1. **Target environment**

The application was developed by a team primarily working on Windows environments and has been successfully deployed on a Linux server (Railway). While the backend services and frontend build tools are cross-platform and can run on Windows, Linux, and macOS, the application also integrates Redis for in-application caching, which is not officially supported on Windows.

Given the stability, performance, and deployment history, **Linux is the recommended operating system** for production environments. However, , **Windows or macOS** can also be used without any significant issues, especially because Redis is not a critical dependency.

Hardware Requirements

To support **up to 1000 concurrent users** with a mix of API and WebSocket traffic, the estimated hardware requirements are as follows:

- **CPU**: 2–4 vCPUs (depending on workload intensity and expected peak usage)
- **RAM**: 4–8 GB (to handle Redis, Node.js processes, and WebSocket memory overhead)
- **Storage**: 10–20 GB SSD (for logs, temporary files, and potential data caching; persistent storage needs depend on whether database and media storage are offloaded to external services)
- **GPU**: Not required

Cloud vs. On-Premise

The team **recommends cloud infrastructure** (e.g., AWS, Azure, Railway, Render, or similar platforms) due to:

- Easier scalability
- Built-in monitoring and deployment pipelines
- Lower upfront cost and simplified maintenance
- High availability and distributed edge servers (improving latency for users in different regions)

That said, **on-premise deployment is also feasible** if required for compliance or data sovereignty. The application is fully containerizable and relies on environment variables for external services, which allows seamless configuration across cloud and local environments.

2. **Software dependencies**

Required Runtimes

The application is built using:

- **Node.js** (v20.12.2 recommended): Required for running the backend server, handling HTTP APIs, WebSocket connections, and integrating Redis for in-memory caching.
- **npm** (Node Package Manager): Used to install and manage backend and frontend dependencies.
- **React.js**: Used for building the frontend user interface, compiled into static assets during the build process.

Libraries, Packages, and Frameworks

- **Backend (Node.js) – all packages are already handled by npm and are listed in package.json, some examples:**
  - express
  - socket.io
  - redis
  - dotenv
- **Frontend (React.js) — all packages are already handled by npm and are listed in package.json, some examples:**
  - react-router-dom – For client-side routing
  - axios – For API requests
  - socket.io-client – For real-time updates from the backend
- **External Services**
  - **Redis**: Used for caching and storing real-time session or transient data. It is not a hard dependency, but it's strongly recommended for scalability and performance in production.

Containerization

At this stage, **Docker or any other containerization technology is not currently implemented**. The application is deployed directly via cloud services using Node.js environments.

3. Installation and Configuration

Installation Method

The software is currently distributed as a **Git repository**, which can be cloned and run locally or on a remote server. It is not packaged as an installer or container at this stage.

To install the application:

Clone both repositories:

```
git clone https://github.com/Procure-Hub-Org/procure-hub-be.git

git clone https://github.com/Procure-Hub-Org/procure-hub-fe.git
```

Install dependencies:

```
npm install
```

Environment Variables and Configuration

The application uses environment variables for managing external service connections and runtime settings. A sample file is provided:

- **.env.example** – Contains all required environment variable names and placeholders. To configure the application, copy this file and fill in actual values:

```
cp .env.example .env
```

System Initialization Steps

To fully initialize the system after installing dependencies and configuring environment variables:

(Optional) Set up Redis if caching is needed (recommended for production)

Run database migrations/initialization scripts

After setting up the database and backend project environment variables run the following command: npx sequelize-cli db:migrate

Add a default admin user by running a custom CLI command: npm run add-admin

This script will prompt you for details like email, password, and name, and will create the admin user directly in the database.

Default Users and Passwords

There are **no hardcoded default users or passwords** for security reasons. The application setup intentionally requires a manual creation of the first admin user via the CLI script mentioned above. This ensures credentials are unique and securely handled during installation.

4. **Continuous Integration / Continuous Deployment (CI/CD)**

CI/CD Tools Used

The project uses **GitHub** as the version control platform and integrates with **Railway** for automatic deployment. While no custom CI/CD pipelines (e.g., GitHub Actions or Jenkins) are configured at this stage, **Railway handles deployment automatically based on Git activity.**

Deployment Triggers

Deployment is currently triggered on every **commit to the develop branch.** This process is fully automated via Railway's GitHub integration and includes:

- Pulling the latest code from the repository
- Installing dependencies
- Running build scripts
- Restarting the deployed service

Deployment from the **main branch** can be configured similarly to support production releases. This allows for a **branch-based deployment strategy**, where:

- develop → staging/testing environment
- main → production environment

This setup ensures a clean and consistent workflow between development, testing, and production deployments.

5. Network and Security

Open Ports

The application architecture separates the frontend and backend into distinct services for enhanced security and maintainability. The following ports need to be open depending on the deployment environment:

- **Backend API and WebSocket server**:
  - Typically runs on a single port (e.g., PORT=3000) handling both HTTP (REST API) and WebSocket (socket.io) connections.
  - This port is configurable via environment variables.
- **Frontend (if hosted separately)**:
  - Usually served via static file hosting or CDN (e.g., Netlify, Vercel) and does not require an open port on the backend server.

