ChatGPT

ClassPilot Project Structure Overview

ClassPilot is envisioned as an **AI-powered student hub** that aggregates multiple LMS platforms (Google Classroom, Canvas, Moodle, etc.) into one unified system [1]. To build a solution of this scope, we will use a **monorepo architecture** – a single repository containing all components (frontend, backend, services, etc.) – to ensure cohesive development and easy code sharing across the project. Below, we outline a FAANG-level project structure for ClassPilot, using PascalCase naming for clarity, and incorporating Docker and local dev tools for easy testing and deployment.

Monorepo Architecture Choice

A **monorepo** is a single repository that houses multiple distinct projects with clear relationships [2]. This approach is ideal for ClassPilot because we have several interrelated parts (UI, API, integration modules, AI services) that should evolve in lockstep. Keeping them in one repo simplifies dependency management and ensures **"one source of truth"** for the entire application (Google famously uses a monolithic repository for most of its codebase to achieve this benefit [3]). In practice, a monorepo means all ClassPilot sub-projects (web app, backend services, libraries, etc.) live together, enabling atomic commits across components and easy code sharing. Key advantages of this approach include:

- **Simplified Code Sharing:** All modules can reference common utilities and models without the overhead of separate versioned packages [4] [5]. For example, the frontend and backend can use the same data model definitions and constants, avoiding duplication.
- **Consistent Tooling:** One build/test pipeline and shared config for all parts of the project, which improves developer productivity and reduces "it works on my machine" issues.
- **Atomic Changes:** Synchronized updates across frontend and backend – a single commit can update an API endpoint and the corresponding UI, keeping them in sync.
- **Visibility & Collaboration:** Every team member has visibility into all parts of the code, fostering better collaboration (a practice used at Google to keep thousands of developers aligned in one codebase [3]).

However, we will ensure the repo remains modular, not a single giant code blob. Each major component will be isolated in its own folder with clear boundaries (monorepo ≠ monolith [6]). This way we get the benefits of a monorepo while maintaining a modular, scalable architecture.

High-Level System Components and Modules

Before diving into directories, it's important to identify the **core components** of ClassPilot, each corresponding to major features of the product:

- **Frontend Application (Unified Dashboard UI):** The web or mobile client that students interact with. It presents a **unified dashboard for deadlines, announcements, and tasks** [7], as well as interfaces for AI tutoring, note viewing, etc. This will be a single-page app (e.g. React, Angular, or similar) or a mobile app, responsible for displaying data and providing a smooth user experience.

- **Backend API Server (Core Application Logic):** The server that aggregates data and handles business logic. This core service integrates with external LMS APIs to fetch assignments and updates, computes things like GPA and progress, and exposes a unified API to the frontend. It powers features like the **GPA calculator, goal tracking, and gamified nudges** [8] . This backend will handle authentication, data aggregation, and serve as the **central hub** of ClassPilot.
- **LMS Integration Modules:** Connectors or services that interface with external Learning Management Systems (Google Classroom, Canvas, Moodle). These modules handle API calls to fetch class data, deadlines, and submit assignments. Given ClassPilot **"connects all your LMS accounts"** into one hub [1] , we might have separate sub-components for each LMS (e.g. a GoogleClassroomService, CanvasService, MoodleService) encapsulating the peculiarities of each API. They will feed normalized data to the core backend. (For example, Google Classroom might require special handling since it doesn't allow direct turn-in via API [9] .)
- **AI Tutor Service:** A component providing the AI-powered study help – e.g. answering questions, summarizing notes, generating study timelines [10] . This could interface with external AI APIs (like OpenAI) or host ML models. We might implement this as a module within the backend or as a separate microservice for scalability. It will need access to course materials/notes to "understand your class materials" and answer questions [11] .
- **Gamification & Analytics Module:** Handles **gamified elements like XP, streaks, badges** and tracks user study behavior [9] . Also includes analytics like calculating current GPA, forecasting targets, and computing "minimum grade needed" for goals [12] . This logic lives on the backend (as part of the core service or a worker) and likely interacts with a database to update points and progress.
- **Notes & Content Management:** Manages user notes and uploaded content. ClassPilot plans to integrate notes (handwritten or typed) and even auto-surface them before deadlines [13] . This module might handle storing notes (in a database or cloud storage), linking notes to relevant assignments, and perhaps using AI to summarize or create flashcards from notes. It also might facilitate future features like **auto-transcribing lecture recordings** into notes [14] .
- **Database(s):** Although not a "module" folder, the architecture will include databases (SQL or NoSQL) to store unified data aggregated from all sources – e.g. user profiles, combined assignment lists, grades, note contents, gamification stats, etc. The project structure will include configuration for database migrations or models (for instance, in a `db/` or `models/` directory under the backend).
- **Background Workers (optional):** For certain tasks that are long-running or need scheduling (sending reminders, generating AI summaries in background, syncing with LMS periodically), we may include a worker process or service. In a large-scale setup, this could be a separate service in the repo responsible for asynchronous jobs (as is common in backend architectures [15]). For now, this can be part of the backend process (e.g. using job queues or cron tasks), but the structure will allow peeling it out into its own service when needed.

