

# Collaborative Text Editor

*Final Project Documentation*

Aryan Bhandari & Vanshika Agarwal

DPCS - Collaborative Text Editor

December 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Overview . . . . .	3
1.2	Core Features . . . . .	3
1.3	Technology Stack . . . . .	3
<b>2</b>	<b>User Guide</b>	<b>4</b>
2.1	Registration and Login . . . . .	4
2.2	Documents Dashboard . . . . .	4
2.3	Creating and Opening Documents . . . . .	4
2.4	Sharing and Role Management . . . . .	4
2.5	Real-Time Collaboration . . . . .	4
2.6	Version History and Timeline . . . . .	4
2.7	Export and Download (Optional) . . . . .	5
2.8	Common Issues and Troubleshooting . . . . .	5
<b>3</b>	<b>Technical Documentation</b>	<b>6</b>
3.1	High-Level Architecture and Data Flow . . . . .	6
3.1.1	System Layers . . . . .	6
3.1.2	Conceptual Architecture . . . . .	6
3.2	Database Schema: Models and Relationships . . . . .	6
3.2.1	Key Invariants . . . . .	6
3.3	REST API Routes . . . . .	7
3.3.1	Authentication . . . . .	7
3.3.2	Documents . . . . .	7
3.3.3	Sharing and Permissions . . . . .	7
3.3.4	Version History . . . . .	7
3.4	Socket.IO Real-Time Protocol . . . . .	7
3.4.1	Connection and Authentication . . . . .	7
3.4.2	Document Rooms . . . . .	7
3.4.3	Events and Payloads . . . . .	7
3.4.4	Server-Side Validation . . . . .	7
3.5	Collaboration Model (Operation Pipeline) . . . . .	8
3.5.1	Guarantees (intended) . . . . .	8
3.5.2	Limitations (honest constraints) . . . . .	8
<b>4</b>	<b>Testing Plan</b>	<b>9</b>
4.1	Test Suite Structure . . . . .	9
4.2	Comprehensive Test Cases . . . . .	9
4.3	Testing Methodology, Rationale, and Scope . . . . .	9
4.3.1	Methodology . . . . .	9
4.3.2	Scope achieved vs not achieved . . . . .	10
4.4	Testing Tools and Commands . . . . .	10
4.4.1	Backend Testing . . . . .	10
4.4.2	Frontend Testing . . . . .	10
4.4.3	Manual Multi-User Testing . . . . .	10
4.4.4	Example Commands . . . . .	10
<b>5</b>	<b>Deployment</b>	<b>11</b>
5.1	Local Setup Instructions . . . . .	11
5.1.1	Prerequisites . . . . .	11

5.1.2	Environment Variables . . . . .	11
5.1.3	Database Initialization (Prisma) . . . . .	11
5.1.4	Running the Servers . . . . .	11
5.2	Production Deployment Plan . . . . .	11
5.2.1	Deployment Steps (Generic) . . . . .	12
5.3	Prisma Migrate Deploy Flow . . . . .	12
5.4	Scaling Considerations . . . . .	12
5.4.1	Socket.IO Scaling . . . . .	12
5.4.2	Database Scaling . . . . .	12
5.5	Cloud Deployment (Production) . . . . .	12
5.5.1	Live Deployment URLs . . . . .	12
5.5.2	Deployment Architecture . . . . .	12
5.5.3	Testing Real-Time Collaboration in Production . . . . .	13
5.5.4	Production Troubleshooting . . . . .	13
5.5.5	Deployment Configuration . . . . .	13
<b>6</b>	<b>Appendix</b>	<b>15</b>
6.1	Project Structure . . . . .	15
6.2	Sample Environment Templates . . . . .	15
6.3	Known Limitations and Future Enhancements . . . . .	15
6.3.1	Known Limitations . . . . .	15
6.3.2	Future Enhancements . . . . .	15

# 1 Introduction

## 1.1 Overview

This project implements a browser-based Collaborative Text Editor that enables multiple authenticated users to create documents, share them with role-based permissions, and edit them concurrently with near real-time synchronization. The system combines a REST API (for authentication, document CRUD, sharing, and version history) with a Socket.IO real-time channel (for live editing operations, presence, and cursor updates). Document state is persisted in a PostgreSQL database via Prisma ORM.

