**Manual de Implantação e Arquitetura da Plataforma IAprovas**

**Versão 1.2 - 20 de Junho de 2025**

Este documento descreve o passo a passo completo para configurar, desenvolver e implantar a plataforma IAprovas a partir do zero, incluindo uma descrição da função de cada arquivo no projeto.

**Parte 1: Configuração do Ambiente de Desenvolvimento Local**

Estas são as ações que você executa no seu computador para poder trabalhar no projeto.

**1.1. Procedimento de Início de Dia:** Sempre que for começar a trabalhar, abra o VS Code e o terminal integrado e siga estes passos:

1. Navegue até a pasta do projeto:

Bash

cd caminho/para/seu/projeto/plataforma\_concursos-b2fd3629646d474b81af188342f1f56f9c40bd79

1. Ative o ambiente virtual do Python:

Bash

source venv/bin/activate

1. Inicie o servidor de backend:

Bash

python3 app.py

1. Inicie o servidor do frontend: No VS Code, clique com o botão direito no arquivo index.html e selecione "Open with Live Server".

**1.2. Procedimento de Fim de Dia (Salvar no GitHub):** Ao final de cada sessão de trabalho, para salvar seu progresso de forma segura:

1. No terminal, pare o servidor app.py com Ctrl + C.
2. Verifique as alterações: git status
3. Adicione todas as alterações: git add .
4. Crie um "pacote" com uma mensagem descritiva (Commit):

Bash

git commit -m "fix: Descreve a correção ou feat: para nova funcionalidade"

1. Envie para o GitHub: git push
2. Desative o ambiente virtual: deactivate
3. Feche o VS Code.

**Parte 2: Integração de Serviços Externos**

Configuramos quatro serviços principais: Firebase, Stripe, OpenAI e Resend.

* **2.1. Firebase (Banco de Dados e Autenticação):**
  + **Configuração no Código:** As credenciais de frontend do Firebase (apiKey, authDomain, etc.) estão no arquivo firebase-config.js. A segurança é garantida pelas Regras de Segurança configuradas no painel do Firestore.
  + **Conta de Serviço (Backend):** Para que nosso app.py pudesse acessar o Firestore com permissões de administrador, geramos uma chave de serviço em *Configurações do projeto > Contas de serviço*. O arquivo JSON baixado foi renomeado para serviceAccountKey.json e colocado no ambiente de produção através dos "Secret Files" da Render.
* **2.2. Stripe (Pagamentos e Assinaturas):**
  + **Estrutura de Produtos e Preços:** Utilizamos um único Produto na Stripe chamado "IAprovas Assinatura". Associados a este produto, foram criados múltiplos Preços para cada plano oferecido (Básico, Intermediário, Premium Mensal, Premium Anual).
  + **Portal do Cliente (Customer Portal):** A funcionalidade "Gerenciar Assinatura" é implementada através de uma rota /create-portal-session no app.py. Essa rota utiliza o stripeCustomerId (salvo no Firestore durante o checkout) para gerar um link de acesso seguro ao portal do cliente da Stripe.
  + **Webhooks:** O webhook checkout.session.completed está configurado no painel da Stripe para apontar para https://iaprovas-backend.onrender.com/stripe-webhook. Ele é responsável por atualizar o plano do usuário e salvar o stripeCustomerId no Firestore.
* **2.3. OpenAI (Inteligência Artificial):**
  + **Controle de Custos:** Um limite de uso mensal (budget) foi configurado em *Settings > Limits* no painel da OpenAI para controlar os gastos.
  + **Otimização de Modelo:** Para a geração de cronogramas, utilizamos o modelo gpt-3.5-turbo para otimizar a velocidade e o custo, mantendo a qualidade da resposta para tarefas estruturadas.
* **2.4. Resend (E-mails Transacionais):**
  + **Configuração:** Optou-se pelo Resend devido à sua API moderna e plano gratuito. A chave de API foi configurada como variável de ambiente no backend.
  + **Domínio Verificado:** O domínio iaprovas.com.br foi verificado no Resend para garantir a máxima entregabilidade dos e-mails.

**Parte 3: Deploy na Render (Produção)**

Colocamos a aplicação no ar usando a plataforma Render.