Each of these pieces will be reflected in our project structure either as separate top-level folders or as sub-folders within the frontend or backend code. The goal is to keep each concern isolated but accessible. Next, we detail the repository layout that brings these components together.

Repository Layout and Directory Structure

Below is the proposed top-level structure for the **ClassPilot** monorepo. Each **PascalCase** folder represents a major part of the system:

```
ClassPilot/                  📦 (Monorepo root)
├── FrontendApp/             📱 (Client-side application - unified dashboard UI)
│   ├── src/                 (Source code for the frontend app)
│   │   ├── components/      (UI components, widgets)
│   │   ├── pages/           (Page/screen components or routes)
│   │   ├── services/        (Frontend services for API calls, state management)
│   │   └── ... (other typical frontend structure: e.g., hooks/, styles/, etc.)
│   ├── public/              (Static assets like images, index.html if SPA)
│   ├── test/                (Frontend-specific tests, if any)
│   ├── package.json / ...   (Frontend build configuration, dependencies)
│   └── README.md            (Notes about building/running the frontend)
├── BackendAPI/                (Server-side API application and core logic)
│   ├── src/
│   │   ├── controllers/     (HTTP request handlers, e.g., assignmentController,
authController)
│   │   ├── models/          (Database models or ORM schemas for ClassPilot data)
│   │   ├── services/        (Business logic layer)
│   │   │   ├── integrations/   (LMS integration code for Google, Canvas, Moodle
APIs)
│   │   │   ├── ai/             (AI tutor logic, e.g., OpenAI API integration)
│   │   │   ├── gamification/   (XP, badges, GPA calculation logic)
│   │   │   ├── notes/          (Notes management logic)
│   │   │   └── ... (other domain-specific services)
│   │   ├── utils/           (Utility functions, helpers)
│   │   ├── config/          (Configuration files, e.g., for environment
variables)
│   │   └── app.js/ts        (Main application entry point, server setup)
│   ├── test/                (Backend tests – unit and integration tests for
services/controllers)
│   ├── package.json / ...   (Backend dependencies, scripts)
│   └── README.md            (Notes about running the backend, API documentation)
├── CommonLib/                 (Shared libraries and assets for reuse across
modules)
│   ├── constants/           (Shared constants, e.g., roles, static config
values)
│   ├── types/               (Shared TypeScript interfaces or data models used by
front & back)
│   ├── utils/               (Shared utility functions, e.g., date formatters)
│   └── ...                  (Any other common code that both frontend and
backend use)
├── Infrastructure/          ⚙ (DevOps and environment configuration)
│   ├── Dockerfile           (Docker image recipe for the backend API)
│   ├── Dockerfile.frontend (Docker image recipe for the frontend app, if
needed)
│   ├── docker-compose.yml  (Compose file to run multi-container dev
environment)
```

```
│   ├── env/                (Environment variable files or templates,
e.g., .env.development)
│   ├── nginx/              (Reverse proxy config if using Nginx in container
setup)
│   ├── k8s/                (Kubernetes deployment manifests, if applicable
later)
│   └── scripts/            (Utility scripts for setup, e.g., database
migration, seed data)
├── docs/                     (Documentation)
│   ├── API.md              (API endpoint documentation)
│   ├── architecture.md     (High-level architecture notes, possibly this
outline)
│   └── ... (design decisions, requirement specs, etc.)
└── README.md               (Root README with project overview and setup
instructions)
```

Note: All directory names are written in PascalCase (capitalized) as requested, for consistency and clarity. This structure is modular – each folder is a self-contained piece of the application – but it's all under the single **ClassPilot** repository. This mirrors common full-stack monorepo setups where frontend and backend live side by side [16]. For example, one guide illustrates a project with a React frontend and Node.js backend in the same repo, each in its own folder [16] – our layout follows the same principle.

