# NCS B2B E-Commerce Platform – Month 1 Execution Plan (Docker Compose Edition)

This plan adapts the original Proxmox/LXC setup to a Docker Compose environment suitable for a single developer (macOS or Linux). We retain the modular-monolith architecture and feature scope (Next.js frontend, NestJS backend, PostgreSQL, Redis, BullMQ, PromptPay, Thai tax invoices) from the Technical Design Doc[[1]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=workflows%2C%20bulk%20pricing%20negotiations%2C%20and,and%20the%20outdated%20B2B%20norms). All steps are laid out Day 1–28, with exact CLI commands, acceptance checks, and rollback procedures. The platform will run via docker compose up with isolated bridge networks and persistent volumes for each service.

## 1. Day-by-Day Plan (Day 1–Day 28)

### Day 1: Project Kickoff and Repo Setup

* **Step:** Install Docker.
* *macOS:* Install [Docker Desktop for Mac](https://docs.docker.com/desktop/mac/install/) (download .dmg and follow GUI installer).
* *Linux:* Run sudo apt update && sudo apt install -y docker.io docker-compose.
* **Acceptance:** docker --version returns a version string.
* **Rollback:** Uninstall Docker (mac: use Docker’s uninstall, Linux: sudo apt purge docker.io).
* **Step:** Clone and initialize repo.
* git clone https://github.com/<your-org>/ncs-b2b-ecommerce.git  
  cd ncs-b2b-ecommerce  
  git init  
  git add .  
  git commit -m "Initial commit: project scaffold"
* **Acceptance:** git status shows clean, and ls shows project directory.
* **Rollback:** Remove the directory and reclone if needed.
* **Step:** Scaffold apps. Create backend/ and frontend/ subfolders:
* cd backend && npx @nestjs/cli new . --skip-git  
  cd ../frontend && npx create-next-app@latest . --typescript --eslint
* **Acceptance:** Verify backend has a NestJS project (e.g. ls backend/src); frontend has Next.js files (ls frontend/pages).
* **Rollback:** Delete the backend or frontend folder and re-run scaffolding commands.

*Citation:* The initial setup (version control, dev environment ready, Hello World apps) aligns with the blueprint’s “Phase 0” kickoff.

### Day 2: Docker Compose and Network Configuration

* **Step:** In project root, create docker-compose.yaml with isolated networks and volumes (see Section 2).
* **Step:** Create named Docker networks:
* docker network create web\_net  
  docker network create app\_net
* **Step:** Create persistent volumes:
* docker volume create pgdata redisdata
* **Acceptance:** docker network ls shows web\_net and app\_net; docker volume ls shows pgdata, redisdata.
* **Rollback:** Remove networks/volumes:
* docker rm -f <containers>;   
  docker network rm web\_net app\_net;   
  docker volume rm pgdata redisdata;
* **Step:** Start minimal services:
* cd backend  
  # Create .env file from .env.example later; for now use defaults  
  cd ..  
  docker-compose up -d postgres redis
* **Acceptance:** docker ps shows postgres and redis containers running. Check with:
* docker exec -it <postgres\_container\_id> psql -U postgres -c '\l'  
  docker exec -it <redis\_container\_id> redis-cli ping
* Both commands should report the database list and PONG, respectively.
* **Rollback:** docker-compose stop postgres redis && docker-compose rm -f postgres redis.

### Day 3: Backend Initial Build & DB Connection

* **Step:** In backend/, install TypeORM or Prisma. For Prisma:
* cd backend  
  npm install prisma @prisma/client  
  npx prisma init
* **Step:** Configure .env with PostgreSQL credentials:
* echo "DATABASE\_URL=postgresql://postgres:mysecretpassword@postgres:5432/ncsdb" >> .env
* **Step:** Create initial DB schema in Prisma (prisma/schema.prisma):
* datasource db { provider = "postgresql" url = env("DATABASE\_URL") }  
  generator client { provider = "prisma-client-js" }  
  model User {  
   id String @id @default(uuid())  
   email String @unique  
   password String  
   name String  
  }
* **Step:** Run migrations:
* npx prisma migrate dev --name init
* **Acceptance:** The migration applies, creating a User table with UUID PK. Connect from NestJS:
* npm install @nestjs/prisma  
  # Or configure TypeORM similarly if chosen
* Deploy NestJS boilerplate to test DB:
* docker-compose up -d backend  
  docker exec -it <backend\_container> npm run start:dev
* Verify the NestJS app can query the DB.
* **Rollback:** If errors, revert .env to previous, or drop and recreate migration:
* npx prisma migrate reset

*Citation:* We use Postgres and an ORM (Prisma/TypeORM) as recommended[[2]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=features%20%28JSON%20support%2C%20full,or%20partition%20data%20if%20needed)[[3]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=Cache%20%26%20Session%20Store%3A%20Redis,or%20queues%20for%20background%20tasks). This matches the plan’s “Hello World” and simple DB connection deliverables.

