EditFlow Project Documentation

1. Introduction

1.1 Introduction to the Project

EditFlow is a web-based freelance marketplace that connects clients with professional video editors and content creators. The platform provides an end-to-end workflow for posting video-editing jobs, receiving proposals from editors, managing projects, exchanging project files, and tracking payments. It is designed as a modern, production-ready application with a React-based frontend and a Node.js/Express backend, backed by a relational database (SQLite) and secured with industry-standard authentication, authorization, and security practices.

The system targets small businesses, content creators, agencies, and individual clients who require reliable, high-quality video editing services and want a structured, professional way to collaborate with editors. By centralizing requirements, communication, deliverables, and payments in one place, EditFlow aims to streamline the collaboration lifecycle and improve trust and transparency for both clients and editors.

1.2 Objective of the Project

The primary objectives of the EditFlow project are:

- To provide a dedicated marketplace for video editing and content creation services.
- To enable clients to post clearly defined jobs with budgets, timelines, and requirements.
- To allow editors to discover relevant jobs and submit structured proposals.
- To convert accepted proposals into managed projects with milestones, files, and status tracking.
- To offer secure user authentication and role-based access control (Client, Editor, Admin).
- To ensure reliable handling of project files, including uploads and downloads.
- To provide a user-friendly, responsive frontend interface for all major workflows.
- To follow professional software engineering practices, including logging, validation, error handling, and documentation, making the system maintainable and production-ready.

1.3 Limitations of Existing System

Traditional and existing methods of hiring video editors—such as informal messaging, generic freelance platforms, or manual referrals—suffer from several limitations:

- **Fragmented Communication**: Requirements, feedback, and file sharing are often scattered across emails, chats, and cloud drives, leading to confusion and loss of context.
- **Lack of Domain Focus**: Generic freelance platforms are not tailored to video-editing workflows, making it harder to express specific technical requirements (formats, resolutions, revisions, etc.).
- **Poor Transparency**: Clients may struggle to track project status, milestones, and deliverables; editors may lack clarity on expectations and acceptance criteria.
- **Limited Trust & Quality Control**: There is often no structured mechanism to review past work, ratings, or portfolios specific to video editing.
- **Manual Tracking of Payments and Milestones**: Payments, partial milestones, and revisions are frequently managed manually, increasing risk of disputes.

- **Scalability Issues**: As the number of projects grows, informal systems become unmanageable and error-prone.

These limitations highlight the need for a specialized, structured platform with built-in workflows tailored to video-editing projects.

1.4 Proposed System

The proposed system, EditFlow, addresses the above limitations by offering a centralized, feature-rich marketplace specifically designed for video-editing and content creation projects. Key characteristics of the proposed system include:

- **Role-Based Access**: Distinct flows and permissions for Clients, Editors, and Admins.
- **Job Management**: Clients can create, view, update, and manage job postings with clearly defined scopes, budgets, and timelines.
- **Proposal Management**: Editors can submit proposals including pricing, delivery estimates, and custom messages; clients can review and accept or reject them.
- **Project & Milestone Management**: Accepted proposals become projects with trackable statuses and milestones, enabling phased delivery and clear progress tracking.
- **File Management**: Secure upload and retrieval of project-related files using server-side storage and controlled access.
- **Secure Authentication & Authorization**: JWT-based authentication, password hashing, and authorization middleware to protect resources.
- **Validation & Error Handling**: Input validation on all critical endpoints and centralized error handling to ensure reliability and predictable behavior.
- **Logging & Monitoring**: Structured logging and health checks to support maintenance, debugging, and production monitoring.

Overall, the proposed system provides a structured, secure, and scalable environment for managing video-editing collaborations end to end.

1.5 Project Feasibility Analysis

1.5.1 Economic Feasibility

EditFlow is implemented using open-source technologies such as Node.js, Express, React, Vite, and SQLite. This significantly reduces licensing and infrastructure costs. The system can initially be deployed on affordable cloud infrastructure or even minimal on-premise setups. As the architecture is lightweight and container-ready (via Docker), hosting and scaling costs can be controlled and optimized over time.

Because the project primarily requires development effort, basic hosting, and storage for project files, the overall economic risk is low. The potential benefits—such as improved efficiency, reduced coordination overhead, and the ability to onboard more clients/editors—justify the development and operational costs.

1.5.2 Technical Feasibility

From a technical perspective, the project is highly feasible:

- The tech stack (Node.js/Express, React, SQLite) is mature, widely adopted, and well supported.

- Required features—such as JWT authentication, file uploads, input validation, and API documentation—are directly supported by robust libraries already integrated into the codebase.
- The modular backend structure (config, controllers, middleware, models, routes) and the component-based frontend structure follow common patterns, simplifying further enhancements.
- Docker and docker-compose support simplify environment setup and deployment.

There are no extraordinary hardware or software requirements, and the system is designed to scale incrementally as usage grows.

1.5.3 Operational Feasibility

The system is designed with usability and maintainability in mind:

- Users interact through a modern, responsive web UI built with React, minimizing training needs.
- Workflows such as job posting, proposal submission, and project management follow intuitive steps.
- Role-based views and protected routes ensure that users see only relevant features and data.
- Centralized logging, health checks, and clear error messages assist administrators in monitoring and troubleshooting.

Given these factors, day-to-day operation of the system by non-technical users (clients, editors, and admins) is feasible and practical.

1.6 Project Management Approach – Software Process Model

The project follows an **incremental, Agile-inspired software process model**. Instead of attempting to deliver all functionality in a single phase, features are planned and implemented in iterations, each delivering a working subset of the system.

Key characteristics of the chosen approach include:

- **Iterative Development**: Core features (authentication, jobs, proposals, projects) are developed first, followed by enhancements (logging, validation, documentation, Dockerization, etc.).
- **Continuous Feedback & Refinement**: Architecture and features are refined based on feedback and evolving requirements.
- **Modular Design**: Clear separation of concerns in backend modules (config, controllers, middleware, models, routes) and frontend modules (pages, components, context, API helpers) supports incremental extension.
- **Documentation-Driven**: README files, API docs, and project documentation are updated alongside code changes to maintain alignment.