Let's break down each major directory and its contents in detail:

FrontendApp (Unified Dashboard UI)

This folder contains the **client-side** application, which provides the unified student dashboard and interface. The exact technology can be chosen as needed (e.g. a React app, an Angular project, or even a React Native mobile app). Inside **FrontendApp/**, we have a typical structure for a modern web app:

- `src/`: The source code of the application. Key sub-folders might include:
- `components/` for reusable UI components (buttons, calendars, etc.).
- `pages/` or `screens/` for page-level components or route handlers (dashboard page, assignments page, profile page, etc.).
- `services/` for front-end logic like API clients (e.g. a module to call the BackendAPI endpoints), state management (could also be in a `store/` if using Redux or context providers), and utility functions specific to the UI.
- Additional folders as appropriate (e.g., `styles/` for styling, `hooks/` for custom React hooks, `context/` for context providers, etc.).
- We will keep this organized by feature if the app grows large (for example, a folder for "Assignments" components vs. "Notes" components).
- `public/`: Static files such as the main HTML file, favicon, and any static assets. If using a framework like Create React App or Next.js, this is where public files or static exports reside.
- `env/` **or config files**: The frontend may have environment-specific config (like API base URL, feature flags). For instance, a production vs. development `.env` or an `environments/` folder (commonly used in Angular projects).

- **Tests**: We include a placeholder `test/` directory (or we may colocate tests next to components). While you indicated tests might not be needed immediately, it's good practice to structure for them. We can add unit tests for UI components and integration tests for critical flows when ready.
- **Build/Config files**: Files like `package.json`, a bundler config (webpack, vite, etc., if not using a framework that abstracts it), and potentially a `tsconfig.json` (if using TypeScript). These reside in the FrontendApp root.
- **Dev server setup**: The FrontendApp will have scripts to run a local dev server for the UI. In a combined Docker environment, the frontend might be served either by a dev server or via a reverse proxy (like Nginx) container. For development ease, we can also proxy API calls to the backend (e.g., if using Create React App, set up `proxy` in package.json to forward `/api` calls to BackendAPI).

Overall, **FrontendApp/** encapsulates all code related to the user interface. Developers can run and test the frontend in isolation (for example, with mock API data), but in our monorepo they can also run it alongside the backend easily.

BackendAPI (Server + Business Logic)

The **BackendAPI/** directory contains the server-side application, which is the heart of ClassPilot's functionality. This is where we implement the APIs and internal logic that power the unified dashboard and AI features. Key parts of this directory include:

- `controllers/`: Defines the web API endpoints (e.g., RESTful endpoints or GraphQL resolvers). Each controller handles a specific slice of the API:
- For example, `AssignmentController` might handle routes like `/api/assignments` (listing upcoming deadlines, marking tasks as done, turning in assignments via LMS APIs), `AuthController` for authentication routes (login, OAuth with Google for Google Classroom access, etc.), `NotesController` for uploading/fetching notes, etc.
- Controllers orchestrate the underlying services and return responses to the client.
- `models/`: Represents the data models and database schema definitions. If using an ORM or ODM (like Prisma, Sequelize, Mongoose, etc.), those models live here. We'll have models for things like User, Course (or Class), Assignment (aggregated from different LMS), Note, Grade, etc. These correlate with the core data that ClassPilot manages internally.
- `services/`: This is the **business logic layer** containing the core modules of our backend. Each major feature area has its own service sub-module for separation of concerns:
- `integrations/`: Code for communicating with external LMS APIs. For each platform (Google Classroom, Canvas, Moodle), we could have separate service classes or modules (e.g., `GoogleClassroomService`, `CanvasService`, etc.). These handle authentication with the third-party API, fetching data (assignments, announcements, grades), and posting data (like submitting assignments). They abstract away the API details and expose simple methods to the rest of our app (e.g., `fetchAssignments(user)` which internally calls the respective LMS API). By modularizing this, if one LMS changes its API or if we add a new LMS in future, we can do so without touching other parts of the system.
- `ai/`: The AI Tutor service logic. This might include an interface to an AI platform (sending course materials or questions and getting answers). For example, an `OpenAIClient` integration could live here to handle Q&A and summaries. It might also include logic to cache AI responses or manage context so that the AI "understands your class materials" and can provide relevant help [17].

