Movie Theater Ticketing System

Software Requirements Specification

Version 4.0

8/05/2025

Guled Hussein

Prepared for

CS 250- Introduction to Software Systems

Instructor: Gus Hanna, Ph.D.

Fall 2023

Revision History

Date	Description	Author	Comments
7/15/2025	Version 1.0	Guled Hussein	Specified the Requirements
7/21/2025	Version 2.0	Guled Hussein	Software Design Specified
7/29/2025	Version 3.0	Guled Hussein	Test Plan
8/4/2025	Version 4.0	Guled Hussein	Architecture Design w/ Data Management

Document Approval

The following Software Requirements Specification has been accepted and approved by the following:

Signature	Printed Name	Title	Date
	Guled Hussein	Software Eng.	
	Dr. Gus Hanna	Instructor, CS 250	

Table of Contents

Table of Figures:

1 Introduction

1.1 Purpose

This Software Requirements Specification (SRS) aims at giving a comprehensive and thorough description of the Movie Theater Ticketing System (MTTS) to be developed. It is a contractual and technical base of all stakeholders in the project. It provides a shared terminology on the behavior, functions, limitations, and interfaces of systems to guarantee congruity among developers, testers, project managers, and representatives of the clients.

The intended audience includes:

- **Software developers** responsible for implementation.
- **Quality assurance teams** verifying system conformance.
- **System architects and UI designers** ensuring adherence to requirements.
- **Theater management** overseeing operational goals and ensuring business needs are met.

The objective of this document is to reduce ambiguity, facilitate traceability and ensure that the system fulfills the user expectations and business goals.

1.2 Scope

The Movie Theater Ticketing System (MTTS) is a software product run over the web environment, which will be used to support the process of the ticket reservation, the choice of seats and make the schedule of a physical movie theater along with the reporting. Customers can use the system to navigate the online resource to see the shows and buy tickets, as well as use self-service terminals, however, containing a back-end interface that allows theater employees to control schedules, pricing, and reporting.

Included Features:

- **Online Booking**: Customers can book tickets via desktop or mobile devices 24/7.
- **Seat Selection**: Real-time visual seat map for selection with automatic availability updates.
- **Ticket Issuance**: E-tickets sent via email or printable from kiosk terminals.
- **Schedule Management**: Admins can add/edit movie listings, showtimes, and screen allocations.

- **Reporting Tools**: Generate daily and weekly attendance, sales, and occupancy reports.
- **User Authentication**: Secure registration and login system for both customers and admins.

Excluded Features (Current Version):

- streaming or digital rentals.
- Online food/concession ordering.
- Third-party ticket platform integrations (e.g., Fandango, BookMyShow).
- Loyalty programs or promotional discount systems (may be added in future versions).

System Assumptions and Dependencies:

- A reliable internet connection is assumed for all online transactions.
- The system assumes the availability of third-party payment gateway APIs (e.g., Stripe, PayPal).
- Users will access the platform via modern web browsers (Chrome, Firefox, Safari).

Primary User Classes:

- **Customers (Moviegoers)**: General users who browse listings, select seats, and purchase tickets.
- **Administrators (Theater Staff)**: Employees responsible for scheduling movies, managing prices, and generating reports.
- **Managers**: Senior personnel who analyze trends and oversee theater performance.
- The aim of the MTTS system is to mitigate bottlenecks in operations and increase customer satisfaction, and to offer end-to-end seamless ticketing services under measurable performance criteria (e.g. <5 seconds transaction time, >99% uptime).

1.3 Definitions, Acronyms, and Abbreviations

This section defines all specialized terms and acronyms used in this SRS. A more extensive glossary may be provided in Appendix A. Key definitions include:

Term / Acronym	**Definition**
MTTS	Movie Theater Ticketing System – The software application described in this SRS.

SRS	Software Requirements Specification – This document, defining system requirements.
GUI	Graphical User Interface – Visual interface through which users interact with the system.
API	Application Programming Interface – External software interfaces (e.g., for payments).
ACID	Atomicity, Consistency, Isolation, and Durability – properties ensuring data transactions consistency and reliability
Customer	End user who uses the system to book, view, or cancel movie tickets.
Administrator	Theater staff with system access to manage movies, schedules, and pricing.
Ticket	A digital or printed reservation confirmation for a movie showing.
Showtime	Scheduled date and time for a movie screening in a particular auditorium.

1.4 References

The following documents and standards have informed the structure and content of this SRS:

1. IEEE Std 830–1998, *Recommended Practice for Software Requirements Specifications*, IEEE Computer Society, 1998.
2. IEEE Std 610.12–1990, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Computer Society, 1990.
3. Sommerville, I. (2015). *Software Engineering* (10th ed.). Addison-Wesley.
4. Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill.
5. CS250 Course SRS Template, Instructor: Dr. Gus Hanna (Fall 2023).
6. Lecture-Use-Cases and Project Notes, CS250 Software Systems Course (2025).

All cited sources are accessible through course resources, university library databases, or the IEEE Xplore Digital Library. Rest of the References are provided in APPENDIX

1.5 Overview

This SRS is organized to provide a clear and systematic understanding of the Movie Theater Ticketing System:

- **Section 2 – General Description:** Provides a description of the product from the user's point of view, foremost including the functions, the user characteristics, the constraints, the dependencies.
- **Section 3 – Specific Requirements**: Details of functional and non-functional system requirements, interfaces and use cases.
- **Section 4 – Analysis Models**: This section gives the system behavior models including sequence diagrams, state diagrams and data flow diagrams.
- **Section 5 – Change Management**: Outlines the process of updating of this document based on the changing requirements.
- **Appendices**: Include such details as definitions, schemes, or an example of test data formats.

2 General Description

2.1 Product Perspective

Movie Theater Ticketing System (MTTS) is an independent software product developed to modernize and automate the ticketing reservation and scheduling process of a real physical movie house. It is not based upon legacy system and is created as new system out of scratch. MTTS is, however, supposed to be able to interface other services like payment gateways (e.g. Stripe or PayPal) and notification services (e.g. email or SMS API), to provide secure payment and e-communication with customers.

The system architecture will follow a three-tier structure:

- **Presentation Layer (Frontend)**: Web-based GUI accessible via browsers and responsive to mobile devices.
- **Application Layer (Backend)**: Business logic, scheduling algorithms, and user/session management.
- **Data Layer (Database)**: Centralized relational database managing user data, movie listings, seat inventory, and transactions.

The MTTS will operate within the theater's IT infrastructure and is expected to be hosted on a cloud-based or on-premise server depending on operational decisions.

2.2 Product Functions

The system will provide the following functions:

- **User Registration and Login**
  They are able to register accounts and log in to access their profiles, reserve with a history of their bookings, and make booking reservations.
- **Movie and Showtime Browsing**
  The customers are able to read about current and upcoming movies with their description, trailers, genre, and timings.
- **Seat Selection, Booking Payment and Ticket Generation**
  Customers are now able to select their own preferred seats via real-time seat maps. The system eliminates cases of double-booking through atomic processing of the transactions.

- There is integrated online payment options customers may use to transact. When a transaction is successful, e-ticket is created and sent via email to a user.
- **Ticket Cancellation**
  The theater policy will provide the window when the booked tickets can be canceled.
- **Admin Functions**
  Admin users can add/retract/edit movie listings, configure showtimes and theaters, pricing, and report.
- **Reporting**
  Internal generating reports are daily sales, occupancy figures, and logs on customer activities.

2.3 User Characteristics

The system will support the following categories of users, each with specific characteristics:

1. **Customers**
   - Non-technical users accessing the system via browser or mobile.
   - Require intuitive, guided user interfaces.
   - May be first-time or recurring users.
2. **Administrators**
   - Trained theater staff who manage schedules, pricing, and reports.
   - Require secure login, data entry interfaces, and reporting tools.
3. **Managers**
   - Executive users with access to high-level reports and analytics.
   - Require minimal interaction but expect summarized, visual dashboards.

The responsive design will be taken into consideration in order to make it mobile compatible, and the navigation will be simple to understand by non-expert users.

2.4 General Constraints

- The system should also adhere to data privacy laws (e.g., the GDPR or national alternatives).
- Web frontend should be based on modern browsers (Chrome, Firefox, Safari, Edge).
- All user communications have to be done through secure HTTPS links.

- Database should be compliant to ACID to keep integrity of transactions.
- The booking should be atomic to avoid race conditions (e.g. two users reserving the same seat).
- The system has to support at least 1000 concurrent users at peak times.

2.5 Assumptions and Dependencies

The assumptions and external dependencies applicable to the development and operation of MTTS include the following:

- The third-party payment processing services (e.g., Stripe, PayPal) will be accessible and usable.
- There will be access to computers and reliable internet connection to carry out backend administration by the theater staff.
- Email/SMS APIs (e.g. SendGrid, Twilio) will be added to send confirmation of tickets.
- The system shall run under a Linux based server unless where otherwise indicated.
- Clients will have rudimentary knowledge of internet platforms and online payment.

3 Specific Requirements

This section provides summary of functional and the external interface requirements of the Movie Theater Ticketing System (MTTS). Such requirements determine what the system will do and how it will communicate with the users, the existing hardware, other software systems, and the communication technologies. All the requirements are unambiguous, testable and traceable to user requirements or business aims.

3.1 External Interface Requirements

3.1.1 User Interfaces

The MTTS shall provide two main user interfaces:

1. **Customer Interface (Web/Mobile Browser):**
   The customer interface will be a responsive web application accessible via latest browsers on desktops and in mobile devices. This interface will facilitate, searching movies, check showtimes, seat selection through interactive seat maps, make purchase of tickets and manage personal bookings. It will be focused on usability, having a clean and intuitive layout catering to all users with different technical sophistication. Their profile dashboard will enable them to choose a booking history and digital tickets as well as to log in and register on the site safely.
2. **Administrator Interface (Dashboard):**
   An administrator interface will be well-secured web-based portal, which only certain staff can use. In this interface, a dashboard and data input forms about movie listing, showtime, prices and auditorium setups etc will be provided. The administration will be able to access cancellation records, audit logs, and the setting of systems as well. Each of the interfaces will be created so that navigation is easy, they meet accessibility requirements (e.g., WCAG 2.1 AA), and they have cross-browser compatibility.

3.1.2 Hardware Interfaces

A well-encrypted web-based portal of an administrator will be an administrator interface, and only specific staff members will be able to use this administrator interface. In such an interface, a dashboard and data input forms regarding movie listing, showtime, prices and auditorium setup etc would be given. The administration will also have an access to cancellation history, audit logs and system set-up. All the interfaces will be designed in such a way that it is easy to navigate through them, they will be accessible (e.g. WCAG 2.1 AA), and they will be cross-browser compatible.

3.1.3 Software Interfaces

MTTS will rely on the connection to multiple external software services. It will be linked with third-party payment gateway API (e.g., Stripe, PayPal or a local provider) to securely process online payments. On receipt of the payment, the system will integrate with an e-mail notification API (e.g. SendGrid) to send electronic tickets and confirmations. The frontend and backend services will internally communicate via RESTful APIs and data will be stored in a relational database such PostgreSQL or MySQL, possibly based on deployment preferences. The server-side part will be created on the framework, e.g., Django or Node.js, and the frontend technologies, e.g., React or Vue.js, will be used to render and engage with the users.

3.1.4 Communications Interfaces

The transmission of any information between the client and server components will be implemented through HTTPS-based secure communications. The APIs within the system are going to be REST-based, with elements getting listed by utilizing GET, reservations being made with POST, updating per-user or per-seat with PUT, and canceling the reservation with the DELETE. The database will be stored on a confidential server and will be secured against malicious attacks namely firewalls, access control lists and encryption-at-rest policies. Transaction records and logs will be timestamped and auditable in order to enable a post-event review and compliance.

3.2 Functional Requirements

In this section, essential characteristics of Movie Theater Ticketing System (MTTS) will be outlined. All features are described in the same way: introduction, inputs, processing, outputs and errors. Section 3.3 includes the use-case descriptions, and use-case diagrams where the user can find the non-functional requirements in 3.5.

3.2.1 User Registration and Authentication (FR1)

3.2.1.1 Introduction

New users will be able to register and old users will be able to authenticate in the system. This feature allows customization of services including booking history, ticket cancellation and saved preferences.

3.2.1.2 Inputs

- Full Name (text)
- Email Address (validated, unique)
- Password (must meet complexity rules)
- Login credentials (email and password)

3.2.1.3 Processing

When registering the system checks uniqueness of email and password validity. Passwords are salted and hash. In the process of logging-in, authentication takes place against the database based on credentials. Secure tokens are used to create a session.

3.2.1.4 Outputs

- Success confirmation on registration.
- Session token on successful login.
- User redirected to dashboard.

3.2.1.5 Error Handling

- Invalid email format or existing email: Display error and block submission.
- Weak password: Prompt user to follow complexity rules.
- Incorrect login credentials: Display message with retry option.
- Multiple failed logins: Lock account temporarily and notify user.

3.2.2 Browse Movies and Showtimes (FR2)

3.2.2.1 Introduction

Users can explore available movies and showtimes through a categorized and filterable interface.

3.2.2.2 Inputs

- Filter selections (genre, date, time, language)
- Search keyword (movie title or actor)

3.2.2.3 Processing

The system to access the movies and related metadata in the database, query them using user input and output available showtimes in the results list. Every film listing entails title, poster, description and screenings schedule.

3.2.2.4 Outputs

- A dynamic list of movie cards with showtimes.
- Visual timeline of showings by date and auditorium.

3.2.2.5 Error Handling

- No results found: Display "No movies match your criteria."
- Database connection failure: Show fallback message and retry option.

3.2.3 3.2.3 Seat Selection, Booking Payment and Ticket Generation (FR3)

3.2.3.1 Introduction

Customers can select specific seats from a visual seat map and proceed to book them through a guided checkout process.

3.2.3.2 Inputs

- Selected showtime
- Selected seats
- User authentication token (session)
- Payment method and details

3.2.3.3 Processing

The system displays the diagram of the place and seats; depending on the layout of the auditorium and also indicates booked seats. Pages with chosen seats are locked within a certain amount of time (e.g. 5 minutes). Once the payment gets approved through the payment gateway, the seats are then closed permanently and a confirmation is created.

3.2.3.4 Outputs

- Confirmation page with booking ID
- E-ticket generated (PDF format or QR code)
- Email confirmation sent to user

3.2.3.5 Error Handling

- Attempted double booking: Prevent and prompt user to reselect.
- Payment failure: Notify user and unlock held seats.
- Session timeout: Expire seat lock and request rebooking.

3.2.4 Ticket Cancellation (FR4)

3.2.4.1 Introduction

Customers may cancel their ticket(s) through the system within an allowable cancellation period and, depending on policy, receive a refund.

3.2.4.2 Input

- Booking ID
- User authentication (must match booking account)

3.2.4.3 Processing

The system confirms and the booking and eligibility of cancellation. In case it is valid, the booking is canceled, the seat is freed, and the refund is done using the payment API.

3.2.4.4 Output

- Cancellation confirmation message
- Email notification to the customer
- Refund transaction (if applicable)

3.2.4.5 Error Handling

- Cancellation deadline passed: Inform user and disallow action.
- Unauthorized cancellation attempt: Reject and log event.
- Refund API failure: Retry and escalate to admin if needed.

3.2.5 Admin Movie and Schedule Management (FR5)

3.2.5.1 Introduction

Authorized theater staff can add, edit, or remove movie listings and configure auditorium schedules and pricing.