This process model is well-suited for academic and professional projects where requirements may evolve, and progressive enhancement is desired.

1.7 Project Timeline – Gantt Chart / Timeline Chart

The following high-level timeline illustrates the major phases of the EditFlow project. Durations are indicative and can be adapted to the actual schedule (e.g., weeks or sprints).

Phase No.	Phase	Description	Duration

Phase No.	Phase	Description	Duration
1	Requirements & Analysis	Gather requirements, define roles and main workflows	1 week
2	System & Database Design	Design architecture, database schema, and APIs	1 week
3	Backend Core Implementation	Implement auth, jobs, proposals, projects, basic CRUD	2 weeks
4	Frontend Core Implementation	Implement main pages, routing, and basic UI	2 weeks
5	Enhancements & Hardening	Add validation, logging, security, Swagger, Docker	2 weeks
6	Integration & Testing	Integrate frontend-backend, fix defects, refine flows	1–2 weeks
7	Documentation & Finalization	Prepare technical/docs, deployment setup, final review	1 week

In a Gantt-style view, these phases can overlap where appropriate (e.g., frontend core development beginning while backend core APIs are nearing completion), enabling parallel work and shorter overall delivery time.

2. Project Analysis – Software Requirement Specification (SRS)

2.1 Purpose of the Project

The purpose of the EditFlow system is to provide a specialized, reliable, and secure platform for managing end-to-end collaboration between clients and video editors/content creators. The project aims to:

- Formalize the process of posting video-editing jobs and receiving proposals.
- Provide a structured project lifecycle from requirement gathering to delivery and payment.
- Reduce miscommunication and ambiguity by centralizing all project-related information.
- Offer a role-based, secure environment for different stakeholders.
- Demonstrate professional software engineering practices suitable for academic and real-world deployment.

This SRS defines the functional and non-functional requirements that guide the design, development, testing, and maintenance of the system.

2.2 Scope of the Project

The scope of EditFlow includes the following major capabilities:

- User registration, authentication, and profile management for Clients, Editors, and Admins.
- Job posting by clients with detailed descriptions, budgets, categories, and timelines.
- Browsing and searching of available jobs by editors.
- Proposal submission by editors and proposal review/decision by clients.
- Conversion of accepted proposals into projects with milestones and status tracking.
- Upload and download of project-related files.

- Basic wallet/budget tracking and payment status representation (as supported by the backend models).
- Administrative capabilities to oversee users, jobs, and platform health.

Out of scope for the current version are: real-time messaging, integrated payment gateways, advanced analytics dashboards, and mobile applications. These are identified as potential future enhancements.

2.3 Functional Requirements

The functional requirements describe what the system must do. At a high level, the main functional modules are:

2.3.1 User Management

- The system shall allow users to register with roles (Client or Editor).
- The system shall authenticate users using email/username and password.
- The system shall provide JWT-based session management for authenticated requests.
- The system shall allow users to view and update their profile information (where applicable).

2.3.2 Job Management (Client)

- The system shall allow clients to create new job postings with title, description, budget, and expected duration.
- The system shall allow clients to view a list of their own jobs.
- The system shall allow clients to edit or close their job postings.
- The system shall allow public or authorized users (e.g., editors) to view available jobs.

2.3.3 Proposal Management (Editor & Client)

- The system shall allow editors to submit proposals for open jobs, including proposed budget and delivery time.
- The system shall allow editors to view a list of proposals they have submitted.
- The system shall allow clients to view all proposals for a specific job.
- The system shall allow clients to accept or reject proposals.

2.3.4 Project & Milestone Management

- The system shall automatically create a project when a proposal is accepted.
- The system shall allow tracking of project status (e.g., In Progress, Completed).
- The system shall support milestones associated with a project, including descriptions and due dates.
- The system shall allow updating milestone status.

2.3.5 File Management

- The system shall allow authenticated users to upload files associated with a project.
- The system shall store uploaded files in a secure server-side directory.
- The system shall allow authorized users to download project files.

2.3.6 Payment & Wallet (High-Level Representation)

- The system shall store basic payment-related data (e.g., amount, status) associated with projects/milestones as defined in the backend models.

- The system shall provide views for users to see their payment or wallet-related information (as per implemented UI and models).

2.3.7 Administration & Analytics

- The system shall provide administrative capabilities to view users, jobs, and system statistics (as supported by admin and analytics controllers).
- The system shall provide a health check endpoint to verify application status.

2.3.8 Security & Access Control

- The system shall restrict protected endpoints to authenticated users only.
- The system shall enforce role-based access for sensitive operations (e.g., only clients can post jobs, only editors can send proposals, only admins can access admin routes).

2.4 Non-Functional Requirements

Non-functional requirements define quality attributes and constraints of the system.

2.4.1 Performance

- The system should respond to typical API requests (authentication, job listing, proposal submission) within acceptable time limits under normal load (e.g., within a few hundred milliseconds on development hardware).
- The system should handle multiple concurrent requests using Node.js's asynchronous model.

2.4.2 Reliability & Availability

- The system should be stable during normal operation without unexpected crashes.
- Health check endpoints and structured logging should support quick detection and diagnosis of failures.

2.4.3 Security

- Passwords must be hashed before storage using a secure algorithm (bcrypt).
- JWT tokens must use a strong secret and have an expiration time.
- Security headers must be set using Helmet.
- CORS must be configured to only allow trusted origins.
- Rate limiting must be used to reduce the risk of brute-force attacks.

2.4.4 Usability

- The frontend must provide a clean, intuitive, and responsive UI for both desktop and mobile users.
- Navigation between major features (Jobs, Proposals, Projects, Wallet, Admin) should be straightforward.

2.4.5 Maintainability & Extensibility

- The codebase shall follow modular design principles, with separation between controllers, models, routes, middleware, and configuration.
- API documentation (Swagger) shall be available to support future developers.
- Configuration shall be handled through environment variables validated by a schema.