- `gamification/`: Encapsulates the gamified elements – calculating XP for completed tasks, managing streaks, issuing badges, etc. It also works with the Grades/GPA tracking: e.g., a function to calculate current GPA and a function to compute "target grade needed" for a goal [12]. This module can also handle analytics like generating progress visuals or sending nudges if the student is falling behind.
- `notes/`: Manages student notes and any content uploads. For instance, storing references to note files, generating AI summaries or flashcards from notes [13], and retrieving notes when needed (e.g., showing relevant notes before an exam or deadline). This service might interact with file storage (if notes are uploaded as images/PDFs) and possibly use OCR or transcription services (for future expansion like **lecture transcription** [14]).
- (Potential) `notifications/`: In future, if we add real-time notifications or email reminders, a service for handling those can be included.
- Other domain-specific services as needed: e.g., `userService` for profile and settings management, `analyticsService` for any tracking of user study habits, etc.
- `utils/` (within `src/`): General utility functions or helpers specific to backend (e.g., date/time utilities for deadlines, common formatting, error handling helpers, etc.). These are distinct from the **CommonLib** utilities, as they might be more specialized or not needed on the frontend.
- `config/`: Configuration files for the backend. This may include:
- Environment-specific config (like separate JSON or YAML files for development, testing, production settings).
- Initialization code for things like database connection, external API keys, etc. For instance, loading a `.env` file or a config module that reads environment variables for DB connection strings, API credentials (Google API keys for Classroom integration, OpenAI API keys for AI, etc.).
- Possibly an `express.js` or `app.ts` that sets up the web framework, if not at root.
- `app.js` **/** `server.ts` **/ etc.**: The main entry point of the backend service that ties everything together – e.g., setting up an Express server, registering controllers (routes) with their handlers, connecting to the database, and starting the server.
- **Tests**: A `test/` directory for backend tests. We can organize tests in a parallel structure to `src/` (e.g., `test/services/ai.test.js` to test AI service, `test/controllers/assignmentController.test.js` for controllers). Initially, this can be minimal or even empty, but the structure is there to encourage writing tests as the project grows. We might also include integration tests (for example, simulating a full flow of fetching LMS data and computing a GPA).
- **Dependencies & Build**: The backend likely has its own `package.json` (if Node.js) or equivalent (if we use another language/framework). This defines dependencies like Express/Nest (for API), Axios or Google SDK (for calling Google Classroom), database library, etc. If using TypeScript, a `tsconfig.json` lives here. If using something like a Python backend (Django/Flask), this folder could instead contain a `requirements.txt` or similar and the Django project structure – the concept remains the same. In any case, this folder is self-contained so the backend can be developed and run on its own.

Importantly, the backend is designed to be **modular**. This means, for example, the LMS integration code is segregated from the core logic – potentially even substitutable if we change data sources. The AI module can be developed relatively independently and could scale out (we could run it as a separate service later by extracting it, while still sharing the repo). This separation follows good design practices and prepares us for future growth or even microservice extraction if needed [15]. (E.g., in a high-scale scenario, one could spin off a separate **worker service** for heavy tasks like generating AI study plans or handling bulk LMS sync, similar to how large systems use background worker processes [15].)

