Project – High Level Design
on

IaC Provisioning for Finance System

Course Name: Cloud Infrastructure & DevOps
Institution Name: Medicaps University – Datagami Skill Based Course

Student Name(s) & Enrolment Number(s):

Sr No	Student Name	Enrolment Number
1	Aadarsh Adwani	EN22EL301001
2	Yash Prasad Shrivastava	EN22CS3011107
3	Yash Wadhwani	EN22CS3011112

Group Name:	10D10
Project Number:	DO-23
Industry Mentor Name:	
University Mentor Name:	Prof. Avnesh Joshi
Academic Year:	2025 – 26

Table of Contents

1. Introduction

This document presents the High Level Design (HLD) for the **IaC Provisioning for Finance System** project, developed as part of the Medicaps University – Datagami Skill Based Course. The underlying system is **SpendSmart**, a full-stack personal finance tracking application that enables users to manage incomes and expenses. SpendSmart is built on the MERN stack (MongoDB, Express.js, React, Node.js) and is fully containerised using Docker and Nginx for production-ready deployment. Infrastructure provisioning, environment configuration, and CI/CD automation are treated as code throughout the project.

1.1. Scope of the Document

This document covers the high-level architectural and design decisions for the IaC Provisioning for Finance System project. It describes the system components of the SpendSmart application, their data flows, external interfaces, and non-functional requirements. It serves as the primary reference for development, review, and deployment teams. Low-level design, code-level documentation, and operational runbooks are outside the scope of this document.

1.2. Intended Audience

This document is intended for students and developers working on the project, university and industry mentors reviewing the design, DevOps engineers responsible for containerised deployment, and evaluators assessing the project for the Medicaps University – Datagami Skill Based Course.

1.3. System Overview

SpendSmart is a decoupled, RESTful web application built on the MERN stack. The React + Vite Single Page Application (SPA) communicates with an Express.js backend over versioned API endpoints (/api/v1/). MongoDB, accessed via Mongoose, serves as the persistent data store. Nginx acts as both a static-file web server and a reverse proxy, routing /api/* requests to the Node.js backend container. The entire application is containerised using a multi-stage Docker build, resulting in a compact, production-hardened Nginx + Node image pair.

Problem Statement: Design and provision the infrastructure for a full-stack Finance Tracking System (SpendSmart) using containerisation and modern DevOps practices. The project involves Docker-based deployment with Nginx reverse proxy, automated dependency management, JWT-based authentication, and a scalable MERN architecture — eliminating manual configuration errors and enabling repeatable, environment-parity deployments.

2. System Design

2.1. Application Design

The application follows a three-tier decoupled architecture comprising the Presentation Layer, the Application Logic Layer, and the Data Storage Layer — all orchestrated within Docker containers.

Layer	Tools / Components	Responsibility
Presentation & Proxy Layer	React 19, Vite, Nginx (Alpine)	Serves the React SPA as static files; proxies all /api/* requests to the Express backend container.
Application Logic Layer	Node.js, Express.js, JWT, Cookie Parser, CORS	Exposes versioned RESTful API endpoints; handles authentication, business logic, and routing.
Data Storage Layer	MongoDB (via Mongoose)	Persists user accounts, income records, and expense records.
State Management	Redux Toolkit, React-Redux	Manages global client-side state (auth session, income/expense arrays).
Infrastructure & DevOps	Docker (multi-stage), Nginx, ESLint	Containerises frontend and backend; multi-stage build minimises image size.

