AxonAI - Visão Geral do Sistema e Proposta de Valor

1. O Problema

Desenvolver projetos com apoio de IA hoje ainda é um processo desarticulado:

- Conversas com LLMs geram ideias valiosas, mas sem estrutura para execução real.
- Usuários perdem tempo copiando e colando instruções em ferramentas como Jira ou Trello.
- Cada nova interação com a IA sofre com perda de contexto e decisões duplicadas.
- O resultado: projetos fragmentados, com baixa rastreabilidade e esforço manual desnecessário.

2. A Solução AxonAI

O AxonAI é um orquestrador inteligente que transforma conversas com IA em planos de projeto executáveis, mantendo foco e contexto em cada etapa.

- Você conversa com a IA.
- Ela transforma a conversa em tasks e checklists organizados (Kanban-ready).
- Com um único clique, você aprova esse plano e cria o projeto.
- A partir daí, cada item do checklist se torna um chat contextualizado através do nosso

  "Modo Foco", eliminando a necessidade de repetir o histórico da conversa.

3. O Diferencial

- **Aprovação única, não microgestão:** O controle humano existe apenas no ponto crítico — a criação do plano inicial. Depois disso, o fluxo é ágil.
- **Contexto Vivo e Granular com o Modo Foco:** Cada item de checklist ativa seu próprio **Modo Foco**, alimentando a IA apenas com o histórico relevante para aquela microtarefa específica.

- **Integração natural com o ecossistema do time:** Kanban automático (Trello, Jira, Linear etc.), mas sem obrigar a adoção de mais uma plataforma isolada.
- **Independência de fornecedor de IA (BYOK):** O usuário conecta suas próprias chaves de API, mantendo controle de custos e flexibilidade tecnológica.

4. Impacto Esperado

- 50% menos esforço operacional ao estruturar projetos assistidos por IA.
- Redução drástica de perda de contexto, aumentando a precisão da IA em tarefas complexas.
- Maior rastreabilidade e governança sobre decisões assistidas por IA, sem burocracia extra.

5. O Posicionamento

O AxonAI não é mais um gestor de tarefas. Não é apenas um copiloto. É a ponte entre ideação e execução, transformando o caos de conversas em um plano vivo e acionável.

AxonAI - Documento de Iniciação de Projeto

1. Objetivo, Escopo e Filosofia do Produto

- **Problema Resolvido:** Resolve a perda de contexto e a ineficiência do fluxo de trabalho ao usar LLMs para tarefas de desenvolvimento de software.
- **Utilizador Final:** Desenvolvedores de software, tanto individuais quanto em pequenas equipas.
- **Visão do Produto:** Criar um ambiente de execução de projetos que integre nativamente a ideação assistida por IA com a execução estruturada de tarefas, validando que um fluxo de trabalho com contexto focado aumenta significativamente a produtividade.
- **Filosofia de Engenharia:** Este projeto será guiado por princípios de engenharia de software que visam a construção de um sistema robusto, manutenível e seguro.
    - **Arquitetura Limpa e Testável:** O design favorece o baixo acoplamento e a alta coesão, com uma separação rigorosa entre o núcleo de negócio e a infraestrutura, garantindo a testabilidade e a longevidade do sistema.
    - **Segurança por Design:** A segurança não é uma camada adicional, mas um pilar da arquitetura, com padrões robustos implementados em todas as camadas.
    - **Escalabilidade e Resiliência:** A arquitetura é desenhada para ser operada em produção, com componentes que suportam escalabilidade e políticas de resiliência para lidar com falhas.

2. Arquitetura e Plataforma de Execução

- **Padrão Arquitetural:** Monolito Modular aplicando os princípios da Arquitetura Hexagonal (Ports and Adapters).
    - **Justificativa:** Garante uma organização de código e separação de responsabilidades de alto nível. Isola o núcleo de negócio da infraestrutura, permitindo a extração futura para microsserviços, se necessário.

