**SM CINEMA BOOKING SYSTEM**

A Deployment Documentation Presented to the

Faculty of Datamex College of Saint Adeline, Inc.

In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science in Information Technology

By:

Jerusalem, Auldey Glen A.

October 2025

**DEPLOYMENT DOCUMENTATION**

**INTRODUCTION**

The Web-Based Cinema Booking System for SM Cinema Grand Central is a full-stack web application designed to simplify the movie ticket reservation process for customers and provide operational management tools for cinema administrators. The system allows users to browse movie listings, select screening times, and choose specific seats in real-time. On the administrative side, the platform provides a dashboard for managing movie schedules and monitoring reservations. The purpose of this system is to enhance customer convenience, reduce manual booking processes, and improve overall operational efficiency for SM Cinema.

The primary objective of this deployment is to make the Cinema Booking System publicly accessible and fully operational for end-users and administrators. This deployment aims to transition the application from a local development environment to cloud-based production servers, ensuring reliable performance, data persistence, scalability, and security. By deploying on industry-standard platforms (Vercel, Render, Supabase, Upstash Redis, and Cloudinary), the system will provide real-time seat availability, secure data management, and responsive user experience across desktop and mobile browsers.

This is a full deployment of the Cinema Booking System, covering both frontend and backend components. The deployment scope includes:

* Frontend deployment on Vercel (React + Vite)
* Backend API deployment on Render (Django REST Framework)
* Database hosting on Supabase (PostgreSQL)
* Caching implementation using Upstash Redis for request throttling
* Media storage configuration on Cloudinary for movie posters and related assets

The deployment is executed in a staged manner, with frontend and backend deployed separately but integrated through API endpoints. Post-deployment activities include functionality testing, performance monitoring, and addressing any critical issues that arise.

**DEPLOYMENT PLAN**

The deployment strategy for the Cinema Booking System follows a phased approach that prioritizes backend stability before frontend integration. The process began with environment preparation, where accounts were created and configured across multiple cloud platforms including Render, Vercel, Supabase, Upstash Redis, and Cloudinary.

The backend API was deployed first on Render to establish a stable foundation for data handling and business logic. Once the backend was confirmed operational, the frontend was deployed on Vercel and connected to the backend through API endpoints. This approach allowed for continuous development and testing in a live environment, where bugs and performance issues could be identified and resolved in real-time.

Database migration, caching setup, and media storage integration were handled during the initial deployment phase to ensure all services were properly connected before extensive user-facing testing began. The deployment maintains an ongoing testing cycle, where functionality is monitored and improvements are applied as needed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Phase | Description | Start Date | End Date | Status |
| Pre-Deployment | Account creation and configuration across deployment platforms, environment variable setup, and service integration planning | 08/20/2025 | 08/22/2025 | Completed |
| Deployment | Backend deployment on Render, frontend deployment on Vercel, database and caching service connection, media storage configuration | 08/23/2025 | 09/01/2025 | Completed |
| Post-Deployment | Live system testing, bug identification and resolution, performance monitoring, ongoing adjustments | 09/01/2025 | Ongoing | In Progress |

Table 1. Deployment Schedule and Milestones

**DEPLOYMENT ENVIRONMENT**

This section outlines the technical requirements and infrastructure configuration necessary for the successful deployment and operation of the Cinema Booking System. It covers both client-side hardware specifications and server-side hosting architecture.

**Hardware Requirements**

**Client Devices (End Users):**

* Processor: Intel Core i3 or AMD Ryzen 3 (or equivalent)
* Memory: Minimum 4GB DDR4 RAM
* Storage: 256GB SSD or HDD
* Display: Minimum resolution of 1366x768
* Network: Stable internet connection via Ethernet or WiFi
* Browser: Latest versions of Chrome, Firefox, Safari, or Edge

The server infrastructure relies entirely on cloud-hosted platforms. The backend runs on Render while the frontend is hosted on Vercel, both utilizing free-tier resources. Database operations are handled by Supabase, caching services are provided by Upstash Redis, and media storage is managed through Cloudinary. All of these services operate on cloud infrastructure, eliminating the need for physical server hardware.