### Day 4: Frontend Initial Build & Hello World

* **Step:** In frontend/, install dependencies and run dev server:
* cd frontend  
  npm install  
  npm run dev &
* **Acceptance:** Open http://localhost:3000 in browser; the Next.js starter page appears.
* **Rollback:** Close the dev server process; revert any front-end file changes.

### Day 5: GitHub Actions CI Pipeline

* **Step:** Create .github/workflows/ci.yml (see Section 4) with jobs for linting, building, and testing both apps. For example:
* name: CI  
  on: [push, pull\_request]  
  jobs:  
   backend:  
   runs-on: ubuntu-latest  
   steps:  
   - uses: actions/checkout@v3  
   - name: Use Node.js  
   uses: actions/setup-node@v3  
   with: { node-version: '18' }  
   - run: |  
   cd backend  
   npm ci  
   npm run lint  
   npm run test  
   npm run build
* (Similarly for a frontend job.)
* **Acceptance:** Push to GitHub triggers the workflow; verify the CI run completes without errors.
* **Rollback:** Remove .github/workflows/ci.yml and revert commit if CI is failing critically.

*Citation:* Automated testing and deployment pipelines were part of the proposed strategy[[4]](file://file-99vRGybfzD8cjE33eWqvvY#:~:text=We%E2%80%99ll%20implement%20CI%2FCD%20pipelines%20for,keeping%20previous%20version%20containers%20ready)[[5]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=DevOps%20%26%20Deployment%3A%20We%20will,web%20vs%20API). We implement a GitHub Actions workflow for both projects as required.

### Day 6: Admin Console Scaffolding

* **Step:** Plan admin interfaces. In backend/, generate modules for admin management:
* cd backend  
  npx nest g module admin  
  npx nest g controller admin  
  npx nest g service admin
* **Step:** In frontend/, add an /admin route. For example, create pages/admin/index.tsx that lists orders. Use a basic HTML table or a React component.
* **Acceptance:** docker-compose restart backend frontend; then accessing http://localhost:3000/admin shows the new page (empty for now).
* **Rollback:** Delete the generated admin module and pages/admin file if restructuring is needed.

### Day 7: .env Configuration and Secrets

* **Step:** Create .env.example (see Section 3) with placeholders:
* POSTGRES\_PASSWORD=mysecretpassword  
  REDIS\_HOST=redis  
  REDIS\_PORT=6379  
  JWT\_SECRET=supersecretjwtkey  
  PROMPTPAY\_QR\_KEY=your-promptpay-key  
  NEXT\_PUBLIC\_API\_URL=http://localhost:3000
* **Acceptance:** Running docker-compose config (if using env) shows variables substituted.
* **Rollback:** Remove sensitive .env from version control; ensure .gitignore includes it.

### Day 8: Authentication & Company Accounts

* **Step:** Implement user auth (register/login). In backend/, install Passport/JWT:
* npm install @nestjs/jwt @nestjs/passport passport passport-jwt bcrypt
* Generate AuthModule and UserModule. Create endpoints for POST /auth/signup and POST /auth/login.
* **Step:** Define Company and Role models in schema. For Prisma:
* model Company {  
   id String @id @default(uuid())  
   name String @unique  
   users User[]  
  }  
  model User {  
   id String @id @default(uuid())  
   email String @unique  
   password String  
   name String  
   company Company @relation(fields: [companyId], references: [id])  
   companyId String  
   role Role @default(USER)  
  }  
  enum Role { USER ADMIN }
* Run npx prisma migrate dev --name auth\_company.
* **Step:** Seed an admin user and a sample company. In prisma/seed.ts:
* import { PrismaClient } from '@prisma/client';  
  const db = new PrismaClient();  
  async function main() {  
   await db.company.create({ data: { name: 'DefaultCo', users: { create: { name: 'Admin', email: 'admin@ncs.com', password: 'hashedpwd', role: 'ADMIN' } } } });  
  }  
  main();
* Run npx prisma db seed.
* **Acceptance:** Use a tool like Postman: POST http://localhost:3000/auth/signup with JSON body { "name": "User", "email": "u@co.com", "password": "pass", "company": "DefaultCo" }. Expect a new user tied to company “DefaultCo”. Login should return a JWT. Verify JWT\_SECRET environment is used.
* **Rollback:** Drop and revert migrations if schemas are incorrect: npx prisma migrate reset.