3.2.5.2 Inputs

- Movie metadata (title, description, poster, duration, rating)
- Showtimes (date, time, auditorium ID)
- Pricing tiers
- Admin credentials/session token

3.2.5.3 Processing

Validated inputs are held in the backend database. Updates of the movie listings are reflected on to the user facing interfaces instantaneously. The system eliminates time conflicts and checks auditorium capacity.

3.2.5.4 Outputs

- Confirmation of new or updated entries
- Immediate visibility of changes on the customer portal

3.2.5.5 Error Handling

- Schedule conflicts or overlapping showtimes: Reject entry and notify admin.
- Missing or malformed fields: Display input validation messages.
- Unauthorized access: Redirect to login and log attempt.

3.2.6 3.2.6 Reporting (FR6)

3.2.6.1 Introduction

The system shall support internal reporting features to help theater managers and administrators monitor performance, identify trends, and make informed operational decisions.

3.2.6.2 Inputs

- Date range
- Movie title (optional)
- Auditorium ID (optional)
- Report type (sales, occupancy, booking trends)

3.2.6.3 Processing

Approved inputs are stored in a backend database. Any updates of the movie listings are projected onto user facing interface in real time. The technology removes time clash and auditorium confirmation.

3.2.6.4 Outputs

- PDF or CSV exportable reports
- On-screen summaries with charts and key metrics
- Dashboard view for quick insights

3.2.6.5 Error Handling

- No data for selected criteria: Display "No results found" message
- System/database error: Log issue and alert administrator

3.3 Use Cases

In this part, the following core use cases are outlined in structured way. Any use case consists of actors, preconditions, normal flow of events, and extensions. It aims at explaining interactive/reactive behaviour of system and aids design and testing.

3.3.1 Use Case 1: Book Movie Ticket

- **Use Case ID:** UC1
- **Use Case Name:** Book Movie Ticket
- **Actor(s):** Customer
- **Description:** This use case describes the process by which a registered user browses showtimes, selects seats, makes a payment, and receives a digital ticket.
- **Preconditions:**
  - The user must be logged into the system.
  - The movie and showtime must be available.
- **Main Flow:**

1. The customer navigates to the movie listings.
2. The customer selects a movie and showtime.
3. The system displays a seat map for the selected show.
4. The customer selects one or more seats.
5. The system locks the selected seats temporarily.
6. The customer proceeds to payment and enters payment details.
7. The system processes the payment via a payment gateway.
8. On successful payment, the booking is confirmed.

9. The system updates seat status to “booked.”
10. An e-ticket is generated and sent to the customer via email.

- **Postconditions:**
    - Seats are reserved and marked as booked.
    - A booking record is stored in the user’s account.
- **Exceptions/Alternate Flows:**
    - Payment fails: User is notified and seats are released.
    - Session expires: User is redirected to login and must restart the booking.

3.3.2 Use Case 2: Cancel Ticket

- **Use Case ID:** UC2
- **Use Case Name:** Cancel Ticket
- **Actor(s):** Customer
- **Description:** This use case describes how a customer can cancel a previously booked ticket and request a refund.
- **Preconditions:**
    - The user must be logged in.
    - The booking must be within the allowable cancellation window.
- **Main Flow:**

1. The customer navigates to the "My Bookings" page.
2. The customer selects the booking to cancel.
3. The system checks if the booking is eligible for cancellation.
4. If eligible, the user confirms the cancellation.
5. The system updates the booking status to "cancelled.
6. Seats are released and marked available.
7. A refund request is sent to the payment gateway.
8. A cancellation confirmation is shown and emailed to the user.

- **Postconditions:**
    - The ticket is marked cancelled and seat is freed.
    - A refund process is initiated.
- **Exceptions/Alternate Flows:**

- Cancellation deadline has passed: Inform user and disallow action.
- Payment refund fails: Escalate to admin with notification.

3.3.3 Use Case 3: Add Movie and Schedule (Admin)

- **Use Case ID:** UC3
- **Use Case Name:** Add Movie and Schedule Showtimes
- **Actor(s):** Administrator
- **Description:** This use case describes how an admin adds a new movie and schedules showtimes with specific auditorium and pricing configurations.
- **Preconditions:**
  - Admin is logged into the system.
  - Auditorium exists and has available time slots.
- **Main Flow:**

1. Admin accesses the dashboard and navigates to "Add Movie."
2. Admin enters movie title, description, genre, language, rating, and upload poster.
3. Admin selects available auditoriums and times for showings.
4. Admin sets seat pricing and seat configuration.
5. The system checks for scheduling conflicts.
6. If valid, the movie and showtimes are saved in the database.
7. New listings appear on the customer portal immediately.

- **Postconditions:**
  - The movie is added to the system.
  - Showtimes and seats are available for booking.
- **Exceptions/Alternate Flows:**
  - Schedule conflicts detected: System prompts admin to reschedule.
  - Missing mandatory fields: Show validation errors and prevent submission.

3.4 Classes / Objects

It is in this section that the main classes (objects) that make up the core of Movie Theater Ticketing System are identified. These classes are mentioned as the main entities which will be modeled during the design stage of the system and which will be implemented into the code. Every class has a short overview of its properties and its connections with other things.

3.4.1 User

3.4.1.1 Attributes

- user_id (Integer): Unique system-generated identifier for the user.
- name (String): Full name of the user.
- email (String): Unique login identifier, validated format.
- password_hash (String): Encrypted password for authentication.
- phone_number (String): Optional contact number for communication.
- role (String): Either 'customer' or 'admin', used for access control.
- created_at (DateTime): Account creation timestamp.

3.4.1.2 Functions

- Register a new account (Refer to Functional Requirement 3.2.1, Use Case UC1)
- Log in to the system (Refer to Functional Requirement 3.2.1, UC1)
- View and manage profile
- View booking history (Linked to Booking object)
- Perform authorized actions based on role (admin functions – UC3)

3.4.2 Movie

3.4.2.1 Attributes

- movie_id (Integer): Unique identifier for each movie.
- title (String): Name of the movie.
- description (String): Brief synopsis of the movie.
- genre (String): Classification such as Action, Drama, etc.
- language (String): Language of the film.
- duration (Integer): Runtime in minutes.
- poster_url (String): Path to the poster image.

- rating (String): Age or content rating (e.g., PG-13).

3.4.2.2 Functions

- Display movie listings for customers (Refer to Functional Requirement 3.2.2, Use Case UC1)
- Add/edit/delete movie records (Admin only; Refer to UC3)

3.4.3 Auditorium

3.4.3.1 Attributes

- auditorium_id (Integer): Unique identifier for the auditorium.
- name (String): Label or name of the auditorium.
- total_seats (Integer): Capacity of the room.
- seat_layout (JSON/Object): Layout structure defining seat rows and columns.

3.4.3.2 Functions

- Store configuration for seat map rendering (Referenced in UC1 and UC3)
- Associate with specific showtimes for scheduling
- Enable seat assignment (Functional Requirement 3.2.3)

3.4.4 Showtime

3.4.4.1 Attributes

- showtime_id (Integer): Unique identifier for a scheduled show.
- movie_id (Foreign Key): Links to a specific movie.
- auditorium_id (Foreign Key): Links to a specific auditorium.
- start_time (DateTime): When the movie begins.
- end_time (DateTime): Automatically computed from duration.
- base_price (Decimal): Ticket cost before discounts or category surcharges.

3.4.4.2 Functions

- Associate movie and auditorium for booking purposes (UC1)
- Display showtime availability to users (Functional Requirement 3.2.2)
- Allow admins to configure new showings (UC3)

3.4.5 Seat

3.4.5.1 Attributes

- seat_id (Integer): Unique identifier for the seat.
- showtime_id (Foreign Key): Links to a specific showtime.
- seat_label (String): Code like A1, B3, etc.
- seat_type (String): Standard, Premium, VIP, etc.
- status (String): Available, Reserved, Booked.

3.4.5.2 Functions

- Display real-time seat map to users (3.2.3, UC1)
- Prevent double booking through seat locking (3.2.3)
- Allow cancellation and release of seat (UC2)

3.4.6 Booking

3.4.6.1 Attributes

- booking_id (Integer): Unique identifier for the booking.
- user_id (Foreign Key): Refers to the user who booked.
- showtime_id (Foreign Key): Refers to the scheduled movie.
- seat_ids (List/Array): Set of selected seats.
- payment_status (String): Paid, Unpaid, Refunded.
- booking_time (DateTime): Timestamp of booking.
- cancellation_time (Nullable DateTime): Timestamp if cancelled.

3.4.6.2 Functions

- Store confirmed ticket data (UC1)
- Support viewing history (3.2.1)
- Enable cancellation (3.2.4, UC2)
- Integrate with Payment and Seat objects

3.4.7 Payment

3.4.7.1 Attributes

- payment_id (Integer): Unique identifier for the transaction.
- booking_id (Foreign Key): Associated booking.
- amount (Decimal): Total paid by the user.

- method (String): Payment channel (e.g., credit card, PayPal).
- status (String): Completed, Failed, Refunded.
- timestamp (DateTime): Time of transaction.

3.4.7.2 Functions

- Process transactions securely (3.2.3)
- Enable refunds during cancellations (3.2.4)
- Update booking status based on transaction result

3.5 Non-Functional Requirements

The quality attributes and the performance limitations of the Movie Theater Ticketing System include the following non-functional requirements. These are system-wide requirements, which maintain reliability, security, usability, maintainability, and performance of all modules.

3.5.1 Performance Requirements

- At peak time, the system must serve a minimum of 1,000 concurrent users without declining responsiveness.
- 95 percent of the transactions (booking of tickets, payment, cancellation) will be performed within 3 seconds on normal load.
- The number of available and sold seats as well as seat reservation will be updated in real time with a no greater than 1s latency of synchronization.
- Summarizational reporting and data analysis: Daily reporting and data analytics will produce summary reports (e.g., revenue, occupancy) in less than 5 minutes of query.
- The response time of the basic queries (e.g., movie listings, showtimes) should not be more than 2 seconds on average (99-percentile).

3.5.2 Availability and Reliability

- The system will have an uptime of 99.5 percent and above in a month, minus scheduled maintenance.
- Routine maintenance should not take more than 2 hours per week and it should take place at least at off-peak times (12 AM- 6 AM local time).

- The system should accommodate automatic recovery of the system to restore the service in not more than 5 minutes in case of server failure.
- Mean Time Before Failures (MTBF) will be above 30 days.
- There will be no single point of failure; it should have redundancy of the database and application servers.

3.5.3 Security Requirements

- User passwords will be stored through a hash and a salt that meets industry encryption standards (e.g., SHA-256 or better).
- Communication between clients and servers should be in HTTPS/TLS (TLS 1.2 or later) in all cases to avoid intercepting data in communication.
- The payment processing will be in line with PCI-DSS of secure credit cards.
- The system will use role-based access control (RBAC) to differentiate among customers, administrators and managers.
- Account lockout shall be in place due to failed logins of 5 logins in number consecutively to last a maximum of 15 minutes, after which the process can be unlocked manually.

3.5.4 Usability Requirements

- The system will have a Graphical User Interface (GUI), webpage and mobile browser (UIResponsiveness) accessible.
- Users should be able to fill and book tickets in not more then 5 clicks or steps from movie selection to confirmation.
- The interface should comply with accessibility standards, it should be able to navigate through the interface with a keyboard and screen readers (WCAG 2.1 Level AA).
- On-screen Tooltips and error messages SHALL lead users through frequent tasks and form corrections.

3.5.5 Maintainability and Supportability

- The system will be modular in order to enable it to make individual modules (e.g. payment module, user management) to be upgraded in a singular fashion.

- System logs and errors will be stored with timestamps and will have to be available to the administrator within 2 minutes.
- New movies, showtime, and prices updates are allowed to be configured by the administrators using the dashboard without alteration of code.
- Documentation (developer API, deployment guide, admin manual) will be updated and supplied every release cycle.

3.5.6 Portability

- The MTTS application will be platform-independent and will run on all modern operating systems (Windows, Linux, macOS), also with JavaScript-enabled browsers.
- The system will be fully functional on Chrome, Firefox, Safari, and Edge with 95+ percent consistency between versions.
- It will be containerized during the deployment (e.g. Docker) to enable the replication of the environment at both development and service servers.

3.6 Inverse Requirements

Inverse requirements state behaviors, and outcomes that the Movie Theater Ticketing System (MTTS) must specifically avoid. These are conditions that are important to the system so that the system has a functional integrity, it also upholds the business rules and any misuse of the system by the user or vice versa. The inverse requirements can be listed as following:

- **Unauthorized Access Prevention**:
  The system should not give a user with unauthenticated or unregistered credentials any access to making ticket reservation, or examining booking history, or administrative dashboard.
- **Duplicate Booking Restriction**:
  The system should be in such a way that, no two users are able to book the same seat successfully at the same show time. Simultaneous booking requests have to be dealt with atomically to avoid race conditions.
- **Payment Circumvention Block**:
  No ticket shall be verified as confirmed or issued until the related payment transaction with the integrated payment gateway will be finalized.

- **Data Manipulation Prohibition**:
  The ordinary users should not have an ability to view data that are beyond their limits or edit the data. As an example, a user is not allowed to modify the other user bookings or access the administrative reports.
- **Improper Showtime Configuration**:
  The program should not give administrators the opportunity to assign the same show at the same time to a show to the same auditorium, as well as allow the addition of shows with date that is already past.
- **Improper Form Submissions**:
  Regarding registration, booking or processing of payment, the system should not accept incomplete or malformed input data. Both the client-side and server-side validations should be imposed.

3.7 Design Constraints

Designing and implementing the Movie Theater Ticketing System (MTTS) is restricted by a series of preordained limitations that are imposed by technical requirements, policy guidelines, law or code, and hardware restrictions. These limitations are hard limits with which one must negotiate on architectural choices, technology stack, interface design and deployment configurations.

Platform and Browser Compatibility

- The system has to be open using contemporary web browsers such as Google Chrome, Mozilla Firefox, Microsoft Edge and Safari.

Mobile responsiveness (minimum supported resolution: 360x640 pixels) to be able to use on smartphones and tablets.

Backend and Deployment Environment

- It is an application that must be deployed within a Linux based server platform (e.g. Ubuntu 20.04 or above) to support hosting and maintenance policies.
- Deployment should allow containerization with Docker and developers should be able to track versions with Git and GitHub.

Database and Storage

- The relational database will be PostgreSQL because this is internal standardized and tested to be ACID compliant.
- All long-term data have to be stored in encrypted volumes to work in accordance with data protection requirements.

Data Privacy and Regulatory Compliance

- The design has to be focused on data privacy laws on the one hand, including GDPR (General Data Protection Regulation) or local variants.
- Personally Identifying Information (PII) which includes user emails and payment information will need to be kept in storage and in its transfer with secure encryption (e.g., TLS 1.2+ and AES-256).

Security Requirements

- Authenticated user logins have to be encrypted to use hashed and salted passwords with algorithms such as bcrypt or Sha-256.
- Secure generation and handling of session tokens should be done, and there should be expiration timeouts to avoid session hijacks. Policy Restrictions in Organization.

Organizational Policy Constraints

- The use of backend dashboard is limited to authorized administrative users. Admin functions should be created according to the inner access control procedures (Role-Based Access Control -RBAC).
- The IT audit requirements should be able to log and report according to the company requirements, as well as retain the data at 12 months minimum.

Payment Gateway Integration

- The system can only allow the use of Stripe or PayPal to make online payments in line with the company agreement with validated payment providers.
- The design should facilitate fallback and retry algorithms on the event of payment gateway downtimes.

Development and Testing Constraints

- The development has to be Agile methodology-based and follow the iterative sprints, Git-based collaboration, and issue tracking via GitHub Projects..
- At least 80 percent of the code coverage will be performed on the critical modules with the help of testing tools like Jest, Selenium and Postman..

