ChatGPT

Plano de Implementação Técnica do Módulo Didático de ECG (Monorepo MEGA)

Visão Geral da Arquitetura Monorepo

Para iniciar o projeto **MEGA** (primeiro módulo de ECG) adotaremos uma abordagem **monorepo**, ou seja, um único repositório hospedando múltiplos projetos/serviços relacionados, com um claro delineamento de responsabilidades. Essa estrutura centralizada facilita o compartilhamento de código, consistência de ferramentas e versionamento unificado de frontend, backend e demais utilitários [1] [2]. No monorepo **MEGA**, teremos as seguintes pastas principais:

- `/frontend` – Aplicação frontend usando Next.js e TypeScript (conteúdo interativo, mídia e UI).
- `/backend` – Aplicação backend em Python usando FastAPI (API REST, lógica de negócio e integrações futuras).
- `/modulos/ecg` – Módulo de conteúdo didático de ECG (material estático: textos, imagens de traçados, dados de quizzes, etc., possivelmente consumido por frontend e backend).
- `/cli` – Scripts de automação e ferramentas de linha de comando (shell ou Python) para tarefas como build, deploy e geração de conteúdo.
- `/infra` – Configurações de infraestrutura e CI/CD (por ex. workflows do GitHub Actions, Dockerfiles, infraestrutura-as-código para deploy em cloud, etc.).

Essa organização modular torna o repositório **monorepo** porém **não monolítico**, permitindo desenvolver e implantar partes de forma independente quando necessário. Por exemplo, poderemos disparar pipelines separados caso haja mudanças apenas no frontend ou apenas no backend, graças a recursos de filtragem de caminhos do GitHub Actions [3]. A seguir, detalhamos a configuração de cada componente.

Frontend: Next.js + TypeScript com Conteúdo Interativo

No diretório `/frontend` será inicializada uma aplicação **Next.js** (última versão) em **TypeScript**. O Next.js fornece um framework React robusto, com suporte a geração de sites estáticos e dinâmicos, roteamento simplificado e otimizações de desempenho out-of-the-box. Com TypeScript garantimos maior segurança de tipos e manutenção facilitada do código. A interface será projetada para conteúdo **multimídia interativo** sobre ECG, incluindo:

- **Gráficos e Traçados ECG:** Utilizaremos componentes React customizados ou bibliotecas de gráficos (ex: D3.js, Chart.js ou até canvas/SVG puro) para renderizar ondas e sinais do ECG em tempo real. A ideia é permitir visualizações interativas dos complexos QRS, ondas P/T, etc., com possibilidade de animações e zoom em segmentos do traçado.
- **Simulações e Cases Clínicos:** Podemos incorporar simulações simples – por exemplo, um controle deslizante para variar frequência cardíaca e observar mudanças no traçado – e estudos de caso onde o usuário vê um ECG e tem que interpretar (nesses casos, o frontend pode exibir imagens ou gráficos provenientes do módulo de conteúdo).

- **Vídeos Educativos:** Integração de vídeos explicativos curtos diretamente nas páginas (via componentes de vídeo HTML5 ou embeds do YouTube/Vimeo), ilustrando conceitos como arranjos de derivações ou mecanismos de arritmias.
- **Quizzes Interativos:** Implementaremos componentes de quiz (perguntas de múltipla escolha, verdadeiro/falso, etc.) para reforço do aprendizado. Os quizzes podem ser definidos em arquivos de conteúdo (por exemplo, um JSON/markdown em `/modulos/ecg`) e o frontend os carrega, exibindo perguntas e avaliando respostas em tempo real. Para respostas, inicialmente a validação pode ser feita no próprio front (para quizzes simples) ou delegada ao backend (especialmente se quisermos gerar explicações ou registrar resultados).

Em termos de estrutura Next.js, usaremos a organização de páginas e rotas para modularizar o conteúdo do ECG. Por exemplo, poderemos ter rotas como `/ecg/fundamentos`, `/ecg/ritmos`, `/ecg/quizzes` etc., correspondentes a seções do módulo. Cada página Next pode consumir dados estáticos do módulo ECG (via import de arquivos markdown/JSON) ou dados dinâmicos via chamadas à API FastAPI (discutido adiante).