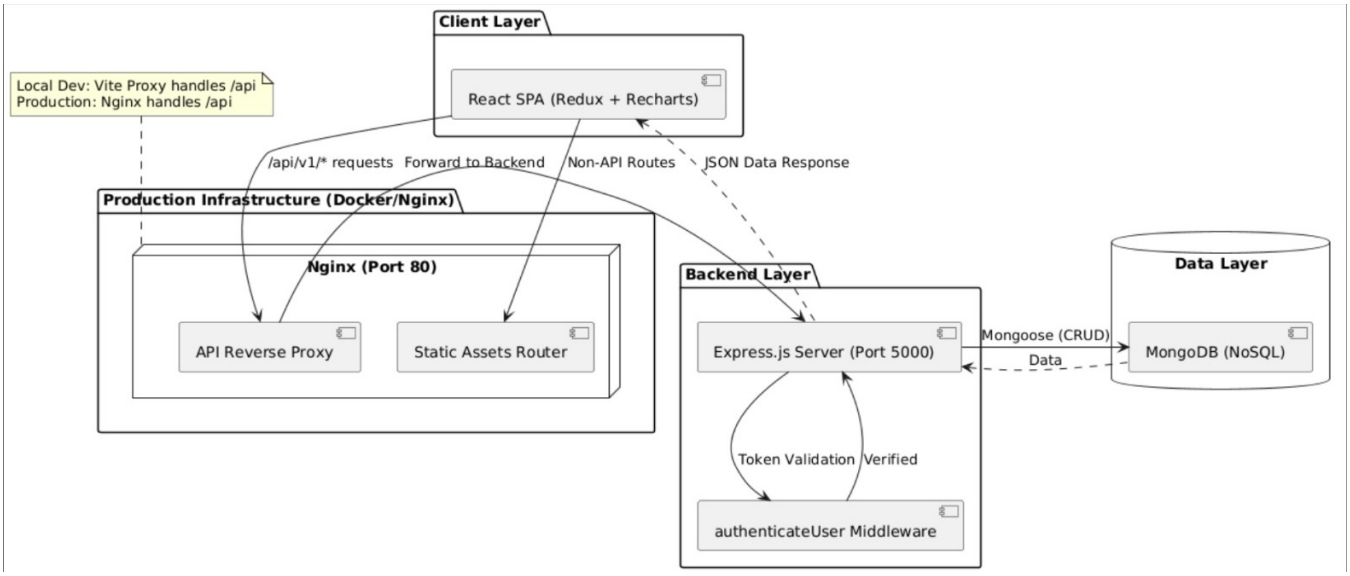

Figure 2.1 – SpendSmart Application Architecture Diagram

2.2. Process Flow

The end-to-end request lifecycle follows a sequential eight-step workflow:

Step	Description
Step 1	The user opens the browser and navigates to the application URL (port 80/443).
Step 2	Nginx serves the compiled React SPA static assets (HTML, CSS, JS) for all non-API routes.
Step 3	The React SPA initialises, Redux store is hydrated, and the user is directed to the Login / Register page.
Step 4	On login, the SPA sends a POST request to /api/v1/users/login; Express validates credentials and issues an HTTP-only JWT cookie.
Step 5	Subsequent finance API calls (/api/v1/incomes, /api/v1/expenses) pass through the authenticateUser middleware which verifies the JWT cookie.
Step 6	Authorised requests are processed by the relevant controller (Income/Expense), which reads from or writes to MongoDB via Mongoose.
Step 7	The API response (JSON) is returned to the React SPA; Redux state is updated and the UI re-renders with the latest data.
Step 8	Recharts visualises income vs. expense summaries on the Dashboard in real-time.

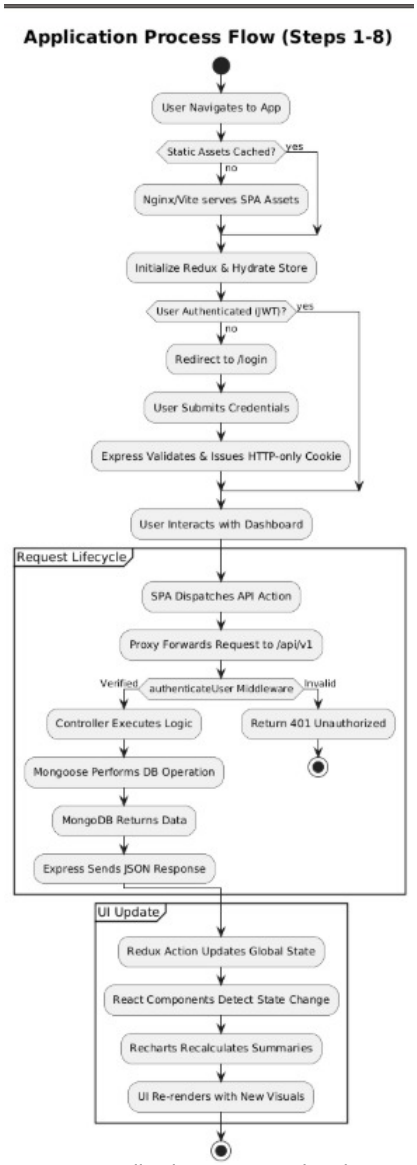

Figure 2.2 – Application Process Flow (Steps 1-8)