Performance Baseline

- The system should be able to achieve some performance levels: 2 seconds or less response time of any booking related action on normal workload..
- The application is supposed to scale upwards to achieve a capacity of 1,000 concurrent users with no impairment of services.

3.8 Logical Database Requirements

The Movie theater ticketing system (MTTS) will utilize a relational database as the means of persisting and handling all the important information essential to the system. The application will rely on the database as its backbone, which will keep data on users, lists of movies, reservations, payment, and metadata. The logical requirements are the following:

Database Type

- The system will use PostgreSQL, a powerful, free, open source relational database management system..
- It supports justification on ACID transactions, referential integrity, complex joins, indexing, and concurrency control that is necessary on the structured and transactional data used by the system..

Data Format and Schema

- All data in storage will be anchored to an organized formula that is per 3rd Normal Form (3NF)..
- Key tables will include:
  - Users: Contains user credentials and profile information.
  - Movies: Stores movie details including title, duration, genre.
  - Showtimes: Links movies to auditoriums with scheduled times.
  - Auditoriums: Defines screen layouts and seating capacity.
  - Seats: Represents seat status and configurations.
  - Bookings: Records seat reservations and user references.
  - Payments: Captures payment transactions and statuses.

Data Integrity Requirements

- **Primary and Foreign Keys** will be enforced to ensure referential integrity between tables (e.g. each booking should refer to a valid booking, showtime and user).

- **Unique Constraints**: Verifies that booking IDs, email addresses and payment transactions identifiers can be used only once.
- **Validation Rules**: Universal rule sets are applied on a field containing email addresses, payment amounts guideline, and password strength.
- **Cascade Rules**: Described in deletions and update (e.g. to delete a movie cascades to related showtimes).

Data Storage and Scalability

- The first system should be capable of supporting up to 100,000 records on a primary entity (e.g., bookings, users).
- Indexing strategies (e.g., on showtimeID, userID, booking_time) as well as partitioning will be introduced to facilitate scalability and improved retrieval with growing data volume.

Data Retention and Archiving

- Operational records (bookings, transactions) should not be destroyed until auditing and analyses are taken care of (at least 12 months of retention).
- Older, dormant data can be periodically archived at secondary tables or external storage (e.g. AWS S3 or cold storage resources).

Backup and Recovery

- The database should enable the automatic backups during a day to be stored at least 7 days.
- Recovery Point Objective (RPO): <=1 day.
- Recovery Time Objective (RTO): <=4 hours.

Security Requirements

- **Data at Rest**: Encrypted using AES-256 or equivalent.
- **Data in Transit**: It is secured further through TLS (HTTPS) used during client-server interactions.
- Access to sensitive tables (e.g., an information about payment details) will be restricted with the help of role-based access control (i.e., only administrators should be able to see the mention of payment details).

3.9 Other Requirements

This section provides other system requirements that may not necessarily be classified as functional, interface or non-functional specifications but cannot be left out in the successful delivery and operation of the system.

Localization and Internationalization

- The system will be able to accommodate the support of the future deployment of multiple languages (e.g., English, Spanish, Urdu) through externalization of texts elements and labels to language resource files.
- Dates and Times will be stored in UTC format in the database but when presented to the user will be in their local time zone by client-side formatting.

Accessibility Compliance

- The frontend must comply with WCAG 2.1 Level AA guidelines so that people with visual, motor, or cognitive disabilities could use it.
- Amongst the features, there is keyboard accessibility, screen reader compatibility, proper color contrast, and alt text to describe non-textual content.

Audit Logging

- All user and administrative actions that should be considered as critical including:
  - Login attempts
  - Bookings and cancellations
  - Showtime modifications by administrators
- The necessary elements of logs should be presented as timestamps, user IDs, IP addresses, and description of actions..
- The logs must be retained at least within 90 days

Legal and Compliance

The system is required to meet the GDPR or local data protection laws by:

- Obtaining explicit consent before storing user data.
- Allowing users to request data deletion or export.
- Avoiding unnecessary data collection beyond operational use.

Deployment and Versioning

- The containerization tool (e.g. Docker) needs to be used in packaging the software so that it can be consistently deployed in all stages across development, test and production..

- The management of source codes will be done through a version control system (e.g. Git) where releases will be tagged and documented..

Mobile Compatibility

It will be made accessible via responsive web interface, which will work well with most mobile gadgets (smartphones, tablets) and will not need a native mobile application..

Future Scalability

- The system architecture should enable the future extension of the system and be expandable to handle multiple theater support even though at present, the system will be created to support one theater only..
- Database schema and Admin dashboards must be capable of supporting flexible theater schedules, pricing and seat designs.

4 Analysis Models

This section gives the analysis models that were utilized in supporting the derivation of the specific requirements of the system as explained in Section 3 of the same document. These models provide a pictorial and textual comprehension of how the system behaves and data paths, which allow enhanced traceability, clarity to the design, and verification.

All the models presented here were chosen in such a way that they fit in and support at least one functional or non-functional requirement mentioned above. Such models also facilitate communication among stakeholders and there is a common ground to the proceedings of the systems.

4.1 Sequence Diagrams

4.1.1 Introduction

Sequence diagrams model the dynamic behavior of the system by showing the interactions between different system components and external actors over time. For the Movie Theater Ticketing System (MTTS), these diagrams help visualize how users interact with the system during critical operations like registration, booking, payment, and cancellation. These diagrams were created to align with the use cases defined in Section 3.3 and functional requirements outlined in Section 3.2.

Each diagram uses standard notation to illustrate lifelines (objects or actors), messages (calls or responses), and control flow. The diagrams also reflect the layered system architecture, including User Interface (UI), Backend Services, and Database.

4.1.2 User Registration and Login

Narrative:
This diagram represents the interaction when a new user registers and then logs into the system.

Textual Sequence Diagram:

```
User -> UI Layer: Open Registration Page
User -> UI Layer: Submit Registration Info (name, email, password)
UI Layer -> AuthController: validateRegistration()
AuthController -> UserService: createUser()
UserService -> Database: INSERT INTO Users
Database --> UserService: Success
UserService --> AuthController: User Created
AuthController --> UI Layer: Registration Success Message

User -> UI Layer: Enter Login Credentials
UI Layer -> AuthController: validateLogin()
AuthController -> UserService: authenticateUser()
UserService -> Database: SELECT * FROM Users WHERE email = ?
Database --> UserService: User Record Found
UserService --> AuthController: Token Issued
AuthController --> UI Layer: Redirect to Dashboard
```

4.1.3 Seat Selection and Booking

Narrative:

This sequence details how a logged-in user selects seats, confirms the booking, and initiates payment.

Textual Sequence Diagram:

```
User -> UI Layer: View Movie Listings
UI Layer -> MovieController: fetchMovies()
MovieController -> Database: SELECT * FROM Movies
Database --> MovieController: Movie List
MovieController --> UI Layer: Display List
```

```
User -> UI Layer: Select Showtime and Seats
UI Layer -> BookingController: lockSeats(seatIDs)
BookingController -> SeatService: verifyAndLock()
SeatService -> Database: UPDATE Seats SET status='Locked'
Database --> SeatService: Success
SeatService --> BookingController: Seats Locked
BookingController --> UI Layer: Proceed to Payment
```

```
User -> UI Layer: Submit Payment Details
UI Layer -> PaymentController: processPayment(details)
PaymentController -> PaymentGatewayAPI: Verify Payment
PaymentGatewayAPI --> PaymentController: Payment Success
PaymentController -> BookingService: confirmBooking()
BookingService -> Database: INSERT INTO Bookings
BookingService -> Database: UPDATE Seats SET status='Booked'
Database --> BookingService: Success
BookingService --> PaymentController: Booking Confirmed
PaymentController --> UI Layer: Show E-Ticket and Confirmation
```

4.1.4 Ticket Cancellation and Refund

Narrative:

The user initiates a cancellation request, which results in refund processing and database updates.

Textual Sequence Diagram:

```
User -> UI Layer: Open 'My Bookings'
UI Layer -> BookingController: fetchUserBookings(userID)
BookingController -> Database: SELECT * FROM Bookings WHERE userID=?
Database --> BookingController: Booking List
BookingController --> UI Layer: Display List
```

```
User -> UI Layer: Click 'Cancel Booking'
UI Layer -> BookingController: cancelBooking(bookingID)
BookingController -> BookingService: validateCancellationWindow()
BookingService -> Database: SELECT booking_time FROM Bookings
Database --> BookingService: Timestamp
BookingService -> PaymentService: initiateRefund(bookingID)
PaymentService -> PaymentGatewayAPI: Refund API Call
PaymentGatewayAPI --> PaymentService: Refund Successful
PaymentService -> Database: UPDATE Payments SET status='Refunded'
Database --> PaymentService: Success
```

```
BookingService -> Database: UPDATE Bookings SET status='Cancelled'
BookingService -> Database: UPDATE Seats SET status='Available'
Database --> BookingService: Success
BookingService --> BookingController: Cancellation Complete
BookingController --> UI Layer: Show Confirmation
```

4.1.5 Traceability to Requirements

Sequence Diagram	Functional Requirement	Use Case
User Registration and Login	FR1	UC1 (Book Ticket)
Seat Selection and Booking	FR3	UC1 (Book Ticket)
Ticket Cancellation and Refund	FR4	UC2 (Cancel Ticket)

4.2 Data Flow Diagrams (DFD)

Data Flow Diagrams (DFDs) serve to convey the flow of data in the Movie Theater Ticketing System (MTTS). They present a picture of the primary processes, external objects, and data stores associated with responding to input by users as well as responding to system commands. Such diagrams are essential to model logical flow of information and to make certain that all the necessary data interactions have been recorded and verified in the process of the system architecture.

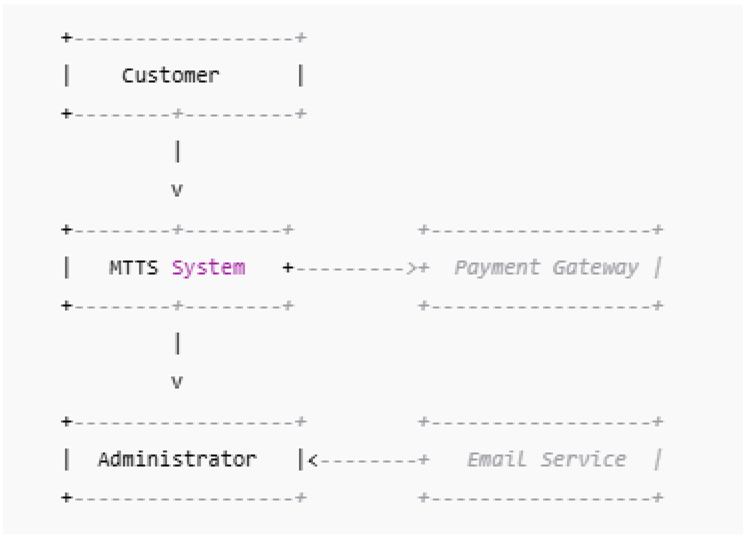
This section gives overview (Level 0) and detailed (Level 1) description of data flow of the MTTS.

4.2.1 Level 0 DFD – Context Diagram

Narrative:

The Level 0 DFD provides a broad overview of the MTTS. It shows the system as a single process and depicts its interactions with external entities like users, administrators, and third-party services.

Textual Representation:



```
+-----------------+
|    Customer     |
+--------+--------+
         |
         v
+--------+--------+         +-----------------+
|   MTTS System   +--------->+  Payment Gateway |
+--------+--------+         +-----------------+
         |
         v
+-----------------+         +-----------------+
|  Administrator  |<--------+   Email Service  |
+-----------------+         +-----------------+
```

Entities:

- **Customer**: Initiates bookings, cancellations, and account actions.
- **Administrator**: Manages schedules, movies, and reports.
- **Payment Gateway**: Processes payments.
- **Email Service**: Sends tickets and notifications.

Central Process:

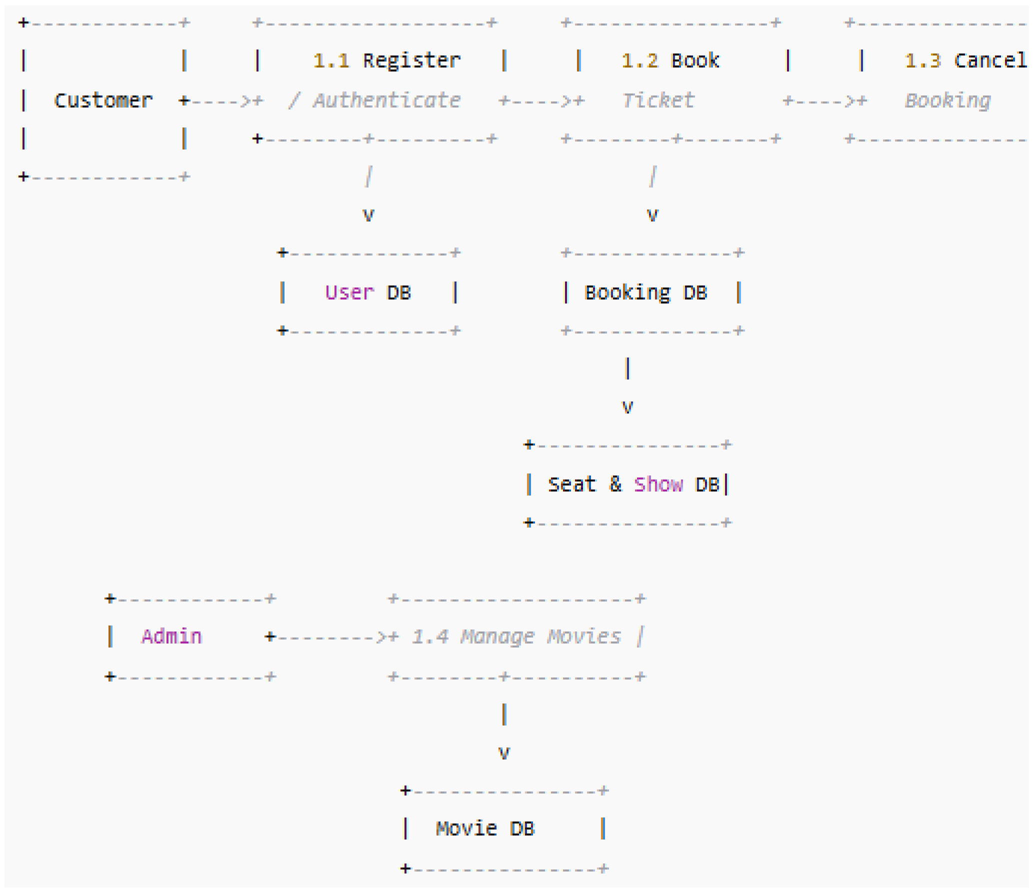
- **MTTS System**: Facilitates all core functions (registration, booking, seat allocation, payments, etc.)

