverage ensures that all branch points are tested for both true and false outcomes. For this, we evaluate all cases within the select statement.

   Test Case = POST[(“Islamabad”) AND (“Lahore”) AND (“Karachi”) AND (“Atlanta”)] AND GET

   i. GET Request
      - Input: A non-POST request.
      - Expected Behavior: The function will enter the else block and render the location.html page.
   ii. Valid Location
      - Input: “Islamabad”, “Lahore”,“Karachi”
      - Expected Behavior: The function will convert the input location into lowercase, find a matching location in the platform database and scroll to the Islamabad branch section.
   iii. Invalid Location
      - Input: “Atlanta”
      - Expected Behavior: The function will convert the input location into lowercase, not find a matching location in the platform database and display the error message popup.

c. Path Coverage
   Path coverage ensures that all possible paths through the code are tested, considering all combinations of decisions and loops.

   Test Case = POST[(“Islamabad”) AND (“Lahore”) AND (“Karachi”) AND (“Atlanta”)] AND GET

   i. GET Request
      - Input: A non-POST request.

        - Expected Behavior: The function will enter the else block and render the location.html page.

    ii. Valid Location
        - Input: “Islamabad”, “Lahore”,“Karachi”
        - Expected Behavior: The function will convert the input location into lowercase, find a matching location in the platform database and scroll to the Islamabad branch section.

    iii. Invalid Location
        - Input: “Atlanta”
        - Expected Behavior: The function will convert the input location into lowercase, not find a matching location in the platform database and display the error message popup.

User Page

The user page manages user login by verifying the inputted username and password by using the platform database. The page functionality includes checking if the submitted username exists in the database and then if it does, checking that the submitted password matches the stored one. On successful authentication, it returns a success message. Otherwise, it responds with appropriate error messages.

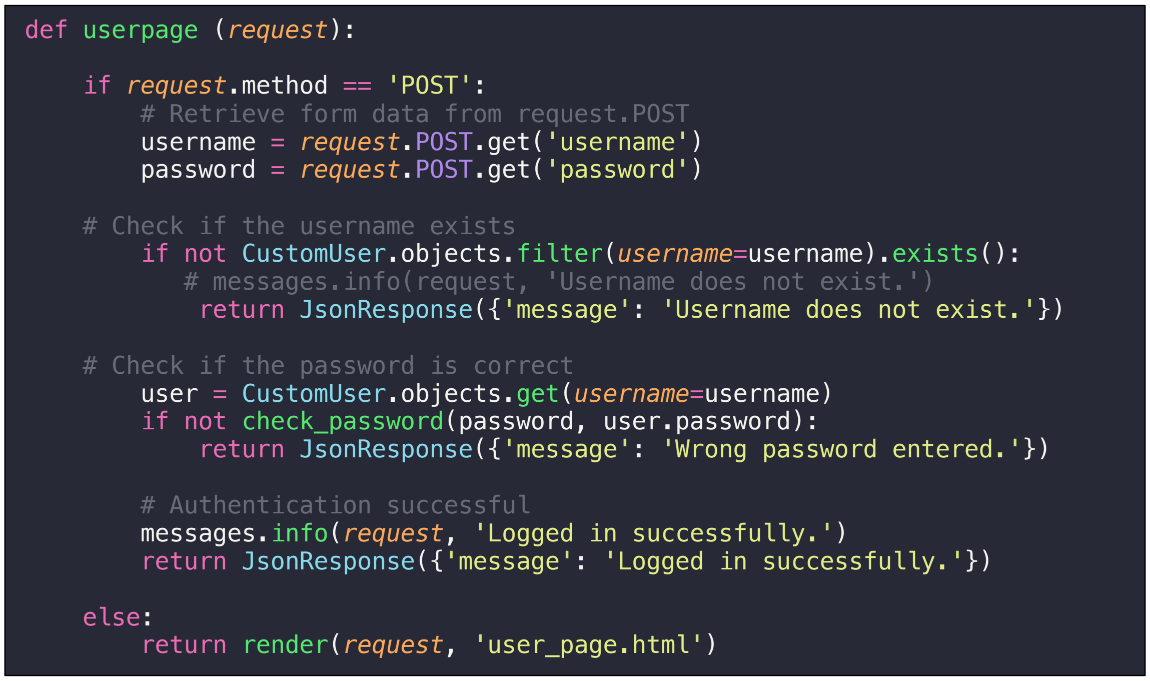
- Black Box Testing
    a. Equivalence Class Partitioning
        i. Valid Credentials
            - Equivalence Class:
                - Username that exists in the database.
                - Password matches the stored password for that username.
            - Expected Behavior:
                - Authentication is successful.
                - Purchase is completed by saving the user’s order in the database.
                - User is redirected to the invoice page.
        ii. Invalid Credentials
            - Equivalence Class:
                - Username does not exist in the database.
                - Password does not match the stored password for that username.
            - Expected Behavior:
                - Error message popup displayed informing users about the issues.
    b. Boundary Value Analysis
        i. Username Boundary Tests:
            - Empty Username: This tests the lower boundary of the username.
            - Valid Username: This tests the valid usernames existing in the database.