2.3. Information Flow

Information flows unidirectionally from the client browser through Nginx to the Express API and finally to MongoDB. During local development, Vite's built-in dev server proxy (vite.config.js) forwards /api requests to http://localhost:5000, mirroring production Nginx behaviour and eliminating CORS friction. In production, all traffic enters through the Nginx container on port 80; static file requests are served directly from /usr/share/nginx/html, while API requests are proxied to the backend container at http://backend:5000. JWT tokens are transmitted exclusively via HTTP-only cookies, preventing client-side script access. MongoDB connection strings and sensitive environment variables are loaded via dotenv and are never committed to the repository.

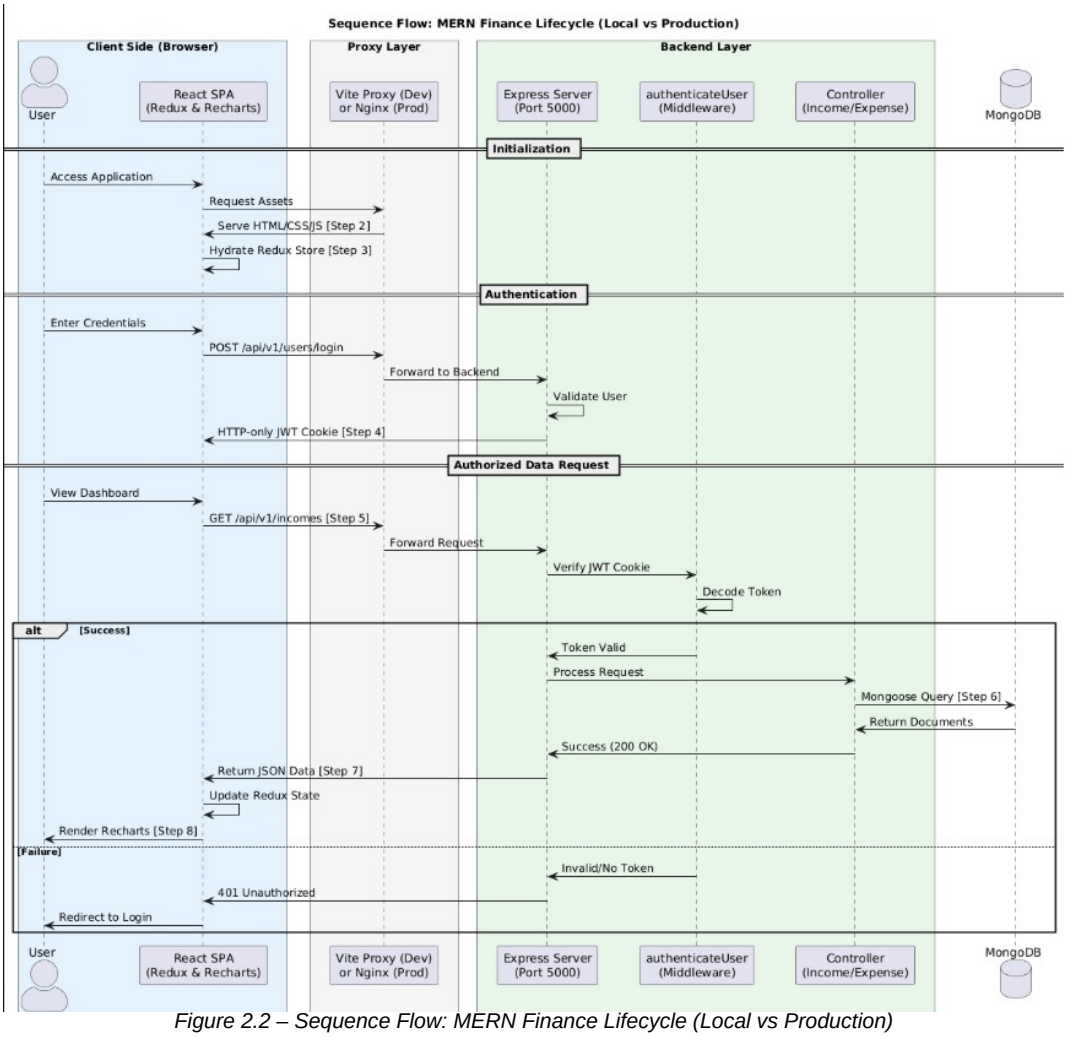

Figure 2.2 – Sequence Flow: MERN Finance Lifecycle (Local vs Production)