2.4.6 Portability & Deployment

- The system shall be containerized using Docker so that it can run consistently across environments.
- Environment-specific configuration shall be externalized via `.env` files and Docker Compose.

2.5 Hardware Requirements

The hardware requirements for developing and running EditFlow in a typical academic or small-production environment are modest:

- **Development Machine**:
  - Processor: Modern multi-core CPU (e.g., Intel i5/AMD Ryzen 5 or above).
  - RAM: Minimum 8 GB (16 GB recommended for smoother multitasking and running Docker containers).
  - Storage: At least 10–20 GB free disk space for source code, node modules, database files, and logs.
  - Network: Stable internet connection for dependency installation and API testing.
- **Server/Deployment Environment (Initial Scale)**:
  - CPU: 1–2 vCPUs.
  - RAM: 2–4 GB.
  - Storage: Sufficient for database and uploaded files; can be expanded as the number of projects grows.

2.6 Software Requirements

The following software stack is required to develop, run, and deploy the system:

- **Backend**:
  - Node.js (v14 or higher).
  - npm or yarn for dependency management.
  - SQLite3 database engine.
  - Express.js and associated middleware (Helmet, express-validator, express-rate-limit, compression, etc.).
- **Frontend**:
  - Node.js and npm.
  - React with Vite as the build tool.
  - Modern web browser (Chrome, Edge, Firefox, etc.).
- **Development & Deployment Tools**:
  - Git for version control.
  - Docker and Docker Compose (for containerized deployment).
  - An IDE or code editor (e.g., VS Code).

2.7 Stakeholders of the System

The main stakeholders of the EditFlow system are:

- **Clients**:
    - Individuals, businesses, or organizations that need video editing or content creation services.
    - Post jobs, review proposals, manage projects, and approve deliverables.
- **Editors (Video Editors / Content Creators)**:
    - Professionals who provide editing and content services.
    - Discover jobs, submit proposals, work on projects, and deliver final outputs.
- **Administrators**:
    - Personnel responsible for overseeing platform health, user management, and policy enforcement.
    - Monitor logs, handle disputes, and ensure the system runs smoothly.
- **Development & Maintenance Team**:
    - Developers, testers, and DevOps engineers responsible for implementing new features, fixing bugs, and managing deployments.
- **Academic Evaluators / Supervisors (for academic context)**:
    - Faculty or reviewers assessing the project's technical quality, documentation, and adherence to software engineering standards.

Each stakeholder group has different expectations and interactions with the system, which are reflected in the functional and non-functional requirements defined above.

3. Requirement Models

This section describes the key analysis and design models used to understand and structure the EditFlow system. The diagrams can be prepared using standard UML tools (e.g., Draw.io, Lucidchart, StarUML) based on the textual descriptions below.

3.1 Use Case Diagram

The use case model captures the interactions between actors (stakeholders) and the system. The primary actors are **Client**, **Editor**, **Administrator**, and **System Services** (e.g., Email/Notification service if added later).

Key use cases include:

- **Client**:
    - Register / Login
    - Manage Profile
    - Post Job
    - View Own Jobs
    - View Proposals for a Job
    - Accept / Reject Proposal

    - View Projects and Milestones
    - Upload / Download Project Files
    - View Payment / Wallet Information

- **Editor**:

    - Register / Login
    - Manage Profile
    - Browse / Search Jobs
    - Submit Proposal
    - View Own Proposals
    - View Assigned Projects
    - Upload / Download Project Files

- **Administrator**:

    - Login
    - View All Users
    - View Jobs, Proposals, Projects
    - Monitor System Analytics/Reports
    - Manage Platform Settings / Policies (conceptual)

- **System**:

    - Authenticate User (JWT)
    - Validate Input
    - Log Requests and Errors
    - Generate API Documentation (Swagger)

In the graphical use case diagram, actors are shown outside the system boundary, with arrows to the above use cases grouped within the EditFlow system.

3.2 Context Level Diagram

The Context Level Diagram (Level 0 DFD) shows EditFlow as a single high-level process interacting with external entities.

System: EditFlow Platform

External Entities:

- Clients
- Editors
- Administrators
- (Optional) External Services (Email provider, Payment gateway – for future enhancements)

Data Flows (Conceptual):

- Clients send job details, view jobs, receive proposals and project updates.
- Editors send proposals, upload deliverables, receive job details and project assignments.
- Administrators view system data and manage platform configuration.

In the diagram, EditFlow is a central process bubble, with data flow arrows to/from the external entities and high-level data stores (e.g., Database).

3.3 Data Flow Diagram

The Data Flow Diagram (DFD) can be decomposed beyond the context level into major processes:

- **Process 1: User Management**
    - Inputs: Registration data, login credentials.
    - Outputs: User accounts, JWT tokens, profile data.
    - Data Stores: User data in the database.
- **Process 2: Job Management**
    - Inputs: Job creation/edition requests from clients.
    - Outputs: Job listings, job details.
    - Data Stores: Job records.
- **Process 3: Proposal Management**
    - Inputs: Proposal submissions from editors.
    - Outputs: Proposal lists, proposal decisions.
    - Data Stores: Proposal records.
- **Process 4: Project & Milestone Management**
    - Inputs: Accepted proposals, milestone updates.
    - Outputs: Project and milestone status, project details.
    - Data Stores: Project, milestone, and activity records.
- **Process 5: File Management**
    - Inputs: File upload requests.
    - Outputs: File download responses, file metadata.
    - Data Stores: File metadata in DB; binary files in uploads directory.
- **Process 6: Payment & Wallet Management**
    - Inputs: Payment updates, milestone completion events.
    - Outputs: Payment status, wallet/transaction views.
    - Data Stores: Payment records.

Level 1 and Level 2 DFDs can be drawn to further elaborate internal flows within each major process.

3.4 Entity Relationship Diagram

The Entity Relationship Diagram (ERD) models the main data entities and their relationships. Based on the backend models and schema, key entities include:

- **User**:
    - Attributes: id, name, email, passwordHash, role (Client/Editor/Admin), profile fields.