CommonLib (Shared Code & Utilities)

The **CommonLib/** directory contains code that is shared across different parts of the project – essentially a small internal library. By centralizing shared pieces here, we avoid duplicating logic in the frontend and backend. Typical contents:

- `constants/`: Application-wide constants. For example, enumerations for user roles, names of events, or even API endpoint paths. If both front and back need to know about certain static values (say, a list of supported LMS names or error codes), defining them once here ensures consistency. In a monorepo, sharing constants and types is a big benefit [5].
- `types/`: Shared TypeScript types/interfaces or data models. If we use TypeScript on both frontend and backend, we can define interfaces for entities (like an `Assignment` interface with properties like title, dueDate, sourceLMS, etc.) once and import it in both places. This ensures the frontend's expectations of the API match the backend's output exactly [18]. Even if different languages are used, this can store API contract definitions (like OpenAPI/Swagger schemas or GraphQL schema definitions) that both sides adhere to.
- `utils/`: Utility functions that are generic enough to be used anywhere. For example, date formatting helpers (to display due dates nicely in the UI and also use in backend reports), parsing functions, validation logic that might run on both client and server (like input sanitization), etc.
- **Shared UI components or styling** (if applicable): If we eventually have a design system that might be used in both a web app and a potential mobile app, those could be here (though often frontend-only shared components might reside under FrontendApp or a `libs/frontend` folder in larger setups [19]). Since currently we have one main frontend, we can keep UI components there; CommonLib is more for non-UI shared code.
- **Configuration schemas**: Another use-case – if both front and back need access to some configuration (like a list of feature flags or something), it could be here.

By having CommonLib, we uphold the DRY (Don't Repeat Yourself) principle across our monorepo. This is one of the key advantages of monorepos, as it **avoids inconsistent duplication** that would happen if separate repos had their own copies of common logic [4]. For example, instead of defining a Grade calculation function twice, we put it in CommonLib and use it in both the GPA tracking service in the backend and perhaps in a client-side what-if GPA calculator for immediate feedback. (If using different languages, we might rewrite it, but at least the formula or pseudo-code lives in docs/common.)

Infrastructure (DevOps, Docker, and Environment Configuration)

The **Infrastructure/** folder (or we could call it **DevOps/** or **Deployment/**) contains all the configuration and tooling for running the application outside of code. This includes Docker setup for local development, and can expand to CI/CD config or cloud deployment scripts as the project matures. Key elements:

- **Dockerfiles**: We include separate Dockerfiles for each major component that we might containerize. For example, `Dockerfile` for the backend API and a `Dockerfile.frontend` for the frontend (or we could place the latter inside `FrontendApp/` for context). These Dockerfiles define how to package the app into a container (install dependencies, build the app, etc.). For development, we might use lighter weight images (with hot-reload capability), and for production, a multi-stage build to get optimized images. The presence of these files means any contributor can spin up the app in a containerized environment and not worry about host machine setup issues.

- **Docker Compose**: A `docker-compose.yml` at the root (or in this folder) to orchestrate multi-container setup. Using **docker-compose** is extremely useful for local dev and testing, as it can bring up the entire stack with one command. For ClassPilot, our compose setup would likely include:
- A service for the backend (built from the BackendAPI Dockerfile).
- A service for the frontend (if we serve it via a container, e.g., using Nginx to serve static files or a node dev server).
- A database service (e.g., a Postgres or MongoDB container) so that developers don't need to install a DB locally. This ensures consistent environment across the team.
- Possibly a Redis or message queue service if we use one for background jobs or caching.
- An Nginx service as a reverse proxy to route requests (common in Docker setups to route `web` to frontend and `/api` to backend, for example).

With Docker compose, a new developer can run `docker-compose up` and have the whole system running – avoiding the "works on my machine" problem by standardizing the environment [20]. This approach ensures we develop seamlessly without OS-level configuration errors [20]. - **Environment Variable Files**: Often we don't want to hard-code config in images. We might have an `env/` directory containing env var files (e.g., `.env.development`, `.env.production`) that are used by Docker compose or by the applications themselves. These would store things like database credentials (for local dev DB), API keys for third-party services (in dev maybe dummy keys), etc. For security, sensitive values wouldn't be committed – instead we commit sample env templates (like `.env.example`) and each dev or deployment provides the real values. - **Kubernetes or Deployment Scripts**: In a production scenario, we might deploy via Kubernetes or another platform. We can include a `k8s/` folder with YAML manifests (deployment, service definitions for each component) or a Helm chart. If using Terraform or other Infra-as-code, those scripts could reside here as well. This is more for later production deployment; initially, focus is on Docker for local and maybe simple cloud run. - **Scripts**: Any miscellaneous scripts that aid development and ops. For example, a `db_migrate.sh` to run migrations, a `seed_demo_data.js` to seed the database with sample data, or CI scripts (if not in a separate `.github/workflows` folder for GitHub Actions or similar). We may also include VSCode devcontainer config here if we want to support VSCode Remote Containers (so that opening the repo in VSCode sets up containers automatically).

