ChatGPT

Plano de Desenvolvimento do MVP – Assistente Fiscal SaaS para MEIs

Visão Geral e Objetivo do Projeto

Este projeto propõe um **MVP (Produto Mínimo Viável)** de um sistema SaaS para auxiliar **Microempreendedores Individuais (MEIs)** na declaração de impostos. O sistema contará com um **agente de IA (LLM)** integrado que responderá perguntas fiscais e utilizará os **documentos do usuário** para fornecer respostas personalizadas. O objetivo principal é entregar, em **1 mês**, uma aplicação funcional em ambiente de produção (local, via Docker) com foco apenas nas funcionalidades essenciais descritas, incluindo:

- Autenticação de usuários via JWT;
- Dashboard com resumo do faturamento do MEI (mensal/anual) comparado ao limite permitido;
- Upload de documentos (notas fiscais, recibos etc.) com processamento OCR assíncrono;
- Chat conversacional com um assistente LLM para tirar dúvidas fiscais, exibindo e permitindo edição dos dados extraídos dos documentos;
- Mecanismo de faturamento por uso (contagem de tokens de IA), sem limites restritivos no ambiente de testes.

A seguir detalhamos a arquitetura técnica proposta, o backlog de funcionalidades priorizado e um plano de execução semana a semana.

Arquitetura Técnica Proposta

A arquitetura adota uma abordagem de **microsserviços** containerizados, com comunicação via APIs REST e orquestração pelo Docker Compose. Abaixo descrevemos os componentes e tecnologias principais:

Frontend e Gateway

- **Frontend:** Desenvolvido em **React** (JavaScript/TypeScript) com **Tailwind CSS** para agilizar o design responsivo. Será uma aplicação de página única (SPA) servindo a interface web para os MEIs. Incluirá páginas para login, dashboard (painel) e chat com o assistente.
- **API Gateway (NGINX):** Na frente dos microsserviços haverá um container NGINX configurado como **reverse proxy**. Ele roteará as requisições do frontend para os serviços correspondentes (p. ex., `/api/auth` para serviço de Auth, `/api/docs` para Documentos, etc.), além de servir os arquivos estáticos do frontend em produção. O NGINX também atuará como ponto central de entrada, podendo futuramente implementar balanceamento de carga ou verificação de autenticação. Nesta fase MVP, a validação JWT pode ser feita diretamente nos microsserviços de backend, mas o NGINX garante uma interface unificada.

Microsserviços de Backend (Python/FastAPI)

Cada funcionalidade nuclear do backend será isolada em um microsserviço **FastAPI** (Python 3.x LTS) distinto, favorecendo a modularidade e escalabilidade. Os serviços se comunicarão via chamadas HTTP

REST internas (e eventualmente via filas para processamento assíncrono). Todos serão containerizados (imagens base Python 3.11 LTS) e orquestrados no Docker Compose. Os principais serviços são:

- **Auth Service:** Responsável por registro de usuários, login e emissão de tokens JWT para autenticação. Utilizará OAuth2 Password + Bearer JWT conforme as práticas do FastAPI [1], armazenando credenciais (e.g. e-mail, hash de senha) em um banco **PostgreSQL**. O JWT assinado garante sessões stateless e seguras para acesso aos outros serviços. *(Ex.: Ao logar, o usuário recebe um token JWT válido por X horas, incluído nos headers para chamadas subsequentes.)*

- **Documentos Service:** Gerencia upload e processamento de documentos fiscais do usuário. Suas funções incluem:

- Endpoint para **upload** de arquivos (PDFs, imagens): ao receber um arquivo autenticado, o serviço o salva em um armazenamento de objetos **MinIO** (compatível com S3) e cria um registro no banco de dados (MongoDB) indicando status "processando".
- Integração com **OCR**: O serviço envia uma tarefa para um **worker Celery** (ver abaixo) para extrair texto do documento via **Tesseract OCR**. Esse processamento é pesado, então é delegado a um worker assíncrono para não bloquear o fluxo principal [2]. Quando o OCR termina, o texto extraído e eventuais dados estruturados (por ex. valores de notas) são salvos no MongoDB.
- **Armazenamento de dados estruturados:** Os textos completos OCR (ou campos como valor, data, CNPJ do emitente, etc., se extraídos) serão armazenados em um banco **MongoDB** (banco NoSQL, ideal para guardar JSONs dos documentos). Cada documento pode ter uma coleção com seu texto e metadados (usuário, data upload, status, etc.).

- **Geração de embeddings:** Após obter o texto via OCR, o serviço de documentos ou o serviço de agente (conforme implementação) gerará **vetores de embeddings** para o conteúdo do documento, usando o modelo de embedding da API LLM (Gemini) ou outra biblioteca de IA. Esses vetores serão armazenados em uma tabela no PostgreSQL com extensão **pgvector** para viabilizar buscas semânticas (ver detalhes adiante).

- **Agent Service:** Fornece a inteligência conversacional. Ele expõe endpoints de **chat** onde o frontend envia perguntas do usuário. Ao receber uma pergunta, este serviço implementa o fluxo **RAG (Retrieval Augmented Generation)**:

- **Recuperação de contexto:** O agente consulta o banco vetorial (Postgres/pgvector) usando o embedding da pergunta para encontrar os documentos ou trechos mais relevantes do usuário relacionados à dúvida. A extensão pgvector permite realizar busca por similaridade de forma eficiente diretamente no Postgres [3].
- **Composição do prompt:** Com base nos documentos recuperados, o serviço monta um prompt contendo: (a) contexto relevante extraído (por ex., "Você tem R$50.000 faturados até agora, limite anual é R$81.000 [4] ..." ou trechos de notas fiscais pertinentes), (b) possivelmente informações gerais sobre regras do MEI (podemos incluir um conhecimento básico pré-definido sobre obrigações do MEI), e (c) a pergunta do usuário. Essa técnica RAG de fornecer contexto adicional ao modelo **"ensina" o LLM sobre informações específicas do usuário**, aumentando a precisão e reduzindo alucinações [5].
- **Chamada ao LLM (Gemini):** O prompt composto é enviado via requisição HTTP para a API do **Google Gemini** (modelo de linguagem de última geração do Google). Utilizamos a versão apropriada (por ex. *Gemini 2.5* "Flash" ou similar) via endpoint REST. O Gemini retorna a resposta em linguagem natural.