Desenvolvimento e Build: Durante o desenvolvimento local, usaremos `next dev` para hot-reload da interface. Para produção, configuraremos scripts de build (`npm run build`) e export estático (`next export`) para gerar uma versão estática do site. A opção de **static export** do Next.js nos permite gerar um site puramente estático (HTML, CSS, JS) a partir das páginas, adequado para hospedagem no GitHub Pages [4]. Como estaremos hospedando no próprio GitHub Pages, definiremos no `next.config.js` as propriedades `output: 'export'` e possivelmente `basePath` (caso o site esteja em um sub-path, e.g., `username.github.io/MEGA`) para garantir que os recursos estáticos carreguem corretamente [4]. Adicionalmente, criaremos no processo de build um arquivo `.nojekyll` na pasta de saída para impedir o Jekyll (engine do GitHub Pages) de interferir na entrega dos arquivos do Next (necessário pois o Next usa pastas iniciando com `_` como `_next`) [5].

Recursos adicionais: Focaremos em uma UI responsiva e moderna. Poderemos adotar um framework de CSS-in-JS ou utilitário como **Tailwind CSS** para agilizar o design responsivo e manter consistência visual. Componentes de interface repetíveis (botões, cards, navbar) podem ser padronizados – possivelmente aproveitando kits como **ShadCN UI** ou similares (como visto em projetos semelhantes [6]). Também integraremos ferramentas de qualidade no frontend: **ESLint** e **Prettier** serão configurados para padronização de código e limpeza, com checagens rodando localmente (via `npm run lint`) e no CI.

Backend: FastAPI em Python para API e Simulações

O backend reside em `/backend`, estruturado como um projeto Python usando **FastAPI** – um framework web moderno, de alta performance e voltado a APIs REST, que provê validação de dados via Pydantic e documentação automática (OpenAPI) integrada [7]. Essa escolha nos dá rapidez de desenvolvimento e a possibilidade de facilmente criar endpoints e posteriormente integrar lógica de IA ou computação científica (Python tem forte ecossistema para isso).

Estrutura do Código: Dentro de `/backend`, criaremos um pacote Python (por exemplo, um módulo `app/` contendo `main.py`, `routers/`, `models/`, etc.). O arquivo `main.py` irá instanciar a aplicação FastAPI e incluir rotas definidas em submódulos. Desde o início, configuraremos ao menos:

- **Rotas de Teste/Status:** Um endpoint GET em `/api/ping` ou `/api/health` que retorna algo como `{"status": "ok"}` para verificar rapidamente se o backend está no ar.

- **Rotas do Módulo ECG:** Um roteador específico, e.g. `/api/ecg/...`, onde implementaremos funcionalidades ligadas ao módulo. Inicialmente podemos ter um GET `/api/ecg/conteudo` que retorna o conteúdo estático do módulo (ex.: texto de teoria ou lista de perguntas de quiz) carregando dados de arquivos em `/modulos/ecg`. Assim, o frontend pode optar por obter o conteúdo via API ao invés de embutir tudo no bundle, facilitando atualizações de conteúdo sem refazer o deploy do front. Outra rota potencial é `/api/ecg/avaliar`, para receber respostas de quiz e retornar pontuação ou feedback (esta pode ser expandida com IA no futuro).
- **Preparação para Simulações/IA:** Deixaremos ganchos (endpoints ou serviços internos) para integrar inteligência artificial e simulações clínicas. Por exemplo, poderemos ter um endpoint `/api/ecg/simular` que no futuro chamará uma função de simulação de eletrocardiograma (dado certos parâmetros, retornar um traçado ou parâmetros calculados). No presente, esse endpoint pode retornar dados fictícios ou uma mensagem "simulação não implementada" – mas a presença dele e sua estrutura já adianta a integração futura com módulos de IA ou algoritmos especializados.

Integração com IA (visão backend): Pensando adiante, definiremos desde já como o backend poderia chamar serviços de IA. Poderemos incluir um módulo interno, ex `services/ia.py`, com funções placeholder que eventualmente usarão APIs do OpenAI (GPT-5) ou Google (Gemini) para gerar conteúdo. Por exemplo, uma função `gerar_questoes(topico: str)` que (quando conectada às credenciais e SDK apropriados) envie um prompt a um modelo de linguagem pedindo N perguntas sobre *topico* e retorne sugestões. No FastAPI, um endpoint `/api/ecg/gerar-perguntas` poderia usar essa função. Assim, o esqueleto para **geração de conteúdo assistida por IA** já fica disponível – mesmo que no início retorne respostas estáticas ou de exemplo.