4.2.2 Level 1 DFD – Core Process Breakdown

Narrative:

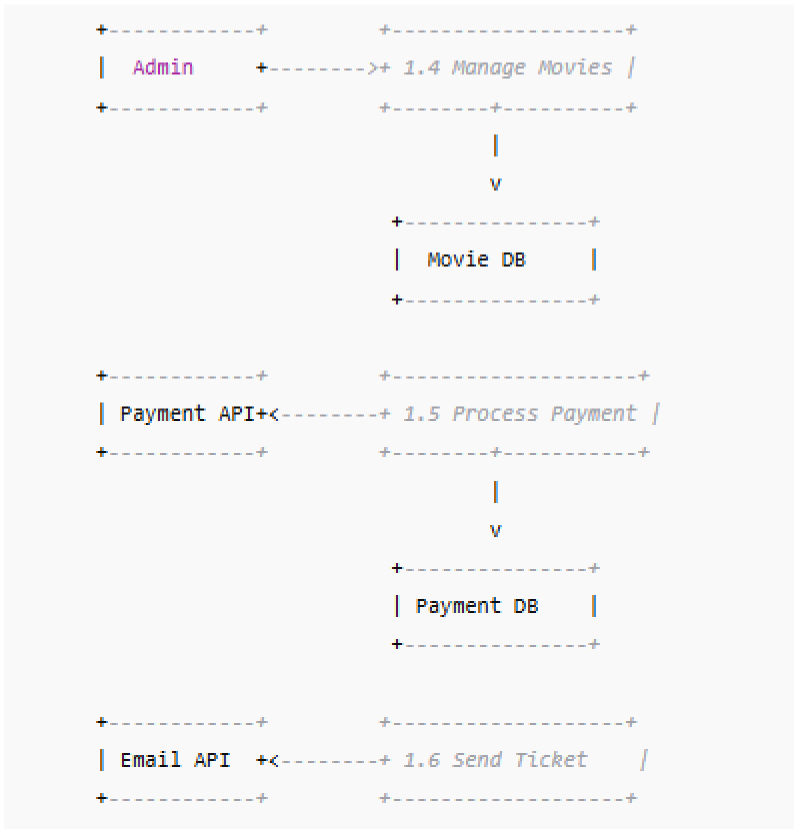
The Level 1 DFD decomposes the MTTS into major sub-processes and shows data stores used for persistence. It helps visualize the internal working components of the system and their data interactions.

Textual Representation:



```
+------------+     +------------------+     +----------------+     +--------------
|            |     |    1.1 Register  |     |   1.2 Book     |     |   1.3 Cancel
|  Customer  +---->+ / Authenticate   +---->+   Ticket       +---->+   Booking
|            |     +--------+---------+     +--------+-------+     +--------------
+------------+              |                        |
                            v                        v
                   +-------------+          +-------------+
                   |   User DB   |          | Booking DB  |
                   +-------------+          +-------------+
                                                 |
                                                 v
                                        +---------------+
                                        | Seat & Show DB|
                                        +---------------+

       +------------+         +-------------------+
       |  Admin     +-------->+ 1.4 Manage Movies |
       +------------+         +--------+----------+
                                       |
                                       v
                               +---------------+
                               |  Movie DB     |
                               +---------------+
```



```
       +------------+         +-------------------+
       |  Admin     +-------->+ 1.4 Manage Movies |
       +------------+         +--------+----------+
                                       |
                                       v
                               +---------------+
                               |  Movie DB     |
                               +---------------+

       +------------+         +--------------------+
       | Payment API+<--------+ 1.5 Process Payment |
       +------------+         +--------+-----------+
                                       |
                                       v
                               +---------------+
                               | Payment DB    |
                               +---------------+

       +------------+         +-------------------+
       | Email API  +<--------+ 1.6 Send Ticket   |
       +------------+         +-------------------+
```

Explanation of Processes:

- **1.1 Register / Authenticate**: Handles user sign-up and login credentials validation. Stores data in User DB.
- **1.2 Book Ticket**: Books seat(s) for a showtime. Updates Seat and Booking DBs.
- **1.3 Cancel Booking**: Releases seat and updates booking status. May initiate refund.
- **1.4 Manage Movies**: Admin adds or updates movie listings and showtimes.
- **1.5 Process Payment**: Initiates and confirms transaction with third-party payment service.
- **1.6 Send Ticket**: Sends booking confirmation and e-ticket to user via email service.

Data Stores:

- **User DB**: Stores user credentials and profiles.
- **Booking DB**: Records all booking details and statuses.
- **Seat & Show DB**: Tracks showtime schedules and seat availability.
- **Movie DB**: Stores movie metadata, genres, and durations.
- **Payment DB**: Maintains payment records, transaction history, and refunds.

Traceability to Functional Requirements

DFD Process	**Linked Functional Requirement**
Register / Authenticate	FR1
Book Ticket	FR3
Cancel Booking	FR4
Manage Movies	FR5
Process Payment	FR3
Send Ticket	FR3

4.3 State-Transition Diagrams (STD)

State- transition diagrams are used to describe the dynamic expectations of any entity, or to express responses of an individual to events. They demonstrate the migration of the given object or system between states in accordance with triggers and requirements. Within the Movie Theater Ticketing System (MTTS), state-transition diagrams are used to capture lifecycle flows of important components namely Booking, Payment, and Seat.

Such diagrams are especially useful in spotting corner cases and developing a consistent response to them like cancellations, failed payments, or multiple seat selections.

4.3.1 Booking Lifecycle State Diagram

Narrative:

This diagram shows the different states a booking can go through, including creation, confirmation, cancellation, and expiration.

Diagram:

```
[Idle]
  |
  | (User selects seats and proceeds to booking)
  v
[Pending]
  |
  | (Payment Successful)
  v
[Confirmed]
  |
  | (User cancels within allowed window)
  v
[Cancelled]
  |
  | (Booking expires before payment)
  ^
  | (Timeout or user inactivity)
  |
[Expired]
```

States:

- **Idle**: No booking initiated yet.
- **Pending**: Seats locked; awaiting payment.
- **Confirmed**: Booking finalized after successful payment.
- **Cancelled**: Booking manually canceled by the user.
- **Expired**: Timeout occurred; booking automatically released.

4.3.2 Payment Process State Diagram

Narrative:

This diagram shows the state changes for a payment process, from initiation through success or failure, and potentially a refund.

Diagram:

```
[Initialized]
  |
  | (Attempt Payment)
  v
[Processing]
  |
  | (Success Response from Payment Gateway)
  v
[Completed]
  |
  | (User cancels booking and requests refund)
  v
[Refunded]
  ^
  | (Gateway refund fails)
  |
[Failed]
```

States:

- **Initialized**: Payment session is created.
- **Processing**: Payment request sent to external API.
- **Completed**: Payment was successful and stored.
- **Refunded**: Amount returned to user after cancellation.
- **Failed**: Payment attempt unsuccessful or refund rejected.

4.3.3 Seat Reservation State Diagram

Narrative:

This diagram captures how a seat moves through different availability states during the booking process.

Diagram:

```
[Available]
  |
  | (User selects seat)
  v
[Locked]
  |
  | (Payment completed)
  v
[Booked]
  |
  | (User cancels or booking expires)
  v
[Available]
```

States:

- **Available**: Seat can be selected by any user.
- **Locked**: Temporarily held during booking (e.g., for 5 minutes).
- **Booked**: Confirmed after successful transaction.
- **Available (again)**: Returned to pool after cancellation or expiration.

Traceability to Requirements

Diagram	**Functional Requirement**	**Use Case**
Booking Lifecycle	FR3, FR4	UC1, UC2
Payment Process	FR3, FR4	UC1, UC2
Seat Reservation	FR3	UC1

5 Change Management Process

Change Management Process describes the procedure of how changes to Software Requirements Specification (SRS) and other project documents used in the software development process of Movie Theater Ticketing System (MTTS) will be managed during the software development lifecycle. Any transformation might be caused by the emergence of new user requirements, customer responses, error detection, security issues, or upgrades in technology. An organized method guarantees that any of the suggested amendments may be considered whether there is an open variety or an ability to implement it as well as how they affect the rest of the project.

5.1 Roles and Responsibilities

- **Change Requester**: Either an instructor or a team member or stakeholder who perceives a need to change. It is their duty to place a formal change request.
- **Change Control Board (CCB)**: The CCB in group projects is generally the entire development team. In the individual projects, the student is the only decision-making force with the course instructor as a consultant. CCB assesses, supports, rejects, or postpones change request.
- **Project Lead (or Group Leader)**: Coordinates the versioning of the projects, the implementation of change and the updating of documentation of the approved changes.

5.2 Change Request Procedure

1. **Initiation**
   A request to perform a change is formatted into a structured form (e.g., Change Request Form or GitHub Issue) expressing:
   - Description of the proposed change
   - Justification for the change
   - Potential benefits and risks
   - Affected components (e.g., database, UI, APIs)
2. **Logging and Tracking**
   Each request is tracked via the issue tracker or project board on the GitHub repository with a specific ID, date and status (e.g., “Open,” “Under Review,” “Approved,” “Rejected”).

3. **Impact Analysis**
   The team assesses how the proposed change would affect:
   - Existing requirements
   - Architecture and design
   - Codebase and test coverage
   - Project timeline and workload
4. **Approval or Rejection**
   The CCB votes on the change based on impacts and benefit assessment. Group decisions can be superseded in an academic context by the instructor feedback.
5. **Implementation**
   Once accepted, the change is assigned to the assignee as the corresponding team member or member of multiple teams, then is dedicated into GitHub with a descriptive note and is merged into the main code branch after a peer review.
6. **Documentation Update**
   Any SRS and other artifacts (test cases, UML diagrams, user stories) are updated to show the change. The document version is incremented (e.g., v1→ v2).
7. **Verification**
   After the implementation, change is validated via regression testing to check whether it has caused defects or inconsistencies.

5.3 Tools and Version Control

- **Versioning**
  Every change to the SRS has a revision number and a revision date in the Revision History section. New minor version release occurs as a result of major design changes.
- **GitHub**
  The complete version history of the code, documentation, and diagrams are managed by using the commit history, pull requests, and issues on GitHub. Change logs can be viewed by any collaborator.
- **Backup and Recovery**
  Earlier versions of documents are stored in the repository in case of rollback ability in case a modification leads to unforeseen problems.

5.4 Review Frequency

- **Weekly Team Reviews**: The team is having regular review sessions in order to know what the group should update on and open requests.
- **Pre-Milestone Checkpoints**: This review guarantees that the SRS and software are in the most up to date state before any assignment is submitted or milestone delivered.

5.5 Communication Plan

All change discussions, resolutions and additions can be found in:

- GitHub Pull Requests and Issues
- Communication channels in groups (e.g., Slack, Discord, MS Teams)
- Meeting minutes shared among team members

Necessary formal communication channels are used in notifying stakeholders of critical or functional changes that have been made.

6 Software Design Specification

6.1 System Description

A modular multi-tiered software, the Movie Theater Ticketing System (MTTS) is tailored in such a way that it gives optimization of the ticket-management process of a cinema facility. As a developer, MTTS can be viewed as service-oriented system, in which loosely coupled but highly cohesive components are combined into a service-based system, communicating using clearly specified interfaces.
In principle, MTTS intends to facilitate the processing of two key types of the user: Customers (moviegoers) and Administrators (theater staff). Customers can view movie listings, showtimes, pick seats and buy tickets either online or via kiosk. Administrators can set up shows, modify movie content, layout auditoriums and produce reports. Its architecture makes the system scalable, responsive and consistent in data.
To facilitate such features, the system will logically consist of the following parts:
Presentation Layer: This entails the customer facing and admin facing. The customer user interface gives users ability to select movies and book seats and payment, and the administrative dashboard gives the user the ability to upload movies and edit as well as change schedule.
Application Layer: The business logic of the system is here. It manages the action of seat primer, transaction clearance, cancellation policy, and schedule conflict.
Data Access Layer: This layer negotiates with the database. It summarizes all the logic that it needs to extract, insert, modify, or delete the records pertaining to the movies, show times, users, seats, and payments.
External Service Integration: This has secure 3rd party payment gateway porting, email ticket services, and analytics. The API contracts make them resilient and fall back in case of failed transaction.
Database Layer: Structured data will be stored and handled using a relational database. The records will be kept in tables containing information about the users, booking, seat, schedule, transaction and audit logs.
The system will use a Model-View-Controller (MVC) architectural pattern that encourages maintainability, testability and separation of concerns. MTTS will be designed to support

concurrent users so that the availability of seats is always updated in real-time and business rules (e.g. cancellation cutoffs, pricing policies etc) are applied to all modules.
The design is cross-platform but will be realized with the help of modern web development technologies, including Python/Django backend and ReactJS frontend. This makes the system future-proof so that they can be installed in a cloud and on-premises environment.

6.2 Software Architecture Overview

Movie Theater Ticketing System (MTTS) adheres to the layered modular approach that is based on Model-View-Controller (MVC) paradigm. The given system is segmented into the modules to which presentation, business logic, data access, and external integrations are encapsulated which makes the system to be scalable, maintainable, and testable. The architecture will be web/mobile client-compatible and will have a lean back-end suitable to run administrative activities and generate reports.

6.2.1 System Components and Responsibilities

1. **Client Interfaces**
   The system allows the use of several access points of users:
   - Web App (developed in React.js)
   - Mobile Application which can be developed using React Native or Flutter.
   - Kiosks (On-site touch-screen enabled browser interface)

Under these interfaces, the consumers are able to sign up, log in, navigate movies, pick some seats, pay and obtain electronic tickets.

2. **Authentication and Session Management**
   Intermediate authentication module deals with:
   - User login and registration
   - Role-based access control (admin, customer)
   - Tokenized sessions (e.g., JWT)
3. **Presentation Layer (Frontend)**
   - **Technology:** React.js (or standard HTML/CSS/JS)

- **Responsibility:** User interface rendering, handling client-side logic, and input collection.
- **Interaction:** Sends authenticated requests to backend via RESTful APIs.

4. **Application Layer (Backend/API Server)**
   Technology: Node.js / Express.js or Django (Python)
   Responsibility:
   - Business logic and request validation
   - Session and token management
   - Interaction with the database
   - Communicates with external APIs (e.g., payment gateway, email service)

5. **Controller Layer**
   The controller layer is a traffic director that distributes a HTTP request to a service logic. Each endpoint is made to support a particular exchange like ticket booking, getting movies, or updates on an admin.
6. **Service Layer (Business Logic)**
   This is the core of the system, holding upon it the services that enforce the fundamental rules of the business. Some major services include:
   - **UserService**: Manages profiles, authentication, and history.
   - **MovieService**: Handles movie listings, metadata, and categorization.
   - **ScheduleService**: Manages showtimes, auditorium assignment, and availability.
   - **BookingService**: Facilitates real-time seat selection, ticket issuance, and confirmation.
   - **PaymentService**: Coordinates with external payment gateways (e.g., Stripe) for secure transactions.
   - **ReportService**: Generates real-time and historical sales reports for admins.
7. **Data Access Layer (DAL)**
   Components of DAL take care of queries and updates on the PostgreSQL database. All services have a clear way of communicating with the DAL which needs to only be well-defined repositories to ensure logic and persistence separation.

8. **PostgreSQL Database**

   The relational database is where all persistent data will be found:

   - User accounts and credentials
   - Movies and showtime schedules
   - Seat maps, bookings, and payment records

   Responsibility:

   - Persistent storage of structured data (Users, Bookings, Movies, Showtimes, Seats, Payments)
   - ACID-compliant transactions
   - Backup and recovery

9. **External Integrations**

   - **Payment Gateway (e.g., Stripe or PayPal):** Used for secure credit/debit card transactions.
   - **Email Service (e.g., SendGrid or Twilio)**: Sends booking notifications and tickets to the users by e-mail (e.g., via SendGrid).