- Above Maximum Length: This tests the upper boundary of the username to ensure it doesn't match above maximum length usernames i.e usernames with more than 150 characters.

ii. Password Boundary Tests:
- Empty Password: This tests the lower boundary of the password.
- Valid Username: This tests the valid passwords existing in the database.
- Above Maximum Length: This tests the upper boundary of the passwords to ensure it doesn't match above maximum length passwords i.e encrypted passwords with more than 128 characters.

- White Box Testing
  All coverages will have the same test cases as each input combination corresponds to its own if blocks, which will be tested in all cases.

Test Case = (Valid Username AND Valid Password) AND (Valid Username AND Invalid Password) AND (Invalid Username)



```python
def userpage (request):

    if request.method == 'POST':
        # Retrieve form data from request.POST
        username = request.POST.get('username')
        password = request.POST.get('password')

    # Check if the username exists
        if not CustomUser.objects.filter(username=username).exists():
           # messages.info(request, 'Username does not exist.')
            return JsonResponse({'message': 'Username does not exist.'})

    # Check if the password is correct
        user = CustomUser.objects.get(username=username)
        if not check_password(password, user.password):
            return JsonResponse({'message': 'Wrong password entered.'})

        # Authentication successful
        messages.info(request, 'Logged in successfully.')
        return JsonResponse({'message': 'Logged in successfully.'})

    else:
        return render(request, 'user_page.html')
```

Signup Page

The signup page enables users to create a new account by inputting personal and login details, ensuring the uniqueness of fields like email and username. Upon successful registration, the user's information is stored in the database.

- Black Box Testing
  a. Equivalence Class Partitioning
     i. Valid Credentials
        - Equivalence Class:

        - Username does not exist in the database.
        - Email does not exist in the database.
        - Fields like password, first name, last name, street, city, house, area and telephone number are non-null, length appropriate inputs.
    - Expected Behavior:
        - User creation is successful and a new entry is made in the database.
        - Purchase is completed by saving the user's order in the database.
        - User is redirected to the invoice page.

ii. Invalid Credentials
- Equivalence Class:
    - Username exists in the database.
    - Email exists in the database.
    - Fields like password, first name, last name, street, city, house, area and telephone number are null or are not length appropriate inputs.
- Expected Behavior:
    - Error message popup displayed informing users about the issues.

b. Boundary Value Analysis

i. Username and Email Boundary Tests:
- Empty: This tests the lower boundary of the username.
- Valid: This tests that the identifier fields do not exist in the database and is length appropriate.
- Above Maximum Length:
    - This tests the upper boundary of the username with above maximum length usernames i.e. usernames with more than 150 characters.
    - This tests the upper boundary of the email address with above maximum length emails i.e. email addresses with more than 254 characters.

ii. Password, Name, Address, Boundary Tests:
- Empty: This tests the lower boundary of the credentials.
- Valid: This tests the correct credentials, ensuring that they are saved correctly. .
- Above Maximum Length: This tests the upper boundary of the credentials to ensure inappropriate length inputs are not considered.

- White Box Testing

  All coverages will have the same test cases as each input combination corresponds to its own if blocks, which will be tested in all cases.

