MIGX — Full Stack Technical Challenge Documentation

Engineering Review Submission

Edu Martorell Ortuño

1. Executive Summary

Project Name: migx
Goal: Develop a full-stack application capable of performing full CRUD operations on a Participant entity for a clinical trial management dashboard.
Challenge Context: Designed as part of a 4-hour technical challenge for Data & AI Engineering — evaluating architecture thinking, full-stack capability, and communication clarity.
The migx project provides a foundational full-stack architecture connecting a FastAPI backend with a Vue 3 frontend. It demonstrates separation of concerns, a clean modular structure, and Docker-based containerization for reproducibility.

2. System Architecture Overview

2.1 Frontend

Tech Stack:
- Framework: Vue.js 3.5.22 + TypeScript
- Build Tool: Vite
- Styling: TailwindCSS 4.1.16 + PrimeVue 4.4.1
- State Management: Pinia 3.0.3

Architecture Structure:
- assets/: Static images and icons
- domain/: Domain classes and interfaces shared across components
- pages/: Vue components bound to routes
- partials/: Modular Vue components organized by entity (e.g., Participant)
- services/: API service files per entity; return domain objects when possible
- stores/: Pinia stores separated by entity or functionality
- utils/: Helpers such as httpCommon and general utilities

Core files: App.vue, router/index.ts, main.ts

2.2 Backend

Tech Stack:
- Language: Python 3.11.7
- Framework: FastAPI
- Database: MongoDB 7.0
- ORM/DAO Pattern: Custom DAO abstraction per entity

Architecture Structure:
- control/: Controllers implementing main functionality
- data/: DAOs per entity; handles MongoDB interaction
- domain/: Domain models for validation and serialization
- routers/: API routers for each entity
- services/: Shared service implementations (e.g., mongo_dao)
- main.py: FastAPI application entrypoint
- settings.py: Configuration and .env imports

2.3 Infrastructure & Containerization

Containerization:
- Orchestration: Docker + Docker Compose
- Services:
  - mongodb — MongoDB database with initialization scripts
  - backend — FastAPI server
  - frontend — Vite-built Vue app served via lightweight container

docker-compose.yml summary:
- Shared network: migx-network
- Volumes: mongodb_data, mongodb_config
- Automatic service health checks for MongoDB
- Backend starts after database initialization
- Frontend depends on backend availability

Additional Artifacts:
- db/: Database init scripts (initdb.sh, participants.json)
- frontend/Dockerfile: Frontend build and runtime image
- backend/Dockerfile: Backend image setup

2.4 Git and version control

Version control for this project is managed through **Git**, hosted on **GitHub** under the following repository:

edmaor/migx-full-technincal-challenge

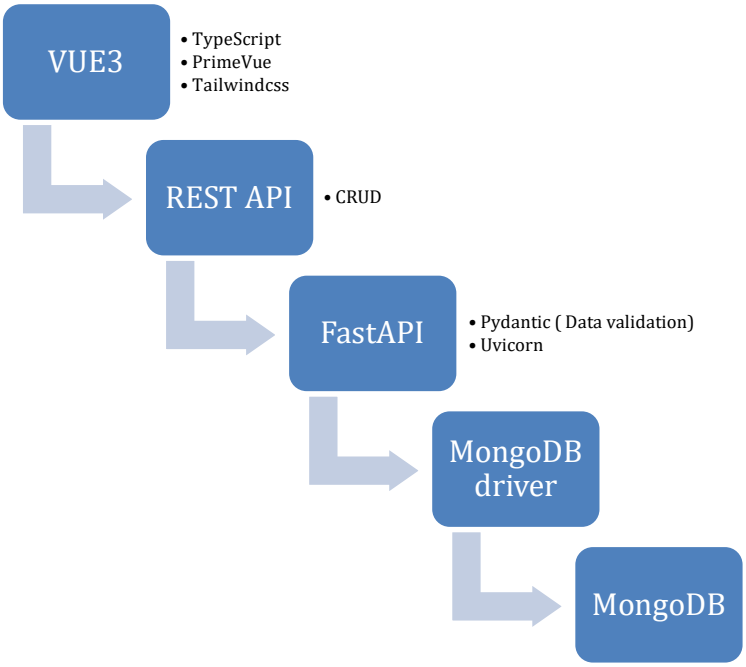
This repository serves as the single source of truth for the entire project — including backend, frontend, and infrastructure components — enabling reproducibility, collaboration, and traceability of all development changes.

Best Practices Applied

- Regular commits to maintain incremental progress visibility.
- Clear separation between frontend and backend modules.
- All Docker-related configurations versioned for reproducibility.
- `.env` files excluded from version control to protect sensitive credentials.
- GitHub repository configured for public access (for evaluation purposes).

```
Repository Base Structure:
├── backend/
├── frontend/
├── docker-compose.yml
└── README.md
```

3. Simple Data Flow Diagram



4. Architecture Decision Record (ADR) Summary

ADR-001 — Frontend Framework: Vue.js 3

Context: Needed a modern JavaScript framework with reactive components and TypeScript support.
Decision: Vue.js 3 with Composition API and Vite build tool.
Alternatives Considered:

- React (larger ecosystem but more setup overhead)
- Angular (heavy, more suitable for enterprise-scale apps)

Rationale: Vue offers high productivity, clean syntax, and a gentler learning curve for rapid prototyping. Native TypeScript support simplified type safety.
Consequences: Vue's smaller ecosystem compared to React limits third-party integrations but benefits maintainability and readability.