**Software Requirements**

On the client side, the system is accessible from any operating system that supports modern web browsers, including Windows, macOS, Linux, iOS, and Android. The application is entirely browser-based, requiring no additional software installation from users. Server-side operations run on Linux-based containers that are automatically managed by the respective cloud hosting providers.

The frontend is built using React 18.x with Vite as the build tool, leveraging JavaScript ES6+ features along with HTML5 and CSS3 standards. Styling and responsive design are handled through Tailwind CSS, ensuring the interface adapts properly across different screen sizes and devices. The application maintains compatibility with all major modern browsers in their latest versions.

For the backend, the system runs on Python 3.12, which was the latest stable version available as of August 2025. The API is built using Django REST Framework version 3.15.x, also the most current stable release at the time of development. Data persistence is managed through PostgreSQL version 16.x, which provides the relational database functionality required for storing movie schedules, reservations, and user data.

Several supporting dependencies and frameworks are integrated into the system. Django CORS Headers enables secure cross-origin resource sharing between the frontend and backend. Django REST Framework's built-in authentication and permissions modules handle access control. On the frontend, React Router manages page navigation while Axios handles HTTP requests to the backend API. Additional libraries and packages are included as specified in the project's requirements files.

**Hosting Information**

The system deployment is distributed across multiple cloud platforms, all operating on free-tier subscription plans. This approach keeps operational costs at zero while still providing reliable service for the intended use case.

**Frontend Hosting:**

* Platform: Vercel
* Deployment Type: Automatic deployment from Git repository
* Domain: cinema-booking-system-chi.vercel.app
* Tier: Free

**Backend API Hosting:**

* Platform: Render
* Deployment Type: Web service with automatic deployment
* Domain: cinema-booking-system-wlkj.onrender.com
* Tier: Free

**Database Hosting:**

* Platform: Supabase
* Database Type: PostgreSQL 16.x
* Configuration: Managed cloud database with automated backups
* Tier: Free

**Caching Service:**

* Platform: Upstash Redis
* Purpose: Request throttling and performance optimization
* Tier: Free

**Media Storage:**

* Platform: Cloudinary
* Purpose: Storage and delivery of movie posters and related media assets
* Tier: Free

Render's free tier includes automatic sleep after periods of inactivity, which may result in slightly longer initial response times for the first request after idle periods. Supabase provides connection pooling and automated backups to ensure data integrity without requiring manual database administration. Cloudinary serves media files through their content delivery network, optimizing image delivery based on the requesting device and connection speed.

**DEPLOYMENT PROCEDURES**

This section provides a detailed step-by-step guide on how the Cinema Booking System was deployed to production environments. The deployment process is divided into three main phases: pre-deployment preparation, deployment execution, and post-deployment verification.

**4.1 Pre-Deployment Steps**

**Backup Existing Data**

Since this was a fresh deployment with no prior production data, no backup procedures were necessary. The system was deployed from scratch without migrating any existing information.

**Set Up the Required Environment**

The pre-deployment phase involved preparing both local development environments and cloud service accounts. Local installations included Node.js for running the React Vite frontend and Python for the Django REST Framework backend. These local setups allowed for thorough testing before pushing code to production servers.

Cloud service accounts were created and configured across multiple platforms. GitHub was used for version control and repository hosting. Vercel and Render accounts were set up for frontend and backend hosting respectively. Supabase was configured for database management, Upstash Redis for caching services, and Cloudinary for media storage. Each platform required initial configuration and connection to the GitHub repository for automatic deployment capabilities.

Environment variable files were prepared for both frontend and backend applications. These files contained sensitive configuration data such as API keys, database connection strings, and service credentials. A main branch was established in the GitHub repository to serve as the production deployment source. All code changes were tested locally before being pushed to this branch to trigger automatic deployments.