CORS e Segurança: Configuraremos o middleware de CORS no FastAPI para permitir que o frontend (possivelmente servido em domínio do GitHub Pages) consiga fazer requisições AJAX à API. Colocaremos as origens permitidas (durante dev, `http://localhost:3000`; em produção, o domínio do Pages). Também preveremos um esquema básico de autenticação para futuras funcionalidades (por exemplo, se depois houver contas de usuário, progresso, etc., mas isso pode não ser usado no módulo inicial).

Documentação Automática: Uma grande vantagem do FastAPI é gerar automaticamente uma UI de docs (Swagger/OpenAPI) em `/docs` com todos os endpoints disponíveis [7]. Isso será útil para testarmos manualmente a API e também para outros desenvolvedores entenderem e validarem as chamadas. Manteremos os modelos de request/response bem definidos usando **Pydantic** (por ex., criar classes Pydantic para representar uma resposta de quiz, etc.), de forma que a documentação seja clara.

Execução e Deploy Backend: Para desenvolvimento local, incluiremos no README instruções para rodar o backend com `uvicorn`. Também poderemos criar um script no Makefile/CLI (`make run-backend`) que execute `uvicorn app.main:app --reload` na porta 8000. Em termos de deploy, como etapa futura, planeja-se hospedar o backend em algum serviço de nuvem (AWS, Heroku, Fly.io, etc.) ou containerizar e rodar em plataforma Kubernetes. Faremos desde já um **Dockerfile** para o backend (no `/backend/Dockerfile`), de modo que seja fácil construir uma imagem Docker do serviço. Isso nos permitirá tanto rodar local via Docker Compose (especialmente se houver banco de dados no futuro) quanto publicar essa imagem em um registry para deploy. No cenário atual (MVP do módulo ECG), o backend pode não ser imediatamente publicado, mas sim rodado local para servir o front (que estará publicado estaticamente). Porém, deixamos tudo pronto para no futuro integrar o backend em produção sem grandes refatorações.

Automação de Build, Deploy e Dev (CLI e GitHub Actions)

Para manter o fluxo de trabalho ágil, implementaremos automações tanto locais (scripts CLI) quanto em pipeline CI/CD (GitHub Actions):

Scripts CLI (Makefile / Shell / Python): No diretório `/cli` criaremos utilitários para facilitar tarefas comuns:

- *Makefile no raiz:* Um Makefile com comandos abreviados. Exemplos: `make dev` para rodar frontend e backend em paralelo (poderemos usar dois shells ou uma ferramenta como `concurrently` via npm para um único comando disparar ambos). `make build` para construir frontend (gera `/frontend/out`) e backend (imagem Docker ou distribuição). `make lint` para rodar linters em ambos subprojetos. `make test` no futuro para executar testes automatizados.
- *Scripts de deploy:* Um script shell como `deploy_frontend.sh` que execute o build do frontend e em seguida use Git para publicar o conteúdo estático no branch `gh-pages`. Similarmente, `deploy_backend.sh` que possa construir a imagem Docker e subi-la a um registry ou fazer deploy em ambiente cloud (poderá chamar CLI de nuvem ou kubectl, dependendo do alvo escolhido).
- *Ferramentas de setup:* Um script Python ou shell para configurar ambiente local (por exemplo, criar arquivo `.env` a partir de `.env.example`, checar dependências instaladas, etc., para onboard de novos devs).
- *Geração de conteúdo:* (Opcional, futuramente) scripts que usem IA para auxiliar na criação de material. Por exemplo, poderemos ter `gerar_conteudo_ecg.py` que consome funções do backend ou chamadas diretas à API do OpenAI/Gemini para produzir rascunhos de textos ou perguntas de ECG automaticamente. Esse tipo de script reforça a ideia de **conteúdo assistido por IA** no fluxo de criação, não apenas em tempo de execução.