Only the backend server needs access to the database and other internal services, keeping the frontend decoupled from direct data access for security.

SSL/TLS Requirements

- **HTTPS is required** in all production environments to ensure secure communication between clients and the server.
- If deployed via platforms like Railway or Vercel, **SSL is handled automatically** through their managed infrastructure.

WebSocket connections are also expected to use **WSS (WebSocket Secure)** when served over HTTPS.

Authentication Mechanism

- The application uses **JWT (JSON Web Tokens)** for stateless authentication.

- Tokens are issued by the backend server upon successful login and include user identification data and expiry information (tokens expire after 24 hours).
- All protected API routes validate the JWT on each request.
- Tokens are typically sent in the Authorization header as a **Bearer token**.

This approach ensures secure, scalable session handling and works well in both REST and WebSocket contexts.

6. **Database Deployment**

DBMS Required

The application currently uses **PostgreSQL** as the primary database management system during development and production.

However, since the project uses **Sequelize** (an ORM for Node.js), the underlying database dialect can be changed through configuration. Sequelize supports multiple dialects such as:

- **PostgreSQL** (default and recommended)
- **MySQL / MariaDB**
- **SQLite**
- **Microsoft SQL Server**

The dialect is set via environment variables allowing flexibility in DBMS selection depending on the deployment environment.

Initialization Scripts and Migrations

The database schema is managed using Sequelize’s built-in migration system. All required tables, relationships, and initial data can be set up using the following commands:

```
npx sequelize-cli db:migrate
```

```
npx sequelize-cli db:seed:all
```

- **Migrations** define the structure and schema of the database (tables, columns, indexes).
- **Seeders** include optional sample or default data such as roles, admin users, or settings.

This setup ensures repeatable and version-controlled database initialization for both development and production environments.

Database Hosting

The database is currently hosted on **Supabase**, which provides a managed PostgreSQL service with built-in support for real-time subscriptions, RESTful APIs, and security.

However, since the connection is handled via environment variables, the application is compatible with various hosting options:

- **Local development** using Dockerized or native PostgreSQL instances
- **Cloud-hosted managed databases** (e.g., Supabase, AWS RDS, Heroku Postgres)

Self-hosted databases on private servers or VMs

7. Rollback and Recovery

Deployment Rollback

Currently, rollbacks are handled manually through **Git version control**. Since the deployment is triggered via commits (e.g., on the develop branch), rolling back to a previous state can be done by:

- **Reverting to a previous commit**
- **Using Railway's deployment history**:
  Railway provides a deployment history where previous successful deployments can be redeployed with a single click. This allows quick recovery if a faulty deployment occurs.

A more robust CI/CD pipeline with pre-deployment checks and staging validation can be added in the future to reduce the need for manual rollback.

Database Backup and Recovery

At the moment, there is **no automated backup procedure** in place. However, since the database is hosted on **Supabase**, it provides:

- **Automated daily backups**
- **Manual backup and restore** options through the Supabase dashboard
- **SQL export** features for ad-hoc backups

8. Monitoring and Logging

Tools Used

Currently, no dedicated external tools such as **Prometheus**, **Grafana**, or the **ELK Stack (Elasticsearch, Logstash, Kibana)** are integrated for monitoring or centralized logging.

However, basic logging and monitoring are handled through the following:

- **Console Logging**:
  - During development, logs are written to the console using standard console.log, console.error, etc.

    - These logs provide real-time feedback on application behavior, errors, and key events.
- **Railway Dashboard**:
    - Railway provides **build logs**, **deployment logs**, and **runtime logs** directly in its web interface.
    - These logs are accessible in real time and are useful for debugging crashes, failed builds, or runtime exceptions.

Logs Generated and Their Location

- Logs are not currently persisted or shipped to an external storage or analysis service.
- Logs generated by the application (API requests, errors, warnings) appear in the Railway runtime console, and are **not stored long-term**.
- No structured logging (e.g., JSON format, request tracing) is currently implemented.

9. User Access and Roles

Access to the Deployed System

The system is **publicly accessible**, meaning anyone with the URL can access the application and register for an account. However, access to core features is restricted through an approval workflow.

User Registration and Provisioning

- Upon registration, a new user account is created in a **"pending" state**.
- The user can login, but cannot actively engage with the application until an **administrator manually approves** the account.
- This mechanism helps ensure only verified users gain access to the system.

User Management and Role Control

User accounts are **provisioned and managed by administrators** through an internal admin interface. Admins have the ability to:

- **Approve or reject pending accounts**
- **Delete user accounts**
- **Assign or change user roles** (e.g., buyer, seller)
- **View user activity or logs**

This approach provides a controlled access system while allowing for role-based permissions and scalable user management.

10. Testing in Deployment Environment

Testing in Deployment Environment

After each deployment, a **manual smoke test** is performed to verify that the core functionality of the application is working correctly. This includes:

- Checking that the application loads without errors
- Testing user login and registration
- Verifying that API endpoints return expected responses
- Ensuring WebSocket communication is functional
- Confirming that admin and user dashboards are accessible

Currently, these checks are done manually by the development team using the live system.

Automated post-deployment testing is not yet implemented, but may be considered in the future to reduce human error and speed up verification.