    - Relationships: One user can create many jobs, proposals, projects, portfolios, reviews, etc.

- **Job**:
    - Attributes: id, clientId, title, description, budget, duration, status, timestamps.
    - Relationships: Each job belongs to one client (User), and has many proposals.
- **Proposal**:
    - Attributes: id, jobId, editorId, proposedAmount, estimatedTime, status, message, timestamps.
    - Relationships: Each proposal belongs to one job and one editor (User). An accepted proposal may be linked to one project.
- **Project**:
    - Attributes: id, jobId, clientId, editorId, status, startDate, endDate, etc.
    - Relationships: Each project is associated with one job, one client, and one editor; it has many milestones, files, activities, and payments.
- **Milestone**:
    - Attributes: id, projectId, title, description, amount, dueDate, status.
    - Relationships: Each milestone belongs to one project.
- **Payment**:
    - Attributes: id, projectId or milestoneId, amount, status, timestamps.
    - Relationships: Each payment is associated with a project (and optionally a milestone).
- **ProjectFile**:
    - Attributes: id, projectId, fileName, filePath, uploadedBy, uploadedAt.
    - Relationships: Each file belongs to one project.
- **ProjectActivity**:
    - Attributes: id, projectId, activityType, description, createdAt.
    - Relationships: Each activity belongs to one project.
- **Portfolio**:
    - Attributes: id, editorId, title, description, media links.
    - Relationships: Each portfolio belongs to one editor.
- **Review**:
    - Attributes: id, projectId, reviewerId, revieweeId, rating, comment, timestamps.
    - Relationships: Each review is associated with a project and relates one user to another.

In the ERD, foreign key relationships (e.g., userId, jobId, projectId) are drawn as one-to-many or many-to-one connectors between the entities above.

3.5 Activity Diagram

Activity diagrams describe the workflow for core processes. Important activity diagrams for EditFlow include:

- **User Registration & Login Flow**:
    - Start → User submits registration form → System validates data → Account created → User logs in → System issues JWT token → End.
- **Job Posting & Proposal Flow**:
    - Client logs in → Navigates to "Create Job" → Enters job details → Submits job → Job stored and published → Editors browse jobs → Editor submits proposal → Proposal stored → Client views proposals → Client accepts or rejects proposal.
- **Project Execution Flow**:
    - Proposal accepted → Project created → Milestones defined → Editor uploads files for milestones → Client reviews deliverables → Milestones marked complete → Payments updated (conceptually) → Project marked completed.

These activity diagrams can be modeled in UML with initial and final nodes, decision points, and swimlanes for each actor (Client, Editor, System).

3.6 Other UML Diagrams (If Required)

Depending on the level of detail required, additional UML diagrams can be prepared:

- **Class Diagram**:
    - Representing backend models (User, Job, Proposal, Project, Milestone, Payment, Portfolio, Review, ProjectFile, ProjectActivity) and their associations.
- **Sequence Diagrams**:
    - Example: Login sequence (User → Frontend → Auth API → DB → Response).
    - Example: Proposal submission sequence (Editor → Frontend → Proposals API → DB → Notification to Client).
- **Component Diagram**:
    - Showing high-level components such as Frontend (React), Backend API (Express), Database (SQLite), and File Storage, and their dependencies.

These diagrams further clarify system behavior and structure and can be included as needed in the final report.

3.7 Flowcharts (If Any)

Flowcharts can be used to visually describe decision-heavy logic. Useful flowcharts for EditFlow include:

- **Authentication Flowchart**:
    - Start → User enters credentials → Validate input → Check user in DB → Password match? → If yes, generate JWT and grant access; if no, show error and retry.

- **Proposal Decision Flowchart**:
    - Start → Client views proposals → Selects a proposal → Decision? (Accept / Reject) → If Accept, create project and notify editor; if Reject, update status and optionally request new proposal.

These flowcharts complement the activity diagrams by emphasizing decision paths and outcomes.

3.8 Project Flow / Architecture

EditFlow follows a modern web application architecture with clear separation between frontend, backend, and database layers.

- **Frontend (Presentation Layer)**:
    - Built with React and Vite.
    - Implements pages such as Login, Signup, Dashboard, Jobs, Job Details, Proposals, Projects, Wallet, and Admin Panel.
    - Uses React Context (AuthContext) for global authentication state and ProtectedRoute components for route-level access control.
    - Communicates with the backend via Axios, using interceptors to attach JWT tokens.
- **Backend (Application Layer)**:
    - Built with Node.js and Express.
    - Organised into config, controllers, middleware, models, routes, and utils.
    - Exposes RESTful APIs for authentication, jobs, proposals, projects, files, payments, analytics, and admin operations.
    - Implements security (Helmet, rate limiting, CORS), validation (express-validator, Joi), logging (Winston, Morgan), and centralized error handling.
- **Database & Storage (Data Layer)**:
    - Uses SQLite as the relational database.
    - Stores all structured data (users, jobs, proposals, projects, milestones, payments, portfolios, reviews, activities, file metadata).
    - Uses a server-side uploads directory for storing binary files.
- **Infrastructure & Deployment**:
    - Dockerfiles for backend and frontend.
    - docker-compose.yml for orchestration, including API, frontend, and supporting services.
    - Nginx configuration for serving the frontend in production.

Overall, the architecture supports scalability, maintainability, and clear separation of concerns.

3.9 Database Schema

The database schema is defined in the SQL script and model files provided in the backend. At a high level, the schema includes the following tables (names may vary slightly by implementation):

- **users**: Stores user accounts and roles.

    - Key columns: id (PK), name, email (unique), password_hash, role, created_at, updated_at.