- **Resposta ao usuário:** O Agent Service encaminha a resposta gerada de volta ao frontend. Opcionalmente, inclui referências aos dados usados (ex.: "Conforme sua nota de Jan/2025 no valor X...") ou oferece a possibilidade de editar dados incorretos.

O Agent Service também interage com o serviço de billing para contabilizar tokens usados em cada resposta. Além disso, toda chamada ao Agent é autenticada (JWT) para garantir que apenas o usuário dono dos dados acesse seu agente.

- **Billing Service:** Responsável por monitorar e gerenciar o **faturamento do SaaS** baseado no uso do LLM (contagem de tokens consumidos, número de perguntas, etc.). No MVP, não haverá cobrança real, mas ainda assim o sistema irá:
- Registrar em banco PostgreSQL cada interação ou quantidade de tokens gastos pelo usuário (p. ex., 1 pergunta = ~N tokens usados).
- Expor endpoints para o frontend consultar o consumo (ex.: para exibir "Você utilizou 50 tokens este mês").
- Preparar o terreno para no futuro aplicar limites de plano ou cobrança. No MVP de demonstração, esses limites estarão abertos (ilimitado), mas o registro é feito.

Este serviço pode também enviar notificações se necessário (fora do escopo MVP) ou gerar faturas. Mas inicialmente, foca-se no **registro do uso**. Os dados de billing (tokens usados por usuário, planos, etc.) residem em tabelas no PostgreSQL.

Tecnologias chave: Todos os microsserviços Python usarão **FastAPI** (versão estável mais recente) devido à sua performance e suporte a APIs assíncronas. Cada serviço rodará via servidor ASGI (Uvicorn ou Gunicorn+Uvicorn workers) dentro do container. Serão utilizadas versões **LTS** das plataformas e linguagens: Python 3.11+, Node.js 20 LTS (para build do frontend), NGINX estável, etc., assegurando estabilidade. As dependências incluem: `PyJWT` para tokens JWT no Auth, `pydantic` / `FastAPI` for models, `Celery==5.x` para fila de tarefas, `redis==7` para broker, driver do Mongo (`motor` ou `pymongo`), SQLAlchemy ou similar para Postgres, cliente MinIO (`minio` SDK em Python) para operações S3, e bibliotecas de IA (cliente HTTP do Gemini, ou fallback para OpenAI API se necessário para embeddings).

Armazenamento de Dados e Arquivos

- **PostgreSQL 15**: Banco de dados relacional principal. Usado pelo serviço Auth (tabela de usuários, senhas hash), pelo Billing (tabelas de uso, planos, pagamentos se houvesse) e também para armazenar os **vetores de embedding** dos documentos para busca semântica. Com a extensão **pgvector**, o Postgres suporta colunas do tipo vetor e consultas de similaridade (nearest neighbors) de forma nativa [3]. Essa escolha simplifica a arquitetura ao **manter dados estruturados e vetoriais no mesmo SGDB**, evitando adicionar outro banco especializado [6]. Todos os microsserviços que precisam armazenar/consultar dados relacionais se conectarão ao PostgreSQL (cada serviço com credencial/privilégios restritos às suas schemas/tabelas). Os contêineres do Postgres serão configurados já com a extensão pgvector habilitada (via variável ou script `CREATE EXTENSION`). Dados sensíveis (senhas) serão armazenados apenas como hashes seguros (e.g. Bcrypt via PassLib).

- **MongoDB 6**: Banco de dados NoSQL orientado a documentos. Escolhido para armazenar os **dados extraídos dos documentos fiscais** dos usuários, em formato flexível (JSON). Cada documento enviado pode gerar um registro contendo campos como `user_id`, `file_name`, `text_ocr` (texto completo extraído) ou mesmo estruturas de campos reconhecidos (ex: `{"date": "...", "amount": 123.45, "issuer": "..."}`) para fácil consulta. O

MongoDB facilita evoluir o esquema de dados extraídos sem migrações complexas – útil porque diferentes documentos podem ter diferentes estruturas de informação. O serviço Documentos se comunica com o Mongo para inserir e ler esses documentos. Índices podem ser usados (ex: por `user_id` para listar documentos de um usuário).

- **MinIO (Objeto Storage S3-like)**: Para armazenar os arquivos binários (uploads) usamos o **MinIO**, um servidor open-source compatível com API S3 da Amazon. Isso permite salvar PDFs/imagens de forma durável e acessá-los via HTTP assinados ou via SDK, simulando um bucket S3 localmente [7] [8]. O MinIO roda em um contêiner separado, com um volume montado para persistir os dados. Na arquitetura, o serviço Documentos, ao receber um upload, usará o SDK Python do MinIO para gravar o arquivo no bucket (por exemplo, bucket "documents"). O uso do MinIO elimina a dependência de nuvem (AWS S3) no MVP, mantendo a solução self-hosted. Os arquivos podem ser recuperados pelo ID quando necessário (por exemplo, o agente de IA poderia acessar a imagem original se precisasse – não previsto agora). As credenciais de acesso MinIO (chave e segredo) serão configuradas via variáveis de ambiente e compartilhadas somente entre Documentos service (e possivelmente o Celery worker, caso ele precise ler o arquivo para OCR).