**Database Configuration**

The database setup involved creating a PostgreSQL instance on Supabase and obtaining the connection URL. This database was configured separately from the backend application and later connected through environment variables in the Render deployment settings. The separation of database hosting from application hosting provides better resource management and data persistence even if the application server restarts.

**Ensure Network Connectivity and System Compatibility**

Local testing confirmed that all components could communicate properly before deployment. The frontend was tested to ensure it could successfully make API requests to the backend. Database connections were verified locally using the Supabase connection string. Cloudinary and Upstash Redis integrations were tested to confirm proper configuration. All compatibility checks passed successfully before proceeding to live deployment.

**4.2 Deployment Execution**

**Backend Deployment on Render**

The backend API was deployed by connecting the GitHub repository to Render's web service platform. Render was configured to automatically deploy from the main branch whenever new commits were pushed. The following environment variables were

configured in the Render dashboard:

* ALLOWED\_HOSTS: Specifies which domains can access the API
* CLOUDINARY\_API\_KEY, CLOUDINARY\_API\_SECRET, CLOUDINARY\_CLOUD\_NAME: Credentials for media storage
* DATABASE\_URL: Connection string for the Supabase PostgreSQL database
* DEBUG: Set to False for production environment
* FRONTEND\_URL: The Vercel domain for CORS configuration
* PYTHON\_VERSION: Specifies Python 3.12
* SECRET\_KEY: Django secret key for cryptographic signing
* UPSTASH\_REDIS\_URL: Connection string for Redis caching service
* USE\_CLOUDINARY: Boolean flag to enable Cloudinary integration

Build and start commands were configured to handle dependency installation, static file collection, database migrations, and application startup. The build command executes pip install -r requirements.txt && python manage.py collectstatic --noinput && python manage.py migrate, which installs all required Python packages, collects static files for serving, and applies database schema migrations. The start command gunicorn config.wsgi:application launches the application server using Gunicorn.

**Frontend Deployment on Vercel**

The frontend was deployed by connecting the GitHub repository to Vercel's platform. Similar to Render, Vercel was configured for automatic deployment from the main branch. The build command was set to npm run build with an install command of npm install. The output directory was specified as dist, which is where Vite places the compiled production files.

Environment variables configured in Vercel include:

* VITE\_API\_URL: Points to the backend API domain on Render
* VITE\_FAVICON\_URL: URL for the site favicon
* VITE\_LOGO\_URL: URL for branding logo
* VITE\_REF\_COMPANY: Reference company name for display purposes

**Service Integration**

Connecting external services to the backend was straightforward. Supabase integration involved copying the database connection URL and adding it to Render's environment variables as DATABASE\_URL. Django automatically handles PostgreSQL connections using this URL. Upstash Redis integration followed a similar process, with the REST API URL added as UPSTASH\_REDIS\_URL. The backend application consumes this URL to establish caching connections. Cloudinary integration required three separate credentials (API key, API secret, and cloud name), which were added to the environment variables. The backend uses these credentials to upload and retrieve media files.

**Database Initialization and System Configuration**

After the initial deployment, database migrations were executed automatically through the build command. Django's migration system created all necessary tables and schema structures in the Supabase PostgreSQL database. A superuser account was created manually using Django's built-in command through the Render shell interface. This administrative account provides access to the Django admin panel and full system management capabilities. The superuser creation is built into Django REST Framework and only needs to be performed once after initial deployment.

**4.3 Post-Deployment Steps**

**Functionality Verification**

Testing procedures focused on verifying core system functionality across different environments. All public-facing pages were accessed and tested on both desktop and mobile devices to ensure responsive design worked correctly. The movie browsing interface was tested to confirm proper display of listings and details. Seat selection functionality was verified to ensure real-time availability updates and proper booking flow. The administrative dashboard was tested to confirm proper authentication and management capabilities.