- **jobs**: Stores job postings created by clients.
    - Key columns: id (PK), client_id (FK → users.id), title, description, budget, duration, status, created_at, updated_at.
- **proposals**: Stores proposals submitted by editors.
    - Key columns: id (PK), job_id (FK → jobs.id), editor_id (FK → users.id), amount, estimated_time, message, status, created_at, updated_at.
- **projects**: Represents accepted work derived from proposals.
    - Key columns: id (PK), job_id (FK → jobs.id), client_id (FK → users.id), editor_id (FK → users.id), status, start_date, end_date, created_at, updated_at.
- **milestones**: Breaks down projects into smaller deliverables.
    - Key columns: id (PK), project_id (FK → projects.id), title, description, amount, due_date, status, created_at, updated_at.
- **payments**: Tracks payment-related information.
    - Key columns: id (PK), project_id (FK → projects.id), milestone_id (optional FK → milestones.id), amount, status, created_at, updated_at.
- **project_files**: Stores metadata for uploaded files.
    - Key columns: id (PK), project_id (FK → projects.id), file_name, file_path, uploaded_by (FK → users.id), uploaded_at.
- **project_activities**: Logs important actions within a project.
    - Key columns: id (PK), project_id (FK → projects.id), activity_type, description, created_at.
- **portfolios**: Stores portfolio items for editors.
    - Key columns: id (PK), editor_id (FK → users.id), title, description, media_url, created_at, updated_at.
- **reviews**: Stores feedback between users after project completion.
    - Key columns: id (PK), project_id (FK → projects.id), reviewer_id (FK → users.id), reviewee_id (FK → users.id), rating, comment, created_at, updated_at.

Indexes and foreign key constraints are defined to ensure referential integrity and improve query performance. The schema is designed to support the current feature set while remaining extensible for future enhancements such as advanced analytics or payment integrations.

4. Project Development / Implementation

This section describes how the EditFlow system has been developed and implemented, focusing on user interface design, configuration design, key output screens, and testing.

4.1 User Interface Designs (With Description)

The frontend of EditFlow is implemented using React and Vite, with a component-based architecture. Major UI elements are organized under `src/pages`, `src/components`, and `src/context` in the frontend codebase.

Main UI Components and Pages

- **Navigation Bar (Navbar)**
  - Provides global navigation links such as Home/Dashboard, Jobs, My Projects, My Portfolio, Wallet, Admin Panel (for admin users), Login, and Logout.
  - Displays context-aware options based on the user's authentication status and role.
- **Authentication Screens (Login, Signup)**
  - **Login Page**: Allows existing users to enter email/username and password. On successful login, a JWT token is stored (through AuthContext) and used for subsequent API calls.
  - **Signup Page**: Allows new users to register as Client or Editor, capturing basic profile information. Validations ensure required fields are filled and passwords meet criteria.
- **Dashboard**
  - Provides a high-level overview for logged-in users, such as recent jobs, proposals, or projects related to the current user.
  - For clients, the dashboard may highlight recently posted jobs and incoming proposals; for editors, it may emphasize available jobs and active projects.
- **Jobs & Job Details (Jobs, CreateJob, JobDetails)**
  - **Jobs Page**: Displays a list of available jobs with summary information (title, budget, duration, client, status). Editors can view and decide which jobs to open.
  - **CreateJob Page**: Allows clients to create a new job by entering job title, description, budget, skills, and expected timeline. Client-side validation is performed before submitting to the backend.
  - **JobDetails Page**: Shows full job information, associated proposals, and contextual actions such as submitting a proposal (for editors) or reviewing proposals (for clients).
- **Proposals & Job Proposals (JobProposals)**
  - Displays all proposals submitted for a particular job, including editor details, proposed amount, estimated delivery time, and messages.
  - Clients can accept or reject proposals from this screen.
- **Projects & Project Details (ProjectDetails, Dashboard sections)**
  - Shows a list of active and completed projects associated with the logged-in user.
  - Project details include linked job, client, editor, milestones, files, and status.
- **Portfolio & Editors (MyPortfolio, Editors)**
  - **MyPortfolio**: Enables editors to showcase their previous work, descriptions, and sample media links.

    - **Editors Page**: Allows clients to view available editors and their portfolios (depending on implemented features).

- **Wallet & Payments (Wallet)**
    - Displays a summarized view of payment or balance-related information as supported by the backend models.

- **Admin Panel (AdminPanel)**
    - Provides administrators with access to platform-level data such as users, jobs, proposals, and basic analytics.

Design Style and UX Considerations

- Modern, responsive layouts with support for desktop and mobile devices.
- Clear separation of areas using cards, panels, and sections.
- Consistent typography and color palette aligned with the EditFlow brand.
- Contextual error and success messages to guide user actions.
- Loading indicators and disabled states during asynchronous operations to improve perceived performance.

4.2 Configuration Designs

Configuration is centralized and environment-driven to support multiple deployment scenarios (development, testing, production).

Backend Configuration

- **Environment Variables (`.env` / env.js)**
    - The backend reads configuration values from a `.env` file, validated using a schema in the environment configuration module.
    - Key variables include `PORT`, `NODE_ENV`, `JWT_SECRET`, `DATABASE_URL`, `CORS_ORIGIN`, logging level, rate limiting settings, and file upload limits.

- **Database Configuration (db.js)**
    - Defines the connection to the SQLite database file and manages initialization using the SQL schema.
    - Ensures proper handling of connections and error reporting.

- **Logger Configuration (logger.js)**
    - Configures the Winston logger with multiple transports (console and log files) and log levels.
    - Handles exception and rejection logging, log rotation, and formatting.

- **Multer Configuration (multer.js)**
    - Defines storage destination, file naming strategy, and file size limits for uploads.

- **Swagger Configuration (swagger.js)**

- Generates Swagger/OpenAPI documentation based on JSDoc comments and annotations.
- Serves interactive API documentation at the `/api-docs` endpoint.

- **Middleware Configuration**
  - Express middlewares for Helmet, CORS, rate limiting, compression, request logging (Morgan), and centralized error handling are configured in the server entry point.

Frontend Configuration

- **Environment Configuration (`.env`, Vite config)**
  - Frontend uses environment variables (e.g., `VITE_API_BASE_URL`) to configure the backend API base URL.
  - Vite configuration handles development server settings and build options.