Including Docker early on is a conscious choice to make onboarding and testing easier. By containerizing, we ensure that every contributor and environment is running the same stack, eliminating the "it works on my machine" issue [21]. It also demonstrates modern DevOps practices, which is a plus. For example, one tutorial uses Docker Compose to run a React frontend and Node backend together for a dev setup [22] [23] – we'll do the same for ClassPilot. This not only speeds up development but also lays the groundwork for deploying the app as containers in production.

Docs (Documentation)

The **docs/** directory holds project documentation, which is crucial for a large project (FAANG-level architects always emphasize thorough docs). We will maintain documents such as:

- **Architecture Overview:** Explaining system architecture, perhaps including diagrams of how ClassPilot integrates with external LMS APIs, data flow diagrams for how an assignment goes from Google Classroom into our database and then to the UI, etc. This can be an `architecture.md` or a set of Markdown files.

- **API Documentation:** A file (or generated docs) listing each API endpoint, its request/response format, authentication needed, etc. Could be `API.md` or we might integrate a tool (like Swagger/OpenAPI YAML) – if the latter, the YAML/JSON spec can live in docs or a dedicated folder.
- **Feature Specs:** Any detailed specifications or proposals for features (for instance, a spec on the Gamification system logic, or how GPA is calculated, etc. referencing the business logic).
- **Setup Guides:** Developer guide for how to run the project, how to use Docker dev environment, etc. This might also be covered in the main README, but docs can have more detailed troubleshooting, etc.
- **Pitch or Design Docs:** Since we have a pitch outline (the PDF you provided), such conceptual documents can be stored here for reference, ensuring everyone understands the vision and problem space.

Storing these in the repo ensures that as the code evolves, the documentation can easily be updated alongside it (and versioned). This is another benefit of a monorepo – even docs and code stay in sync in one place.

(The user-provided pitch outline already highlights why ClassPilot is needed and its unique value [1] [24]*; we would expand those ideas into technical requirements in our docs.)*

Root Files and Miscellaneous

At the root of the repository, we have the overarching **README.md**. This is the first entry point for anyone looking at the repo. It will give a high-level description of ClassPilot, instructions for setup (perhaps summarizing the Docker compose usage or how to run frontend/backend separately), and link to more detailed docs. We might also include in the root any configuration that spans the whole monorepo, such as:

- **Lint/Format configs** (e.g., an `.eslintrc` or `.prettierrc`) to enforce code style uniformly across frontend and backend.
- **.gitignore** file covering all sub-projects.
- CI config: If using GitHub Actions, a `.github/workflows/` folder might exist at root. If using another CI, their config files (like a CircleCI config.yml) would be here.
- Package management config: If using a tool like Yarn Workspaces or Lerna to manage dependencies in a monorepo, the configuration (like a root `package.json` listing workspaces, or a `lerna.json`) would live at root. This can simplify running install for all parts at once. (For now, this is optional – one can also manage each subproject separately, but workspaces can streamline things.)
- License file, contribution guidelines, etc., can also be present at root.

Naming Conventions and Organization

We have chosen **PascalCase** naming for project directories (e.g., `FrontendApp`, `BackendAPI`, `CommonLib`, etc.) to keep names readable and distinct. PascalCase (capitalizing each word) is commonly used for project names and classes in many corporate codebases, as it clearly delineates words. For example, `GoogleClassroomService` in the integrations is easier to read than a lowercase or underscored name. Consistent naming convention helps new contributors navigate the repo quickly. Inside the code, we will of course follow language-specific conventions (e.g., classes in PascalCase, variables in camelCase for JavaScript, etc.), but at the structural level PascalCase for folders makes the major pieces stand out.