6.2.2 Architectural View (Diagram)

See the architecture diagram in the following that illustrates the layers in the system and the data flows:

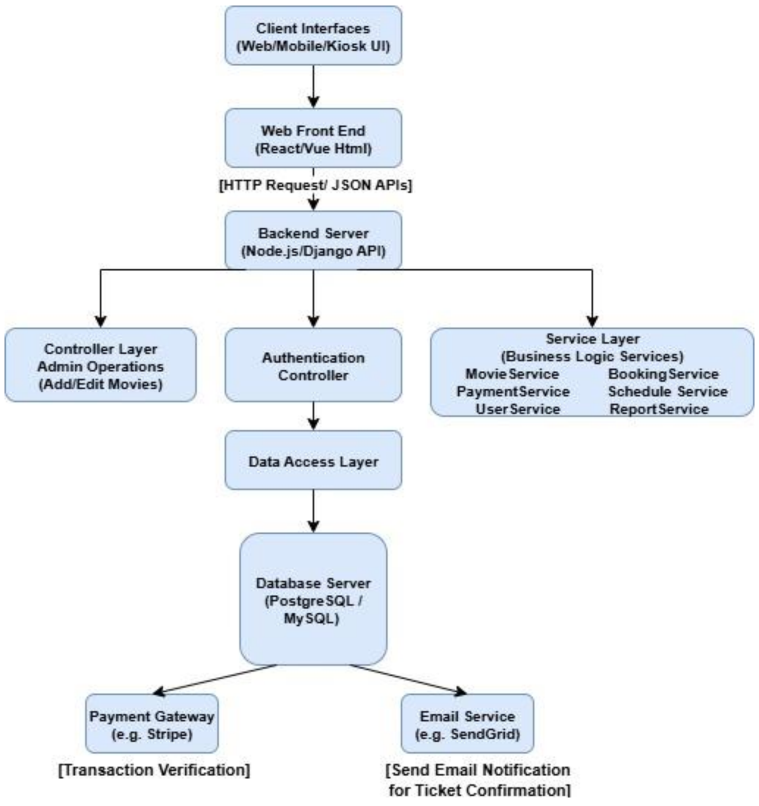

Client Interfaces (Web/Mobile/Kiosk UI)
Web Front End (React/Vue Html)
[HTTP Request/ JSON APIs]
Backend Server (Node.js/Django API)
Controller Layer Admin Operations (Add/Edit Movies)
Authentication Controller
Service Layer (Business Logic Services) MovieService BookingService PaymentService Schedule Service UserService ReportService
Data Access Layer
Database Server (PostgreSQL / MySQL)
Payment Gateway (e.g. Stripe)
[Transaction Verification]
Email Service (e.g. SendGrid)
[Send Email Notification for Ticket Confirmation]

Figure 1:System Architecture Overview

6.2.3 Data Flow Summary

- **User Actions**: Interact with the frontend to browse movies, book seats, cancel tickets, etc.
- **Frontend to Backend**: Inputs are sent over secure HTTPS to backend routes (e.g., POST /register, GET /movies).
- **Backend Logic**: Applies input validation, initiates database transactions, or makes external API calls.
- **Database Transactions**: Interact with normalized SQL tables for persistent storage.
- **External APIs**: Only triggered when critical operations (e.g., payment) are validated and committed.
- **System Feedback**: Result of operations (confirmation, error messages) returned to the user.

6.2.4 Design Principles Applied

- **Separation of Concerns (SoC)**: UI, logic, and data are strictly separated.

- **Security by Design**: All data communication over TLS; backend enforces authorization and role-based access.
- **Scalability**: Stateless API design allows for horizontal scaling.
- **Maintainability**: Modular structure supports independent updates to UI, backend, and data schema.
- **Consistency & Reliability**: SQL-based transactional integrity ensures predictable operations and failure rollback.

6.3 UML Diagram Description

Movie Theater Ticketing System (MTTS) UML Class Diagram represents the static components of the system, but it reveals the classes that are important to the system and their attributes, operations and depict the relations that exist between the classes. The diagram is the basis of system implementation and it is a simple way to give an actual picture of the interaction between core components of the system.

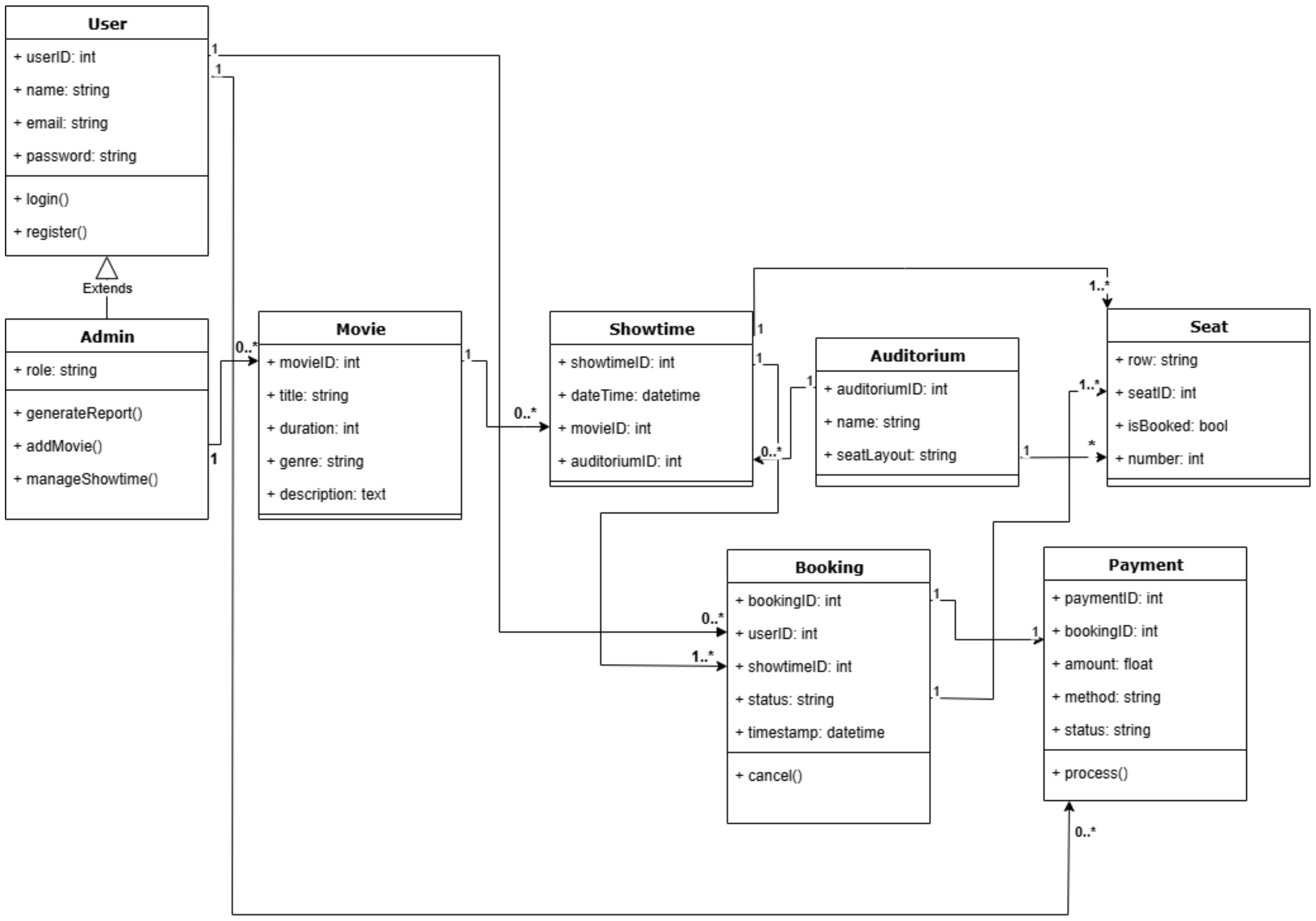


Figure 2:UML Class Diagram

6.3.1 Class Descriptions:

6.3.1.1 User

- **Attributes:**
    - userID: int – Unique identifier for each user.
    - name: string – User's full name.
    - email: string – Email used for login and contact.
    - password: string – Encrypted password string.
- **Functions:**
    - login(): void – Authenticates the user.
    - register(): void – Registers a new user.

- **Relationships:**
    - One User can have **zero or more (0..*)** Booking records.
    - Users can view available movies in the system, though they do not create or modify them.
    - The Admin class extends User.

6.3.1.2 Admin (inherits from User)

- **Attributes:**
    - role: string – Specifies the role (e.g., "admin").
- **Functions:**
    - generateReport(): void – Generates system usage or financial reports.
    - addMovie(): void – Adds a new movie entry to the system.
    - manageShowtime(): void – Manages and modifies showtime data.
- **Relationships:**
    - Inherits all relationships and behaviors of User.

6.3.1.3 Movie

- **Attributes:**
    - movieID: int – Unique movie identifier.
    - title: string – Name of the movie.
    - duration: int – Duration in minutes.
    - genre: string – Genre such as Action, Drama, Comedy.
    - description: text – Summary or storyline.
- **Relationships:**
    - One Movie is associated with **zero or more (0..*)** Showtime entries.
    - Each Showtime is linked to exactly **one (1)** Movie.

6.3.1.4 Showtime

- **Attributes:**
    - showtimeID: int – Unique identifier for the showtime.
    - dateTime: datetime – Date and time of screening.

  - movieID: int – Reference to the associated movie.
  - auditoriumID: int – Reference to the auditorium.
- **Relationships:**
  - One Showtime is for **one (1)** Movie.
  - One Showtime occurs in **one (1)** Auditorium.
  - One Showtime may have **zero or more (0..*)** Booking entries.

6.3.1.5 Auditorium

- **Attributes:**
  - auditoriumID: int – Unique identifier.
  - name: string – Auditorium name (e.g., Hall A).
  - seatLayout: string – Layout details in encoded format.
- **Relationships:**
  - One Auditorium can host **zero or more (0..*)** Showtime sessions.
  - One Auditorium contains **one or more (1..*)** Seat entries.

6.3.1.6 Seat

- **Attributes:**
  - seatID: int – Unique identifier.
  - row: string – Row label (e.g., "A").
  - isBooked: bool – Indicates booking status.
  - number: int – Seat number within the row.
- **Relationships:**
  - Each Seat belongs to exactly **one (1)** Auditorium.
  - Each Seat can be associated with one or more bookings indirectly using availability check during the seat selection process.

6.3.1.7 Booking

- **Attributes:**
  - bookingID: int – Unique booking reference.
  - userID: int – Reference to the booking user.

  - showtimeID: int – Reference to the showtime.
  - status: string – Booking status (e.g., Confirmed, Cancelled).
  - timestamp: datetime – When the booking was made.
- **Functions**
  - cancel(): void – Cancels the booking and updates related entities.
- **Relationships:**
  - Each Booking is linked to exactly **one (1)** User.
  - Each Booking refers to exactly **one (1)** Showtime.
  - One Booking has exactly **one (1)** Payment.

6.3.1.8 Payment

- **Attributes:**
  - paymentID: int – Unique payment reference.
  - bookingID: int – Reference to the related booking.
  - amount: float – Total payment amount.
  - method: string – Payment method used.
  - status: string – Status (e.g., Completed, Failed).
- **Functions:**
  - process(): void – Executes the payment transaction.
- **Relationships:**
  - Each Payment is linked to exactly **one (1)** Booking.

6.3.2 Design Summary:

The model is modular, maintainable and its responsibilities are separated very well:

- Important individual domain entities (movie, showtime, user, booking, and so on) are modeled in an individual way.
- Referential integrity exists between components as a result of foreign key relationships.
- Methods like cancel() and process() allow explaining important class level behaviors.
- The structure is extendable, e.g. go ahead and add Review, Concession, or Discount classes in the future.

The diagram has been developed in such a way that essential system flows, including browsing a movie, selection of a seat, booking and payment should be traced through the relationship between the objects and their operations.

6.3.3 Design Changes from Assignment 2

After a serious review of the assets and needs of the Movie Theater Ticketing System (MTTS) software, there was no need to make any modifications of the UML Class Diagram provided in Assignment 2. The current design is still relevant in expressing all important aspects of the system and completely consistent with the extended test plan created in this assignment.
To make it more clear and complete the following are added as minor descriptive additions in Section 4.3.1 (Class Descriptions):

- **Seat to Booking Relationship Clarification:** There is no direct relationship in the UML but now the description stipulates that the relationship is checked (availability) and linked operationally when plotting a seat during the booking operation.
- **User to Movie:** It has been clarified that the user only interacts with the movies available as a read-only experience (e.g. browsing and selecting) but only administrators create new movies.
- **Terminology Consistency:** There was a standardization of the headings of functions to be called as Operations to emulate the UML terminologies in all the classes.
- **Return Types:** Return types added on all operations where applicable.
- **Multiplicity Verification:** Multiplicities (such as 0..*, 1..*) were checked and confirmed the reading and accuracy of the system requirements.

Note: No new features or parts were added to this phase, that is why the class diagram alone does not differ in this phase as compared to the one of Assignment 2. The descriptive refinements offer better alignment with the test plan and better understanding in its entirety without changing the system structure.

6.4 Development Plan and Timeline

Designing of Movie Theater Ticketing System (MTTS) is an iterative and systematic procedure. This section provides the stages of development of the project, the personal roles, project schedule plan, as well as the practices of the contribution. Since it is a one-member project, the related issues of development such as design, implementation and testing will be carried out by one person.

6.4.1 Phased Timeline

The process of development is conceptualized into different, sequential stages which are spaced out within 5 weeks. Every stage will feature critical deliverables that will facilitate the fulfillment of the MTTS.

Phase	**Week**	**Key Deliverables**
Phase 1: Requirements Analysis	Week 1	Completion of Software Requirements Specification (SRS), Use Cases
Phase 2: Design Specification	Week 2	Software Architecture Diagram, UML Class Diagram, Class Descriptions
Phase 3: Core Implementation	Week 3–4	Development of core modules: registration, seat booking, payment, and admin panel
Phase 4: Testing and Debugging	Week 5	Execution of test plans, validation of use cases, bug fixes, performance tuning

6.4.2 Responsibilities

Because I am the only developer, I will be in charge of:

- SRS documentation and SRS requirements elicitation
- Software architecture and class modeling Developing software architecture and class models
- Front end and backend implementation
- Incorporation of 3 rd party services (e.g. payment gateway)
- Unit and integration testing Conducting unit and integration testing
- Version control and use of GitHub repository
- Capturing of progress and keeping the system traceable

6.4.3 GitHub Contribution and Tracking

On the basis of course requirements:

- I will source code and documentation, diagrams to the group GitHub repository
- The version control conventions (a clear commit message, well-structured branches) will be adhered to at all times
- This repository will picture entire history and traceability of work throughout the phases
- Every major deliverable will have at least one significant commit made

6.4.4 Development Tools and Stack

- **Frontend**: HTML, CSS, JavaScript (React optional)
- **Backend**: Node.js with Express or Python Flask
- **Database**: PostgreSQL or MySQL
- **Version Control**: Git, hosted on GitHub
- **Design Tools**: Draw.io (Architecture + UML diagrams)
- **IDE**: Visual Studio Code

7 Test Plan

This section contains the Movie Theater Ticketing System (MTTS) testing strategy, objective and scope. The point is to ensure that the system is good to fulfill all functional and non-functional requirements as stipulated in previous sections of this paper. The testing will take place in accordance with the industry standard procedures to provide system correctness, usability, reliability, and security. Unit-level component testing, feature-explicit functionality testing and entire system-level workflow testing will be carried out. It also has a traceability matrix and structured test cases enabling mapping of all requirements to testing.

7.1 Test Plan Overview

The testing process of the MTTS will allow the product to meet its intended application in a manufactured environment. In the testing process, the main aims are:
Correctness: Be sure that every functional part works within the specification and the business rules.

- **Reliability:** Ensure the system is reliable in terms of the loads it is expected to handle and how it is used.
- **Security:** Confirm that only allowed personnel will access sensitive functions or make administrative action.
- **Usability:** evaluate the usability and familiarity of the customer and administrator user interfaces.