- **API Client Configuration (Axios)**
  - A centralized Axios instance is configured with a base URL and interceptors.
  - Request interceptors automatically attach JWT tokens from AuthContext/local storage.
  - Response interceptors handle common error cases (e.g., unauthorized) and can trigger logout or redirection.

- **Auth Context Configuration (AuthContext)**
  - Manages the authentication state (current user, token) globally.
  - Provides helper functions for login, logout, and automatic token persistence across page refreshes.

This configuration design ensures that sensitive values are not hard-coded, and that behavior can be adjusted per environment without code changes.

4.3 Output Screen Designs / Report Screenshots Images

The implemented user interfaces result in several key output screens. For inclusion in the final report, screenshots can be captured for the following representative screens (with annotations describing key UI elements):

- **Login Screen**
  - Fields for email/username and password.
  - Error messages for invalid credentials.

- **Signup Screen**
  - Registration form with role selection (Client/Editor).
  - Validation messages for missing or invalid inputs.

- **Dashboard Screen**
  - Overview cards showing counts of jobs, proposals, and projects.
  - Quick links to primary actions (Post Job, Browse Jobs, View Projects).

- **Jobs Listing Screen**
    - Tabular or card-based display of available jobs.
    - Filters or search (if implemented) to refine job viewing.
- **Job Details & Proposals Screen**
    - Full job description, budget, and timeline.
    - List of proposals with actions (Submit Proposal for editors; Accept/Reject for clients).
- **Project Details Screen**
    - Display of project metadata, milestones, and files.
    - Buttons or forms to upload files and update milestones.
- **My Portfolio Screen**
    - Display of an editor's portfolio entries (titles, descriptions, media links).
- **Admin Panel Screen**
    - Summary of platform statistics (e.g., number of users, jobs, projects).
    - Lists of users or high-level analytics (depending on implemented features).

These screenshots serve as visual evidence of implementation and help reviewers understand the system's look and feel. In the final document, each screenshot should be captioned and briefly described.

4.4 Project Testing

Testing is an essential part of ensuring that EditFlow functions correctly and reliably. The project adopts a combination of manual and automated testing approaches, with a roadmap for further test automation.

4.4.1 Backend Testing Approach

- **Unit and Integration Testing (Planned/Extensible)**
    - The backend structure (controllers, routes, middleware, models) is designed to support unit and integration tests.
    - Test scripts are defined in the package configuration, and popular testing frameworks (e.g., Jest, Supertest) can be integrated to test API endpoints, validation, and error handling.
- **Manual API Testing**
    - During development, key endpoints (authentication, jobs, proposals, projects, files) are exercised using tools like Swagger UI and API clients.
    - Typical test scenarios include:
        - Successful user registration and login.
        - Access control enforcement (e.g., only clients can post jobs, only editors can submit proposals).
        - Job creation, listing, and retrieval.
        - Proposal submission, listing, and acceptance/rejection flows.
        - Project and milestone creation and status updates.

        - File upload and download permissions.

- **Error Handling and Edge Cases**
    - Tests verify that invalid inputs are rejected with meaningful error messages (e.g., missing required fields, invalid IDs).
    - Authentication and authorization failures return appropriate status codes.

4.4.2 Frontend Testing Approach

- **Manual UI Testing**
    - Pages are manually tested in a browser to ensure navigation, form submissions, and state updates work as expected.
    - Responsive behavior is validated on different viewport sizes.
    - Authentication flows (login/logout, protected routes) are verified using different user roles.
- **Planned Automated Testing**
    - The React component structure is compatible with future unit tests using frameworks like Jest and React Testing Library.
    - End-to-end testing tools (e.g., Cypress, Playwright) can be introduced to automate critical user journeys (login, post job, submit proposal, accept proposal, manage project).

4.4.3 Non-Functional Testing

- **Performance & Load (Basic Checks)**
    - Basic checks are performed to ensure that typical API calls return within acceptable time under development conditions.
- **Security Testing (Baseline)**
    - Verification that key security features (JWT authentication, password hashing, CORS, Helmet, rate limiting) are correctly configured.
- **Regression Testing**
    - When new features are added, previously implemented flows (authentication, job posting, proposals, projects) are re-tested to ensure no regressions occur.

Overall, the testing strategy ensures that core functional paths are verified, with a clear path for introducing more comprehensive automated tests as the project evolves.

5. Software Testing

5.1 Software Testing

Software testing for EditFlow focuses on verifying that the implemented features meet the specified functional and non-functional requirements, and that regressions are minimized as the system evolves. The testing strategy combines manual validation with a structure that is ready for automation.

Testing Levels

- **Unit Testing (Planned/Supported)**
    - Targets individual functions, controllers, middleware, and utility modules in the backend.
    - Example: Testing the authentication controller's login logic with valid and invalid credentials.
- **Integration Testing (Planned/Supported)**
    - Tests interactions between components, such as controllers, routes, and the database layer.
    - Example: Sending HTTP requests to the `/api/jobs` endpoints and verifying that data is correctly stored and retrieved from the database.
- **System/End-to-End Testing (Partially Manual, Automation-Ready)**
    - Validates complete user workflows across frontend and backend.
    - Example: A client logs in, posts a job, an editor submits a proposal, and the client accepts it to create a project.
- **Non-Functional Testing (Baseline)**
    - Basic security, performance, and usability checks are performed to ensure the system behaves reliably under typical conditions.

Testing Techniques

- **Black-Box Testing**
    - Most tests are designed from the user's perspective, focusing on inputs and expected outputs without inspecting internal code.
- **Positive and Negative Testing**
    - Positive tests confirm that the system behaves correctly when valid data is provided.
    - Negative tests ensure that invalid data, unauthorized access, and edge cases are handled gracefully with clear error messages.
- **Regression Testing**
    - When new features or fixes are introduced, core flows such as authentication, job posting, proposal submission, and project management are re-tested to ensure existing behavior remains correct.

The architecture (modular backend and component-based frontend) is deliberately structured to facilitate the later introduction of automated tests using common JavaScript testing frameworks.