* **3.1. Arquivos de Configuração para Deploy:**
  + requirements.txt: Lista todas as bibliotecas Python que o backend precisa.
  + build.sh: Script executado pelo Render para instalar as dependências (pip install -r requirements.txt).
  + Procfile: Define o comando para iniciar o servidor: web: gunicorn app:app. Este arquivo foi ajustado para remover o timeout, otimizando o processo de geração no backend.
* **3.2. Deploy do Backend (Web Service):**
  + Criamos um "Web Service" na Render conectado ao nosso GitHub.
  + **Configurações:** Runtime: Python 3, Build Command: bash build.sh, Start Command: gunicorn app:app.
  + **Variáveis de Ambiente:** Configuramos um "Environment Group" na Render com as chaves: OPENAI\_API\_KEY, STRIPE\_SECRET\_KEY (produção), STRIPE\_WEBHOOK\_SECRET (produção) e o conteúdo do serviceAccountKey.json como um "Secret File".
* **3.3. Deploy do Frontend (Static Site):**
  + Criamos um "Static Site" na Render conectado ao GitHub.
  + **Configurações:** Publish Directory: . (ponto), para publicar todos os arquivos HTML/CSS/JS.

**Parte 4: Arquitetura do Projeto (O que cada arquivo faz)**

**4.1. Arquivos HTML (A Estrutura):**

* cronograma.html: Página para geração e visualização de cronogramas. O formulário foi atualizado para incluir uma lista de matérias comuns (checkboxes) e um campo de "tags" para adicionar outras matérias, melhorando a experiência do usuário.

**4.2. Arquivos de Estilo CSS (A Aparência):**

* dashboard-lovable.css: Adicionados novos estilos para a grade de checkboxes de matérias, melhorando a organização visual do novo formulário.

**4.3. Arquivos JavaScript (A Interatividade):**

* main-app.js: Orquestrador principal do dashboard. Gerencia o estado de autenticação e dispara o evento userDataReady para sincronizar a inicialização de outras páginas.
* api.js: Centraliza todas as chamadas fetch para o backend. A API\_BASE\_URL é definida dinamicamente para funcionar em ambiente local e de produção.
* **cronograma-page.js (Atualizado):** Gerencia a página de cronogramas com uma arquitetura assíncrona.
  + **Geração Assíncrona:** Ao submeter o formulário, o script chama uma rota que inicia um **trabalho assíncrono** no backend.
  + **Listener em Tempo Real:** A página utiliza um listener em tempo real do Firestore (onSnapshot) para "ouvir" as atualizações do cronograma que está sendo gerado no servidor.
  + **Feedback de Status:** A interface exibe o status "Gerando..." e atualiza automaticamente para "Abrir" ou "Falhou" quando a tarefa no servidor é concluída, sem precisar recarregar a página.
  + **Renderização:** A função exibirPlanoNaTela foi corrigida para ler a estrutura de dados correta (dias\_de\_estudo) enviada pelo backend, resolvendo o problema do cronograma em branco.

**4.4. Backend e Arquivos de Servidor (app.py):**

* **app.py (Atualizado):** O motor da plataforma, com a rota de geração de cronograma totalmente reformulada.
  + **Geração de Cronograma Assíncrona:** A rota /gerar-plano-estudos agora é assíncrona. Ela recebe a solicitação, pré-processa os dados (ex: corrigindo "Terca" para "Terça-feira"), cria um documento "placeholder" no Firestore com status: 'processing', inicia a chamada para a OpenAI em um threading.Thread (tarefa de segundo plano) e retorna uma resposta imediata 202 Accepted.
  + **Tratamento de Erros:** A tarefa em segundo plano possui um bloco try...except que, em caso de falha, atualiza o status do plano no Firestore para failed, fornecendo feedback claro ao usuário.
  + **Prompt Inteligente:** O prompt enviado à OpenAI foi significativamente refinado para forçar a IA a usar todos os dias e horas disponíveis, aplicar uma variedade de métodos de estudo de forma pedagógica (Teoria -> Exercícios -> Revisão) e respeitar todas as matérias informadas pelo usuário, garantindo planos de alta qualidade.