To attain the above objectives the following testing strategies will be used:

- **Black-Box Testing:** It aims at the validation of functional behavior with local expertise of the code. It will be highly applied to UI and feature-level testing (e.g., login, booking, cancellation).
- **White-Box Testing:** Usable with unit testing on the back-end tasks including payment processing and user validation, and also guaranteeing the coverage of right execution paths in the codes.

- **Boundary Testing:** This is utilized to verify the edge cases and limit values, including incorrect seat reservations, maximum showdate availability, or maximum concurrent users.

The tests will be executed in a staging environment that replicates production, conditions. The server environment where the system will be implemented will be controlled (e.g., Heroku or localhost) and it will include the following components:

Front-End: web application (React.js)

Back-End: Node.js using Express.js API

Database: PostgreSQL containing test data

Test Tools:

- Unit testing Joe and Jest along with Mocha
- API with Postman testing
- Automated UI Selenium testing
- GitHub and Excel documentation and version control

Each test will be run based on specified scripts and the results will be recorded in files of structured formats that can be evaluated.

7.2 Features to Be Tested

Among the key functionalities of MTTS platform, the test plan will comprise the following:

1. **Login and Registration of Users**

   The capacity to open new accounts, sign onto the accounts using credentials and the profile features. This covers form validation, policy enforcement on passwords and authentication choreography.
2. **Browsing Movie and Showtime**

   Customers should have the opportunity to see a list of available movies that have metadata (title, duration, genre, description) and respective showtime metrics by date and location.
3. **Choice of Seats and Reservation**

   Seat availability has to be shown in real time. They should be able to reserve, and the users should have the option to reserve seats and there are no over booking or double booking.
4. **Ticket and Generation of Payments**

Connection to safe payment gateways to purchase tickets. When transaction success is accomplished, the users are issued with electronic ticket(s) containing a booking identifier.

5. **Ticket Cancellation**

   The system is to enable users to cancel reservations within a permissible period, adjust the seat availability, and where necessary, start the process of refunds.

6. **Admin Dashboard (Movie and schedule Management)**

   The administrative users must be in a position to append, modify and delete movies listing, showtimes, auditorium layout and pricing structures.

7. **Reporting and Analytics**

   The system should produce operational reports such as sales summaries, occupancy rates and user activities logs. Such reports must be exportable and must be snapshots in time or daily.

All these features will be tested with the proper test levels and traced to certain requirement in the Traceability Matrix (Section 6.6).

7.3 Features Not to Be Tested

As far as Movie Theater Ticketing System (MTTS) is concerned, although it is an exhaustive system, the following would fall outside the scope of testing in this assignment, or in the test environment in cases when there will not be enough resources available:

- **Third-Party Analytics tools integration**

  Outside services such as Google Analytics, Mixpanel or other integration will not be tested yet since they are not embedded in the platform. These will be taken into consideration later during system iterations, or deployment.

- **Kiosk Printing Hardware Interaction**

  Self-service kiosks or actual printers to print tickets are not provided in the present test system. The process of generating tickets will be emulated by digital download or PDF in email.

- **Merchandise/Concession Sales Modules**

  Our system is only centered on ticking functions, at the moment. Other features such as food delivery, merchandise sales or loyalty rewards are not part of this release and are therefore, not part of the testing plan.

7.4 Test Strategy

The testing shall follow a tiered-based implementation that spans the testing approaches of unit, functional, and system testing, which are all geared towards verification and validation. All the layers possess their own objectives, boundaries and entrance/departure values as follows:

7.4.1 Unit Testing

Scope:
Tests the individual components of software, including functions and modules independently, so that each of them can perform its designated functions.
Examples:

- registerUser(email, password)
- validateSeatAvailability(showtimeID, seatID)
- processPayment(bookingID, paymentDetails)

Entry Criteria:

- Component/module has been implemented and internally reviewed
- The required dependencies have been stubbed or mocked

Exit Criteria:

- All unit test cases are passed without critical errors
- Code coverage is at least 80% for core modules
- No memory leaks or exceptions under expected input ranges

Tools Used: Jest, Mocha, Sinon

7.4.2 Functional Testing

Scope:
The most important features are tested end-to-end to ensure they pass the functional requirements.
Examples:

- User registers and a confirmation email is sent

- Booking a seat, making payment, and receiving e-ticket
- Admin adds new showtime and assigns an auditorium to it

Entry Criteria:

- All relevant modules have passed unit testing
- Front-end and back-end integration is complete

Exit Criteria:

- All functional test scenarios produce correct outputs
- Validations, UI interactions, and error messages work as specified
- No blocking defects in tested workflows

Tools Used: Selenium, Postman, Manual UI walkthroughs

7.4.3 System Testing

Scope:

Confirms the behavior of the whole system as near-production. The forms are stress testing, usability testing, and reliability of the interface.

Examples:

- 50 concurrent users booking seats simultaneously
- System response time under normal vs. peak load
- Full booking and cancellation lifecycle with payments and rollbacks

Entry Criteria:

- All functional tests have passed
- Environment is configured to simulate production settings

Exit Criteria:

- All critical workflows perform as expected under load
- No security violations or authentication bypasses
- No crash or deadlock problem with edge-case conditions

Tools Used: JMeter (for load), Selenium (regression testing), BrowserStack (cross-browser)

7.5 Test Deliverables

The test plan of Movie Theater Ticketing System (MTTS) will have the following deliverables that will be generated and handed in:

- **Test Plan Document**
  This document (i.e., Section 5 of SRS) sets out the details of testing strategy, what should be tested, testing levels, tools, and test environment.
- **10 Sample Test Cases**
  A sample of test cases made in the format of Excel with the detail of unit, functional, and system-level testing scenarios based on the given MTTS_TestCases.xlsx template.
- **Traceability Matrix**
  A trace of test cases and type of tests between the functional requirements and system approach to the use of templates, which illustrates total coverage of the required features.
- **GitHub Link to Repository**
  The GitHub repository with publicly available source code, SRS document, the test plan, UML diagrams, and Excel-based Test cases.

7.6 Traceability Matrix

Requirement ID	Requirement Description	Test Case ID	Test Level	Component
FR-1	User registration with valid credentials	MTTS_Unit_1	Unit	User_Auth_Module
FR-2	Payment processing using valid payment details	MTTS_Unit_2	Unit	Payment_Module
FR-3	Booking seats for selected movie and showtime	MTTS_Func_1	Functional	Booking_Module
FR-4	Booking cancellation updates status and seat availability	MTTS_Func_2	Functional	Cancellation_Module
FR-5	Admin can create new showtime entries	MTTS_Func_3	Functional	Showtime_Module

FR-6	Complete booking journey from registration to payment	MTTS_Sys_1	System	System_Integration
FR-7	System maintains performance under 100 concurrent users	MTTS_Sys_2	System	Performance_Testing
FR-8	isBooked flag updates accurately upon successful booking	MTTS_Unit_3	Unit	Seat_Management
FR-9	Admin can generate booking reports for selected criteria	MTTS_Unit_4	Unit	Admin_Module
FR-10	Failed payment transactions are handled gracefully with appropriate error messaging	MTTS_Sys_3	System	Error_Handling

7.7 Test Environment Setup

The system will undergo a controlled test in an environment that approaches reality of production deployment. The set up entails:

- **Application Platform**:
  Developed under localhost. The end production will adopt either Heroku or AWS EC2 based on the performance needs.
- **Database**:
  Primary Option: PostgreSQL 14
  Alternative (for testing): MySQL 8
- **Tools and Frameworks**:
  Testing:
  - *Jest*: For unit testing JavaScript components

- *Selenium*: For automated UI and regression testing
- *Postman*: For API endpoint testing and validation

Development:

- *VSCode*: Code editing and integration
- *GitHub*: Version control and submission tracking

7.8 Test Cases

The Movie Theater Ticketing System (MTTS) testing process uses ten sample test cases, where three dimensions of software testing are employed:

- **Unit Testing**: Unit testing checks that the separate components, including functions of registration of the user and processing of payment, work well.
- **Functional Testing** – Verifies specific business rules in booking, cancellation and movie browsing.
- **System Testing** – Tests the end-to-end behavior of the system, performance and integration.

At each level of granularity at least two test cases are ready. The template in the spreadsheet MTTS_TestCases.xlsx contains all test cases organized and recorded in it. Each test case in the spreadsheet contains the following information:

- Test Case ID
- Component
- Priority (P0, P1, P2 and P3)
- Description/Test Summary
- Pre-requisites
- Test Steps
- Expected Result
- Actual Result
- Pass/Fail Status
- Test Executed By

These test cases are traced to this SRS, and stored in the GitHub repository to be able to establish traceability and repeatability during validation and regression testing.

Test Cases									
TestCaseId	**Component**	**Priority**	**Description/Test Summary**	**Pre-requisites**	**Test Steps**	**Expected Result**	**Actual Result**	**Status**	**Test Executed By**
MTTS_Unit_1	User_Auth_Module	P1	Verify successful user registration with valid credentials.	User registration page is accessible.	1. Navigate to the registration page. 2. Enter valid name, email, and password. 3. Click on 'Register'.	User should be registered and redirected to the login page.	User registered and redirected.	Pass	Guled
MTTS_Unit_2	Payment_Module	P1	Validate processPayment() with valid payment details.	Booking completed and payment page available.	1. Select a valid booking. 2. Enter valid payment details. 3. Click 'Pay Now'.	Payment should be processed and confirmation displayed.	Payment processed and confirmation displayed.	Pass	Guled
MTTS_Func_1	Booking_Module	P0	Ensure user can select	User is logged in,	1. Select seats.	Seats should be booked	Booking confirmed.	Pa	Guled

			seats and make a booking.	movie showtime selected.	2. Click 'Book Now'. 3. Confirm booking.	successfully and confirmation displayed.		s s	
MTTS_Func_2	Cancellation_Module	P1	Verify cancellation of an existing booking.	User has an active booking.	1. Navigate to My Bookings. 2. Select booking. 3. Click 'Cancel'.	Booking should be canceled and seat availability updated.	Booking canceled successfully.	Pass	Guled
MTTS_Func_3	Showtime_Module	P2	Ensure admin can add a new showtime.	Admin logged in, showtime form accessible.	1. Fill in movie, auditorium, and time. 2. Click 'Add Showtime'.	New showtime should be created and visible in the listings.	Showtime created and listed.	Pass	Guled
MTTS_Sys_1	System_Integration	P0	Verify complete user journey from registration to ticket booking.	System deployed and all components integrated.	1. Register a new user. 2. Login. 3. Select movie and seats. 4. Make payment.	End-to-end booking should complete successfully.	Booking flow completed.	Pass	Guled
MTTS_Sys_2	Performance_Testing	P1	Check system behavior under 100 concurrent users.	Load testing environment setup.	1. Simulate 100 users accessing booking module.	System should maintain response times < 2s.	All requests handled within 2s.	Pass	Guled

MTTS_Unit_3	Seat_Management	P2	Validate isBooked flag is updated correctly after booking.	User selects seat for booking.	1. Book an available seat. 2. Check isBooked flag in DB.	isBooked = true after booking.	Flag updated as expected.	Pass	Guled
MTTS_Unit_4	Admin_Module	P3	Check admin can generate a report.	Bookings exist for the selected date range.	1. Login as Admin. 2. Select report criteria. 3. Click 'Generate'.	Report should be generated with relevant data.	Report generated successfully.	Pass	Guled
MTTS_Sys_3	Error_Handling	P1	Verify graceful handling of failed payment transactions.	Mock payment gateway failure.	1. Attempt payment with invalid details. 2. Observe behavior.	Payment failure message and retry option shown.	Error handled gracefully.	Pass	Guled

7.9 GitHub Repository Link

All documents, diagrams, and source code have been uploaded to the group repository: https://github.com/Kenzo4888/movie-theater-system

8 Data Management Strategy

8.1 Data Management

Movie Theater Ticketing System (MTTS) is a complex system that uses persistent, relational data to sustain its functions including user registration, seat reservations, payments and schedules. There needs to be a solid data management strategy that would provide transactional integrity, data consistency, scalability, and support reporting and analytics. Here the selected database type, schema design, entity relationships and data organization logic have been presented together with the tradeoffs and alternatives discussion section.

8.1.1 Database Type and Justification

Database Type: PostgreSQL

A Relational Database Management System (RDBMS) is chosen after considering the type of the MTTS dataset assessed as well as operational needs. In particular, PostgreSQL is suggested as the main data store since it supports the structured data, transactional integrity and querying with the SQL language. MySQL is another system that the system can utilize instead of meeting similar needs.

Rationale for Selecting SQL (PostgreSQL)

- **Structured Data Requirements**: MTTS is dealing with structured and related data of users, movies, showtimes, bookings, payments and seats. The data structures are appropriate in tabular formats with specific schemas and foreign key requirement.
- **Relational Integrity**: These issues involve many-to-one and one-to-many relations as an essence of MTTS (e.g. one user might have many bookings; one showtime coincides with one movie). Such relationships are supported by a relational database.
- **Transactional Operations**: Transactions such as booking of tickets, and processing payment need to have ACID (Atomicity, Consistency, Isolation, Durability) compatibility to guarantee consistency. A ticket cannot be booked twice in a row, and booking needs to be made only after the payment. SQL databases natively offers these guarantees.
- **Advanced Querying and Reporting**: Utilising the features such as advanced querying/reporting including advanced SQL queries, indexing, and JOIN, PostgreSQL helps the generation of real-time reporting on critical business indicators including

occupancy rates and most accurate revenue summaries and customer activity trend analysis.

- **Scalability and Maintenance**: PostGreSQL provides perf optimization tools, schema migrations, indexing techniques and extensions (like PostGIS to handle spatial data or TimeScaleDB to higher-performance time-series analytics).

Alternative Considered: NoSQL (e.g., MongoDB)

NoSQL systems such as MongoDB were also found to be less suitable to serve the MTTS case, although flexibility and scaling are important features of that system. NoSQL is particularly suited to unstructured or fast- changing data models, something that is not required most in this project.

Aspect	**SQL (PostgreSQL)**	**NoSQL (e.g., MongoDB)**
Schema	Fixed, defined schema	Schema-less, flexible
Data Relationships	Strong foreign key support	Requires manual embedding or linking
Transaction Support	Full ACID compliance	Limited (unless using transaction wrappers)
Query Complexity	Supports complex joins and analytics	Simpler query language, limited relational joins
Use Case Fit (MTTS)	Excellent	Suboptimal

8.1.2 Schema and Logical Grouping

The MTTS application usages one logical database based on a number of normalized tables defining the main system entities. This pattern makes administration easy, data consistency clear and leads to the establishment of faithful business logic across modules..

Database Structure Overview

The primary entities are:

- **Users**: Registered accounts, including customers and administrators.
- **Movies**: Metadata about films such as title, genre, and duration.
- **Auditoriums**: Screen names and seating configurations.
- **Showtimes**: Movie schedule associated with specific auditoriums.
- **Seats**: Individual seats, their availability status, and labels.

- **Bookings**: Records of reservations made by users.
- **Payments**: Transactions linked to bookings and their status.

Table Relationships

The system follows a normalized structure:

- **One-to-Many**:
  - One **User** → Many **Bookings**
  - One **Movie** → Many **Showtimes**
  - One **Auditorium** → Many **Seats** and **Showtimes**
- **One-to-One**:
  - One **Booking** → One **Payment**
- **Many-to-One**:
  - Multiple **Showtimes** → One **Movie**, One **Auditorium**
  - Multiple **Seats** → One **Auditorium**

Textual ER Diagram

```
Users (userID PK)
  └────< Bookings (bookingID PK, userID FK)
        └────< Payments (paymentID PK, bookingID FK)
        └────< Showtimes (showtimeID FK)
              └────< Movies (movieID FK)
              └────< Auditoriums (auditoriumID FK)
Auditoriums
  └────< Seats (seatID PK, auditoriumID FK)
```

Foreign key constraints are used on each table which ensures referential integrity between all modules. We have keys, like the userID, movieID and showtimeID, which are propagated in form of references used to tie entities together..