5.2 Test Cases

Below are representative test cases that illustrate how key features of EditFlow are validated. They can be formalized into a test case document or implemented as automated tests.

Authentication & Authorization

1. **TC-AUTH-01: Successful User Registration**

- Precondition: User is not registered.
- Steps: Submit valid registration data with role = Client.
- Expected Result: API returns success; new user record is created; user can subsequently log in.

2. **TC-AUTH-02: Login with Valid Credentials**

- Precondition: User account exists.
- Steps: Submit correct email/username and password.
- Expected Result: API returns JWT token; frontend stores token; user is redirected to dashboard.

3. **TC-AUTH-03: Login with Invalid Credentials**

- Steps: Submit wrong password.
- Expected Result: API returns error (401/400) with appropriate message; no token is issued.

4. **TC-AUTH-04: Access Protected Route without Token**

- Steps: Call a protected API (e.g., `/api/jobs` create) without Authorization header.
- Expected Result: Request is rejected with 401 Unauthorized.

5. **TC-AUTH-05: Role-Based Access Control**

- Steps: Logged-in editor attempts to call a client-only operation (e.g., create job).
- Expected Result: Request is rejected with 403 Forbidden or equivalent error.

Job and Proposal Management

6. **TC-JOB-01: Create Job (Client)**

- Precondition: User is logged in as Client.
- Steps: Fill and submit job form with valid data.
- Expected Result: Job is created; appears in client's job list and public job list (if applicable).

7. **TC-JOB-02: View Job List (Editor)**

- Precondition: At least one job exists.
- Steps: Editor navigates to Jobs page.
- Expected Result: Jobs are listed with key details (title, budget, status).

8. **TC-PROP-01: Submit Proposal (Editor)**

- Precondition: Editor is logged in; open job exists.
- Steps: Editor opens JobDetails and submits a proposal with valid data.
- Expected Result: Proposal is created, associated with the job and editor, and visible to the client.

9. **TC-PROP-02: Accept Proposal (Client)**

- Precondition: Client has at least one proposal for one of their jobs.
- Steps: Client opens JobProposals and accepts a proposal.
- Expected Result: Proposal status changes to "Accepted"; a project is created and linked to the job, client, and editor.

Project, Milestone, and File Management

10. **TC-PROJ-01: View Projects**
    - Precondition: At least one project exists for the logged-in user (client or editor).
    - Steps: User navigates to Projects/ProjectDetails view.
    - Expected Result: Relevant projects are listed; selecting one shows its details, milestones, and files.
11. **TC-MILE-01: Update Milestone Status**
    - Precondition: Project with milestones exists.
    - Steps: Authorized user updates a milestone status (e.g., from Pending to Completed).
    - Expected Result: Status is updated in the database and reflected on the UI.
12. **TC-FILE-01: Upload Project File**
    - Precondition: User is authenticated and authorized for the project.
    - Steps: User selects a file and uploads it from ProjectDetails.
    - Expected Result: File is saved on the server; metadata is stored in DB; file is listed under project files.
13. **TC-FILE-02: Download Project File**
    - Precondition: At least one file exists for the project.
    - Steps: User clicks download link.
    - Expected Result: File is downloaded successfully; access is restricted to authorized users.

Frontend UI & Navigation

14. **TC-UI-01: Protected Route Behavior**
    - Steps: Try to access a protected page (e.g., Dashboard) without being logged in.
    - Expected Result: User is redirected to Login page.
15. **TC-UI-02: Role-Specific Menu Options**
    - Steps: Log in as Client, then as Editor.
    - Expected Result: Navbar and available routes differ appropriately for each role (e.g., Create Job for clients, Browse Jobs for editors).

Non-Functional / Security

16. **TC-SEC-01: Rate Limiting**
    - Steps: Rapidly send multiple authentication requests beyond configured threshold.
    - Expected Result: API starts responding with rate-limit errors after the threshold, protecting against brute-force attempts.
17. **TC-SEC-02: CORS Configuration**
    - Steps: Attempt to call the API from an untrusted origin.
    - Expected Result: Browser blocks the request due to CORS policy; only configured origins are allowed.

These representative test cases can be expanded into a full test case matrix including priority, test data, and actual results, and then automated gradually as the project matures.

6. Limitations

While EditFlow has been designed and implemented using professional practices, the current version has certain limitations that should be acknowledged:

- **No Integrated Payment Gateway**
    - Payments are modeled conceptually in the database and UI (e.g., wallet and payment status), but full integration with real payment providers (such as Stripe or PayPal) is not implemented.
- **Limited Real-Time Collaboration Features**
    - The system does not currently support real-time messaging, live notifications, or in-browser media preview. Communication is assumed to occur through external channels or basic status updates.
- **Basic Analytics and Reporting**
    - Only foundational analytics and admin views are available. Advanced reporting dashboards, detailed usage analytics, and business intelligence features are not yet implemented.
- **Single-Database, Single-Instance Deployment**
    - The reference implementation targets a single SQLite database instance and a relatively small user base. For very high traffic or large-scale deployments, migration to a more scalable DBMS and multi-instance architecture would be required.
- **Testing Coverage Not Fully Automated**
    - Although the architecture is test-friendly and representative test cases are defined, full automated unit, integration, and end-to-end test suites are not yet implemented.
- **Email/Notification Services Not Integrated**
    - Automated email notifications (e.g., for account verification, password reset, proposal updates) and push notifications are not part of the current scope.
- **Accessibility and Localization**
    - The UI is primarily designed for a single language and may not fully conform to all accessibility guidelines (e.g., WCAG) without further refinement and testing.

These limitations do not prevent the system from functioning as a working freelance marketplace prototype, but they highlight areas for improvement and potential future work.

7. Future Enhancements

Several enhancements can be implemented in future iterations to extend EditFlow from a robust prototype into a full-featured, large-scale production system.