Cross-browser compatibility was checked using Chrome, Firefox, and Safari. Mobile testing included both iOS and Android devices to ensure consistent experience across platforms. All test cases passed successfully, confirming that the deployment was functional and ready for use.

**Performance Monitoring and Stability**

System performance is monitored through built-in platform tools provided by Vercel and Render. Vercel's web application dashboard provides analytics showing page load times, visitor statistics, and deployment history. The observability tools display real-time metrics and error logging for the frontend application.

Render's dashboard offers similar monitoring capabilities for the backend API. The logs section displays application output and error messages in real-time. Events tracking shows deployment history and system status changes. Metrics provide insights into memory usage, CPU utilization, and response times. These monitoring tools allow for quick identification and resolution of any performance issues or errors that arise during operation.

**User Training and Support**

User training requirements are minimal due to the system's design. The customer-facing interface is intuitive and does not require user accounts or authentication, making the booking process straightforward and accessible. Customers can browse movies, select seats, and complete reservations without prior instruction or training.

Administrative training focuses on the staff members who manage movie schedules and monitor reservations. Training materials cover how to access the admin dashboard, add or modify movie listings, update showtimes, and view booking reports. Since the admin interface follows standard Django conventions, users familiar with basic web applications can navigate it with minimal guidance. Support documentation includes screenshots and step-by-step instructions for common administrative tasks.

**USER TRAINING & SUPPORT**

This section outlines the training approach, documentation resources, and support channels available to system users. Given the intuitive design of the customer interface and the straightforward nature of the administrative dashboard, training requirements are minimal and focused primarily on administrative staff.

**Training Schedule for Users**

No formal training sessions were scheduled for end-user customers, as the public-facing booking interface is designed to be self-explanatory and requires no prior instruction. The seat selection process, movie browsing, and reservation workflow follow common web application patterns that most users are already familiar with.

For administrative staff who manage the system, informal training was provided on an as-needed basis. This included demonstrations of the admin dashboard functionality, guidance on adding and updating movie schedules, and instruction on monitoring reservation activity. The training approach was practical and hands-on, allowing administrators to learn while performing actual system management tasks.

**Documentation or Manuals Provided to Users**

Basic system documentation is available through the administrative interface itself. The Django admin panel includes built-in help text and field descriptions that guide users through common tasks. Context-sensitive labels and form validations provide immediate feedback to prevent errors during data entry.

For customers, no separate documentation is required as the booking interface uses clear visual cues, tooltips, and straightforward navigation. Button labels and page headings clearly indicate available actions, making the reservation process intuitive without requiring external reference materials.

**RISKS & CONTINGENCY PLAN**

This section identifies potential risks that may impact the deployment and operation of the Cinema Booking System, along with their anticipated impact levels and corresponding mitigation strategies. Understanding these risks allows for proactive planning and quick response when issues arise.