Integração Contínua (CI) com GitHub Actions: Configuraremos múltiplos workflows do GitHub Actions no diretório `.github/workflows` (mantido dentro de `/infra` se preferirmos lógica agrupada):

- *CI de Build/Test:* A cada push ou PR para a branch principal, dispararemos um workflow que faz checkout do repo e roda: 1) **Instalações** – setup do Node (npm ou pnpm) e Python (poetry/pip). 2) **Linters e Testes** – executa `npm run lint` no front, `flake8` ou similar no back, e rodará testes se houver (ex: `npm test` e `pytest`). 3) **Build** – executa o build do front e talvez constrói a imagem do back (usando docker build). Isso garante que o código em cada mudança está compilando e passando em checagens básicas. Podemos aproveitar o suporte a monorepo do Actions para otimizar: usando o filtro de paths no gatilho, evitamos rebuildar o front se só arquivos do backend mudaram e vice-versa [3].
- *Deploy Contínuo (CD):* Uma ação para automatizar o deploy do frontend no GitHub Pages. Ex: ao dar push de tag de release ou push na branch `main`, rodamos o build do frontend (`next build && next export`) e então usamos uma ação como **JamesIves/github-pages-deploy-action** ou o próprio **Github Pages Action** para publicar a pasta `out` resultante no branch `gh-pages` [8]. Já incluiremos no fluxo a criação do arquivo `.nojekyll` e configuração do `basePath` se necessário. Depois de configurado, o site estará disponível via Pages (endereço do GitHub Pages do repo ou custom domain futuramente).
- *Deploy Backend:* Embora o backend possivelmente não seja implantado imediatamente, deixaremos esboçado um workflow para ele. Por exemplo, usar GitHub Actions para buildar e publicar a imagem Docker do FastAPI no GitHub Container Registry, ou até mesmo implantar em

um serviço (caso definido, poderíamos integrar com Heroku deploy ou AWS). Isso ficaria talvez manual inicialmente (executado quando mantainer quiser). Em todo caso, o pipeline de backend vai assegurar que o Dockerfile builda sem erros e possamos rodar `uvicorn` no container.
- *Badges & Status:* Configuraremos o repo para exibir badges de CI passando no README, para rápida visualização.

Segurança e Segredos: No GitHub Actions, utilizaremos Secrets para quaisquer credenciais (ex: se integrar OpenAI API key ou credencial do deploy na nuvem). Também aplicaremos princípios de menor privilégio nos tokens. Como dica futura, podemos adotar as novas Actions de AI para auxílios – por ex., **Gemini CLI GitHub Action** – falaremos disso a seguir.

Integração com IA e Desenvolvimento Assistido (Gemini CLI e GPT-5)

Um diferencial desse projeto será preparar o terreno para ampla utilização de **Inteligência Artificial** tanto no processo de desenvolvimento quanto na geração de conteúdo educacional. Os pilares serão o **Gemini CLI** (ferramenta da Google para agentes de IA em desenvolvimento) e o **GitHub Copilot (GPT-5)** quando disponível.

Gemini CLI no Workflow de Dev: Planejamos incorporar o Gemini CLI de duas formas: localmente e via GitHub Actions. Localmente, os desenvolvedores podem usar o CLI para gerar trechos de código, explicar partes complexas ou até produzir esboços de documentação. Para isso, adicionaremos no repositório um arquivo de contexto (por exemplo, `GEMINI.md`) contendo detalhes do projeto e guidelines para o agente – facilitando que o Gemini CLI entenda a estrutura do nosso monorepo e nossas convenções. Além disso, poderemos criar um arquivo de configuração em `.gemini/settings.json` no projeto, caso seja necessário customizar algo (por exemplo, restringir comandos que o agente pode rodar, ou definir nosso arquivo de contexto preferencial) [9] [10].