### Day 9: Catalog – Data Models and Seed

* **Step:** Add Category and Product models to schema.prisma:
* model Category {  
   id String @id @default(uuid())  
   name String  
   products Product[]  
  }  
  model Product {  
   id String @id @default(uuid())  
   name String  
   description String?  
   price Float  
   category Category @relation(fields: [categoryId], references: [id])  
   categoryId String  
   stock Int  
  }
* Run npx prisma migrate dev --name product\_model.
* **Step:** Seed core catalog data. For example, in prisma/seed.ts, add:
* await db.category.create({  
   data: {  
   name: 'Networking',  
   products: {  
   create: [  
   { name: 'Router X1000', description: 'High-speed router', price: 250.0, stock: 10 },  
   { name: 'Switch S200', description: '24-port switch', price: 150.0, stock: 5 }  
   ]  
   }  
   }  
  });
* Rerun npx prisma db seed.
* **Acceptance:** In NestJS, create a ProductModule and a controller GET /products. Query from DB:
* curl http://localhost:3000/products | jq
* Expect a JSON list of the two products seeded.
* **Rollback:** If bad data, delete products via npx prisma studio or drop and reseed.

*Citation:* Defining product and company data models aligns with the blueprint’s data model requirements. Seeding an admin and core products ensures acceptance of “core catalog” and admin user data.

### Day 10: Frontend – Catalog Browsing and Search

* **Step:** In frontend/, create pages: /products (list) and /products/[id] (detail). Use fetch or axios to call the backend API endpoints.
* **Step:** Implement a search bar on /products that queries /products?search=keyword.
* **Acceptance:** In browser, go to http://localhost:3000/products. A list of products (“Router X1000”, “Switch S200”) appears. Searching “Router” filters the list.
* **Rollback:** Remove or revert changes to pages if layout breaks.

### Day 11: Shopping Cart (Client and Server)

* **Step:** In frontend/, add a “Cart” page (e.g., /cart) and a React context or state to hold cart items. Provide “Add to Cart” buttons on product pages.
* **Step:** In backend/, create CartItem model in schema.prisma:
* model CartItem {  
   id String @id @default(uuid())  
   product Product @relation(fields: [productId], references: [id])  
   productId String  
   quantity Int  
   user User @relation(fields: [userId], references: [id])  
   userId String  
  }
* Run npx prisma migrate dev --name cart.
* **Step:** Create POST /cart and GET /cart API endpoints for adding and viewing cart items (filter by authenticated user).
* **Acceptance:** Log in as a user, add a product to cart via frontend. Verify GET /cart returns the item. The front-end /cart page shows the item and quantity.
* **Rollback:** If errors, remove the CartItem model and rollback migration: npx prisma migrate reset.

### Day 12: Cart CSV Upload

* **Step:** Install a CSV parser on frontend (e.g. papaparse) and add a file input on /cart:
* cd frontend  
  npm install papaparse
* In /cart page, parse uploaded CSV of format productId,quantity.
* **Step:** For each row, call POST /cart to add items.
* **Acceptance:** Use a test CSV (e.g., router.csv containing Router X1000,3). After upload, cart should contain 3 units of “Router X1000”.
* **Rollback:** Remove the CSV upload code if parse fails; allow manual entry as fallback.

### Day 13: RFQ and Quote Workflow

* **Step:** In backend/, create Quote and QuoteItem models:
* model Quote {  
   id String @id @default(uuid())  
   user User @relation(fields: [userId], references: [id])  
   userId String  
   status String @default("PENDING")  
   items QuoteItem[]  
   createdAt DateTime @default(now())  
  }  
  model QuoteItem {  
   id String @id @default(uuid())  
   quote Quote @relation(fields: [quoteId], references: [id])  
   quoteId String  
   product Product @relation(fields: [productId], references: [id])  
   productId String  
   quantity Int  
   unitPrice Float  
  }
* Run npx prisma migrate dev --name rfq.
* **Step:** Implement POST /quotes to create a quote from the current cart (convert cart items into quote items with current prices). Clear the cart after quoting.
* **Step:** Implement POST /quotes/:id/accept to convert the quote into an Order (see next day).
* **Acceptance:** As a user with 2 items in cart, call POST /quotes. A new quote ID returns. Calling GET /quotes/:id shows the quote details with status “PENDING”.
* **Rollback:** If incorrect, drop the Quote models and redo migration or correct schema.

### Day 14: Order Placement (Quote → Order)

* **Step:** In backend/, create Order and OrderItem models:
* model Order {  
   id String @id @default(uuid())  
   quote Quote @relation(fields: [quoteId], references: [id])  
   quoteId String  
   createdAt DateTime @default(now())  
   total Float  
   status String @default("PROCESSING")  
  }
* Run npx prisma migrate dev --name order.
* **Step:** Implement POST /orders/:quoteId to finalize an order: it consumes a quote and writes an order record with total amount. Mark quote as “COMPLETED”.
* **Acceptance:** After quoting, calling POST /orders/:quoteId returns an order. Verify GET /orders lists the new order with correct total.
* **Rollback:** If issues, drop Order models and reset with npx prisma migrate reset.