## 1.2 Core Features

- **Authentication (JWT):** Register, login, and session validation
- **Documents CRUD:** Create, list, open, rename/update, and delete documents
- **Sharing & Permissions:** Share documents with users using explicit roles (VIEW/COMMENT/EDIT) and revoke access
- **Real-Time Collaboration:** Multiple users can edit the same document concurrently with live updates
- **Presence & Cursors:** Show who is currently in a document with optional colored cursors per user
- **Version History:** Snapshots of document content over time with timeline visualization
- **Export:** Download documents as PDF/DOCX/plain-text (optional feature)

## 1.3 Technology Stack

- **Frontend:** React + TypeScript (Vite), React Router, Socket.IO client
- **Backend:** Node.js + TypeScript, Express, Socket.IO
- **Database:** PostgreSQL, Prisma ORM
- **Security:** JWT-based authentication with server-side authorization

## 2 User Guide

### 2.1 Registration and Login

1. Navigate to the application URL
2. Register with name/email/password or login using existing credentials
3. On successful login, you are redirected to the Documents Dashboard

### 2.2 Documents Dashboard

The dashboard lists all documents that you own or have been granted access to. Each document entry shows basic metadata (title, last updated) and displays your effective role for that document.

### 2.3 Creating and Opening Documents

- Click **New Document** to create a fresh document
- Click on a document row to open it in the editor view

### 2.4 Sharing and Role Management

1. Open the document you want to share
2. Open the Share panel
3. Add a collaborator by email and assign a role (VIEW/COMMENT/EDIT)
4. The collaborator will see the document in their dashboard and can access it according to their role
5. To remove access, use **Remove Access** for that collaborator

### 2.5 Real-Time Collaboration

1. Open the same document in two or more browsers (or incognito sessions) with different accounts
2. Start typing; changes appear for all connected users in near real-time
3. Presence indicators show which users are currently active in the document
4. Optional: Cursor indicators show each user's cursor/selection in a unique color

### 2.6 Version History and Timeline

The application maintains a version history (snapshots) for each document. In the UI, users can:

- View a list of versions with timestamps and authors
- Preview earlier snapshots
- Restore a snapshot (optional feature)

## 2.7 Export and Download (Optional)

If implemented, the editor provides export actions:

- Download as PDF
- Download as DOCX
- Download as plain text

## 2.8 Common Issues and Troubleshooting

Issue	Solution
"Failed to fetch" / API errors	Ensure backend is running and the API base URL is configured correctly
Refresh breaks editor route	Ensure the app rehydrates auth state and refetches document on page load
Not seeing a shared document	Confirm the owner shared with the correct email and role; verify permissions in backend
Edits appear duplicated or jittery	Indicates client/server operation ordering issues; review collaboration model and resync logic