- **Tecnologias Principais:**
  - **Backend:** Java 21 LTS, Spring Boot 3.5.5, Maven.
  - **Frontend:** React 19, Vite.
  - **Base de Dados:** PostgreSQL.
- **Estratégia de Implantação (Deployment):**
  - **Plataforma:** Render (PaaS - Plataforma como Serviço).
  - **Justificativa:** Abstrai a complexidade de infraestrutura, oferecendo "Git push-to-deploy", gestão integrada de bases de dados, gestão de segredos e escalabilidade simplificada.
- **Gestão de Configuração:**
  - **Método:** Utilização de perfis do Spring Boot e variáveis de ambiente para todas as configurações, incluindo segredos e URLs de conexão.
  - **Justificativa:** Padrão nativo do Spring que se integra perfeitamente com a gestão de segredos de plataformas PaaS.
- **Documentação da API:**
  - **Padrão:** OpenAPI 3 com a biblioteca `springdoc-openapi`.
  - **Justificativa:** Gera automaticamente uma especificação OpenAPI a partir dos controllers, garantindo que a documentação esteja sempre atualizada e fornecendo uma UI Swagger para testes interativos.

3. Arquitetura de Software de Referência

A arquitetura hexagonal é implementada com uma separação rigorosa de responsabilidades.

- **Domínio e Persistência:**
  - **Estratégia:** O Modelo de Domínio é puro e isolado, contendo a lógica de negócio. A camada de persistência é um `Adapter` que utiliza Entidades JPA e Mappers para traduzir os objetos de domínio para o formato do banco de dados, implementando os `Repository Ports`.
  - **Justificativa:** Desacopla totalmente a lógica de negócio da tecnologia de persistência, alinhando-se com os princípios da Arquitetura Hexagonal.
- **Camada de Interação com IA:**

    - **Estratégia:** A interação com a IA é gerida por um pipeline de componentes (`AIFacade` , `PolicyEngine` , `ProviderRouter` , `ApiKeyVault`), cada um com responsabilidades específicas.
    - **Justificativa:** Esta abordagem modular e extensível substitui um serviço unificado, permitindo a implementação de políticas de segurança, governança e resiliência de forma mais robusta e manutenível.
- **Controle de Custos de IA:**
    - **Estratégia:** Um `PolicyEngine` irá gerir o uso com base em "unidades de IA", com custos diferentes por tipo de operação e limites associados a planos de subscrição.
    - **Justificativa:** Cria um mecanismo de controle de custos granular, justo e escalável, essencial para a viabilidade do produto.

4. Estratégia de Segurança

A segurança é implementada com padrões de nível de produção.

- **Autenticação:**
    - **Estratégia: Refresh Token Rotation** com `Access Tokens` (JWT RS256) de curta duração e `Refresh Tokens` opacos de longa duração.
    - **Justificativa:** Oferece um mecanismo robusto para gestão de sessão e revogação, mitigando os riscos de roubo de tokens.
- **Segurança das Chaves BYOK:**
    - **Estratégia: Envelope Encryption** com um KMS (Key Management Service) dedicado.
    - **Justificativa:** Garante a proteção das chaves de API dos utilizadores com um padrão de segurança empresarial, separando a gestão de chaves da aplicação.
- **Segurança da Aplicação (Essenciais):**
    - **Hashing de Senha:** Bcrypt é mandatório.
    - **Proteção de Input:** Validação rigorosa de input e proteção contra SSRF são implementadas para mitigar riscos elevados.

5. Arquitetura Frontend

- **Gestão de Estado:**
  - **Estado do Servidor:** TanStack Query para automatizar o fetching, caching e sincronização de dados da API.
  - **Estado Global do Cliente:** Zustand para um estado global minimalista e de alto desempenho.
- **Biblioteca de Componentes:** Shadcn/ui ou Mantine para um desenvolvimento de UI rápido e de alta qualidade.
- **Qualidade e Testes:** A arquitetura inclui estratégias definidas para testes (unitários, integração e E2E), qualidade de código (ESLint, Prettier) e sincronização de tipos com o backend via OpenAPI.

6. Funcionalidades Essenciais e Estado Inicial

- **Funcionalidades Essenciais:**
  1. Autenticação de utilizador (local e Google) com gestão de sessão segura.
  2. Criação e gestão de Workspaces e Projetos.
  3. Mecanismo de "proposta e aprovação" para criação de tarefas via IA.
  4. Visualização do projeto com tarefas e checklists.
  5. "Focus Mode": chat contextual focado num `ChecklistItem`.
- **Estado Inicial do Sistema (Data Seeding):**
  - **Estratégia:** Scripts de migração de schema geridos pelo Flyway e um `CommandLineRunner` do Spring Boot para dados iniciais.
  - **Justificativa:** Garante um processo de inicialização do sistema automatizado, repetível e versionado.

7. Plano de Evolução da Arquitetura

- **Transição do MVP:** Este documento formaliza a superação da "dívida técnica consciente" assumida durante a fase de MVP.
- **Gatilho para Refatoração:** As simplificações da fase de MVP (modelo unificado, segurança pragmática) foram substituídas pelas estratégias

robustas descritas neste documento como parte da transição para um produto de produção.