2.4. Components Design

Component	Technology	Description
Nginx Web Server & Reverse Proxy	Nginx (stable-alpine)	Serves React SPA static files; proxies /api/* to Express backend; handles static asset caching.
React SPA	React 19, Vite, Tailwind CSS v4, NextUI	Single Page Application; renders Dashboard, Auth, and Finance views.
Redux Store	Redux Toolkit, React-Redux	Global state: authSlice (session), financeSlice (income & expense arrays).
Express API Server	Node.js, Express.js (Port 5000)	Versioned RESTful API; middleware chain: CORS → cookieParser → authenticateUser → controller.
Auth Middleware	JWT, Cookie Parser	Validates HTTP-only JWT cookie on protected routes (/incomes, /expenses).
User Service	Express Router, Mongoose	Handles registration, login, logout; stores hashed passwords in MongoDB.
Income Service	Express Router, Mongoose	CRUD operations on Income collection; records linked to userId.
Expense Service	Express Router, Mongoose	CRUD operations on Expense collection; records linked to userId.
MongoDB Database	MongoDB (via Mongoose)	Persistent data store; collections: Users, Incomes, Expenses.
Docker (Multi-stage)	Docker, node:20-alpine, nginx:stable-alpine	Stage 1: builds React dist; Stage 2: copies dist into Nginx image — minimising production image size.

2.5. Key Design Considerations

Decoupled Architecture: Frontend and backend are independently deployable services communicating only over HTTP/REST, enabling team-parallel development and independent scaling.

Security by Default: JWT tokens are stored in HTTP-only cookies (XSS-resistant). CORS policy in app.js whitelists only localhost:5173 and the production Vercel domain. No secrets are stored in the codebase.

Environment Parity: The Vite dev-proxy mirrors Nginx production routing, minimising environment-specific bugs during development.

Optimised Container Image: A multi-stage Docker build separates the Node.js build environment from the lightweight Nginx runtime, producing a minimal, secure production image.

SPA Routing: Nginx is configured with try_files $uri /index.html so that all React Router client-side routes resolve correctly without server-side 404 errors on page refresh.

Global Error Handling: A catch-all Express middleware returns structured JSON 404 responses for unknown routes, preventing HTML error pages from leaking to API consumers.

Modular Codebase: Backend routes, controllers, and middleware are segregated by domain (users, incomes, expenses) for maintainability and testability.

2.6. API Catalogue

The SpendSmart backend exposes versioned RESTful API endpoints under /api/v1/. All finance routes are protected by the authenticateUser JWT middleware.

Route	Method	Auth Required	Description
/api/v1/users/register	POST	No	Register a new user; stores hashed password.
/api/v1/users/login	POST	No	Authenticate user; set HTTP-only JWT cookie.
/api/v1/users/logout	POST	Yes	Clear the JWT cookie and end session.
/api/v1/users/profile	GET	Yes	Retrieve authenticated user profile.
/api/v1/incomes	GET	Yes	Fetch all income records for the logged-in user.
/api/v1/incomes	POST	Yes	Create a new income record (amount, date, source).
/api/v1/incomes/:id	PUT	Yes	Update an existing income record by ID.
/api/v1/incomes/:id	DELETE	Yes	Delete an income record by ID.
/api/v1/expenses	GET	Yes	Fetch all expense records for the logged-in user.
/api/v1/expenses	POST	Yes	Create a new expense record (amount, date, category).
/api/v1/expenses/:id	PUT	Yes	Update an existing expense record by ID.
/api/v1/expenses/:id	DELETE	Yes	Delete an expense record by ID.

3. Data Design

3.1. Data Model

The SpendSmart application uses MongoDB as its document store. The three primary collections are:

Collection	Key Fields	Description
Users	_id, name, email, password (hashed), createdAt	Stores registered user credentials and profile data. Passwords are hashed using bcrypt before storage.
Incomes	_id, userId (ref: Users), amount, source, date, description, createdAt	Each document records a single income event linked to a user via userId reference.
Expenses	_id, userId (ref: Users), amount, category, date, description, createdAt	Each document records a single expense event linked to a user via userId reference.