Processamento OCR Assíncrono (Celery & OCR)

Para extrair texto dos documentos enviados, o sistema usará **OCR (Optical Character Recognition)** através do mecanismo **Tesseract**. Como essa operação pode ser demorada para arquivos grandes ou múltiplos, adotamos uma arquitetura assíncrona com fila de tarefas:

- **Celery Worker (Python):** Um ou mais contêineres rodarão o worker do **Celery**, que fica escutando tarefas enviadas na fila. Quando um usuário faz upload de um documento, o serviço Documentos enfileira uma tarefa (via Celery) do tipo "processar OCR do arquivo X para usuário Y". O **Redis** serve como **message broker** do Celery (fila) e também como backend de resultados (ou alternativamente poderíamos salvar resultado direto no MongoDB). O Celery permite escalonar consumidores se necessário e mantém a API web responsiva – ou seja, o upload retorna rápido com status "em processamento" enquanto o OCR ocorre em background [2]. Isso evita travar o serviço web em requisições longas.

- **Tesseract OCR:** No container do worker, estará instalada a engine Tesseract (versão estável 5.x) e possivelmente wrappers Python como `pytesseract` para facilitar chamada. Quando a tarefa Celery é consumida, o worker acessa o arquivo: pode baixar do MinIO (o arquivo fica acessível via URL interna ou usando o SDK), ou montamos o volume do MinIO no worker para acesso direto. O Tesseract então é executado na imagem/PDF e retorna o texto extraído. Esse texto é post-processado se necessário (p.ex., conversão de caracteres especiais, extração de números). Em seguida, o worker armazena o resultado:

- Insere o texto no **MongoDB** (atualiza o registro do documento com o campo `text_ocr` preenchido e status "processado").
- Aciona o cálculo de **embedding vetorial**: o worker ou o serviço Agent será chamado para gerar embedding do texto para futura busca. Uma opção: o Celery worker envia uma requisição interna para o Agent Service tipo "embed this text", ou chama diretamente a API de embedding (Gemini Embeddings) e insere o vetor no Postgres. Integrar via Agent centraliza a lógica de IA lá; porém, para simplificar, o próprio worker poderia usar a API de embedding do Gemini (se disponível) ou uma biblioteca local de embedding (ex: SentenceTransformers) apenas para o MVP. Optando pela arquitetura dada, podemos supor o uso do **Gemini Embeddings model** via API HTTP para consistência – i.e., o worker envia texto para endpoint de embedding, recebe o

vetor e então usa psycopg2/SQLAlchemy para inserir no Postgres (tabela de vetores) referenciando o `document_id`.

- **Notificação de conclusão:** Assim que o OCR (e embedding) finaliza, o sistema marca o documento como pronto. O frontend poderá consultar o status periodicamente via API (polling) ou receber um websocket/sinal (dependendo do tempo, provavelmente polling simples). Ao usuário, no dashboard, aparecerão os dados extraídos do documento após pronto.

Essa arquitetura assíncrona garante que mesmo vários uploads grandes não irão deixar a aplicação lenta – o trabalho pesado fica nos workers, e podemos escalar mais instâncias de worker se necessário.

Integração com LLM (Gemini) e RAG

O diferencial do sistema é o assistente inteligente para dúvidas fiscais, potenciado por um **LLM de última geração (Google Gemini)**. Detalhes dessa integração:

- Usaremos a **API do Gemini** fornecida pelo Google (acesso via chave de API). O Gemini possui modelos de texto com alta capacidade e um contexto extenso (há versões com janela de até 1 milhão de tokens, permitindo passar bastante conteúdo [9]). Para nosso MVP, escolheremos um modelo balanceado (ex: *Gemini 2.5 Flash*) para respostas rápidas e custo menor, e utilizaremos também o **modelo de embeddings do Gemini** para converter textos em vetores de alta dimensão, adequados ao nosso banco vetorial.
- O fluxo de uma pergunta do usuário no chat será: o frontend envia a questão para o Agent Service, que irá buscar **documentos relevantes** do usuário no Postgres (via similaridade de embeddings). Essa etapa de **retrieval** garante que a resposta do LLM seja fundamentada nos dados atualizados e específicos do usuário (por exemplo, valores de faturamento, datas de notas, etc.), mitigando lacunas de conhecimento do modelo [10]. Em seguida, o Agent compõe o **prompt** combinando:
- Um *system prompt* curto explicando o papel do assistente (por ex: "Você é um assistente fiscal que ajuda MEIs. Responda de forma clara, em português, e cite dados do usuário quando relevante.").
- Contexto inserido com os trechos dos documentos recuperados (limitando para caber no contexto do modelo).
- A pergunta do usuário em si (user prompt).
- Essa montagem é enviada via POST HTTP para o endpoint de geração de conteúdo do Gemini (ex.: `v1beta/models/gemini-2.5:generateText`). A resposta retornará provavelmente um texto de várias frases respondendo à dúvida. O Agent então envia essa resposta de volta ao frontend.
- **Edição de dados extraídos:** Caso o usuário perceba que algum dado usado na resposta está incorreto (por erro de OCR, por ex.), ele poderá acionar uma funcionalidade de editar. Isso pode ser implementado de duas formas no MVP:
- Via interface no dashboard: exibir os dados extraídos de cada documento (como uma tabela de valores) com opção de editar campos. Ao salvar, atualizamos o MongoDB e possivelmente recalculamos o embedding (para refletir o dado corrigido) no Postgres.
- Via chat: o usuário pode escrever "O valor da nota X está errado, deveria ser Y". Essa entrada poderia ser detectada e o sistema atualiza os dados. Dado o tempo curto, a primeira abordagem (UI de edição manual) é mais viável. Essa feature é considerada de prioridade menor no MVP, mas planejada caso o tempo permita, pois aumenta a confiabilidade das respostas.
- O **conhecimento fiscal estático** (regras gerais de MEI) não estará necessariamente em nenhum banco específico no MVP; confiaremos no conhecimento interno do LLM para perguntas gerais. Porém, poderíamos melhorar isso carregando um pequeno conjunto de documentos de referência (legislação do Simples Nacional para MEI, por ex.) também como embeddings. Em

RAG, isso seria simplesmente um conjunto global de vetores consultados além dos documentos do usuário. Devido ao tempo, podemos não implementar essa base de conhecimento externa explicitamente, mas é uma possibilidade futura.