**Potential Deployment Issues and Solutions**

|  |  |  |
| --- | --- | --- |
| Risk | Impact | Mitigation Strategy |
| Render Free Tier Cold Starts | High | Accept initial delay as trade-off for free hosting; consider implementing a scheduled health check to keep service active during peak hours; inform users that first load may take 30-60 seconds |
| Vercel/Render Service Outages | High | Monitor platform status pages regularly; communicate any known downtime to users in advance; rely on platform's built-in redundancy and recovery procedures |
| Database Connection Limits (Supabase Free Tier) | Medium | Implement connection pooling in Django settings; monitor concurrent connection usage through Supabase dashboard; optimize queries to reduce connection duration |
| API Rate Limiting | Medium | Utilize Upstash Redis for request throttling; monitor API usage through Render metrics; implement caching strategies to reduce redundant requests |
| Environment Variable Misconfiguration | High | Maintain backup documentation of all environment variables; test deployments in staging branch before pushing to production; verify all variables after each configuration change |
| CORS Configuration Issues | High | Ensure ALLOWED\_HOSTS and FRONTEND\_URL are properly configured in backend settings; test cross-origin requests during deployment verification; maintain updated CORS headers configuration |
| Media Upload Failures (Cloudinary) | Medium | Implement error handling for failed uploads with user-friendly messages; monitor Cloudinary storage usage and bandwidth limits; validate file types and sizes before upload attempts |
| Concurrent Seat Booking Conflicts | High | Implement database-level transaction locking for reservation operations; use Django's select\_for\_update() to prevent race conditions; display real-time seat availability updates |
| Browser Compatibility Issues | Low | Test application across major browsers during development; use standard web technologies with broad support; provide graceful degradation for older browsers |
| Mobile Responsiveness Problems | Low | Utilize Tailwind CSS responsive utilities consistently; test on various device sizes throughout development; conduct cross-device testing before major releases |
| Slow Page Load Times | Medium | Optimize image delivery through Cloudinary's CDN; minimize JavaScript bundle sizes through code splitting; implement lazy loading for below-the-fold content |
| Data Migration Errors | Low | Run migrations in test environment first; maintain database backups before applying schema changes; use Django's migration rollback capabilities if issues occur |

Table 2. Risk Assessment and Mitigation Strategies

Each identified risk has been evaluated based on its potential impact on system functionality and user experience. The mitigation strategies outlined provide practical approaches to minimize disruption and maintain service reliability throughout the system's operational lifecycle.

**DEPLOYMENT VERIFICATION & SIGN-OFF**

This section documents the successful completion of deployment testing and provides formal acknowledgment from project stakeholders that the system has been deployed and verified according to specifications

**Summary of Successful Deployment Tests**

The deployment verification process involved comprehensive testing of all core system functionalities across multiple environments and devices. All critical components were tested and confirmed to be operating as expected in the production environment.

The movie listings interface was verified to display current films with accurate information including titles, descriptions, genres, and showtimes. Navigation between different movie details and screening schedules functioned correctly without errors. The seat selection functionality was thoroughly tested to ensure proper display of theater layouts, real-time availability updates, and accurate seat reservation capabilities. Multiple booking scenarios were simulated to confirm that the selection logic prevented double-booking and maintained data integrity throughout the reservation process.

The complete booking workflow was tested from start to finish, verifying that users could successfully browse movies, select preferred showtimes, choose available seats, and complete reservations. Each step of the process was confirmed to provide appropriate feedback and guidance to users. The administrative dashboard was accessed and tested to ensure proper authentication, authorization, and full management capabilities. Admin users were able to add new movies, update showtimes, modify schedules, and view reservation data without encountering access issues or functional limitations.

Responsive design implementation was validated across various screen sizes and devices. The interface was tested on desktop computers, tablets, and mobile phones to confirm that layouts adapted appropriately and all functionality remained accessible regardless of device type. Touch interactions on mobile devices were verified to work smoothly for seat selection and navigation.

Cross-browser compatibility testing was conducted using Chrome, Firefox, Safari, and Edge browsers. The application displayed consistently across all tested browsers with no significant rendering issues or functionality gaps. Database operations were tested extensively to ensure proper data persistence, retrieval, and modification. Create, read, update, and delete operations for movies, showtimes, and reservations all functioned correctly without data loss or corruption.

Media file delivery from Cloudinary was verified to ensure movie posters and related images loaded properly and displayed at appropriate quality levels. Image optimization and CDN delivery performed as expected, contributing to acceptable page load times. API endpoint functionality was tested to confirm proper communication between the frontend and backend. All REST API calls returned expected responses with correct status codes and data formats. Error handling was verified to provide meaningful feedback when issues occurred.

Performance monitoring through Vercel and Render dashboards confirmed stable operation with acceptable response times. No critical errors or service disruptions were observed during the testing period. All deployment verification tests completed successfully, confirming that the system is ready for operational use.