3.2. Data Access Mechanism

The Express.js backend accesses MongoDB exclusively through the Mongoose ODM library. The database connection is established at server startup in server/config/db.js; if the connection fails, the process exits to prevent serving requests without a database. All database queries use Mongoose model methods (find, findById, create, findByIdAndUpdate, findByIdAndDelete). Finance routes (incomes, expenses) are protected by the authenticateUser middleware, which validates the HTTP-only JWT cookie and attaches the decoded user identity to the request object before any database operation is performed. This ensures that users can only access their own records.

3.3. Data Retention Policies

Data Type	Retention Period	Policy
User Account Records	Until account deleted	Retained in MongoDB as long as the user account is active; deleted on explicit account deletion.
Income Records	Until user deletes them	DELETE /api/v1/incomes/:id permanently removes the document from MongoDB.
Expense Records	Until user deletes them	Same policy as Income Records.
JWT Cookies	Session / token expiry	HTTP-only cookie expires as per JWT expiry setting; cleared on logout.
Application Logs	Runtime only (current)	Console logs are emitted during runtime; no persistent log storage in current implementation.
Docker Build Cache	Until cache invalidated	Multi-stage build layers are cached by Docker daemon; cleared on docker system prune.

3.4. Data Migration

Schema evolution is managed at the application level via Mongoose schema definitions. Since MongoDB is schema-flexible, adding new fields to a schema does not require a migration script for existing documents; the new fields will be undefined for old records until updated. For structural changes (e.g., renaming fields or changing types), MongoDB's updateMany operator is used to back-fill existing documents. Seed data for development and testing (sample users, income and expense records) is maintained as JavaScript seed scripts executed against the development MongoDB instance. Environment promotion (dev → staging → production) is handled by pointing the MONGO_URI environment variable to the appropriate MongoDB Atlas cluster in the .env file.

4. Interfaces

The SpendSmart system interfaces with the following external systems, tools, and protocols:

Interface	Direction	Protocol	Description
User Browser	Inbound	HTTP/HTTPS (Port 80/443)	End users access the React SPA through a standard web browser.
Nginx ↔ React SPA	Internal	File System / HTTP	Nginx serves compiled static assets from /usr/share/nginx/html.
Nginx ↔ Express API	Internal Proxy	HTTP (Port 5000)	Nginx proxies /api/* requests to the Express backend container.
React SPA ↔ Redux	Internal	In-process	React components dispatch actions to and read state from the Redux store.
Express ↔ MongoDB	Outbound	TCP (MongoDB Wire Protocol)	Mongoose connects to MongoDB using the MONGO_URI connection string.
Vite Dev Proxy	Local Outbound	HTTP	During development, Vite proxies /api to localhost:5000 to mirror production routing.
GitHub / Version Control	Bidirectional	Git / HTTPS / SSH	Source code repository; pull requests trigger code review and CI checks.
Vercel (Frontend Deploy)	Outbound	HTTPS	Production frontend deployment target; CORS in app.js whitelists the Vercel domain.
Docker Daemon	Local	Docker Socket	Docker builds multi-stage images and orchestrates frontend and backend containers.

5. State and Session Management

State management in SpendSmart operates at two distinct levels: server-side session management and client-side application state management.

Server-Side Session Management (JWT + HTTP-Only Cookies): Upon successful login, the Express server issues a signed JSON Web Token (JWT) stored in an HTTP-only cookie. HTTP-only cookies are inaccessible to client-side JavaScript, mitigating XSS-based token theft. The authenticateUser middleware verifies this cookie on every protected route request. On logout, the cookie is cleared by the server response. JWT expiry is configured to automatically invalidate sessions after a defined period without requiring server-side session state storage, making the authentication mechanism stateless and horizontally scalable.

Client-Side State Management (Redux Toolkit): The React SPA manages global application state using Redux Toolkit. The store is structured into slices: authSlice holds the authenticated user object and login status, while a finance slice manages arrays of fetched income and expense records. Redux Thunks handle asynchronous API calls, dispatching loading, success, and error actions to keep the UI in sync with server state. Component-level transient state (form inputs, modal visibility) is managed using React's built-in useState hook.