- **Contagem de tokens:** A API Gemini (assim como OpenAI) deve fornecer contagem de tokens usados na resposta. O Agent Service extrairá essa informação da resposta ou calculará pelo tamanho do prompt+resposta. Em seguida, chamará um endpoint do Billing Service (ex: `POST /usage`) com o usuário e quantidade de tokens, para registro. Esse registro permitirá futuramente cobrar ou limitar. No MVP, apesar de não limitar, queremos **mostrar no dashboard talvez quantas interações foram feitas**.

Segurança e Controle de Acesso: Toda comunicação entre frontend e backend será feita sobre HTTPS (no deploy final; localmente, via HTTP mesmo). O JWT de autenticação será enviado no header Authorization: Bearer. Os microsserviços (Auth, Documents, Agent, Billing) verificarão o JWT em cada requisição privada – por exemplo, o Agent só aceitará perguntas de usuários autenticados e usará o `user_id` do token para filtrar seus dados. Essa abordagem segue as melhores práticas do FastAPI para segurança com JWT [1]. Podemos configurar **CORS** no FastAPI para permitir o frontend origin. O NGINX gateway também pode ser configurado para rejeitar requisições não autenticadas (usando `auth_request` para o Auth service) se quisermos reforçar isso [11], mas inicialmente deixaremos a lógica de auth nos serviços para simplicidade.

Resumo das Tecnologias (Stack):
- *Linguagens:* Python 3.11+ (backend), JavaScript/TypeScript (frontend).
- *Frameworks:* FastAPI (backend REST), React 18 (frontend SPA).
- *UI:* Tailwind CSS 3.x para estilo rápido e consistente.
- *Banco de dados relacional:* PostgreSQL 15 (ou 14) com extensão **pgvector** [3] (armazenamento de usuários, billing e vetores).
- *Banco de documentos:* MongoDB 6 para dados extraídos e possivelmente histórico de conversas (não essencial no MVP).
- *Armazenamento de arquivos:* MinIO (deployment recente, compatível S3) para objetos (documentos enviados) [8].
- *Cache/Mensageria:* Redis 7 para fila Celery e cache se necessário.
- *Fila de tarefas:* Celery 5.x, rodando em container separado, para OCR e outras tarefas offline.
- *OCR:* Tesseract OCR 5 no container do worker (via pytesseract ou tesseract CLI).
- *LLM:* API Google **Gemini** (modelos 2.5 – uso via chamadas REST autenticadas). Em desenvolvimento local, se a API não estiver disponível, poderíamos usar um placeholder (ou OpenAI API) apenas para não bloquear progresso, mas assumiremos acesso ao Gemini conforme proposta.
- *Containerização:* Docker Engine 24+ e Docker Compose v2 (formato Compose YAML 3.x) para definir os serviços. Todos os componentes (frontend, backend services, dbs, etc.) rodarão em containers isolados, facilitando a implantação consistente em qualquer host.
- *Orquestração:* Docker Compose local para desenvolvimento e demonstração. A composição incluirá todos os serviços mencionados. Para ambiente de produção real, essa stack poderia ser implantada em um servidor ou cluster, mas no MVP usaremos o próprio Docker Compose local como "produção simulada".
- *Controle de versão:* Git (monorepo ou repos separados para frontend e cada backend service, dependendo da organização – possivelmente um repositório unificado com subpastas para simplificar o MVP).
- *Outras libs:* bcrypt para senhas, uvicorn server, axios ou fetch no frontend, etc.

Essa arquitetura **prioriza apenas ferramentas essenciais para rodar localmente** – por exemplo, utilizamos MinIO ao invés de S3, e Docker Compose ao invés de Kubernetes, para manter o escopo

enxuto. O foco é entregar um sistema funcional cobrindo o fluxo principal: cadastro/login → upload doc → processamento → chat com IA utilizando os dados.

Backlog de Funcionalidades (Histórias de Usuário)

Abaixo estão listadas as histórias de usuário planejadas para o MVP, escritas do ponto de vista do usuário (MEI) ou do sistema. As funcionalidades consideradas **essenciais** para o MVP de 1 mês estão marcadas como **(Essencial)**. Funcionalidades marcadas como **(Baixa prioridade)** são desejáveis, porém podem ser postergadas caso o tempo não seja suficiente, ou implementadas de forma simplificada.

Funcionalidades Essenciais (MVP)