*Citation:* The RFQ-to-Order flow is core to the design[[6]](file://file-99vRGybfzD8cjE33eWqvvY#:~:text=digitized%20workflows,4). By Day 14, we have implemented quote creation and order placement as planned.

### Day 15: PromptPay Integration (QR Code)

* **Step:** Install a QR code library (e.g. qrcode) in frontend/:
* cd frontend  
  npm install qrcode.react
* **Step:** On checkout, if payment method is PromptPay, generate a QR code. Use a stub PromptPay ID or sandbox API key from the .env. In backend/, create a Payment model:
* model Payment {  
   id String @id @default(uuid())  
   order Order @relation(fields: [orderId], references: [id])  
   orderId String  
   method String  
   status String @default("PENDING")  
   transactionDate DateTime @default(now())  
  }
* Run npx prisma migrate dev --name payment.
* **Step:** In frontend, use <QRCode value="promptpay:<ID>?amount=<total>" /> (for sandbox QR) or call a PromptPay API.
* **Acceptance:** Completing an order as PromptPay payment should show a QR code on screen. Scanning it (simulated) marks payment received. In backend, simulate a webhook: POST /payments/confirm that updates the Payment status to “COMPLETED”. The order status then becomes “PAID”.
* **Rollback:** If QR generation fails, offer a fallback “Pay Offline” button.

*Citation:* Local payments via PromptPay QR were specified as a feature[[7]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=Payments%20%26%20Invoicing%3A%20Offer%20integrated,showing%20VAT%20breakdown%20on%20the). This step ensures QR generation works in sandbox mode.

### Day 16: Thai Tax Invoice PDF

* **Step:** In backend/, install a PDF library (e.g. pdfkit) or use NestJS-export. Generate a tax invoice PDF after payment. Extend Order or Invoice model if needed. For example:
* // In an order-completion service:  
  import PDFDocument from 'pdfkit';  
  const doc = new PDFDocument();  
  doc.text('NCS Networks Co. Ltd.');  
  doc.text(`Order ID: ${order.id}`);  
  // ... list items, VAT, total  
  doc.end();
* **Step:** Create GET /orders/:id/invoice.pdf endpoint that streams the PDF.
* **Acceptance:** Access http://localhost:3000/orders/<orderId>/invoice.pdf; a PDF downloads showing company header, order details, VAT breakdown, total in THB.
* **Rollback:** If PDF errors, revert recent changes and display a simple HTML invoice.

*Citation:* Generating a PDF tax invoice on order completion is required by the blueprint[[8]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=order%20summary%2C%20capturing%20company%20tax,DSS%20requirements%20%28using).

### Day 17: User Accounts and Roles (continued)

* **Step:** Implement multi-user/company roles. Ensure endpoints check user’s company and role. For example, orders from Company A are not visible to Company B’s users. Use Guards in NestJS.
* **Acceptance:** Test with two users in different companies: each sees only their own data.
* **Rollback:** Tighten or relax role restrictions if necessary.

### Day 18: Admin Console – Enhance

* **Step:** Flesh out the admin console. In frontend/, create pages under /admin for managing products, users, quotes, and orders. Use forms calling the backend’s CRUD endpoints (built via NestJS modules).
* **Acceptance:** Log in as an admin and verify you can add/edit products and view all orders.
* **Rollback:** Remove broken admin features as needed and notify stakeholders.

### Day 19: CI/CD Verification & Testing

* **Step:** Add unit and integration tests. In backend/, write a basic Jest test:
* npm install --save-dev jest @nestjs/testing ts-jest
* Create src/app.controller.spec.ts that tests one API endpoint.
* **Step:** In frontend/, install testing library (React Testing Library) and test that key pages render.
* **Acceptance:** Run npm run test in both apps; all tests should pass. The CI workflow should report green.
* **Rollback:** Fix or skip failing tests; ensure CI passes.

### Day 20: Healthchecks and Monitoring

* **Step:** Add simple healthcheck endpoints. In backend/, add GET /health that returns { status: 'ok' }.
* **Step:** Ensure nginx (if used) or docker-compose healthchecks. For example, in docker-compose.yaml:
* healthcheck:  
   test: ["CMD", "curl", "-f", "http://localhost:3000/health"]  
   interval: 30s; timeout: 10s; retries: 3
* **Acceptance:** After docker-compose up, docker ps shows healthy statuses.
* **Rollback:** Remove or adjust healthchecks if they cause false negatives.