6. Caching

Caching in this project operates at two levels: the build tooling level and the browser level.

Build-Time Caching (Docker & Vite): The multi-stage Docker build exploits Docker's layer caching mechanism. The COPY package*.json ./ and RUN npm install instructions are placed before the source code copy step, so the dependency installation layer is only re-executed when package.json or package-lock.json changes — significantly accelerating subsequent builds. Vite's development server maintains a module cache in node_modules/.vite, enabling near-instant hot module replacement (HMR) during development.

Browser-Level Caching (Nginx): Nginx serves React SPA static assets (JS bundles, CSS, images) with standard HTTP caching headers. Since Vite generates content-hashed filenames for all bundled assets (e.g., index-Dj3kA9.js), these files can be cached indefinitely by the browser — new deployments produce new hash values, automatically busting the cache. The index.html entry point is served with no-cache directives to ensure browsers always fetch the latest version on each visit.

API Response Caching (Future Consideration): For frequently accessed, read-heavy data such as monthly expense summaries or exchange rates, a Redis cache layer could be provisioned as an additional Docker service. API responses for such endpoints could be cached with a configurable TTL, reducing MongoDB query load and improving response times for concurrent users.

7. Non-Functional Requirements

7.1. Security Aspects

Authentication & Authorisation: All finance API routes (/api/v1/incomes, /api/v1/expenses) are protected by the authenticateUser JWT middleware. Users can only read and modify their own records, enforced by filtering all database queries with the authenticated userId.

XSS Protection: JWT tokens are stored exclusively in HTTP-only cookies, preventing client-side scripts from accessing authentication credentials.

CORS Policy: The Express CORS configuration in app.js explicitly whitelists only http://localhost:5173 and the production Vercel domain, rejecting cross-origin requests from all other origins.

Password Security: User passwords are hashed before storage in MongoDB. Plaintext passwords are never persisted or logged.

Secrets Management: All sensitive values (MongoDB URI, JWT secret, port numbers) are loaded from a .env file via dotenv and are excluded from version control via .gitignore. No secrets are hardcoded in the codebase.

Minimal Attack Surface: The Docker multi-stage build uses nginx:stable-alpine as the production base image, which contains only the Nginx binary and required OS libraries — minimising the attack surface compared to full OS images.

Input Validation: Client-side form validation is enforced using Yup schema validation on all user inputs before API submission, reducing the risk of malformed or malicious data reaching the backend.

7.2. Performance Aspects

Optimised Bundle Size: Vite's production build applies tree-shaking, code-splitting, and minification, producing compact JavaScript bundles. Combined with the Nginx static file serving, initial page load times are minimised.

Nginx Reverse Proxy Efficiency: Nginx handles static file serving and API proxying in a single, high-performance process, avoiding the overhead of a Node.js server serving static assets.

Hot Module Replacement (HMR): Vite's HMR support during development enables sub-second UI updates without full page reloads, significantly improving developer productivity.

MongoDB Indexing: The userId field on Income and Expense collections should be indexed to ensure O(log n) query performance as the dataset grows, preventing full collection scans on per-user data retrieval.

Stateless API Scalability: The JWT-based stateless authentication design allows the Express backend to be horizontally scaled behind a load balancer without requiring shared session storage, supporting increased concurrent user loads.

Data Visualisation Performance: Recharts renders financial charts client-side using SVG, offloading visualisation computation from the server and ensuring smooth, interactive chart experiences directly in the browser.

8. References

1. React Documentation
https://react.dev/

2. Vite Documentation
https://vitejs.dev/

3. Redux Toolkit Documentation
https://redux-toolkit.js.org/

4. Express.js Documentation
https://expressjs.com/

5. Mongoose ODM Documentation
https://mongoosejs.com/docs/

6. MongoDB Documentation
https://www.mongodb.com/docs/

7. JSON Web Tokens (JWT) Introduction
https://jwt.io/introduction

8. Docker Multi-Stage Builds
https://docs.docker.com/build/building/multi-stage/

9. Nginx Reverse Proxy Guide
https://nginx.org/en/docs/

10. Tailwind CSS v4 Documentation
https://tailwindcss.com/docs

11. Recharts Documentation
https://recharts.org/en-US/

12. Medicaps University – Datagami Skill Based Course
Project Brief / Reference Document (as provided by faculty)