- **Como usuário MEI, quero me cadastrar no sistema com email e senha,** para ter uma conta segura onde meus dados fiscais ficarão armazenados. *(Essencial)*
- **Como usuário, quero fazer login com minhas credenciais e receber um token de acesso,** para acessar as funcionalidades protegidas do sistema. *(Essencial)*
- **Como usuário autenticado, quero visualizar um** dashboard **com meu faturamento acumulado no mês e no ano,** para acompanhar se estou próximo do limite do MEI (R$81 mil anuais [4]). *(Essencial)*
- **Como usuário autenticado, quero registrar (fazer upload) de documentos fiscais (notas, recibos) no sistema,** para centralizar minhas receitas e permitir que o sistema extraia as informações automaticamente. *(Essencial)*
- **Como sistema (serviço de documentos), quero processar cada documento enviado realizando OCR e extração de dados,** para obter texto e valores dos documentos sem intervenção manual. *(Essencial)*
- **Como sistema, desejo armazenar os arquivos enviados de forma segura e acessível,** para que possam ser referenciados posteriormente (ex: reprocessamento ou auditoria). *(Essencial)*
- **Como usuário, quero ver uma lista dos meus documentos enviados e os dados principais extraídos de cada um (data, valor, descrição),** para conferir minhas transações e verificar se estão corretas. *(Essencial)*
- **Como usuário, quero conversar via chat com um** assistente virtual (IA) **dentro do sistema,** para tirar dúvidas sobre minhas obrigações fiscais e receber orientações personalizadas. *(Essencial)*
- **Como usuário, quero que o assistente IA utilize meus dados (faturamento, notas enviadas) ao responder perguntas como "quanto já faturei este ano?" ou "posso exceder o limite do MEI?",** para obter respostas precisas e relevantes à minha situação. *(Essencial)*
- **Como sistema (serviço de agente), quero buscar nos documentos do usuário informações relevantes antes de responder no chat,** para fornecer ao modelo de IA contexto acurado e evitar respostas erradas [10]. *(Essencial)*
- **Como usuário, quero que o assistente exiba em suas respostas alguns detalhes dos meus dados quando útil (por exemplo, "Você já faturou R$50.000 este ano até agora"),** para confiar que a resposta está embasada nas minhas informações. *(Essencial)*
- **Como sistema (serviço de billing), quero contabilizar o uso do assistente AI em termos de tokens ou interações por usuário,** para possibilitar controle de uso e futuramente cobrança pelo serviço. *(Essencial)*
- **Como usuário, quero poder visualizar no dashboard ou perfil quantas consultas/interações de chat fiz no mês,** para ter transparência sobre meu uso (mesmo que ilimitado no teste). *(Essencial)*
- **Como desenvolvedor (e stakeholder), quero que toda a arquitetura rode localmente usando Docker Compose,** para facilitar o deploy de demonstração em qualquer máquina sem configurações manuais complexas. *(Essencial)*

Funcionalidades de Baixa Prioridade (Opcional / Pós-MVP)

- **Como usuário autenticado, quero editar manualmente os dados extraídos de um documento (ex: corrigir um valor ou data),** caso o OCR ou parser tenha cometido algum erro, garantindo que minhas informações estejam corretas. *(Baixa prioridade)*
- **Como usuário, gostaria que o sistema me alertasse no dashboard ou via chat se meu faturamento estiver próximo do limite anual do MEI (por exemplo, acima de 90% do teto),** para eu tomar providências antecipadamente. *(Baixa prioridade)*
- **Como usuário, quero visualizar um histórico das perguntas que fiz ao assistente e suas respostas,** para consultar novamente informações sem precisar perguntar de novo. *(Baixa prioridade)*
- **Como usuário, quero poder fazer logout da minha conta de forma segura,** para encerrar minha sessão em dispositivos compartilhados. *(Baixa prioridade)*
- **Como administrador do sistema (futuro), quero gerenciar planos de assinatura e limites de tokens por plano,** para poder monetizar o SaaS adequadamente. *(Baixa prioridade)*
- **Como administrador, gostaria de ter métricas gerais de uso (número de usuários, docs processados, tokens consumidos no total),** para avaliar a carga e benefícios do sistema. *(Baixa prioridade)*

(Obs: histórias administrativas estão fora do escopo do MVP, mas citadas para contexto futuro.)

O backlog acima está ordenado em prioridade aproximada. Durante o desenvolvimento de 4 semanas, daremos foco máximo às histórias essenciais. As de baixa prioridade serão implementadas apenas se as principais forem concluídas antes do prazo ou poderão ser simuladas na demo.

Plano de Desenvolvimento (Cronograma de 4 Semanas)

A seguir está um plano de alto nível para desenvolver o MVP em **4 semanas**. Cada semana possui objetivos claros e entregáveis. O plano assume uma pequena equipe ou desenvolvedor focado, com iterações semanais. Utilizamos abordagens ágeis, entregando valor incrementalmente (por ex., ao fim da semana 2 já teremos upload e OCR básico funcionando, ao fim da semana 3 o chat integrado, etc.).

Semana 1 – Configuração Inicial e Autenticação:

- **Planejamento e Setup do Projeto:** Configurar repositório (estruturar pastas para frontend e microsserviços backend). Definir no Docker Compose os serviços básicos (PostgreSQL, MongoDB, Redis, MinIO, e placeholders para cada serviço Python e frontend). Garantir que todos os contêineres sobem e se comunicam em rede interna.
- **Serviço de Auth (Backend):** Implementar o microsserviço de autenticação usando FastAPI:
- Criar modelo de usuário (SQLAlchemy ou similar) no PostgreSQL.
- Implementar endpoints `/register` (criação de usuário com hash de senha) e `/login` (geração de JWT ao validar credenciais). Utilizar **JWT (PyJWT)** conforme guia do FastAPI [1] com segredo aplicativo.
- Implementar proteção básica em um endpoint de teste (ex: `/auth/me`) que retorna dados do usuário logado, para validar o mecanismo JWT.
- **Frontend – Autenticação:** Bootstrap do front em React (usando Vite ou Create React App):
- Configurar Tailwind CSS.
- Criar telas simples de **Cadastro** e **Login** com formulários.
- Integrar chamadas à API Auth (usando fetch/axios) para registrar e logar. Armazenar o token JWT no cliente (localStorage ou memory) ao logar.

- Proteger rotas internas do front (usar contexto de auth ou router) – se não logado, redireciona para login.
- **NGINX Gateway:** Escrever configuração de NGINX para rotear `/api/auth/**` para o serviço Auth (que rodará, por exemplo, em `http://auth:8000` dentro da rede Docker). Fazer o mesmo para outros serviços (mesmo se ainda não implementados, já planejar rotas `/api/docs`, `/api/agent`, `/api/billing`). Configurar NGINX também para servir a app React (os arquivos estáticos construídos) na rota `/` padrão.
- **Teste de ponta a ponta (E2E) básico:** Subir todo o ambiente com `docker-compose up`. Testar via navegador o fluxo de cadastro e login:
- Criar usuário novo (verificar no Postgres que foi salvo).
- Login obtendo JWT (verificar presença do token e capacidade de chamar um endpoint protegido, ex: `/auth/me` retornando dados do usuário).
- Ajustar CORS ou NGINX caso haja bloqueios de requisições do front para back.
- **Entrega Semana 1:** MVP parcial com sistema de usuários funcional. Riscos mitigados: garantia que infraestrutura de banco e comunicação via JWT está ok, que Docker Compose é capaz de orquestrar múltiplos serviços. Documentar no README instruções para rodar (útil para demonstração final).