- **Monitoramento Contínuo:** A arquitetura será continuamente revisada para garantir que continue a atender aos requisitos de escalabilidade, segurança e manutenibilidade do produto.

Fontes

Estratégia de CI/CD (Integração e Implantação Contínua) com GitHub Actions

A automação do ciclo de vida do desenvolvimento é fundamental para garantir a qualidade e a velocidade. Adotaremos uma estratégia de múltiplos workflows no GitHub Actions.

Princípios:

- **Automação Total:** Cada commit é um candidato a release, passando por um pipeline automatizado de validação.
- **Segurança no Pipeline:** Todos os segredos (credenciais de banco de dados, chaves de API, segredos de JWT) serão gerenciados via *GitHub Encrypted Secrets* e expostos ao pipeline apenas como variáveis de ambiente.
- **Consistência Ambiental:** Os ambientes de *Staging* e *Produção* serão provisionados com a mesma infraestrutura como código para garantir consistência.

1.1. Workflow de Validação (`on: pull_request`)

Este workflow atua como um portão de qualidade (quality gate) antes que o código seja mesclado à branch principal (`main`).

- **Gatilho:** Abertura ou atualização de um *Pull Request* para a branch `main` .
- **Objetivo:** Validar a integridade, qualidade e correção do código.
- **Passos:**
    1. **Checkout & Setup:** Configuração dos ambientes Java 21 e Node.js.
    2. **Validação do Backend:**
        - Executar testes unitários e de integração (`mvn test`). Os testes de integração utilizarão Testcontainers para instanciar um banco de dados PostgreSQL real, garantindo a compatibilidade.

- Executar testes de arquitetura (`mvn verify`) com ArchUnit para garantir que as dependências entre camadas não foram violadas.
- Compilar o artefato da aplicação (`mvn package`).

3. **Validação do Frontend:**
   - Instalar dependências (`npm install`).
   - Executar Linter e Formatter (`npm run lint`) para garantir a consistência do código.
   - Executar testes de componentes e integração (`npm run test`) com Vitest e React Testing Library.
   - Compilar os ativos estáticos (`npm run build`).

- **Regra de Proteção:** A mesclagem do *Pull Request* será bloqueada se qualquer um dos passos acima falhar.

1.2. Workflow de Implantação para Staging (`on: push, branches: [main]`)

Este workflow implanta a versão mais recente do código em um ambiente de homologação.

- **Gatilho:** Mesclagem de código na branch `main` .
- **Objetivo:** Disponibilizar uma versão funcional para testes de ponta a ponta (E2E) e validação de stakeholders.
- **Passos:**
  1. **Executar Validação:** Re-executar todos os passos do workflow de validação.
  2. **Build & Push da Imagem Docker:** Construir uma imagem Docker da aplicação backend e enviá-la para um registro de contêineres (ex: GitHub Container Registry).
  3. **Implantação no Render:** Acionar um *Deploy Hook* do Render via `curl` . O Render irá automaticamente baixar a imagem mais recente e reiniciar o serviço de Staging com zero downtime. As migrações de banco de dados do Flyway serão aplicadas automaticamente na inicialização da aplicação.
  4. **Executar Testes E2E:** Após a implantação, um job separado iniciará os testes de ponta a ponta (com Playwright) contra o ambiente de Staging.

5. **Notificação:** Enviar uma notificação para um canal do Slack informando o sucesso da implantação.

1.3. Workflow de Implantação para Produção (`on: release, types: [published]`)

Este workflow promove uma versão estável e testada para o ambiente de produção.

- **Gatilho:** Criação de uma nova *Release* no GitHub (ex: `v1.2.0`).
- **Objetivo:** Implantar uma nova versão para os usuários finais de forma controlada e segura.
- **Passos:**
  1. **Aprovação Manual:** O workflow será pausado aguardando a aprovação de um revisor definido (ex: Tech Lead), utilizando o ambiente de `production` do GitHub Actions, que exige aprovação.
  2. **Implantação no Render:** Após a aprovação, o *Deploy Hook* do serviço de Produção do Render é acionado.
  3. **Verificação de Saúde:** Um job de *smoke test* verifica se os principais endpoints da aplicação estão respondendo com status `200 OK` após a implantação.
  4. **Notificação:** Enviar uma notificação de alta prioridade sobre a implantação em produção.