Table 1: Common Issues and Solutions

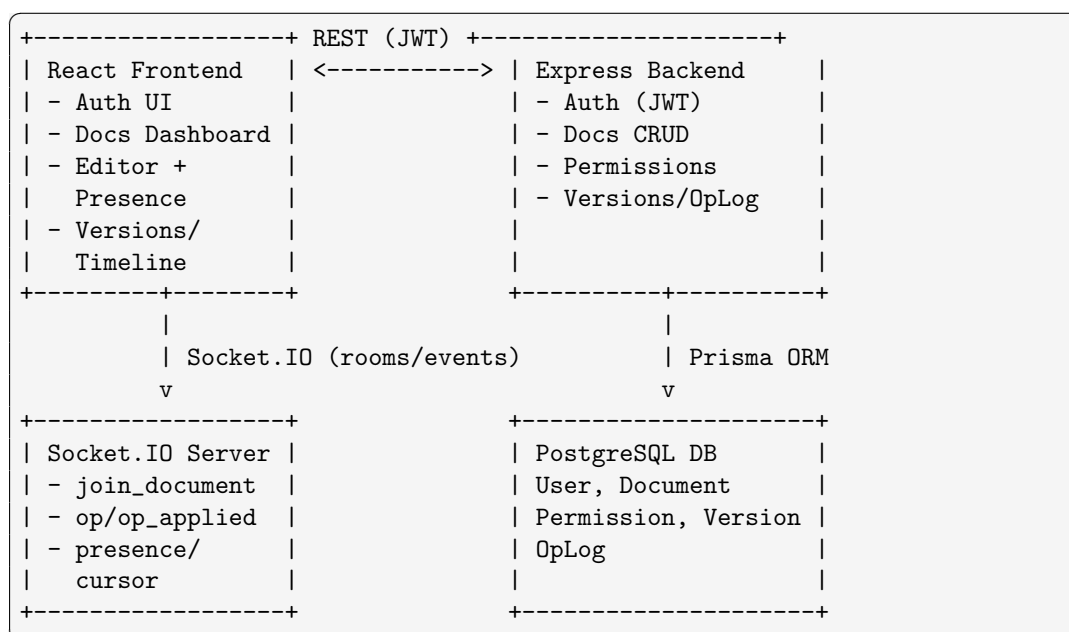
## 3 Technical Documentation

### 3.1 High-Level Architecture and Data Flow

#### 3.1.1 System Layers

- **Frontend (React):** UI, routing, auth state, editor experience; calls REST and maintains real-time connection
- **Backend (Express):** JWT auth, CRUD operations, permissions management, versioning, and real-time gateway
- **Database (PostgreSQL via Prisma):** Durable storage for users, documents, permissions, version snapshots, and operation logs
- **Real-Time Channel (Socket.IO):** Document rooms, presence/cursors, and edit operation propagation

#### 3.1.2 Conceptual Architecture



### 3.2 Database Schema: Models and Relationships

This project uses Prisma ORM to model and query the database. The core models include:

- **User:** Stores user accounts (email, display name, password hash)
- **Document:** Stores document title and canonical content; tracks ownership and currentVersion integer
- **Permission:** Links (userId, documentId, role) to grant VIEW/COMMENT/EDIT privileges
- **Version:** Snapshot records of document content at specific points in time
- **OpLog:** Append-only log of edit operations for debugging and auditability

#### 3.2.1 Key Invariants

- Only the owner (and optionally EDIT role users) can change sharing permissions

- Read access requires owner or VIEW/COMMENT/EDIT permission
- Write access requires owner or EDIT permission
- Document updates in collaboration increment currentVersion monotonically

### 3.3 REST API Routes

#### 3.3.1 Authentication

Endpoint	Description
POST /auth/register	Request: {name, email, password} → Response: {user, token}
POST /auth/login	Request: {email, password} → Response: {user, token}
GET /auth/me	Requires Authorization: Bearer token → Response: {user}

Table 2: Authentication Endpoints

#### 3.3.2 Documents

Endpoint	Description
GET /documents	List documents accessible to user (owned + shared) with role metadata
POST /documents	Create a new document (title optional) → returns created document
GET /documents/:id	Fetch document (title/content/owner/role)
PATCH /documents/:id	Update document fields (e.g., rename; optionally update content)
DELETE /documents/:id	Delete document (owner-only)

Table 3: Document Endpoints

#### 3.3.3 Sharing and Permissions

Endpoint	Description
POST /documents/:id/share	Request: {email, role} → Adds/updates Permission for that user
DELETE /documents/:id/share/:userId	Revoke access by deleting Permission (owner-only)

Table 4: Sharing and Permissions Endpoints

#### 3.3.4 Version History

### 3.4 Socket.IO Real-Time Protocol

#### 3.4.1 Connection and Authentication

- The client connects with a JWT token via Socket.IO connection options
- The server validates the token and associates the socket with a user identity

#### 3.4.2 Document Rooms

Each document corresponds to a Socket.IO room (e.g., `document:<id>`). Users join the room when they open the editor.