Semana 2 – Funcionalidade de Documentos (Upload & OCR):

- **Serviço de Documentos – Upload:** Desenvolver endpoint `/documents/upload` no serviço Documentos:
- Aceitar upload (multipart/form-data) de arquivo. Validar JWT do usuário (via Dependency do FastAPI).
- Ao receber arquivo, gerar um ID (UUID) e nomeá-lo no bucket MinIO. Usar SDK MinIO para fazer **upload do arquivo** ao bucket privado [8]. Salvar registro no MongoDB com campos: user_id, doc_id, nome, status="pending", timestamp.
- Retornar ao frontend uma resposta com status de aceite (talvez os dados do documento cadastrado, sem o conteúdo ainda).
- **Integração Celery (OCR):** Configurar Celery no projeto:
- Criar container Docker para **worker** (usar mesma imagem base do Documentos service ou separate) que roda `celery -A app.tasks worker`.
- Configurar Redis (broker URL no Celery).
- Implementar função tarefa async `process_document(document_id)`:
    - No código da tarefa, obter do Mongo o documento pelo `document_id`.
    - Usar MinIO SDK para baixar o arquivo (ou apontar caminho local se usamos volume compartilhado).
    - Executar **OCR**: usar `pytesseract.image_to_string` para imagens ou PDF (possivelmente convertendo PDF->imagens por `pdf2image` se necessário).
    - Obter texto OCR. Atualizar documento no Mongo: preencher campo `text_ocr` com o texto extraído, e `status="processed"`.
    - (Extra) Tentar extrair campos estruturados simples: ex., buscar por padrões de valor (R$) e datas – se trivial, guardar separados.
    - Enfileirar ou chamar função para **gerar embedding** do texto: podemos aqui mesmo calcular embedding. Ideal: chamar API do Gemini Embeddings:
    - Preparar requisição com o texto (ou pedaços do texto se muito grande).
    - Receber vetor (128-768 dims, dependendo do modelo).
    - Conectar ao Postgres e inserir na tabela de embeddings (colunas: id, user_id, doc_id, embedding vetor).
    - Marcar sucesso ou log de erro. (Em caso de falha OCR, atualizar status="error" no Mongo para feedback.)

- Testar local a execução da tarefa isoladamente (usar `celery worker` logs).
- **Serviço de Documentos – Dispatch de OCR:** No endpoint upload, após salvar registro, chamar de forma assíncrona a tarefa Celery: `celery.send_task('process_document', args=[doc_id])`. (Alternativa: usar Celery delay/apply_async direto). Confirmar que o worker recebe a tarefa (ver logs).
- **Frontend – Upload UI:** Integrar no dashboard uma seção para **upload de documentos**:
- Formulário com campo de arquivo e botão enviar.
- Lista de documentos já enviados abaixo, com colunas: nome, data upload, status (Pendente ou Processado), valor (quando disponível).
- Ao enviar um arquivo, chamar via JS fetch `POST /api/docs/upload` com JWT. Exibir feedback (ex: "Documento enviado, processando...").
- Opcional: implementar polling a cada X segundos para atualizar status dos documentos listados. Ou permitir refresh manual.
- **Dashboard – Cálculo de Faturamento:** Implementar no serviço Documentos ou em um helper no frontend a **soma do faturamento**:
- No backend, criar endpoint `/documents/summary` que soma todos os valores extraídos dos docs do usuário (só dos processados) e retorna total mensal (filtrando por mês atual) e anual (ano atual). *Se não tivermos valores estruturados extraídos automaticamente, podemos simular solicitando que o usuário informe o valor ao fazer upload, ou extrair do nome do arquivo.* Contudo, como MVP, podemos assumir que o OCR ou a nota fiscal contém valor e tentamos parsear. Se for problemático, deixar para o agente responder consultas de total.
- No frontend dashboard, exibir: "Faturamento este mês: R$X de R$6.750; Faturamento este ano: R$Y de R$81.000 [4] " talvez com uma barra de progresso.
- Essa funcionalidade ajuda o MEI a ver se está perto do limite.
- **Teste de ponta a ponta:** Realizar upload de um documento de exemplo (pode ser uma imagem de recibo com texto legível):
- Verificar que o arquivo foi salvo no MinIO (via console web do MinIO em :9001 ou via CLI).
- Verificar que o registro no Mongo foi criado.
- Ver logs do Celery para ver processamento e conclusão.
- Após um tempo, checar que o Mongo documento agora tem status "processed" e campo `text_ocr`.
- Verificar que o embedding foi salvo no Postgres (consulta na tabela embeddings).
- No frontend, verificar que a lista de documentos atualiza para "Processado" e possivelmente mostrar um trecho do texto ou o valor extraído.
- **Entrega Semana 2:** MVP parcial onde usuários autenticados conseguem enviar documentos e tê-los processados. O backend agora integra o pipeline completo de **Upload -> OCR -> Armazenamento**. O dashboard começa a mostrar dados reais (mesmo que básicos). Documentar qualquer step manual (ex: se precisar instalar tesseract no host ou ajustar permissões).