Design Tradeoffs and Alternatives

Design Option	**Description**	**Tradeoff**
Single SQL Database (Chosen)	All tables under one PostgreSQL instance.	Simplifies development and supports relational queries, but limits microservice scaling.

Microservices with Separate DBs	Each service (e.g., booking, payment) has its own DB.	Allows independent scaling but increases overhead in data syncing and consistency enforcement.
Hybrid (SQL + NoSQL)	SQL for transactional data, NoSQL for analytics/logs.	More powerful but complex to manage and deploy.

8.1.3 Data Access and Transactions

The Movie Theater Ticketing System (MTTS) is based on accurate data processing and this is the reason why it needs to support transactions to maintain the consistency of the system, particularly to book tickets, received payments and cancel tickets. This section describes data access and transactional processing in the system so that it has integrity to avoid data abnormalities.

Transaction Management

PostgreSQL supplies the system with ACID-friendly transaction management. Atomic transactions combine critical user actions that entail seat booking and processing payments. An example is the process of booking a seat (i.e. locking the seat, processing pay and recording booking) are considered a single unit of work. Should any step incomplete, the complete transaction is rolled back and no partial data starts, or incomplete data-sets exist.

- **Atomicity**: Makes sure all the steps in a transaction either take place or not..
- **Consistency**: Ensures validity of database state prior to and after processing transactions.
- **Isolation**: Eliminates the possibility of interference of concurrent transactions (e.g. two users to book the same seat).
- **Durability**: Durability is the ability in that after a transaction has ended in a commitment the changes are reflected even after a system crashes.

Access Methods

Data access is layered using a secure API interface:

- **Frontend** (React/HTML): Sends user requests (e.g., booking, cancellation) via HTTPS.
- **Backend API** (Node.js/Django): Verifies and handles requests..
- **Data Access Layer (DAL)**: Deals with access to the PostgreSQL database via Object-Relational Mapping (ORM) facilities (e.g., Sequelize, SQLAlchemy).

This design helps in separation of concerns, maintainability of code and secure access.

Concurrency Control

In order to serve many simultaneous users (e.g. selling tickets concerning a blockbuster movie), MTTS implements:

- **Pessimistic locking** for seat selection: Race conditions are avoided by temporarily locking selected seats.
- **Optimistic locking** for admin updates: Minimises contention on database when performing low-risk activity, such as editing a schedule.

Logging and Auditing

- Every insert/update/delete data modification action is tracked with a timestamp and user ID.
- The logs will be securely stored and periodically archived in order to aid in the debugging and audits..

8.1.4 Backup and Recovery Strategy

Since MTTS is transactional and user-centric, a trustworthy backup and recovery plan is important to maintain business continuity and data protection in the wake of system malfunctions, data corruption, or cyberattack..

Backup Policy

- **Daily Full Backups**: A complete snapshot of the database is taken every 24 hours.
- **Incremental Backups**: Changes since the last full backup are recorded every hour.
- **Retention Schedule**: Daily backups are stored for 30 days; weekly snapshots are stored for 3 months.

Backups are encrypted and located off-site (e.g. cloud storage on AWS S3 or Azure Blob Storage) in order to provide safeguards against physical damage and data center outage.

Recovery Mechanisms

- **Point-in-Time Recovery (PITR)**: Allows you to back out to the exact point prior to failure (e.g. immediately prior to a migration failure event).
- **Failover Server**: A standby server is also kept that would then succeed in case the first one failed.
- **Recovery Time Objective (RTO)**: ≤ 10 minutes.
- **Recovery Point Objective (RPO)**: 1 hour or less.

Tools and Automation

- Backup and replication with PostgreSQL pg_dump and pg_basebackup.
- Automated backups done via cron jobs or regular CI/CD pipelines.
- Real-time status and alerts (e.g., Prometheus, pgBackRest logs) monitoring tools.

Data Validation After Recovery

After restoration, automated tests (unit + data integrity checks) are run to verify:

- Referential integrity between tables
- Accuracy of recent bookings and payments
- Availability of user records and access logs

8.1.5 Data Security and Access Control

In a bid to safeguard sensitive data (e.g., login details, payment details) of the user, the Movie Theater Ticketing System (MTTS) may implement a layered standards-based security approach. The strategy is geared towards data protection at resting point, data in route, roles based access control and auditability..

Data Encryption

- **At Rest**: The strengths of all user passwords are hashed with a powerful algorithm (e.g., bcrypt, Argon2), and sensitive information (e.g., payment tokens) is encrypted with AES-256..
- **In Transit**: Any client-server communication is applied via HTTPS with TLS 1.2 or greater.

Authentication and Authorization

- **Authentication**: WT (JSON Web Tokens) would offer an authorized session management when users log in successfully.
- **Authorization**: We have Role-Based Access Control (RBAC). Movie scheduling, pricing, and reporting related routes are only accessible by certain authenticated admins. Customers will only see their bookings and their profile..

User Role	**Access Rights**
Customer	Register, login, view movies, book/cancel tickets
Admin	Manage listings, schedules, view reports, issue refunds

Secure API Design

- The authentication and user roles will be validated by middleware which protects all the API endpoints.

- Input validation and sanitization stop SQL Injection attacks and XSS issues.
- Rate-limiting and IP throttling to curb DDoS attacks and brute-force attacks are applied.

Database Access Controls

- The character that is giving write access to production database is only the backend service account.
- The character that is giving write access to production database is only the backend service account.
- Database users are restricted by the principle of least privilege.

Monitoring and Logging

- Each authentication attempt, booking, cancellation and payment is recorded along with user ID, IP address and timestamp.
- Inappropriate attempts are detected and cause temporary account locking.
- The logs under the security policy are retained for up to 6 months as required and are archived safely.

8.1.6 Data Lifecycle and Retention Policy

In an attempt to adhere to the principles of data protection and the efficiency of the system, MTTS implements definite rules regarding the period of storing data, the archiving of data as well as the deletion of data in the end.

Data Retention Periods

Data Type	**Retention Duration**	**Action After Expiry**
User Accounts	2 years of inactivity	Soft deletion with notification
Booking Records	5 years	Archived for audit purposes
Payment Records	7 years	Archived, then securely deleted
Audit Logs	6 months	Deleted after archival

Data Archival Process

- Older records than those that need to be retained are transferred to an archival database, which is optimized in terms of storage.
- Each archived data is encrypted and read-only and only auditors and admins have access.
- User may demand removal of his/her data according to the privacy laws (e.g. compliance with GDPR).

Data Deletion and Anonymization

- Personally identifiable information (PII) including name and email: upon the request of the user, PII is anonymized.
- The hard deletion is only after verifying window so that data cannot be accidentally lost.
- The erased information is also recorded and acknowledgement sent to the user via email to be transparent.

Summary of Data Management Approach:

- Decided to use PostgreSQL due to its relational and organized nature of data and good transactional integrity.
- Implemented normalized logical database consisting of single logical database and foreign key relationships.
- The access to data is maintained by layered APIs and transaction-safe endpoints.
- Back up, recovery and retention policies keep operations alive and within compliance laws.
- Any private and sensitive information is made to behave itself and is treated in a responsible, best practice fashion.

8.2 Tradeoff Discussion

When designing the Movie Theater Ticketing System (MTTS), a number of architectural and data related decisions were taken through project objectives, system requirements, scalability expectations and constrained resources. The section assesses the tradeoffs addressed in the selection of the data management technologies and organizational strategies.

8.2.1 SQL vs. NoSQL Database Architecture

Chosen Option: Relational SQL Database (PostgreSQL)

The Movie Theater Ticketing System (MTTS) will use a relational SQL database i.e., PostgreSQL database as the storage database to be used in the project. The basis of this decision is the characteristics of the data in the system which is highly structured, highly relational, and needs good consistency guarantees..

8.2.1.1 Justification:

The MTTS deals with structured things (Users, Bookings, Payments, Movies, Showtimes, and Seats) and each is related to another (e.g. n-to-1 between Users and Bookings, or 1-to-1 between Bookings and Payments). The application logic highly relies on the enforcement of these relationships and makes operations like booking a seat and canceling a ticket atomic, consistent, and durable. The mentioned characteristics scale well with the SQL systems adhering to the ACID model (Atomicity, Consistency, Isolation, Durability). The pre-eminence of postgreSQL amongst other relational systems is represented by:

- **Rich Feature Set**: Support for triggers, stored procedures, and advanced indexing.
- **Standards Compliance**: Full SQL compliance for query accuracy and compatibility.
- **Extensibility**: Native support for JSON, full-text search, and GIS extensions if future features are introduced (e.g., location-based theater search).
- **Performance**: Proven stability and scalability for medium to large-scale applications.

8.2.1.2 Alternatives Considered: NoSQL Databases

NoSQL databases like Mongodb and Firebase fire store was also put on the cards to compare, specifically the capabilities under schema less design and good throughput in distributed systems.

Potential Advantages of NoSQL:

- **Schema Flexibility**: It is beneficial in cases where the data model has been rapidly changing or unstructure.
- **Horizontal Scalability**: Suitable to be used in distributed architectures with automatic sharding.
- **Developer Agility**: Easy data insertion and modification not involving use of preset schemas.

However, limitations include:

- **Weaker Consistency**: Many NoSQL systems use eventual consistency, and may cause race conditions, with two users reserving the same seat.
- **Lack of Joins**: Need to denormalize data, or join at application level, necessitating more effort to develop and resulting in more redundant storage.
- **Limited Transactional Support**: Most NoSQL platforms do not allow complex, multi-document ACID transactions, although this situation is improving recently..

8.2.1.3 Tradeoff Summary:

Criteria	**SQL (PostgreSQL)**	**NoSQL (MongoDB, Firestore)**
Data Consistency	Strong ACID compliance	Eventual or limited consistency

Schema Flexibility	Fixed schema, strongly typed	Schema-less, flexible
Querying	Powerful joins, complex queries supported	Simple querying; joins must be handled manually
Transaction Handling	Robust support for atomic transactions	Limited or non-native transaction support
Use Case Fit	Ideal for structured, relational business logic	Better for real-time analytics or content platforms

8.2.2 Single vs. Multiple Databases

Chosen Option: Single Logical Database Schema

MTTS will be built using single logical model of relational database that integrated all fundamental elements into one schema. This data structure is centralized and all modules including User Management, Booking Engine, Payments, Showtime Scheduling and Reporting are incorporated.

8.2.2.1 Justification:

1. Simplified Maintenance and Security

- **Centralized Backups**: One and the same database make schedules of backup simpler, minimize the risk of incomplete backups, and make the recovery processes easier.
- **Uniform Security Policies**: Access control policies, encryption policies and logging policies are applied consistently between tables.

2. Operational Simplicity

- **Straightforward Joins**: Simple SQL queries by JOIN started in one table and ended in other tables (e.g., Users, Bookings, Payments)..
- **Transaction Management**: Processes that have more than one participant (e.g. a seat reservation and a payment) may be bundled into a single atomic transaction and decrease the complexity of handling the errors.

3. Cost Efficiency

- **Resource Optimization**: Less difficult to deal with resource distribution (RAM, CPU, storage) on one database engine.

- **Faster Development**: The overhead costs associated with developer are lower particularly in small-scale or mid-size applications.

8.2.2.2 Alternatives Considered: Microservices with Independent Databases

An architecture with independent databases per service based on microservices was checked against long-term scalability. In this type of architecture, a service such as Booking Service, Payment Service, User Service has its own schema and persistence layer..

Advantages of Microservices with Separate Databases:

- **Service Isolation**: Failure or scaling of individual components does not impact other components.
- **Technology Flexibility**: A microservice may use any kind of database it desires (e.g. use MongoDB to support real-time chat but use PostgreSQL to handle bookings).

Drawbacks:

- **Orchestration Complexity**: Orchestrating data between services is also overhead intensive where coordination activities are required in going through the multiple steps of booking and payment.
- **Cross-Service Transactions**: Needs distributed transaction protocols (e.g., Saga pattern), and such protocols are not easy to implement or maintain.
- **Data Duplication**: Shared data (e.g. movie metadata) could require duplication across services, creating the problem of potential inconsistency.

Tradeoff Summary:

Criteria	**Single Logical DB**	**Multiple Independent DBs**
Complexity	Low	High
Scalability	Medium (vertical/horizontal scaling)	High (per service)
Development Speed	Faster (easier to implement and debug)	Slower (requires orchestration, coordination)
Maintenance	Simple and centralized	Requires monitoring multiple systems
Transaction Consistency	Easy to enforce	Difficult to manage without distributed logic

8.2.3 Normalization vs. Denormalization

Chosen Approach: Normalized Schema (Up to Third Normal Form - 3NF)
In Movie Theater Ticketing System (MTTS), the normalization of the structure has been taken up to the third normal form (3NF). This data modeling technique will help structure the database in such a way that it minimizes redundancy, update anomalies and the data integrity is maintained by the use of foreign key relationships. The tables correspond to one subject or entity (e.g., Users, Movies, Showtimes, Bookings, Payments, Seats) and are cross-linked by primary and foreign key constraints.

Justification and Advantages

1. **Data Integrity and Consistency**
   Normalization would put a check on inconsistencies in other tables caused by an update in a table. By way of example, a change in the title of a movie only happens in one location and no duplicate or mismatch records is created on overlapping tables such as showing schedule or reservations.
2. **Efficient Storage**
   Redundant information is reduced. As an example, the user profiles are recorded only one time in the Users table and defined in other table by userID, which saves storage space and makes it easier to maintain..
3. **Accurate Querying and Relationship Mapping**
   Normalized tables produce exactly correct joins. As an example, to obtain all bookings of a film of a certain date, there must be trusted connections between the tables Bookings, Showtimes, and Movies.
4. **Modular Updates**
   Schema variants (e.g. adding new user attributes or fields that handle payments) become localized and less likely to have side effects or inconsistencies in other areas of the database.

8.2.3.1 Alternative Approach: Denormalized Schema for Analytics

Denormalization was one possibility, to make read-heavy parts of the system like reporting dashboards more readable, since the system could make common aggregate queries (e.g. revenue per show time, seat occupancy patterns).

Advantages of Denormalization:

- **Improved Read Performance**: Denormalized tables and views can be used to precompute the large reports that involve numerous joins.
- **Simplified Queries**: Eliminates the number of joins necessary in SQL queries hence queries are simple to write and run promptly.

Disadvantages of Denormalization:

- **Data Redundancy**: Results in repetition of user, movie, or showtime information and wastes the storage space.
- **Update Anomalies**: Update of a field (such as movie title) might have to be manually or via batch jobs in many tables.
- **Complex Maintenance**: Greater threat of inconsistency and more effort to maintain redundant data consistent.

8.2.3.2 Tradeoff Summary

Criteria	**Normalization (3NF)**	**Denormalization**
Data Consistency	High (enforced by relational constraints)	Medium (prone to inconsistencies)
Storage Efficiency	High	Low
Query Complexity	High (requires joins)	Low
Performance (Reads)	Moderate	High for pre-aggregated views
Use Case Fit	Best for transactional systems	Best for analytical/reporting systems

8.2.4 Security vs. Usability

Chosen Balance: Layered Security with Minimal Disruption

In the case of Movie Theater Ticketing System, the security model is guided by the need to ensure protection to sensitive information about users and transactions, and also offer the services that are seamless and easy to the user. The strategy is applied in its layered security measures at the backend and non intrusive frontend functionalities that direct and secure users.