2. Estratégia de Observabilidade e Monitoramento

A observabilidade é construída sobre três pilares: Logs, Métricas e Traces.

2.1. Dashboards

Criaremos dashboards específicos para monitorar diferentes aspectos do sistema em uma ferramenta como Datadog ou Grafana.

- **Dashboard de Saúde da API:**
  - **Métricas:** Taxa de requisições (RPM), taxa de erros (separada por 4xx e 5xx), latência (percentis p50, p90, p99), saúde da JVM (uso de heap, CPU).
  - **Objetivo:** Monitorar a saúde técnica e a performance da aplicação.

- **Dashboard de Uso de IA:**
  - **Métricas:** Latência das chamadas à API de IA por provedor, taxa de erros, tokens de entrada/saída por modelo, status dos *Circuit Breakers*.
  - **Objetivo:** Controlar custos, performance e a fiabilidade dos serviços de IA de terceiros.
- **Dashboard de Métricas de Negócio:**
  - **Métricas:** Novos cadastros, número de projetos criados, execuções do fluxo "proposta e aprovação" da IA, usuários ativos diários (DAU).
  - **Objetivo:** Fornecer visibilidade sobre a utilização do produto.

2.2. Estratégia de Alertas

- **P1 - Alertas Críticos (Notificação imediata para a equipe de plantão):**
  - Taxa de erros 5xx > 5% por mais de 5 minutos.
  - Latência p99 da API > 2 segundos.
  - Aplicação fora do ar (falha no health check).
- **P2 - Alertas de Atenção (Notificação em canal de Slack):**
  - Uso de Heap da JVM > 80%.
  - Pico incomum de erros 4xx (pode indicar um bug no cliente ou um ataque).
  - *Circuit Breaker* para um provedor de IA aberto.

3. Gestão de Migrações e Seeding de Dados

3.1. Migração de Schema com Flyway

A evolução do schema do banco de dados será gerenciada exclusivamente pelo Flyway. Todos os scripts DDL (Data Definition Language) serão versionados em

`src/main/resources/db/migration`, garantindo um processo de migração automatizado e repetível.

3.2. Estratégia de Seeding de Dados

O uso de `CommandLineRunner` é limitado. Adotaremos uma abordagem mais robusta:

- **Dados de Referência:** Para dados que devem existir em todos os ambientes (ex: planos de subscrição padrão), usaremos migrações repetíveis do Flyway (`R__*.sql`).
- **Dados de Teste (Ambientes não produtivos):** Utilizaremos um componente Spring ativado por perfil (`@Profile("!prod")`) para inserir dados de teste (usuários, workspaces) apenas em ambientes de desenvolvimento e staging.

4. Fluxos de Gestão de Usuários

As operações de gerenciamento de membros do Workspace serão expostas via API REST.

4.1. Convidar Membro para um Workspace

- **Endpoint:** `POST /api/v1/workspaces/{workspaceId}/invitations`
- **Corpo da Requisição:** `{ "email": "string", "role": "MEMBER" | "ADMIN" }`
- **Lógica do Caso de Uso:**
  1. O serviço de aplicação verifica se o usuário solicitante tem permissão (`OWNER` ou `ADMIN`) no workspace.
  2. Verifica se o e-mail já pertence a um membro para evitar duplicatas.
  3. Cria um registro de convite com um token seguro e data de expiração.
  4. Dispara o envio de um e-mail de convite contendo um link para aceitação.
- **Resposta:** `201 Created`

4.2. Aceitar Convite para um Workspace

- **Endpoint:** `POST /api/v1/invitations/accept`
- **Corpo da Requisição:** `{ "token": "string" }`
- **Lógica do Caso de Uso:**
  1. O usuário deve estar autenticado para aceitar um convite.
  2. O serviço valida o token (existência e data de expiração).
  3. Verifica se o e-mail do usuário autenticado corresponde ao e-mail do convite.