Semana 3 – Integração do Assistente LLM (Chatbot):

- **Serviço Agent – Consulta com RAG:** Implementar no Agent Service o endpoint principal de chat, por exemplo `POST /agent/chat` com corpo `{question: "texto da pergunta"}`:
- Proteger com autenticação JWT.
- No handler, receber a pergunta do usuário. Usar biblioteca de embeddings (ou a própria tabela) para calcular embedding da pergunta. Podemos chamar o modelo de embedding Gemini para a pergunta também, ou utilizar uma aproximação (ex: mesma tabela, inserindo temporariamente e consultando). Simpler: usar `pgvector` funcionalidade de similaridade diretamente: realizar uma query SQL do tipo:

```
SELECT doc_id, embedding <-> :user_embedding AS distance
FROM embeddings
WHERE user_id = :user_id
ORDER BY distance
LIMIT 5;
```

(Assumindo `<->` operador de distância coseno ou L2 indexado). Isso retorna os IDs dos top documentos. Então recuperar do MongoDB os textos ou dados desses documentos.

- Montar o **prompt**: Concat as pieces of document text (ou sumarizá-los se muito longos) junto com a pergunta. Poderíamos preceder cada peça com algo como "Dado o documento: [texto]".
- Chamar a **API do Gemini**:
  - Montar a requisição HTTP para o endpoint de geração de texto. Incluir no body o prompt montado.
  - Incluir talvez parâmetros como temperatura baixa para respostas mais objetivas, e talvez indicar idioma PT-BR.
  - Receber a resposta (com stream desativado no MVP para simplicidade – espera a resposta completa).
- Parsear a resposta (JSON) para extrair o texto respondido.
- Retornar ao frontend o texto da resposta.
- *Implementar breve caching:* Se a mesma pergunta for feita em curto intervalo, poderíamos armazenar a resposta para não gastar tokens duplicados (mas não essencial).
- Registrar o uso de tokens: do objeto de resposta do Gemini, extrair `usage.total_tokens` (se disponível) ou calcular via length. Chamar Billing: `POST /billing/usage` com `{user_id, tokens: n, timestamp}`. Ou colocar numa fila ou direto no DB.
- **Serviço Billing – Registro de uso:** Implementar endpoint no Billing Service para registrar uso:
- Por simplicidade, pode ser que o Agent já tenha acesso ao DB e possa gravar diretamente. Mas seguindo microserviços puros, o Agent faz requisição. Implementar `/billing/usage` que verifica autenticação (talvez somente serviços internos deveriam chamar – poderíamos skip JWT e usar uma secret interna ou IP restrito, mas para MVP pode deixar JWT de user mesmo).
- Billing service insere na tabela `usage` (user_id, tokens, timestamp). Também podemos agregar por dia/mês para facilidade (mas não agora).
- Endpoint `/billing/summary` para o frontend pegar total de tokens usados no mês (soma filtrando tabela).
- **Frontend – Interface de Chat:** Desenvolver no frontend uma tela ou seção de **chatbot**:
- Pode ser no dashboard mesmo: um box de chat onde o usuário digita pergunta e vê a resposta. Ou numa página separada "/chat".
- Implementar um componente simples: lista de mensagens (perguntas e respostas), e um campo input embaixo.
- Ao enviar pergunta, otimisticamente mostrar na lista e limpar input. Chamar API `/api/agent/chat` via fetch com JWT.
- Enquanto aguarda resposta, talvez mostrar "...". Ao receber, exibir a resposta na lista.
- (Baixa prio: permitir feedback ou avaliar resposta? Provavelmente não no MVP.)
- Manter em estado local o histórico durante a sessão. Se quisermos persistir histórico, poderíamos salvar no Mongo (ex: coleção chats com user_id, conversation as array). Mas no MVP, talvez não persistir para simplicidade. O histórico se perderia ao recarregar a página.
- **Dashboard – Atualizar com Billing info:** No componente dashboard principal, adicionar uma seção "Consumo" mostrando, por ex: "Você fez X perguntas este mês" ou "Tokens usados: Y (limite do plano: Ilimitado)". Consumir o endpoint `/billing/summary` ao carregar.
- **Teste de ponta a ponta:**

- Criar ou usar usuário existente, que já tenha alguns documentos processados (usar dados da semana 2).
- Ir à tela de chat, perguntar algo como “Qual meu faturamento este ano?” ou “Estou perto do limite do MEI?”.
- Verificar no backend logs que:
    - Agent service buscou embeddings, encontrou documentos relevantes (talvez aqueles com valores).
    - Fez chamada ao LLM (isso é difícil testar local se não temos chave – poderíamos simular a resposta com alguma lógica estática se necessário para teste).
    - Retornou uma resposta coerente (ex: “Seu faturamento atual é X, ainda está abaixo do limite anual de R$81 mil.”).
    - Billing registrou tokens.
- Ver no frontend a resposta aparecendo. Testar outra pergunta para conferir histórico local.
- Testar que o dashboard token usage atualiza (ex: se antes 0, agora mostra 2 perguntas feitas).
- **Ajustes e Polimento:** Realizar ajustes de UX – por exemplo, limpar lista de docs quando outro usuário loga (caso estado global), melhorar textos exibidos (linguagem amigável). Garantir que erros são tratados (ex: upload de arquivo não imagem deve retornar erro claro).
- **Entrega Semana 3:** Nessa fase, o MVP deve estar praticamente completo: usuários podem logar, subir docs e fazer perguntas ao assistente com base nos dados. Demonstrar o fluxo inteiro funcionando. Resta a última semana para refinamentos, testes adicionais e preparação da apresentação.