Additionally, the project structure is organized by **feature/domain** rather than by technology alone. This means, for instance, under `services/` in the backend, we grouped code by feature (integrations, ai, gamification, etc.), and in the frontend we will likely group components by pages or features. This aligns with how large-scale projects remain maintainable – developers can work on one feature area in isolation and find all related code nearby, rather than chasing through overly generic layer folders. The monorepo approach with clear sub-folders enforces those boundaries while still allowing easy cross-usage of common code.

Development Workflow with Docker and Local Tools

To make development and testing easier, we incorporate Docker from the start. Using Docker ensures that every developer and environment is running the app with the same setup (same OS, dependencies, versions). As one article noted, setting up a consistent dev environment with Docker helps avoid the "it was working on my machine" scenario and OS-level configuration issues [20].

Here's how our dev workflow is structured with the provided tools:

- **Docker Compose Up**: With the provided `docker-compose.yml`, a developer can spin up the entire ClassPilot stack with a single command. This will:
- Build or pull the necessary images (frontend, backend, db, etc.).
- Start containers for each service and network them together. For instance, the frontend container can be reached on localhost:3000 and will proxy API calls to the backend container on localhost:5000 (set up via environment config).
- Mount volumes for code (in dev mode) so that live code changes on the host reflect inside the containers (facilitating hot-reload for the frontend and backend). This way, one can code in their editor and see changes without rebuilding the container each time during development.
- Initialize any required services (the database container might run an init script to create the schema, etc.).
- **Hot reloading and Debugging**: We can configure the containers for dev to use development mode (e.g., Node's `nodemon` for backend auto-restart, React's dev server for HMR). This provides a smooth edit-and-refresh cycle even though code is running in containers.
- **IDE Integration**: Developers can still use their favorite IDE/editor on the host, or even leverage VS Code dev containers. We could include a `.devcontainer` config for VS Code that uses our Docker compose, allowing one-click setup of an isolated dev environment.
- **Local Testing**: The Docker setup also allows running tests in the container environment to ensure consistency. We can have a service in compose for running tests, or simply exec into the running backend container to run `npm test`. This again ensures tests aren't failing due to environment differences.
- **Developer Tools**: We might include other local dev conveniences, such as:
- A script to easily reset the database (useful for testing with fresh data).
- Preconfigured accounts or demo data to simulate multiple LMS accounts connected (perhaps loaded via the seed script).
- Linters and formatters integrated with git hooks (to maintain code quality).

By investing in this robust local setup, we make it easier to onboard new developers and collaborate. A new contributor should be able to read the README, run one or two commands (e.g., clone repo, run `docker-compose up`), and have the whole system running exactly as it should. This reflects best practices seen in professional environments, where reproducibility and developer experience are prioritized [20]. It also sets the stage for deployment: the same Docker containers used in dev can be deployed to a cloud environment or CI pipeline with minimal changes.

Scalability and Future Considerations

The proposed structure is designed with future growth in mind. As ClassPilot evolves, this monorepo can **scale by adding new folders or services** without needing a new repository for each feature. Some forward-looking considerations:

- **Adding a Mobile App**: If later on an iOS/Android app is needed, we can simply add, for example, `MobileApp/` alongside FrontendApp. This could contain a React Native or Swift/Kotlin project. It could reuse the CommonLib for shared logic (or even share some code via web views or a common core in a mono-repo). The current shared libraries (types, constants) would benefit the mobile app as well. The structure supports multiple frontend targets (web, mobile, etc.) in one codebase [25].
- **Multiple Backend Services**: Right now, `BackendAPI` is one monolithic server. In the future, if certain parts of the system need to scale independently or be decoupled for performance, we could split them. For instance, a separate `WorkerService` could be introduced to handle background jobs (sending emails, heavy AI computations, etc.) as hinted before. This could be placed as another top-level folder (e.g., `WorkerService/`) or under Backend (depending on preference). Monorepo allows this flexibility – you can have several deployable services (API, worker, maybe a separate service for real-time notifications, etc.) all versioned together [15]. The Infrastructure folder can then be updated with additional Dockerfiles/compose services or k8s manifests for the new service.
- **Feature Flags and Modules**: As we add major features (like those under *Future Expansion* in the pitch – e.g., **auto-transcription of lectures, group study leaderboards, parent dashboards** [14]), the code for these can be cleanly added. For example, a new `transcription/` service module could be added under backend services to handle audio file processing (perhaps calling an external speech-to-text API). Or a new micro-front-end for a parent dashboard could be added under FrontendApp or as a separate app. Because our structure is by domain, adding new domains is straightforward.
- **Testing and Quality**: In future, we would expand the tests directory and possibly introduce testing at different levels (unit, integration, end-to-end). The structure will accommodate an `e2e/` tests folder possibly at root for end-to-end tests that run the whole system (some monorepos do this to test interactions between services). We have left hooks for testing in place.
- **Performance and Maintenance**: Having all code in one repo means we must also maintain discipline to keep things modular. We should use this structure to enforce boundaries (e.g., the frontend should only access backend through HTTP calls, not by reaching into backend code – obvious in a multi-language scenario, but if both are Node, we still treat them as separate). Using tools or conventions (like never import backend code into frontend, etc.) will maintain a clean separation. Monorepo tools can provide lint rules for this if needed.
- **Collaboration Workflow**: In a team setting, we might have multiple people working on different areas (one on UI, one on integration service, etc.). Git branching strategy and CI should be set up such that changes in one area run relevant tests. Monorepo allows running tests for all affected parts when something changes [18], which is great for catching issues early. We may configure our

CI to only rebuild/test services that are impacted by a given commit (tools like Nx or Turborepo excel at this, but even without them, a well-structured project can script it).

In summary, this project structure is **comprehensive and built for scale**. It imitates the kind of organization you'd see in large-scale apps like those by FAANG companies – clear separation of concerns, modular design, readiness for additional services, and an emphasis on developer experience (via documentation and Dockerized environment). We have one repository **ClassPilot** containing everything needed: from the UI that students see, to the backend logic crunching their data, to the integration hooks that pull info from external LMS platforms [1] – truly a "one hub, not three tabs" solution [24], reflected even in how we manage the code.

By following this structure, the ClassPilot team can confidently build a platform that is maintainable in the long run. As new features roll out or user base grows, the architecture will accommodate those changes with minimal friction. And importantly, any contributor can navigate the PascalCase directories to quickly find what they need (UI vs API vs common logic vs infra) and understand the system's anatomy at a glance.

Sources:

- Potvin, R. & Levenberg, J. *Why Google Stores Billions of Lines of Code in a Single Repository. Communications of the ACM* 59(7): 78-87, 2016. (Google's monorepo strategy) [3]
- Nx Dev Tools – *Monorepo Definition and Best Practices.* (Monorepo explained and benefits) [2] [4]
- Rahman, M. *Building a Scalable Nx Monorepo Structure for Full-Stack Applications.* Medium, 2024. (Example of structuring frontend, backend, and shared libs in a monorepo) [5] [15]
- *ClassPilot Pitch Outline.* (Problem, solution, and feature set for the ClassPilot project) [1] [10] [14]
- Nikhar, T. *Setup Development Environment with Docker for Monorepo.* DEV.to, 2021. (Using Docker Compose for a full-stack app dev environment) [20] [22]

---

[1] [7] [8] [9] [10] [11] [12] [13] [14] [17] [24] ClassPilot.pdf
file://file-1RvH9JRK53oHKwsTJaevAM

[2] [4] [6] Monorepo Explained
https://monorepo.tools/

[3] Why Google Stores Billions of Lines of Code in a Single Repository
https://research.google/pubs/why-google-stores-billions-of-lines-of-code-in-a-single-repository/

[5] [15] [18] [19] [25] Building a Scalable Nx Monorepo Structure for Full-Stack Applications | by Mahabubur Rahman | Medium
https://mahabub-r.medium.com/building-a-scalable-nx-monorepo-structure-for-full-stack-applications-a05ab856ac5d

[16] [20] [21] [22] [23] Setup Development Environment with Docker for Monorepo - DEV Community
https://dev.to/tejastn10/setup-development-environment-with-docker-for-monorepo-3433