### Day 21–24: Testing Flows End-to-End

* **Step:** Systematically test each feature with manual and automated E2E checks:
* **Auth & Accounts:** Sign up, login, logout; check role-based access.
* **Catalog & Search:** Browse categories, search by keyword.
* **Cart & CSV:** Add items manually and via CSV; modify quantities; clear cart.
* **RFQ → Quote → Order:** Create a quote from cart; verify order creation.
* **PromptPay:** Complete payment via QR stub and webhook; confirm order status.
* **Tax Invoice:** Download invoice PDF; verify contents.
* **Acceptance:** All above flows work without errors. Document any bugs.
* **Rollback:** For any failing flow, isolate the last change (use Git) and revert that commit, then fix incrementally.

### Day 25: Final Integration and Demo Prep

* **Step:** Perform a “demo run”: from a fresh clone, run docker compose up, then execute the RFQ-to-order scenario end-to-end.
* **Acceptance:** On a fresh environment, docker compose up launches all containers and the example data is present. A test order can be placed and a PDF invoice downloaded.
* **Rollback:** If something fails, check docker-compose logs and revert recent configs.

### Day 26–27: Documentation and Cleanup

* **Step:** Write README instructions for setup (Docker installation, docker compose up, environment vars).
* **Step:** Polish code: remove unused files, fix lint warnings.
* **Acceptance:** Codebase is clean; all lint jobs pass.
* **Rollback:** If documentation errors, correct typos and re-issue.

### Day 28: Review & Buffer

* **Step:** Review acceptance criteria one last time: verify RFQ-to-Order flow, PromptPay sandbox, tax invoice, and CI pass.
* **Acceptance:** All criteria are met. The developer conducts a brief retrospective.
* **Rollback:** If any criterion is unmet, iterate fixes for that specific feature.

## 2. Docker Compose Configuration (docker-compose.yaml)

version: '3.8'  
services:  
 postgres:  
 image: postgres:15  
 restart: unless-stopped  
 environment:  
 - POSTGRES\_USER=postgres  
 - POSTGRES\_PASSWORD=${POSTGRES\_PASSWORD}  
 - POSTGRES\_DB=ncsdb  
 volumes:  
 - pgdata:/var/lib/postgresql/data  
 networks:  
 - app\_net  
 healthcheck:  
 test: ["CMD", "pg\_isready", "-U", "postgres"]  
 interval: 10s; timeout: 5s; retries: 5  
  
 redis:  
 image: redis:7  
 restart: unless-stopped  
 volumes:  
 - redisdata:/data  
 networks:  
 - app\_net  
 healthcheck:  
 test: ["CMD", "redis-cli", "ping"]  
 interval: 10s; timeout: 5s; retries: 5  
  
 backend:  
 build: ./backend  
 restart: unless-stopped  
 env\_file: ./backend/.env  
 depends\_on:  
 postgres:  
 condition: service\_healthy  
 redis:  
 condition: service\_healthy  
 networks:  
 - app\_net  
  
 worker:  
 build: ./backend  
 command: npm run worker  
 restart: unless-stopped  
 env\_file: ./backend/.env  
 depends\_on:  
 redis:  
 condition: service\_healthy  
 postgres:  
 condition: service\_healthy  
 networks:  
 - app\_net  
  
 frontend:  
 build: ./frontend  
 restart: unless-stopped  
 environment:  
 - NEXT\_PUBLIC\_API\_URL=http://localhost:3000  
 ports:  
 - "3000:3000"  
 networks:  
 - web\_net  
 - app\_net  
  
 nginx:  
 image: nginx:stable  
 restart: unless-stopped  
 ports:  
 - "80:80"  
 volumes:  
 - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro  
 depends\_on:  
 frontend:  
 condition: service\_started  
 backend:  
 condition: service\_started  
 networks:  
 - web\_net  
 - app\_net  
  
volumes:  
 pgdata:  
 redisdata:  
  
networks:  
 web\_net:  
 driver: bridge  
 app\_net:  
 driver: bridge

This compose.yaml defines six services: postgres, redis, backend (NestJS API), worker (BullMQ queue processor), frontend (Next.js), and nginx as a reverse proxy. Each service attaches to one or both networks (app\_net for internal communication, web\_net for public access). The pgdata and redisdata volumes persist database state. Healthchecks ensure containers report “healthy” before dependents start.

## 3. Environment File Template (.env.example)

# PostgreSQL  
POSTGRES\_PASSWORD=yourPostgresPasswordHere  
  
# Redis  
REDIS\_HOST=redis  
REDIS\_PORT=6379  
  
# JWT / Auth  
JWT\_SECRET=yourJWTSecretKeyHere  
  
# PromptPay (sandbox)  
PROMPTPAY\_QR\_ID=XXXXXXXX  
PROMPTPAY\_QR\_KEY=YYYYYYYYYYYYYY  
  
# Frontend  
NEXT\_PUBLIC\_API\_URL=http://localhost:3000

The .env.example above provides placeholders. In practice, **never commit actual secrets**; each developer or deployment sets real values. For local dev, update the .env in backend/ and .env.local in frontend/ accordingly.