#### 3.4.3 Events and Payloads

#### 3.4.4 Server-Side Validation

On receiving an operation (op), the server:

Endpoint	Description
GET /documents/:id/versions	List snapshots for a document
POST /documents/:id/versions	Create a snapshot (optional feature)
POST /documents/:id/restore/:versionId	Restore snapshot (optional feature)

Table 5: Version History Endpoints

Event	Description
join_document	Payload: {documentId} - User joins document room
leave_document	Payload: {documentId} - User leaves document room
presence	Payload: {documentId, users:[...]} - Broadcast current users in room
cursor_update	Payload: {documentId, position, selection?} - Broadcast cursor position
op	Payload: {documentId, type, position, text?, length?, baseVersion} - Edit operation
op_applied	Payload: {documentId, op, newVersion} - Broadcast applied operation

Table 6: Socket.IO Events

1. Verifies socket is authenticated
2. Verifies user has EDIT permission (or is owner)
3. Verifies operation structure (bounds checking, type checking)
4. Validates or reconciles baseVersion with canonical currentVersion
5. Applies operation to canonical state and persists

### 3.5 Collaboration Model (Operation Pipeline)

The collaboration model uses server-authoritative, operation-based synchronization:

1. **Local edit:** User types in the editor; client computes a minimal operation (insert/delete)
2. **Emit operation:** Client emits op over Socket.IO with documentId and baseVersion
3. **Validate & apply:** Server validates permissions, applies operation to canonical content, records in OpLog, increments currentVersion
4. **Broadcast:** Server emits op\_applied to all clients in the document room
5. **Remote apply:** Each client applies the received operation to local state to stay synchronized

#### 3.5.1 Guarantees (intended)

- Near real-time convergence for multiple simultaneous users under typical concurrent typing
- Canonical state always persists in the database and can be recovered on refresh

#### 3.5.2 Limitations (honest constraints)

- This is not a full CRDT with offline merges; edge-case concurrent edits may require simple transforms/resync
- Very high-frequency concurrent edits can expose ordering/resync complexity without ack/pending-op logic

## 4 Testing Plan

### 4.1 Test Suite Structure

- **Unit tests (Backend):** Pure logic tests for permission checks, role resolution, and operation validation helpers
- **Integration tests (Backend):** API endpoints with test database (auth, CRUD, share/revoke, versions list)
- **Realtime tests:** Scripted Socket.IO clients that join rooms and emit ops concurrently
- **Frontend tests (Optional):** Component tests for auth routing, dashboard rendering, editor toolbar
- **Manual acceptance tests:** Repeatable steps for concurrency, refresh, permissions, and exports

### 4.2 Comprehensive Test Cases

Area	Test Case	Type	Expected Result
Auth	Register new user, then login	Integration	Token issued; /me returns user
Auth	Invalid password login	Integration	401 with clear message
Docs	Create document, list, open	Integration	Document visible; content loads
Permissions	Share doc with VIEW role	Integration	Recipient can view, cannot edit
Permissions	Share doc with EDIT role	Integration	Recipient can edit and sync
Permissions	Revoke access from user	Integration	User loses dashboard access
Realtime	3-user concurrent typing	Manual/Scripted	No duplication/loss; convergent
Realtime	Disconnect/reconnect mid-edit	Manual	Rejoin room; content resyncs
Refresh	Refresh /documents/:id route	Manual	No blank screen; auth rehydrates
Versions	Versions list loads	Integration	Snapshot list returns and renders
Export	PDF/DOCX/plain text export	Manual	File downloads and opens correctly
Errors	Permission denied (open/edit)	Integration	403 with user-friendly message

Table 7: Comprehensive Test Cases

### 4.3 Testing Methodology, Rationale, and Scope

#### 4.3.1 Methodology

- Start with backend integration tests to validate correctness of auth, permissions, and CRUD
- Validate real-time correctness with controlled manual concurrency tests and repeatable scripted clients

- Validate refresh/reconnect as a separate acceptance test (common failure mode in SPAs)
- Validate exports as end-to-end UI behavior rather than unit-level behavior

#### 4.3.2 Scope achieved vs not achieved

- **Achieved:** Auth, CRUD, role checks, share/revoke flows, multi-user real-time under typical conditions, refresh recovery
- **Potential gaps:** Exhaustive concurrent-operation edge cases and full offline collaboration (out of scope for course timeline)
- **Optional features:** Exports and snapshot restore may be optional depending on implementation maturity