7.1 Functional Enhancements

- **Integrated Payment Gateway**
  - Integrate secure online payment providers (e.g., Stripe, PayPal) for handling deposits, milestone-based payments, and payouts to editors.
  - Support dispute resolution workflows and transaction history views for clients and editors.
- **Real-Time Communication & Notifications**
  - Add in-app messaging or chat between clients and editors for each project.
  - Implement real-time notifications (e.g., using WebSockets or Socket.io) for events such as new proposals, accepted proposals, file uploads, and review submissions.
- **Advanced Search and Filtering**
  - Provide advanced job and editor search with filters by category, budget range, skills, experience level, and rating.
  - Add saved searches and job alerts.
- **Enhanced Portfolio & Review System**
  - Allow editors to create richer portfolios with categorized work samples, tags, and embedded media previews.
  - Extend the review system with more detailed feedback, public ratings, and response options for editors.

7.2 Technical and Architectural Enhancements

- **Migration to a Scalable Database**
  - Move from SQLite to a more scalable RDBMS (e.g., PostgreSQL or MySQL) for higher concurrency and larger datasets.
  - Introduce database migrations and versioning with a dedicated migration tool.
- **Microservices / Modularization (If Needed)**
  - Gradually separate core domains (auth, jobs, proposals, projects, payments) into modular services if scale demands.
  - Introduce an API gateway and centralized authentication service.
- **Caching and Performance Optimization**
  - Use caching layers (e.g., Redis) for frequently accessed data such as job listings and user profiles.
  - Optimize database queries and add additional indexes based on real-world usage patterns.
- **File Storage Improvements**
  - Move from local file storage to cloud object storage (e.g., AWS S3, Azure Blob Storage) for better scalability, redundancy, and CDN integration.
  - Add secure, time-limited download URLs and in-browser media previews (video/image).

7.3 Quality and Security Enhancements

- **Automated Testing Suite**
  - Implement comprehensive unit, integration, and end-to-end tests for backend and frontend using Jest, React Testing Library, and Cypress/Playwright.
  - Integrate tests into a CI pipeline to prevent regressions.
- **TypeScript Adoption**
  - Gradually migrate backend and/or frontend code to TypeScript for stronger type safety and improved maintainability.
- **Security Hardening**
  - Perform regular security audits and penetration tests.
  - Add features such as two-factor authentication (2FA), more granular permissions, and enhanced audit logging.
- **Monitoring and Observability**
  - Integrate application performance monitoring (APM) tools and centralized log aggregation.
  - Add dashboards and alerts for key metrics (error rate, response time, throughput).

7.4 UX and Product Enhancements

- **Improved Accessibility and Localization**
  - Refine UI to better meet accessibility standards (e.g., keyboard navigation, ARIA labels, contrast checks).
  - Add multi-language support and locale-specific formatting (dates, currencies).
- **Mobile Experience**
  - Further optimize responsive layouts for mobile devices.
  - Optionally develop a dedicated mobile app using technologies like React Native or Flutter, reusing the existing API.
- **Richer Admin Panel**
  - Expand the admin dashboard with more detailed analytics, user management tools, automated moderation aids, and configuration options.

These enhancements provide a clear roadmap for evolving EditFlow into a more powerful, scalable, and user-friendly platform while building on the solid foundation of the current implementation.

8. Conclusion

The EditFlow project delivers a professionally structured freelance marketplace tailored for video editors and content creators. Through a clear separation of frontend, backend, and database concerns, the system demonstrates how modern web technologies can be combined to support real-world collaboration scenarios such as job posting, proposal management, project tracking, and file exchange.

From an engineering perspective, the project applies key software development best practices: secure authentication and authorization, input validation, centralized error handling, structured logging, environment-based configuration, and comprehensive documentation. The inclusion of Swagger-based API documentation, Dockerized deployment, and a modular codebase makes the system easier to understand, maintain, and extend.

The requirement analysis, design models, and implementation details documented in this report show a complete lifecycle—from problem identification and feasibility analysis, through architecture and UI design, to testing and evaluation. While certain advanced features (such as integrated payments, real-time communication, and extensive analytics) are reserved for future work, the current implementation provides a solid, working prototype that can be used as a foundation for academic evaluation and further professional development.

In summary, EditFlow achieves its primary objectives of creating a secure, maintainable, and user-focused platform that connects clients with video editors in a structured and efficient way, while also serving as a strong learning artifact for modern full-stack application development.

9. Bibliography

The following references and resources were consulted during the design and development of the EditFlow system:

1. **Node.js Documentation** – Node.js Foundation. Official documentation for Node.js runtime. Available at: https://nodejs.org
2. **Express.js Guide** – Express.js. Official documentation for Express web framework. Available at: https://expressjs.com
3. **React Documentation** – Meta Platforms, Inc. Official documentation for the React JavaScript library. Available at: https://react.dev
4. **Vite Documentation** – Evan You and Vite Contributors. Official documentation for Vite build tool. Available at: https://vitejs.dev
5. **SQLite Documentation** – SQLite Consortium. Official documentation for SQLite database engine. Available at: https://sqlite.org/docs.html
6. **Swagger / OpenAPI Specification** – OpenAPI Initiative. Documentation for designing and documenting RESTful APIs. Available at: https://swagger.io/specification
7. **JWT (JSON Web Tokens)** – Auth0 Documentation. Concepts and best practices for token-based authentication. Available at: https://jwt.io/introduction
8. **bcrypt Library Documentation** – Official documentation for password hashing with bcrypt (various language bindings).
9. **Helmet.js Documentation** – Helmet Middleware for Express. Guidance on securing Express apps with HTTP headers. Available at: https://helmetjs.github.io
10. **express-validator Documentation** – Express middleware for validation and sanitization. Available at: https://express-validator.github.io
11. **Winston Logger Documentation** – Winston logging library for Node.js. Available at: https://github.com/winstonjs/winston
12. **Docker Documentation** – Docker Inc. Official documentation for containerization and Docker Compose. Available at: https://docs.docker.com

In addition to the above, general web development best practices and various community tutorials, blog posts, and discussion forums were referenced to inform architecture, security, and UX decisions.