ADR-002 — Backend Framework: FastAPI

Context: Required a Python backend that is modern, performant, and integrates well with async I/O and schema validation.
Decision: Chose **FastAPI**.
Alternatives: Flask, Django REST Framework.
Rationale:

- Asynchronous performance.
- Built-in validation and serialization using Pydantic.
- Auto-generated OpenAPI docs (SWAGGER).

ADR-003 — Database: MongoDB 7.0

Context: Needed flexible schema for fast prototype iterations and JSON-like data modeling.
Decision: Chose **MongoDB**.
Alternatives considered: PostgreSQ.
Rationale:

- Natural JSON storage and dynamic document structure.
- Fits NoSQL needs for early-stage projects.

Consequences: Lacks strict schema enforcement — mitigated with domain validation models.

ADR-004 — Containerization and Orchestration

Context: The project must run in a reproducible local environment with backend, frontend, and DB coordination.
Decision: Docker + Docker Compose.
Alternatives: Local Python/Node environments, cloud deployment.
Rationale:

- Guarantees consistent environment for testing and evaluation.
- Enables future scalability to multi-service deployments.

Consequences: Slight increase in setup complexity but full portability.

ADR-005 — Code Organization and Architecture Pattern

Context: Wanted clear modular boundaries between business logic, data access, and presentation.
Decision: Adopted a **layered architecture** with `control`, `data`, `domain`, `routers`, and `services`.
Rationale: Improves readability, facilitates testing, and allows swapping layers (e.g., different DB or UI).
Consequences: Slight verbosity in small projects but essential for scalability.

ADR-006 — Authentication (Planned)

Context: Security is critical for clinical data; however, implementation was out of initial 4-hour scope.
Decision: Defer authentication implementation; frontend ready for JWT integration.
Planned Implementation: JWT-based authentication with route guards in frontend and role-based access control in backend.
Alternatives: OAuth2 or session-based auth for multi-user integration.
Consequences: Current API is public; to be mitigated in next iteration.

ADR-007 — Testing & CI/CD (Planned)

Context: No testing implemented due to time limit.
Decision: Planned use of **Pytest** (backend), **Vitest** (frontend), and **GitHub Actions** for CI/CD.

ADR-008 — AI Tooling and Coding Assistance

Context: Used AI-assisted coding for rapid prototyping and documentation under time constraints.
Decision: Leveraged **ClaudeAI** and **GitHub Copilot** for code scaffolding, bug fixing, and refactoring suggestions.
Rationale: Improved speed while maintaining developer oversight and comprehension.
Consequences: Productivity gain without sacrificing understanding or architectural ownership.

ADR-009 — Repository Structure: Monorepo vs. Multi-repo

Context: Typically, frontend and backend services would live in separate repositories to allow independent versioning, CI/CD pipelines, and scaling. However, the challenge required a single deliverable repository.

Decision: Adopted a **monorepo structure** containing `frontend/`, `backend/`, and shared Docker configuration files.

Rationale:

- The challenge instructions specified a single GitHub submission, making a unified repository the most practical approach.
- Simplifies reviewer setup — a single `docker-compose up` command runs the entire stack.
- Easier to manage shared assets (like `.env`, diagrams, and documentation).

Alternatives Considered: Separate repositories (`migx-frontend` and `migx-backend`) managed under an organization or workspace structure.

Consequences:

- Reduced modularity for independent deployments.
- CI/CD pipelines would need to be scoped carefully to avoid unnecessary cross-service rebuilds.
- Future iterations could split into multiple repos once the project matures beyond initial prototype scope.

5. Setup & Execution Instructions

5.1 Local Environment (development)

Requirements:
- Python ≥ 3.11
- Node.js ≥ 18
- MongoDB ≥ 7.0

Steps:
Backend:
```
cd backend
pip install -r requirements.txt
uvicorn app.main:app –host 0.0.0.0 -–port 8000 --reload
```

Frontend:
```
cd frontend
npm install
npm run dev
```

Access the app at http://localhost:3000

5.2 Docker & Docker Compose

To build and run all services:
```
docker compose up --build
```

Default ports:
- Frontend: http://localhost:3000
- Backend API: http://localhost:8000/docs
- MongoDB: localhost:27017

Environment Variables:
- MONGO_DB_HOST: mongodb
- MONGO_DB_PORT: 27017
- MONGO_DB_ROOT_USERNAME: admin
- MONGO_DB_ROOT_PASSWORD: admin123
- MONGO_DB_DATABASE: migx
- BACKEND_PORT: 8000
- FRONTEND_PORT: 3000

6. Future Improvements

1. Authentication & Authorization — JWT-based auth and protected frontend routes.
2. Participant Metrics Dashboard — data visualization and metrics analytics.
3. Generic DAO Layer — reusable MongoDAO base class.
4. Testing & CI/CD — add Pytest, Vitest, and GitHub Actions.
5. Observability — logging and monitoring integration.

7. Conclusion

The migx project establishes a modular and extensible architecture suitable for scalable clinical trial management solutions. It fulfills core CRUD functionality while laying the groundwork for authentication, analytics, and future automation through CI/CD pipelines. The use of modern frameworks (Vue 3, FastAPI, MongoDB) and containerization ensures maintainability, portability, and efficient local deployment.