Semana 4 – Testes Finais, Refinamentos e Implantação:

- **Teste e Correção de Bugs:** Conduzir uma bateria de testes abrangentes:
- **Testes unitários básicos** em funções críticas (ex: função de OCR parsing, função de consulta vetorial).
- **Testes manuais** de uso: criar vários usuários, testar limites (ex: upload de arquivo grande, comportamento com muitos documentos, perguntas sem contexto).
- Corrigir bugs encontrados: erros de autenticação, tratamento de concorrência (ex: dois uploads simultâneos), etc.
- **Segurança & Robustez:** Verificar configurações de segurança:
- JWT expirando adequadamente (talvez fixar expiração de 1 hora para MVP). Implementar refresh token se tempo permitir (se não, documentar que o usuário precisaria relogar após expirar).
- Garantir que cada serviço rejeita requisições sem JWT ou com JWT inválido com código 401.
- Escapar/validar entradas onde necessário (embora a maioria sejam controladas pelo UI).
- Configurar **CORS** adequadamente no backend (permitir origin do front, bloquear origens desconhecidas em produção).
- **Documentação e Facilitação:** Escrever/documentar no README ou Wiki:
- Instruções para rodar o Docker Compose (como subir, pré-requisitos como Docker instalado).
- Descrição breve de cada serviço e portas.
- Credenciais padrão (se aplicável, ex: talvez um usuário admin ou keys de MinIO etc., embora isso esteja no .env/compose).
- Como obter uma chave de API do Gemini (ou usar variável de ambiente `GEMINI_API_KEY` no compose).
- Qualquer etapa manual para preparar (ex: carregar pgvector extension, que pode ser feito via comando init do Postgres container).
- **Melhorias de UX (se tempo):** Implementar funcionalidades secundárias se o core já estiver sólido:

- Edição de dados extraídos: no dashboard, permitir clicar num valor para editá-lo, salvar no Mongo e talvez recalcular embedding (o recalculo poderia ser postergado ou feito síncrono já que vetor de um doc pequeno não é pesado).
- Alerta visual no dashboard se faturamento anual > 70% do limite (por ex., colorir em amarelo, acima de 90% em vermelho).
- Logout no frontend (simplesmente limpar token e redirecionar).
- Página de erro ou mensagem amigável se o LLM não responder ou estiver indisponível (fallback: "Desculpe, o assistente está indisponível no momento.").
- **Performance e Deploy:** Mesmo em MVP, verificar uso de recursos:
- Ajustar *timeout* das requisições do front ao chat (pois respostas longas podem demorar ~10s).
- Configurar no NGINX proxies timeouts maiores para `/api/agent/chat` if needed.
- Ver memória consumida por Mongo/Redis (talvez limitar no compose).
- Se possível, testar em ambiente semelhante a produção – e.g., rodar Docker Compose em uma VM limpa para verificar passos.
- **Demonstração final:** Preparar alguns dados de exemplo para a demo:
- Cadastrar um usuário de teste.
- Carregar documentos fictícios (por exemplo, 5 notas fiscais com valores variados, totalizando perto de 81k para mostrar limite).
- Deixar pré-processado (ou acelerar processamentos).
- Ensaiar perguntas que demonstrem bem o sistema (por exemplo: "Liste minhas últimas notas fiscais e valores", "Quanto posso faturar a mais sem estourar o limite?", "O que acontece se eu passar do limite do MEI?").
- Assegurar que o LLM responde corretamente – possivelmente ajustar o prompt do agente para orientar a resposta (por ex: "Se a pergunta for sobre limite e o usuário estiver perto, avise-o do percentual..."). Pequenos *fine-tunings* via prompt engineering podem ser feitos nessa hora.
- **Entrega Semana 4:** O sistema completo, rodando em containers, pronto para apresentação. Fazer uma **demonstração** integrando tudo: do cadastro ao uso do chat. Reunir feedback dos stakeholders e estar pronto para discutir próximas etapas (novas features ou deploy em cloud).

Considerações Finais: Este plano busca entregar um MVP funcional com o conjunto mínimo de componentes integrados. Usamos apenas ferramentas open-source e serviços locais (Dockerized) para evitar overhead de infraestrutura. A arquitetura modular com microsserviços Python FastAPI e comunicação REST torna o sistema escalável e de fácil manutenção, enquanto recursos como pgvector e Celery foram incorporados para atender requisitos específicos (IA com RAG e OCR assíncrono) respaldados por práticas recomendadas do mercado. Em 4 semanas, com foco e priorização, o objetivo é ter uma aplicação demonstrável que comprove o valor da ideia: facilitar a vida do MEI na gestão de seus impostos, combinando automação documental e inteligência artificial conversacional.

Fontes Conectadas: As decisões arquiteturais tomaram como base referências de tecnologias mencionadas: uso de JWT para segurança em aplicações FastAPI [1], estratégia RAG para fornecer contexto a modelos de linguagem [5], armazenamento vetorial integrado ao Postgres via pgvector [3], processamento assíncrono com Celery para manter o sistema responsivo [2], e utilização do MinIO como alternativa local ao S3 [7], entre outras. O limite legal de faturamento do MEI (R$81.000/ano) citado no dashboard foi confirmado em fonte confiável [4] para garantir acurácia da informação exibida. Essas referências corroboram as escolhas técnicas feitas e reforçam as melhores práticas seguidas no desenvolvimento deste MVP.

---

1 OAuth2 com Senha (e hashing), Bearer com tokens JWT - FastAPI
https://fastapi.tiangolo.com/pt/tutorial/security/oauth2-jwt/

2 11 Building a High-Availability API with Celery and Microservices Architecture: A Practical Guide | by Dwicky Feri | Medium
https://dwickyferi.medium.com/building-a-high-availability-api-with-celery-and-microservices-architecture-a-practical-guide-97f70351bd53

3 5 6 10 PostgreSQL as a Vector Database: A Pgvector Tutorial | Tiger Data
https://www.tigerdata.com/blog/postgresql-as-a-vector-database-using-pgvector

4 Limite MEI 2025: Teto de faturamento e nova proposta de 2025
https://www.contabilizei.com.br/contabilidade-online/faturamento-mei-2025/

7 8 Setting up Object Storage with MinIO with Docker - Docker - KodeKloud - DevOps Learning Community
https://kodekloud.com/community/t/setting-up-object-storage-with-minio-with-docker/336624

9 Gemini API | Google AI for Developers
https://ai.google.dev/gemini-api/docs