### 4.4 Testing Tools and Commands

#### 4.4.1 Backend Testing

- **Test runner:** Jest/Vitest (recommended)
- **Integration testing:** Supertest for REST endpoints

#### 4.4.2 Frontend Testing

- **Component tests:** React Testing Library (optional)
- **E2E tests:** Playwright/Cypress (optional, recommended if time permits)

#### 4.4.3 Manual Multi-User Testing

- Use 2 normal browser windows + 1 incognito session with three users
- Use devtools console logs on both client and server to confirm join/leave and operation flow

#### 4.4.4 Example Commands

```
# Backend
cd server
npm install
npx prisma generate
npx prisma migrate dev
npm run dev

# Frontend
cd client
npm install
npm run dev
```

## 5 Deployment

### 5.1 Local Setup Instructions

#### 5.1.1 Prerequisites

- Node.js (LTS version recommended)
- PostgreSQL (local install or Docker)
- Docker Desktop for containerized Postgres (optional)

#### 5.1.2 Environment Variables

Frontend (.env):

```
# client/.env
VITE_API_BASE_URL=http://localhost:4000
```

Backend (.env):

```
# server/.env
DATABASE_URL=postgresql://USER:PASSWORD@localhost:5432/cte_db
JWT_SECRET=replace_with_strong_secret
PORT=4000
```

#### 5.1.3 Database Initialization (Prisma)

```
cd server
npm install
npx prisma generate
npx prisma migrate dev
```

#### 5.1.4 Running the Servers

```
# Terminal 1 - Backend
cd server
npm run dev

# Terminal 2 - Frontend
cd client
npm run dev
```

### 5.2 Production Deployment Plan

A typical production setup uses:

- **Frontend hosting:** Static hosting (Vercel/Netlify/GCS bucket) for the Vite build output
- **Backend hosting:** Node runtime (Cloud Run/App Engine/VM/Render/Fly.io) running Express + Socket.IO
- **Managed Postgres:** Cloud-managed Postgres (Cloud SQL/RDS/Neon/Supabase)

### 5.2.1 Deployment Steps (Generic)

1. Provision managed Postgres and set `DATABASE_URL`
2. Deploy backend with environment variables (`DATABASE_URL`, `JWT_SECRET`, `PORT`)
3. Run migrations using `prisma migrate deploy` at release time
4. Deploy frontend build; configure `VITE_API_BASE_URL` to point to backend
5. Ensure CORS is configured to allow the frontend domain

## 5.3 Prisma Migrate Deploy Flow

In production, migrations should be applied in a controlled step:

```
cd server
npx prisma migrate deploy
```

## 5.4 Scaling Considerations

### 5.4.1 Socket.IO Scaling

- With a single backend instance, Socket.IO rooms work naturally
- With multiple instances behind a load balancer, you typically need:
  - Sticky sessions (route a user's websocket to the same instance), and/or
  - A pub/sub adapter (commonly Redis) so events broadcast across instances

### 5.4.2 Database Scaling

- Use connection pooling in production
- Add indexes for common queries (documents by owner/permission, permissions by docId/userId, versions by docId)

## 5.5 Cloud Deployment (Production)

### 5.5.1 Live Deployment URLs

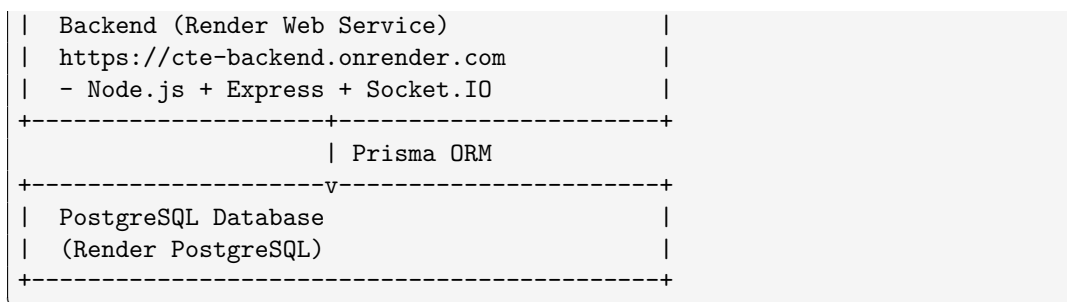
The application is deployed on Render.com with the following endpoints:

- **Frontend:** <https://cte-frontend.onrender.com>
- **Backend API:** <https://cte-backend.onrender.com>
- **Repository:** <https://github.com/DPCS3810/Project-2-Team-3>

### 5.5.2 Deployment Architecture

The production deployment uses Render.com's infrastructure with three main components:

```
+-----+
| Frontend (Render Static Site) |
| https://cte-frontend.onrender.com |
| - React + Vite + TypeScript |
+-----+
| HTTP/WebSocket |
+-----v-----+
```



### 5.5.3 Testing Real-Time Collaboration in Production

To verify the deployment is working correctly:

1. Open <https://cte-frontend.onrender.com> in two browser windows
2. Register/login with different accounts in each window
3. Create a document in one window and share it with the other user
4. Open the same document in both windows
5. Type in one window and verify changes appear instantly in the other window
6. Verify presence indicators show both users are active
7. Test cursor updates and real-time synchronization

### 5.5.4 Production Troubleshooting

Common production issues and solutions:

Issue	Solution
Services not responding	Check Render dashboard for deployment status and logs
Environment variables missing	Verify all required environment variables are configured in Render settings
Database connection errors	Ensure Render PostgreSQL service is running and DATABASE_URL is correct
CORS errors	Verify CORS_ORIGIN environment variable includes the frontend URL
WebSocket connection fails	Confirm backend service is running and both HTTP and WebSocket connections are allowed
Deployment failures	Check build logs in Render dashboard; verify package.json scripts are correct

Table 8: Production Troubleshooting Guide

### 5.5.5 Deployment Configuration

#### Frontend Configuration (Render Static Site):

- Build command: `npm install && npm run build`
- Publish directory: `dist`
- Environment variables:
  - `VITE_API_URL`: Backend API URL
  - `VITE_WS_URL`: Backend WebSocket URL

**Backend Configuration (Render Web Service):**

- Build command: `npm install && npx prisma generate && npm run build`
- Start command: `npm start`
- Environment variables:
  - `DATABASE_URL`: PostgreSQL connection string from Render
  - `JWT_SECRET`: Strong secret for JWT tokens
  - `CORS_ORIGIN`: Frontend URL for CORS configuration
  - `PORT`: Automatically set by Render

**Database Configuration (Render PostgreSQL):**

- Version: PostgreSQL 14+
- Connection pooling: Enabled
- Automatic backups: Configured in Render
- Internal connection: Used by backend service

## 6 Appendix

### 6.1 Project Structure

```
cte-project/  
|-- client/  
|   |-- src/  
|   |   '-- components/  
|   |   '-- documents/  
|   '-- package.json  
|-- server/  
|   |-- src/  
|   |   '-- modules/  
|   |       |-- documents/  
|   |       '-- collab/  
|   |-- prisma/  
|   |   '-- schema.prisma  
|   '-- package.json
```

### 6.2 Sample Environment Templates

**client/.env.example:**

```
VITE_API_BASE_URL=http://localhost:4000
```

**server/.env.example:**

```
DATABASE_URL=postgresql://USER:PASSWORD@localhost:5432/cte_db  
JWT_SECRET=replace_with_strong_secret  
PORT=4000
```

### 6.3 Known Limitations and Future Enhancements

#### 6.3.1 Known Limitations

- Collaboration correctness depends on operation ordering and resync logic; extreme concurrency edge cases may not match Google Docs guarantees
- Snapshot/branch semantics may be limited to linear versions unless merge/prune is implemented server-side
- Export fidelity depends on the chosen library; complex formatting may not fully match editor rendering

#### 6.3.2 Future Enhancements

- Upgrade to a CRDT-first persistence model for stronger offline/merge behavior
- Richer presence features: live user list, cursor trails, comments/annotations
- True timeline branching: branch creation, merging, and pruning with conflict resolution
- Production-grade scaling: Redis adapter for Socket.IO and horizontal scaling