4. Adiciona o usuário à lista de membros do workspace com a função definida no convite.
5. Invalida o token do convite.

- **Resposta:** `200 OK`

4.3. Remover Membro de um Workspace

- **Endpoint:** `DELETE /api/v1/workspaces/{workspaceId}/members/{memberId}`
- **Lógica do Caso de Uso:**
  1. O serviço de aplicação verifica as permissões do solicitante. Um `ADMIN` não pode remover um `OWNER`, e o `OWNER` não pode ser removido se for o único membro com essa função .
  2. Invoca o método `workspace.removeMember(memberId)` no objeto de domínio, que contém as regras de negócio.
  3. Persiste o estado atualizado do agregado `Workspace`.
- **Resposta:** `204 No Content`

Ensino

Esta documentação demonstra que uma arquitetura de software robusta transcende o código e os padrões de design, estendendo-se para a **operacionalidade do sistema**. Tópicos como CI/CD, observabilidade e gestão de dados não são "adicionais", mas sim componentes integrais da arquitetura que garantem que o valor de negócio do software possa ser entregue de forma fiável, segura e contínua. Automatizar o caminho para a produção (CI/CD) e instrumentar o sistema para "perguntar" sobre seu estado (observabilidade) são investimentos que reduzem o risco e aumentam a velocidade de iteração a longo prazo.

Especificação do Modelo de Domínio - AxonAI

1. Objetivo

Este documento descreve o modelo de domínio revisado do AxonAI, incorporando princípios de escalabilidade, consistência e extensibilidade. Ele reflete a evolução a partir do MVP para uma base sólida de produção.

2. Agregados Principais

- **ProjectAggregate**
  - **Campos:**
    - `projectId`
    - `name`
    - `ownerId`
    - `taskIds: List<TaskId>`
    - `createdAt`, `updatedAt`, `version`
  - **Responsabilidade:** Representa o projeto e mantém apenas referências para tarefas.
  - **Métodos principais:**
    - `rename(String newName)`
    - `addTask(TaskId taskId)`
    - `archive(UserId actor)`
  - **Eventos:** `ProjectCreated`, `TaskAddedToProject`, `ProjectArchived`
- **TaskAggregate**
  - **Campos:**
    - `taskId`
    - `projectId`
    - `title`

- `status: TaskStatus`
- `checklistItemIds: List<ChecklistItemId>`
- `mainConversationId`
- `createdAt`, `updatedAt`, `version`

- **Responsabilidade:** Gerencia o ciclo de vida da tarefa e a consistência de seus `ChecklistItems`.
- **Estados permitidos:**
  - BACKLOG → RUNNING | CANCELED
  - RUNNING → DONE | PAUSED | CANCELED
  - PAUSED → RUNNING | CANCELED
  - DONE → RUNNING (reopen)
- **Métodos principais:**
  - `start(UserId actor)`
  - `complete(UserId actor)`
  - `addChecklistItem(String description, UserId actor)`
  - `focusOnChecklistItem(ChecklistItemId id, UserId actor)`
- **Eventos:** `TaskCreated`, `TaskStarted`, `TaskCompleted`, `ChecklistItemAdded`, `ChecklistItemFocused`

- **ChecklistItem**
  - **Tipo:** Entidade (não é mais Value Object)
  - **Campos:**
    - `checklistItemId`
    - `taskId`
    - `description`
    - `status: ChecklistItemStatus`
    - `focusedConversationId`
    - `createdAt`, `updatedAt`
  - **Estados permitidos:**
    - PENDING → FOCUSED | PAUSED | DONE

        - FOCUSED → DONE | PAUSED
        - PAUSED → FOCUSED | PENDING
    - **Métodos principais:**
        - `markFocused(UserId actor)`
        - `markDone(UserId actor)`
        - `pause(UserId actor)`
- **ConversationAggregate**
    - **Campos:**
        - `conversationId`
        - `ownerRef` (TaskId ou ChecklistItemId)
        - `createdAt`, `lastMessageAt`, `messageCount`
    - **Mensagens:** armazenadas separadamente em `ConversationMessage` (append-only), com: `messageId`, `conversationId`, `author`, `content`, `timestamp`.
    - **Benefício:** Escalabilidade e leitura paginada.
- **AIInteraction**
    - **Campos:**
        - `interactionId`
        - `conversationId`
        - `provider`, `model`, `tokensIn`, `tokensOut`, `estimatedCost`
        - `timestamp`, `requestHash`, `responseHash`
    - **Função:** Auditoria e controle de custos de uso da IA.

3. Regras de Negócio e Validação

- **Transições de estado controladas:** Métodos específicos nos agregados lançam `DomainException` se violarem regras de transição de estado.
- **Criação de Tarefas:** Tasks só podem ser criadas em projetos que não estejam arquivados.
- **Invariante de Foco Único (Regra Crítica):**
    - A regra de que

"apenas um `ChecklistItem` pode estar no estado `FOCUSED` por `Task`" é uma invariante protegida pelo `TaskAggregate`.