8.2.4.1 Security Measures Implemented

1. **Role-Based Access Control (RBAC)**

- Differentiates between Admin and User roles.
- Administrative functions (e.g. change of show time, reports generation) only be allowed to authenticated users with high privileges..

2. **Secure API Communication**
   - All API-calls are secured through HTTPS with TLS-encryption, which secure confidentiality and integrity of the network-transferred data.
3. **Input Validation and Sanitization**
   - On the client-side, validations are instant (e.g., password length, required fields).
   - Server-side validations avert injection attacks, improper data entry and intrusions.
4. **Session Management**
   - In-authenticated sessions are managed via tokens and secure cookies and there are time-out periods on inactivity to help thwart session hijacking.
5. **Critical Action Confirmation**
   - In-authenticated sessions are managed via tokens and secure cookies and there are time-out periods on inactivity to help thwart session hijacking.

8.2.4.2 Usability Considerations

- **Intuitive Interface Design**: People are directed in the system through routine layouts, form validations, and tip-toebooks.
- **Minimal Interruptions**: The security mechanisms are applied without annoyance, as far as few prompts and unnecessary verification procedures are to be applied.
- **Responsive Error Handling**: Relevant messages are placed in case of unsuccessful validation or the system has a problem with the access.

8.2.4.3 Tradeoff Summary

Factor	**Security-Oriented Design**	**Usability-Oriented Design**
Data Protection	Strong encryption, backend enforcement	Less visible to user
User Interruptions	Required for critical actions (e.g., cancel)	Minimized for routine tasks

Complexity for Developers	Higher (RBAC, validation, encryption)	Medium
End-User Experience	Slight friction in sensitive areas	Smooth in general workflows
MTTS Approach	Balanced: Security enforced behind the scenes	UI designed for ease of use

8.3 Security, Backup, and Privacy Considerations

8.3.1 Security Strategy

Data that the Movie Theater Ticketing System (MTTS) deals with may be confidential in the sense that they contain personal information, logins, and credits cards. In this capacity, a rigorous security framework has been incorporated in the design and in the information management chain. Security strategy is established based on 4 core domains, which include data protection, authentication, access control, secure communication, and incident response.

1. Data Protection

- **Encryption-at-Rest**: User data, its booking history, and payment logs are placed in an encrypted PostgreSQL database with AES-256 encryption.
- **Encryption-in-Transit**: The communication protocol between the frontend, backend, and third-party APIs (e.g. payment gateways) is HTTPS over TLS 1.3.
- **Password Hashing**: User passwords can be hashed by means of bcrypt with salts in order to thwart rainbow table attacks or reverse-engineering.

2. Authentication and Access Control

- **Role-Based Access Control (RBAC)**: two user roles (Customer and Admin) have limited privileges associated with each role. Admins are able to control schedules and reporting, Customers book and cancel tickets.
- **JWT Tokens**: JWTs are used, also via a secure server key, to provide stateless session management.
- **Login Attempt Throttling**: To prevent brute force attacks a rate-limiting policy is enforced on the APIs, and a lock out policy is enforced after five unsuccessful consecutively attempted logins.

3. Secure APIs and Input Validation

- **Sanitization**: User input and API output are sanitized on the server side to avoid SQL injections, cross-site scripting (XSS) and command injection attacks.
- **Secure Headers**: X-Content-Type-Options, X-Frame-Options, and Content-Security-Policy HTTP headers are enforced.

4. Incident Handling and Monitoring

- **Audit Logs**: Sensitive actions such as cancellations, logins by administrators are time stamped and user references.
- **Intrusion Detection**: Web Application Firewall (WAF) is considered and optional intrusion detection systems such as Snort or Suricata are also considered in production use.
- **Alerting**: Alerts are set on the basis of suspicion, i.e., the mass failure of logging in or the unusual eruption of traffic.

8.3.2 Backup and Recovery Plan

Data durability and high speed of recovery under failure of the system or disasters is pivotal to the reliability of the MTTS platform. A backup and recovery plan has been integrated to serve as a comprehensive process, including redundancy, automated, frequent backup, and testing.

Backup Frequency and Retention

- **Full Backups**: Full backups of the PostgreSQL database occur every day at a time of 2:00 AM in the server.
- **Incremental Backups**: These are back up done within four hours to capture what has changed during the day without making unnecessary duplicates of entire sets of data.
- **Retention Policy**: 30 days retention of backups. The daily backups are kept up to 7 days, weekly backups up to 4 weeks and monthly up to 3 months.

Storage Redundancy

- **Cloud Redundancy**: Automatic Server Backups to geographically distant cloud storage location provided by services such as Amazon AWS S3 or Google Cloud Storage.
- **On-Premises Copy (Optional)**: On hybrid deployment, one copy is stored on-site where the high availability is available on encrypted storage with RAID 1 mirrored status.

Automated and Manual Testing

- **Restore Testing**: The integrity of the backup files should be tested periodically by carrying out test restores and testing the restoration procedures to make sure they are functioning correctly.
- **Checksum Verification**: The backup files are verified with checksums in order to identify corruption or tampering.

4. Recovery Time and Recovery Point Objectives

- **RTO (Recovery Time Objective)**: The system will recoup full functionality within 60 minutes of which a failure occurs.
- **RPO (Recovery Point Objective)**: The maximum loss of data limitation is limited to up to 4 hours according to the daily backup scheme.

5. Disaster Recovery

- An official Disaster Recovery (DR) playbook contains failovers to a standby server, restoring database, re-routing DNS and contacting team procedures.
- The plan of the DR is revised on a quarterly basis with changes being updated when infrastructure or business requirements change.

8.4 Data Flow & Storage Operations

It explains the flow of data into and through the system, including user interactions on one end and storage into the database on the other, and how storage operations are performed at every point. Movie Theater Ticketing System (MTTS) data management pipeline guarantees the transactional consistency, secure manageability of the data, and system reliability of the workflows both in the user environments and in the administrative process..

8.4.1 Data Flow Overview

The MTTS adheres to a model of three tiers and includes:

- **Presentation Layer (Frontend)**
  UI is Web-based (React/HTML/CSS) through a browser. It reads user data and shows system outputs.
- **Application Layer (Backend)**
  Django or Node.js business logic. Input validation, session management, transaction processing, enforcement of security. Data Layer (Database)

- **Data Layer (Database)**
  The relational database PostgreSQL where persistent data is stored (e.g. users, bookings, payments, movies, etc.)

8.4.2 User Interaction to Storage Flow

The following is a step by step account of how an average booking transaction travels through the system:

1. **User Input via Web Interface**
   A user can log in, browse movies, choose a showtime, and choose seats..
2. **API Request to Backend**
   The frontend cooperation makes a HTTPS request to the backend API (e.g., POST /bookings).
3. **Validation & Business Logic**
   - Backend verifies the user and input (e.g., showtime is present, the seats are available)
   - Back end initiates a database transaction.
4. **Database Write Operation**
   - Seat status (isBooked) is updated.
   - A new entry is created in the Bookings table.
   - Payment is initiated and written to the Payments table upon confirmation.
5. **Commit/Rollback**
   - When all operations are successful the database transaction is committed.
   - In case of any mishap (say in case of payment failure), the transaction gets rolled back so that it is not partially booked.
6. **User Feedback**
   - Backend returns a response confirming success/failure.
   - Frontend shows a booking confirmation or a corresponding error message.

8.4.3 CRUD Operation Mapping

Operation	**Method**	**Table(s) Affected**	**Example Endpoint**
Create Booking	INSERT	Bookings, Payments, Seats	POST /bookings

Read Showtimes	SELECT	Movies, Showtimes, Seats	GET /showtimes
Update Booking	UPDATE	Bookings	PUT /bookings/:id
Delete Booking	DELETE	Bookings, Payments	DELETE /bookings/:id

ACID principles are used in all database operations and affect atomicity (or nothing at all), consistency (a valid state), isolation (non-interference) and durability (if committed, it is saved).

8.4.4 Data Retention & Storage Policies

- **User Data**: Stored permanently until removed by user request, based on the data privacy laws.
- **Booking & Payment Records**: They are maintained at least 1 year to be used in Audit/ refunding.
- **Reports**: created every day and saved (archive) as CSV files or views in the database until 6 months.

Data is stored nightly in a secure manner by encryption-at-rest policies on database server. Weekly tests are done on the restorability of the backups.

8.4.5 Flow Diagram (Textual Representation)

[User]
  ↓ *(HTTPS Request)*
[Frontend Web UI]
  ↓ *(POST/GET)*
[Backend API - Node.js/Django]
  ↓ *(SQL Transaction)*
[PostgreSQL Database]
  ↓ *(Write/Read)*
[Confirmation or Error to User]

8.4.6 External Data Communication

- **Payment Gateway (e.g., Stripe)**:

  Backend sends secure tokenized requests to payment API.

  Response determines if booking can proceed.
- **Email/SMS Notification (e.g., SendGrid/Twilio)**:

  After booking, system generates a receipt and sends it via API to the user's contact.

A. APPENDICES

A.1 Appendix A – Glossary of Terms

Term	**Definition**
MTTS	*Movie Theater Ticketing System* – The software system described in this document, supporting movie booking, management, and reporting.
SRS	*Software Requirements Specification* – A formal document detailing functional and non-functional requirements of the MTTS.
GUI	*Graphical User Interface* – The visual interface that allows users (admins and customers) to interact with the MTTS system via web browsers.
Admin	An authorized user role with elevated privileges to manage movies, showtimes, pricing, and generate reports within MTTS.
Customer	An end-user or moviegoer who uses MTTS to register, browse movies, select seats, book tickets, and manage bookings.
Showtime	A scheduled screening of a specific movie in a designated auditorium at a specified date and time.
Booking	A confirmed reservation of one or more seats by a customer for a particular showtime.
E-ticket	An electronic ticket generated upon successful payment, containing booking details and sent via email to the customer.
UML	*Unified Modeling Language* – A standardized visual modeling language used to describe class structures and system behavior.
Commit	An action in version control (Git/GitHub) that records a set of changes to a repository, providing traceability and collaboration.
Change Request	A formal submission made to propose modifications to the system’s requirements, design, or scope.

API	*Application Programming Interface* – Interfaces through which MTTS interacts with external services such as payment gateways or email systems.
Seat	A physical seat in an auditorium. Represented digitally in MTTS for selection and availability tracking.
Auditorium	A designated theater room where movies are screened. Each auditorium has a unique layout and ID in the system.
Payment Gateway	A third-party service that processes online transactions securely (e.g., Stripe, PayPal).
PostgreSQL	The chosen relational database management system for MTTS, supporting structured queries and transaction integrity.
Functional Requirement (FR)	A specification describing a specific function or behavior the MTTS system must perform (e.g., register user, generate report).
Non-Functional Requirement (NFR)	A system-wide requirement related to performance, security, usability, or reliability.
Traceability Matrix	A tabular structure linking system requirements to test cases to ensure full validation and coverage.
Materialized View	A precomputed view of data stored in the database to optimize query performance for read-heavy operations.
Data Normalization	A database design strategy to reduce data redundancy and maintain referential integrity by organizing data into related tables.

A.2 References

1. IEEE. (1998). *IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998)*. The Institute of Electrical and Electronics Engineers, Inc. https://standards.ieee.org/standard/830-19S98.html
2. Sommerville, I. (2015). *Software Engineering* (10th ed.). Boston, MA: Addison-Wesley. ISBN: 978-0133943030
3. Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education.
   ISBN: 978-0078022128
4. ISO/IEC/IEEE 29148:2018. *Systems and software engineering — Life cycle processes — Requirements engineering*.
   https://www.iso.org/standard/72089.html
5. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
   ISBN: 978-0201633610
6. Glinz, M. (2007). *On Non-Functional Requirements*. In *15th IEEE International Requirements Engineering Conference (RE 2007)*.
   https://doi.org/10.1109/RE.2007.45
7. Hanna, G. (2025). *CS250 Course Lecture Notes and Software Requirements Template*, Department of Computer Science, [Your University Name].
8. Hanna, G. (2025). *CS250 Use Case Lecture Examples*, [Course Handouts].
9. Fowler, M. (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison-Wesley.
10. Beizer, B. (1995). *Software Testing Techniques* (2nd ed.). Van Nostrand Reinhold.
11. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley.
    ISBN: 978-0321815736
12. Ambler, S. W. (2005). *The Elements of UML 2.0 Style*. Cambridge University Press.
    ISBN: 978-0521540186

13. Elmasri, R., & Navathe, S. B. (2015). *Fundamentals of Database Systems* (7th ed.). Pearson Education.
ISBN: 978-0133970777
14. Kienzle, D., Elder, M. C., Tyree, D., & Edwards-Hewitt, J. (2002). *Security Patterns Repository: Design for Secure Software*.
Retrieved from: https://buildsecurityin.us-cert.gov
15. Sommerville, I. (2007). *Designing Software Architectures: A Practical Approach Using ADLs*. Addison-Wesley.
16. Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.
ISBN: 978-0201571691
17. Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd ed.). Addison-Wesley.
ISBN: 978-0321267979
18. ISO/IEC 25010:2011. *Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models*.
Retrieved from: https://www.iso.org/standard/35733.html
19. IEEE Std 1012-2012. *IEEE Standard for System and Software Verification and Validation*.
Retrieved from: https://standards.ieee.org/standard/1012-2012.html
20. Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing* (3rd ed.). Wiley.
ISBN: 978-1118031964

A.3 Summary of Changes – Version 4

Section No.	Section Heading	Summary of Changes Made
6.2	Software Architecture Overview	Updated the architectural diagram to include the database server, external APIs (Payment, Email), and system flow.
6.3.3	Design Changes from Assignment z2	Added explanation that no structural changes were made to the UML diagram; minor additions clarified relationships and multiplicities.
7	Test Plan	Verified that the test plan section includes objectives, strategy, features tested, test deliverables, traceability matrix, and test cases.
7.6	Traceability Matrix	Introduced a traceability matrix linking functional requirements to their respective test cases and types.
4.2	Software Architecture Diagram (Updated)	Added text-based version of updated system architecture with frontend-backend-database interaction flow.
8.1	Data Management Strategy	Comprehensive explanation of database strategy, schema design, normalization, and rationale for SQL over NoSQL.
8.2	Tradeoff Discussion	Detailed comparison between SQL vs. NoSQL, single vs. multiple databases, normalization vs. denormalization, and security vs. usability.
8.3	Data Flow and Storage Operations	Described how data moves through the system with storage operations, using narrative and diagrammatic explanation.
3.6	Inverse Requirements	Added inverse requirements such as operations the system must prevent (e.g., double booking, unauthorized access).
3.7	Design Constraints	Documented constraints related to hosting platform, browser support, API dependencies, and response time requirements.

3.8	Logical Database Requirements	Clarified requirements for database type, ACID compliance, referential integrity, and expected data volume.
3.9	Other Requirements	Catch-all section added to list backup/recovery, audit logging, and extensibility requirements.
4	Analysis Models	Section restructured to organize different types of analysis models used in the design and specification.
4.1	Sequence Diagrams	Described system workflows (booking, payment, admin functions) in step-by-step format; diagrams presented as text.
4.2	State-Transition Diagrams	Provided detailed transitions for booking status changes (available, locked, booked, cancelled).
4.3	Data Flow Diagrams (DFD)	Added Level 0 and Level 1 DFDs to describe high-level data processes and their interactions.
5	Change Management Process	Added formal procedure for submitting, reviewing, approving, implementing, and documenting changes to the SRS.