## 4. GitHub Actions Workflow (.github/workflows/ci.yml)

name: CI  
  
on: [push, pull\_request]  
  
jobs:  
 backend:  
 name: Backend CI  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v3  
 - name: Setup Node.js  
 uses: actions/setup-node@v3  
 with: { node-version: '18' }  
 - name: Install & Test Backend  
 run: |  
 cd backend  
 npm ci  
 npm run lint  
 npm run test  
 npm run build  
  
 frontend:  
 name: Frontend CI  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v3  
 - name: Setup Node.js  
 uses: actions/setup-node@v3  
 with: { node-version: '18' }  
 - name: Install & Test Frontend  
 run: |  
 cd frontend  
 npm ci  
 npm run lint  
 npm run test  
 npm run build

This workflow builds, lints, and tests both backend and frontend on every push/PR. Successful runs ensure code quality and that new changes pass all checks, meeting the acceptance criterion that CI must build and test both apps.

## 5. Schema and Migrations

Using **Prisma** (similar steps apply for TypeORM). Our core schema (prisma/schema.prisma) includes UUID primary keys as required. Example (excerpt):

datasource db { provider = "postgresql" url = env("DATABASE\_URL") }  
generator client { provider = "prisma-client-js" }  
  
model Company {  
 id String @id @default(uuid())  
 name String @unique  
 users User[]  
}  
  
model User {  
 id String @id @default(uuid())  
 name String  
 email String @unique  
 password String  
 role Role @default(USER)  
 company Company? @relation(fields: [companyId], references: [id])  
 companyId String?  
}  
  
enum Role {  
 USER  
 ADMIN  
}  
  
model Category {  
 id String @id @default(uuid())  
 name String  
 products Product[]  
}  
  
model Product {  
 id String @id @default(uuid())  
 name String  
 description String?  
 price Float  
 stock Int  
 category Category @relation(fields: [categoryId], references: [id])  
 categoryId String  
}  
  
model Quote {  
 id String @id @default(uuid())  
 status String @default("PENDING")  
 user User @relation(fields: [userId], references: [id])  
 userId String  
 items QuoteItem[]  
 createdAt DateTime @default(now())  
}  
  
model QuoteItem {  
 id String @id @default(uuid())  
 quote Quote @relation(fields: [quoteId], references: [id])  
 quoteId String  
 product Product @relation(fields: [productId], references: [id])  
 productId String  
 quantity Int  
 unitPrice Float  
}  
  
model Order {  
 id String @id @default(uuid())  
 total Float  
 status String @default("PROCESSING")  
 quote Quote @relation(fields: [quoteId], references: [id])  
 quoteId String  
 createdAt DateTime @default(now())  
}  
  
model Invoice {  
 id String @id @default(uuid())  
 order Order @relation(fields: [orderId], references: [id])  
 orderId String  
 pdf Bytes  
 createdAt DateTime @default(now())  
}

After updating schema.prisma, run migrations:

npx prisma migrate dev --name init

This creates SQL migration files and applies them to the ncsdb database. We then seed core data by a script (prisma/seed.ts), which inserts at least one company (“DefaultCo”), an admin user, and sample products as shown on Day 9. Running npx prisma db seed populates the initial catalog and user.

## 6. End-to-End Test Checklists

For each retained feature, verify with scenarios:

* **Authentication:**
* Register a new user (POST /auth/signup).
* Log in (POST /auth/login) and receive a JWT.
* Access a protected endpoint (e.g. /orders) using the JWT in Authorization header – should succeed.
* Logout and verify token is invalidated (optional).
* **Company Accounts & Roles:**
* Create two users under the same company; verify both see company data.
* Create a user in a different company; ensure data is isolated.
* Verify ADMIN role can view all orders, USER role only own orders.
* Change a user’s role (simulate in DB) and verify access levels change accordingly.
* **Catalog Browsing & Search:**
* Visit /products: ensure all seeded products are listed (page load).
* Use the search box to filter by name; results should update correctly (e.g. searching “Router” shows only router products).
* Click a product to go to /products/[id]; confirm details (name, price, stock) are correct.
* **Cart with CSV Upload:**
* Add items to cart manually (click “Add to Cart” buttons). Ensure they appear in /cart page with correct quantities.
* Prepare a CSV file (headers productId,quantity) and upload on /cart. Confirm cart now includes all CSV items.
* Remove an item from cart (if UI allows) and verify cart updates.
* **RFQ to Quote to Order Flow:**
* While logged in, with items in cart, click “Request Quote” (calls POST /quotes). Receive a quote ID.
* View the quote (GET /quotes/:id): it should list line items and status “PENDING”.
* Convert quote to order (POST /orders/:quoteId). Receive an order ID.
* Confirm order appears under /orders and its status is “PROCESSING” or similar.
* **PromptPay QR (Sandbox):**
* At checkout, choose “PromptPay” method. A QR code should display with the order amount.
* Simulate scanning: call the webhook endpoint (POST /payments/confirm) or otherwise notify the backend of payment.
* Verify the Payment record status becomes “COMPLETED” and the corresponding Order status updates to “PAID”.
* **Thai Tax Invoice PDF:**
* After an order is paid, click “Download Invoice”. This calls GET /orders/:id/invoice.pdf.
* A PDF file should be downloaded. Open it: it must show NCS’s header, order details (items, quantities, prices), VAT breakdown, total amount, and invoice numbering.
* Confirm the PDF can be opened without error and content matches the order.