Mais empolgante é usar o **Gemini CLI via GitHub Actions** para automação inteligente: existe uma ação oficial (beta) que permite ao Gemini atuar em issues e pull requests. Vamos configurar essa integração para obter **reviews automatizados de código** nas pull requests. Assim que um PR for aberto, o agente Gemini pode ser acionado para analisar as mudanças e comentar com feedback de qualidade, estilo e possíveis bugs [11]. Isso fornecerá *insights* instantâneos sobre cada alteração, liberando os revisores humanos para focar em aspectos mais complexos do design em vez de problemas triviais [12]. Essa revisão por IA checará aderência aos padrões, pontos de atenção em segurança e sugerirá melhorias, funcionando como um "colega de equipe AI" dentro do repositório. Além disso, habilitaremos a funcionalidade de **colaboração sob demanda**: desenvolvedores poderão mencionar o bot (ex: comentar "@gemini-cli, escreva testes para esta função") em uma issue/PR e o Gemini CLI tentará atender à solicitação (gerando código de testes, por exemplo) [13]. Essas capacidades tornarão o fluxo de desenvolvimento mais ágil e com garantia extra de qualidade.

GPT-5 via Copilot: O GitHub Copilot, ao evoluir para versões GPT-4/5, será outra peça fundamental. Certificaremos que o repositório está **Copilot-friendly**: comentários claros, README bem escrito e talvez um arquivo `CONTRIBUTING.md` com notas de arquitetura, para que o modelo de IA tenha contexto ao sugerir código. Com GPT-5 integrado, esperamos ainda mais capacidade de geração de código complexo e de **suporte à escrita de documentação e conteúdo**. Um uso almejado é na geração assistida de conteúdo didático: por exemplo, usar o Copilot (ou a OpenAI API diretamente) para **gerar explicações ou questões** a partir de descrições curtas. Podemos criar templates onde um desenvolvedor/especialista escreve os tópicos-chave, e o Copilot ajuda a expandir em parágrafos ou

questões de quiz. No CLI, isso pode ser semiautomatizado: um comando que envia um prompt "Gerar 5 perguntas de ECG sobre arritmia X" e recebe sugestões a serem refinadas pelo time humano.

Em suma, a integração de IA será incremental: desde o início já deixamos hooks e configurações para **Gemini CLI** e **Copilot GPT-5**, e conforme essas ferramentas forem sendo usadas, podemos ampliar seu papel. O resultado buscado é acelerar o desenvolvimento (com autocompletes e geração de boilerplate), garantir qualidade (CI com reviews automáticos) e enriquecer o conteúdo educacional (gerando materiais base que os médicos/educadores possam revisar ao invés de partir do zero).

Documentação, Configurações e Deploy Inicial

Por fim, complementaremos a implementação com itens essenciais de documentação e configuração de ambiente:

- **README.md:** Um README inicial será incluído na raiz do repositório detalhando o propósito do projeto, instruções de setup e uso para desenvolvedores e visão geral da estrutura. Ele explicará como rodar o frontend e backend localmente (incluindo dependências necessárias: Node 18+, Python 3.11+, etc.), como executar os scripts disponíveis, e como contribuir. Também listaremos as pastas do monorepo e sua função (resumindo parte do que foi descrito acima), facilitando para qualquer novo colaborador entender a organização. Instruções rápidas de deploy e links para o site (uma vez no ar via GitHub Pages) serão adicionadas. Além disso, colocaremos shields (badges) do CI, versão, licença, etc., no README para transparência e status imediato.
- **Arquivos de Config de Exemplo:** Forneceremos um `.env.example` tanto na raiz quanto possivelmente específicos para frontend e backend. Este arquivo conterá todas as variáveis de ambiente usadas no projeto. Por exemplo: `NEXT_PUBLIC_API_URL=http://localhost:8000` (URL base do backend durante desenvolvimento), chaves de API como `OPENAI_API_KEY=` (vazia, apenas indicando a necessidade), variáveis de configuração do FastAPI (como `DEBUG=` ou credenciais de banco se houver). Assim, o desenvolvedor copia esse `.env.example` para `.env` e preenche as variáveis reais localmente. Para o backend, podemos usar Python-decouple ou python-dotenv para carregar essas variáveis no app, enquanto no frontend usaremos o prefixo `NEXT_PUBLIC_` conforme exigido pelo Next.js para expor variáveis no bundle do cliente.
- **Dependências e Scripts:** No `package.json` do frontend, adicionaremos *scripts* úteis (além dos padrões `dev`, `build`, `lint`, `export`, possivelmente um `analyze` para analisar bundle, etc.). No backend, um arquivo `requirements.txt` (ou `pyproject.toml` se usarmos Poetry) listará as dependências (FastAPI, Uvicorn, Pydantic, etc., e possivelmente `python-dotenv`, `requests` para chamadas externas, etc.). Um detalhe importante é fixar versões mínimas para evitar que updates de libs quebrem o módulo didático no futuro.
- **Setup do GitHub Pages:** Já mencionado anteriormente, mas reforçando: configuraremos o repositório para GitHub Pages. Isso envolve habilitar o Pages nas configurações do repo apontando para a branch `gh-pages` (ou usar o novo método de publicar via artifact do Actions). O DNS padrão será algo como `drmcoelho.github.io/MEGA` caso usado o Pages do usuário, ou `drmcoelho.github.io/MEGA/` se via project page – de qualquer modo, ajustaremos o Next.js para esse base path. Após o primeiro deploy automático pelo Action, verificaremos se todas as rotas estão funcionando estáticas. O site frontend poderá consumir o backend local durante dev e, em produção, apontaremos o `NEXT_PUBLIC_API_URL` para a URL em nuvem quando o backend estiver implantado (por agora, se backend não estiver online, algumas features dependentes disso ficarão inativas no site publicado, mas podemos colocar mensagens informativas "Backend indisponível para esta função").