Test Case = (Valid Credentials, Unique Email, Unique Username) AND (Valid Credentials, Non-Unique Email, Unique Username) AND (Valid Credentials, Unique Email, Non-Unique Username) AND (Invalid Credentials, Unique Email, Unique Username)

```
def signuppage (request):
    if request.method == 'POST':
        # Retrieve form data from request.POST
        firstname = request.POST.get('firstname')
        lastname = request.POST.get('lastname')
        username = request.POST.get('username')
        password = request.POST.get('password')
        telephone = request.POST.get('telephone')
        email = request.POST.get('email')
        city = request.POST.get('city')
        area = request.POST.get('area')
        street = request.POST.get('street')
        house = request.POST.get('house')

        if CustomUser.objects.filter (email = email).exists():
            #messages.info(request, 'Email Already In Use')
            return JsonResponse({'message': 'Email already in use'})


        if CustomUser.objects.filter (username = username).exists():
            #messages.info(request, 'Username Already In Use')
            return JsonResponse({'message': 'Username already in use'})


        user = CustomUser.objects.create(username = username, first_name =
firstname, last_name = lastname,
                                        password= make_password(password), email
= email, street = street, city= city,
                                        house = house, area = area,
telephone=telephone)
        user.save()
        return JsonResponse({'message': 'Account created successfully!'})

    else:
        return render(request, 'signup_page.html')
```

Invoice Page

The invoice page retrieves and displays details for a specific order based on the provided order identification number. It calculates the total price for the order, extracts customer data, and displays the information on the invoice.html page.

- Black Box Testing
    a. Equivalence Class Partitioning
        i. Valid Order ID
            - Equivalence Class:
                - An Order ID corresponding to an existing order in the platform database.
            - Expected Behavior:

        - The order's invoice will be displayed on the web page with appropriate details found from the database.
        - A popup will be displayed informing users of their successful order, thanking them for their business.

    ii. Invalid Order ID
        - Equivalence Class:
            - An Order ID that does not correspond to any existing order in the platform database.
        - Expected Behavior:
            - The order's invoice will be displayed without details as the order does not exist.

  b. Boundary Value Analysis
        - Minimum Order ID: This tests the first order saved in the database to ensure previously saved orders can still be accessed.
        - Maximum Order ID: This tests the most recent order saved in the database to ensure recently saved orders can be accessed without any issues.
        - Random Order ID: This tests to ensure invoices can be generated for all invoices successfully.
        - Empty Order ID: This tests an invalid, null Order ID to test error handling.
        - Alphabetic Order ID: This tests an invalid Order ID to test error handling.

- White Box Testing
  The invoice page view contains a single if block which will be tested by all coverage testing techniques. Even though statement coverage considers the if and else blocks as a single statement, we will create and test two test cases, making sure every line of code is properly tested. Branch and path coverage will test both the true and false conditions in one test case. The test cases for each technique are provided below:

  Test Case for Statement Coverage = (Valid Order ID) OR (Invalid Order ID)
  Test Case for Branch Coverage = (Valid Order ID) AND (Invalid Order ID)
  Test Case for Path Coverage = (Valid Order ID) AND (Invalid Order ID)

```python
def invoice(request):
    invoice_details = None
    order_id = request.GET.get('order_id')

    if order_id:
        invoice_details = Invoice.objects.filter(order_id=order_id)
        invoice_details_first = invoice_details.first()
        customer_data = {
                'first_name': invoice_details_first.first_name,
                'last_name': invoice_details_first.last_name,
                'house': invoice_details_first.house,
                'street': invoice_details_first.street,
                'area': invoice_details_first.area,
                'city': invoice_details_first.city,
                'date_ordered': invoice_details_first.date_ordered.date(),
            }
        # Calculate the total price
        total_price = sum(item.total_price for item in invoice_details)

    else:
        invoice_details = None
        customer_data = None
        total_price = 0

    return render(request, 'invoice.html', {'invoice_details': invoice_details,
'customer_data': customer_data, 'total_price': total_price})
```

CONCLUSION

The e-commerce platform for Teaa Café successfully integrates essential features like product browsing, order placement, invoice generation, and location-based branch search to provide a seamless and user-friendly experience. By combining intuitive design with robust functionality, the platform caters to the diverse needs of the users, ensuring customer satisfaction and operational efficiency.

DISCLOSURE

AI tools were utilized throughout the project for various purposes, including error handling, debugging code, serving as a reference and guide for project planning, assisting with report structuring, and ensuring proper formatting of the final documentation.