Passing all these checks confirms that the MVP’s critical flows operate end-to-end as expected.

## 7. Admin Console Scaffolding Plan

The admin console enables NCS staff to manage the platform. Key points:

* **Roles & Authentication:** Only users with ADMIN role should access admin routes. Implement NestJS guards and protect Next.js /admin pages.
* **Product Management:** In admin/products, allow creating/editing/deleting categories and products. Use forms that call backend CRUD endpoints (e.g. POST /products, PUT /products/:id).
* **User & Company Management:** In admin/users, list all users and their companies. Allow assigning roles and linking users to companies. (Initially, basic JSON display is acceptable.)
* **Quotes & Orders:** Provide pages to view all quotes and orders across companies (for admin). E.g., /admin/quotes and /admin/orders fetch from GET /quotes and GET /orders and display in a table.
* **Re-use Templates:** Use a simple React admin template or Ant Design components to speed up UI development. For example, Ant Design’s Table and Form.
* **Iterative Delivery:** Implement one section at a time (e.g., start with products). Since admin UI is not part of MVP acceptance, it can remain minimal but with navigation links.

This scaffolding ensures admins can oversee core data. It follows the Week 9–10 plan for admin basics, though in Month 1 we only scaffold the framework and basic routes.

## 8. Migration-to-Proxmox Checklist

Although we develop on Docker Compose, future deployment on the original Proxmox architecture will require mapping:

| **Docker Service** | **Proxmox Equivalent** |
| --- | --- |
| postgres | LXC container or VM with Ubuntu and PostgreSQL installed |
| redis | LXC container with Redis installed |
| backend (NestJS) | LXC container (or VM) running Node.js API server |
| worker | LXC container running the NestJS worker (BullMQ processor) |
| frontend (Next) | LXC container (Node) or static file host for Next.js server |
| nginx | LXC container (or VM) running Nginx as reverse proxy |

Per the Day 2 Provisioning Guide, each service runs in its own LXC or VM for isolation. For example, on Proxmox we might have: **CT100** = Postgres, **CT101** = Redis, **CT102** = API + Worker, **CT103** = Frontend, **CT104** = Nginx. The Docker bridge networks correspond to Proxmox virtual network or VLAN segments (e.g. separate LANs for front-tier and database). All environment variables and volumes would be configured via LXC container configuration files or startup scripts. (These mappings ensure parity between our compose dev setup and the target production provisioning.)

*Citation:* The Technical Design Doc specified a Proxmox VE cluster for on-prem deployment[[1]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=workflows%2C%20bulk%20pricing%20negotiations%2C%20and,and%20the%20outdated%20B2B%20norms); our migration checklist matches each Compose service to a Proxmox container.

## 9. Decision Log

* **Docker Compose vs. Proxmox:** We chose Docker Compose for Month 1 to simplify a solo-developer setup[[5]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=DevOps%20%26%20Deployment%3A%20We%20will,web%20vs%20API)[[9]](file://file-99vRGybfzD8cjE33eWqvvY#:~:text=We%20will%20containerize%20the%20application,prem%20install). *Rationale:* No need to manage VMs locally; Compose replicates the containerized architecture. *Reversion:* In future, we can convert each docker-compose service into LXC containers per the provisioning guide.
* **Prisma ORM:** Decided on Prisma for quicker schema migrations and seeding. *Rationale:* Rapid iteration with code-first schema and easy seeding. *Reversion:* Could switch to TypeORM if needed; migrations exist in SQL.
* **NestJS (Backend) and Next.js (Frontend):** Used as per scope. *Rationale:* Matches blueprint’s stack and allows SSR/PWA benefits[[1]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=workflows%2C%20bulk%20pricing%20negotiations%2C%20and,and%20the%20outdated%20B2B%20norms). *Reversion:* Could swap the frontend framework but backend Node/Express is non-negotiable for code reuse.
* **Single Repo (Mono-repo):** Both apps in one repo for simplicity. *Rationale:* Easier version control and CI. *Reversion:* Could split into separate repos; CI would need adjusting.
* **Localhost Domain (no SSL):** Using http://localhost for all services. *Rationale:* Avoid SSL headaches during dev. *Reversion:* In production on Proxmox, we’ll configure Nginx with proper TLS.