- **Qualidade de Código:** Implementaremos configurações de lint/format para todo o monorepo: `.eslintrc.json` e `.prettierrc` no raiz para JS/TS; possivelmente `pyproject.toml` config para `flake8` e `black` no Python. Assim, garantimos um estilo consistente. Poderemos também ativar *pre-commit hooks* usando Husky (para front) e pre-commit (para Python) integrando formatação antes de cada commit.
- **Licença e Coautoria:** Incluiremos arquivo `LICENSE` adequado (caso open-source) e talvez um `CODE_OF_CONDUCT.md` e `CONTRIBUTING.md` dado que o projeto pode envolver colaboração aberta. Esses documentos não impactam diretamente a técnica, mas completam a preparação do repositório.

Com tudo isso implementado, teremos o esqueleto técnico pronto do projeto **MEGA** – módulo ECG. O **frontend Next.js** servirá páginas interativas e estéticas, o **backend FastAPI** fornecerá APIs e futura lógica de IA, ambos integrados em um monorepo coeso. As automações via **CLI e CI/CD** garantirão ciclos de desenvolvimento rápidos e implantação facilitada, enquanto a integração de **ferramentas de IA (Gemini CLI, Copilot)** posiciona o projeto na vanguarda, aproveitando revisão de código automatizada e geração de conteúdo inteligente para maximizar qualidade e eficiência [12]. Em resumo, este primeiro passo estabelece uma base tecnológica sólida sobre a qual poderemos construir rapidamente as funcionalidades educacionais de ECG e escalar para novos módulos no futuro.

Fontes Utilizadas:

- Organização monorepo com Next.js (TypeScript) + FastAPI [2] [7]
- Filtro por paths no GitHub Actions para monorepos [3]
- Configuração do Next.js estático no GitHub Pages (.nojekyll, output export) [5]
- Integração de Gemini CLI para reviews de PR e automação com IA [12]

---

[1] Monorepo Explained
https://monorepo.tools/

[2] [6] GitHub - cording12/next-fast-turbo: A Turborepo featuring a Next.js frontend, FastAPI backend and a fully built and annotated Mintlify documentation site.
https://github.com/cording12/next-fast-turbo

[3] continuous integration - Deploy individual services from a monorepo using github actions - Stack Overflow
https://stackoverflow.com/questions/58136102/deploy-individual-services-from-a-monorepo-using-github-actions

[4] [5] [8] Next.js Deploy as a Static Site using Github Pages - DEV Community
https://dev.to/lico/nextjs-deploy-as-static-site-using-github-pages-3bhm

[7] Rapid Development with Next.js + FastAPI + Vercel + Neon Postgres
https://www.wolk.work/blog/posts/rapid-development-with-next-js-fastapi-vercel-neon-postgres

[9] [10] GEMINI CLI FLAGS
https://www.notion.so/24116e4c3d2980efbb7cf6ec15ff9ff1

[11] [12] [13] Gemini CLI GitHub Actions: AI coding made for collaboration
https://blog.google/technology/developers/introducing-gemini-cli-github-actions/