Full-Stack Engineer (.NET 10 / Next.js) AI-Assisted Build Test (90 Minutes)

Purpose

This test is intentionally designed to allow and encourage the use of AI tools (ChatGPT, Claude Code, Copilot, Cursor, etc.).

We are evaluating: - How you think and prioritize under a strict timebox - How you prompt AI and *critique* its output - How you turn AI assistance into **shippable**, maintainable code - Multi-tenant / multi-branch design maturity (Thailand-first, global-ready)

We are **not** testing memorization.

Timebox

⏱️ **90 minutes (strict)** - You are not expected to finish everything - Trade-offs are expected - Over-engineering is discouraged - **Correct boundaries > feature count**

Scenario

You are building v1 of a **Clinic POS** platform.

This is a **multi-tenant, multi-branch** B2B system: - 1 Tenant → many Branches - 1 Patient belongs to exactly 1 Tenant, and may visit multiple Branches - **Tenant data isolation is mandatory** (no cross-tenant exposure)

You are not required to implement enterprise-grade tenancy enforcement, but your design must be **tenant-safe by default**.

Technology constraints (mandatory)

- Backend: **.NET 10 / C#**
- Frontend: **Next.js**
- Database: **PostgreSQL**
- Cache: **Redis**
- Messaging: **RabbitMQ**

What makes this test

We are looking for candidates who can ship a **thin vertical slice** with: - Clear domain boundaries - Correctness under concurrency - Tenant-safe data access patterns - Minimal but meaningful automated tests - Runnability on another machine with one command

If you try to build “everything”, you will fail the timebox.

Task set — You have 90 minutes

You must complete **ALL of Section A + Section B** and **ANY TWO** of Section C / D / E.

Strong candidates will also include a small test suite and a clean runbook.

Section A — Core slice (mandatory)

A1. Implement a working thin slice (backend + frontend)

Build a usable flow end-to-end: - Create Patient - List Patients

Minimum backend requirements

- REST API with request validation and consistent error responses
- Persistence in PostgreSQL (migrations required)
- **Tenant-safe filtering on all reads/writes**

Minimum frontend requirements

- Next.js UI that can:
    - Create a patient
    - List patients
    - Filter by Branch (optional)
- Basic usability: fast, simple forms; no heavy UI work required

A2. Create Patient

Patient fields (minimum): - FirstName (required) - LastName (required) - PhoneNumber (required) - TenantId (required) - CreatedAt (server-generated) - (Optional) PrimaryBranchId

Rules: - PhoneNumber must be unique **within the same Tenant** (not globally) - Return a safe error on duplicate

A3. List Patients

List with: - Required filter: **TenantId** - Optional filter: BranchId - Sorted by CreatedAt **DESC**

> You may model “patient visits branches” minimally (e.g., PrimaryBranchId or a separate mapping table). Explain your choice.

Section B — Authorization & user management (mandatory)

B1. Roles

Implement role-based permissions: - Admin - User - Viewer

B2. Permissions (minimum)

Define and enforce permissions for: - Creating patients - Viewing patients - Creating appointments (even if you don’t implement appointments)

B3. User management (minimal scope)

Implement API endpoints: - Create User - Assign Role - Associate User with Tenant and one or more Branches

Required enforcement

- Requests must include an authenticated identity (choose JWT, cookie session, or simple token)
- Authorization must be enforced server-side (policy/attribute/middleware)
- **Viewer cannot create patients**

> Stubbing auth is acceptable only if permissions are still enforced reliably.

B4. Seeder (required)

Provide a seeder that creates: - 1 Tenant - 2 Branches - Users for each role (Admin, User, Viewer) - Correct tenant/branch associations

Seeder must be runnable via one command.

Section C — Appointment + messaging (choose any two sections total)

C1. Create Appointment

Implement Create Appointment with: - TenantId, BranchId, PatientId - StartAt (datetime) - CreatedAt

C2. Prevent exact duplicate booking

Prevent duplicates **within the same Tenant** for: - Same PatientId + same StartAt + same BranchId

> Must be safe under concurrency (prefer DB unique constraint + friendly error).

C3. Publish event to RabbitMQ

Publish an event when appointment is created: - Event name of your choice - Payload must include TenantId

Consumer is optional.

Section D — Caching & data access (choose any two sections total)

D1. Cache at least one read path

Cache **List Patients** (or another meaningful GET).

D2. Cache key strategy for tenant isolation

Keys must be tenant-scoped (e.g., `tenant:{tenantId}:patients:list:{branchId|all}`)

D3. Invalidation

On Create Patient (and Create Appointment if implemented): - Invalidate relevant cache keys - Or use versioned keys with tenant-scoped version bump

---

Section E — Architecture & evolution (choose any two sections total)

Choose ONE:

E1. Define one domain event

Example: `PatientCreated` or `AppointmentBooked` - Where it's emitted - What guarantees it provides - Whether it's in-transaction or outbox-based (lightweight explanation ok)

E2. Propose a tenant isolation strategy

In README: - How TenantId is derived (token/claims/header) - How it is enforced in the data access layer - How you prevent accidental missing filters

E3. Microservice-ready without premature microservices

Show how this could split later: - Bounded contexts - Message contracts - Shared vs duplicated schema

---

Mandatory execution requirements

Your submission must be runnable by others.

Docker & one-command run (required)

Provide: - `docker compose up --build` - Starts: PostgreSQL, Redis, RabbitMQ, backend, frontend

Migrations

Database migrations must apply automatically on startup **or** via documented command.

Minimal tests (required)

Provide **at least 3 automated tests**: - 1 backend test: tenant scoping enforced (cannot read other tenant) - 1 backend test: duplicate phone prevented within tenant - 1 frontend or integration smoke test (can be minimal)

> We care more about what you choose to test than coverage.

Required deliverables

```
/src
  /backend
  /frontend
README.md
AI_PROMPTS.md
```

README.md (required)

Must include: - Architecture overview (how tenant safety works) - Assumptions and trade-offs - How to run (one command) - Environment variables (`.env.example`) - Seeded users and how to login - API examples (curl) - How to run tests

---

AI_PROMPTS.md (mandatory)

Include: - Exact prompts used - Iterations - Accepted vs rejected AI outputs - How you validated correctness

> We evaluate your *AI judgment*, not your AI usage volume.

Submission requirements (important)

- Push your solution to **GitHub**
- Repository must be **PUBLIC**
- Email us the **GitHub repository link**

Rules

- Any AI tools allowed
- You may stub non-core infrastructure if needed, but **data flow must be real** for the slice
- Over-engineering is discouraged
- Time management is part of the test

Final instruction

Do not aim to finish everything. Aim to demonstrate: - Tenant-safe thinking - Correctness under real constraints - Minimal, maintainable architecture - Ability to ship a real slice fast