Each deviation preserves functionality; if issues arise, we have clear undo steps (e.g., docker-compose down, revert PRs, or run migrations resets).

*Citation:* Adapting to Docker aligns with the container strategy in the blueprint[[5]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=DevOps%20%26%20Deployment%3A%20We%20will,web%20vs%20API) and respects the original “on-premise” requirement[[1]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=workflows%2C%20bulk%20pricing%20negotiations%2C%20and,and%20the%20outdated%20B2B%20norms) while trading off Proxmox.

## 10. Risk Register

| **Risk** | **Impact** | **Likelihood** | **Mitigation Strategy** |
| --- | --- | --- | --- |
| **Solo dev throughput** | Schedule delays | Medium | Strict daily goals (as above); limit scope to MVP features; use LLM/codegen for boilerplate when possible. |
| **Data loss (local env)** | Lost progress/configuration | Low | Use persistent volumes; commit configuration to Git; regularly backup volumes or export DB dumps. |
| **Payment processing errors** | Failed transactions | Low-Medium | Use sandbox PromptPay; implement idempotent callbacks; log all payment attempts; allow offline payment fallback. |
| **PDF generation issues** | Invoice compliance risk | Low | Validate PDFs early (Day 16); keep a simple HTML invoice as backup; use a tested PDF library. |
| **SAP/ERP sync delay (future)** | Data inconsistency with ERP | Medium | Queue orders for later sync; build robust retry logic; mark orders as “pending” if sync fails. |
| **CI/CD pipeline failure** | Delayed integration | Low | Keep CI simple; commit small increments; ensure ci.yml caches dependencies. Notify on failures immediately. |
| **Docker environment mismatch** | “Works on my machine” bugs | Medium | Document exact Docker versions; use Docker Desktop same version; test on both macOS and Linux VM if possible. |
| **Network configuration issues** | Service connectivity errors | Low | Define clear networks in Compose; test inter-container communication each day; use healthchecks. |

Each risk is monitored daily with backups (e.g. docker volume inspect pgdata), and the rollback steps above provide a fast recovery path.

*Citation:* Proactive risk management (e.g. backup, testing) was advised in the blueprint[[10]](file://file-ELei6cHNQfbdEsYr1r6uo7#:~:text=3,Service%20Gap)[[11]](file://file-ELei6cHNQfbdEsYr1r6uo7#:~:text=3,Gap); our strategies (persistent volumes, test vs. live separation) follow those guidelines.

[[1]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=workflows%2C%20bulk%20pricing%20negotiations%2C%20and,and%20the%20outdated%20B2B%20norms) [[2]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=features%20%28JSON%20support%2C%20full,or%20partition%20data%20if%20needed) [[3]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=Cache%20%26%20Session%20Store%3A%20Redis,or%20queues%20for%20background%20tasks) [[5]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=DevOps%20%26%20Deployment%3A%20We%20will,web%20vs%20API) [[7]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=Payments%20%26%20Invoicing%3A%20Offer%20integrated,showing%20VAT%20breakdown%20on%20the) [[8]](file://file-Pi8X5EKhLZSK4GWXhGGajZ#:~:text=order%20summary%2C%20capturing%20company%20tax,DSS%20requirements%20%28using) Streamlined Product Blueprint – NCS Networks B2B E-Commerce Platform.docx

<file://file-Pi8X5EKhLZSK4GWXhGGajZ>

[[4]](file://file-99vRGybfzD8cjE33eWqvvY#:~:text=We%E2%80%99ll%20implement%20CI%2FCD%20pipelines%20for,keeping%20previous%20version%20containers%20ready) [[6]](file://file-99vRGybfzD8cjE33eWqvvY#:~:text=digitized%20workflows,4) [[9]](file://file-99vRGybfzD8cjE33eWqvvY#:~:text=We%20will%20containerize%20the%20application,prem%20install) 1.2. Streamlined Product Blueprint for NCS Networks B2B E-Commerce Platform.docx

<file://file-99vRGybfzD8cjE33eWqvvY>

[[10]](file://file-ELei6cHNQfbdEsYr1r6uo7#:~:text=3,Service%20Gap) [[11]](file://file-ELei6cHNQfbdEsYr1r6uo7#:~:text=3,Gap) Competing and Winning in Thailand's B2B Networking Equipment Market.docx

<file://file-ELei6cHNQfbdEsYr1r6uo7>