- A única maneira de alterar o foco é através do método `TaskAggregate::focusOnChecklistItem(checklistItemId, actor)`. Este método é responsável por orquestrar a transição de forma atômica, garantindo a consistência do agregado da seguinte forma:
  1. Verificar se a `Task` está em um estado que permite a operação (ex: `RUNNING`).
  2. Localizar e "desfocar" (transitar para `PENDING` ou `PAUSED`) qualquer `ChecklistItem` que esteja atualmente no estado `FOCUSED` dentro da mesma tarefa.
  3. Mover o `ChecklistItem` alvo para o estado `FOCUSED`.
  4. Emitir um evento de domínio `ChecklistItemFocused` para notificar outras partes do sistema.
- Qualquer outra tentativa de alterar o estado de um `ChecklistItem` para `FOCUSED` diretamente deve ser rejeitada pelo domínio.

4. Glossário da Linguagem Onipresente

Este glossário define os termos centrais que conectam a visão de negócio à implementação do domínio.

- **Modo Foco (Focus Mode):**
  - **Descrição de Negócio:** É a funcionalidade principal do AxonAI, que permite ao usuário iniciar uma conversa com a IA isolada e focada no contexto de um único `ChecklistItem`. Todo o histórico da tarefa pai é herdado, mas a nova interação é específica para a microtarefa.
  - **Implementação no Domínio:** O "Modo Foco" é ativado pelo método `TaskAggregate::focusOnChecklistItem()` e representado pelo estado `ChecklistItemStatus.FOCUSED` na entidade `ChecklistItem`. A conversa associada a este modo é identificada pelo `focusedConversationId`.

5. Eventos de Domínio

Eventos serão publicados de forma assíncrona e idempotente para integração com:

- Kanban externo (Trello, Jira, Linear)

- Notificações
- Métricas de uso de IA
- **Principais eventos:**
  - `ProjectCreated`
  - `TaskCreated`
  - `ChecklistItemFocused`
  - `ConversationMessageAdded`
  - `AiInteractionLogged`

6. Persistência e Concorrência

- **Persistência:** JPA/Hibernate com `@Version` para controle otimista de concorrência.
- **Estrutura:**
  - Conversas: armazenadas em coleção/tabela própria, paginada.
  - Tasks: tabela própria com chave estrangeira para `projectId`.
  - ChecklistItems: tabela própria com chave estrangeira para `taskId`.

7. Estratégia de Leituras e Evolução para CQRS

A arquitetura está preparada para a eventual implementação de CQRS, mas a abordagem inicial priorizará a simplicidade para acelerar a entrega de valor. A estratégia será dividida em fases:

- **Fase 1 (Implementação Inicial):** As consultas de leitura (queries) para visualizações complexas, como o `ProjectBoardView` (Kanban), serão realizadas diretamente contra o modelo de persistência dos agregados. Serão criados DTOs otimizados para essas leituras, mas sem a complexidade de um modelo de dados separado ou de atualização por eventos.
- **Fase 2 (Evolução Acionada por Métricas):** A transição para um modelo completo de CQRS, com projeções de leitura (`Read Models`) dedicadas e atualizadas por eventos de domínio, será considerada uma refatoração estratégica. A decisão de iniciar esta fase será acionada por um dos seguintes **gatilhos baseados em dados de monitoramento**:

- A latência no p95 (percentil 95) para as APIs de leitura de painéis (ex: carregar um projeto) exceder consistentemente **500ms**.
- A complexidade das consultas diretas ao banco de dados começar a impactar negativamente a performance das operações de escrita (comandos).

8. Segurança e Retenção

- **BYOK:** chaves criptografadas via Envelope Encryption + KMS.
- **Logs:** sem chaves em logs.
- **Retenção de conversas:** política configurável com suporte a exclusão (LGPD/GDPR).

9. Benefícios da Revisão

- Evita agregados inchados (Project não carrega milhares de tasks na memória).
- Permite escala horizontal de tasks e conversas.
- Melhora governança (auditoria de IA, controle de transições de estado).
- Abre caminho para monetização futura via `AiInteraction`.

Estratégia de Tratamento de Exceções

1. Filosofia e Objetivos

O tratamento de exceções no AxonAI segue três princípios fundamentais para garantir uma aplicação robusta e operável:

1. **Previsibilidade:** Clientes da API, como o frontend, devem receber uma estrutura de erro consistente e previsível para todas as falhas, permitindo um tratamento de erros padronizado na interface do utilizador.
2. **Segurança:** Nenhuma informação sensível ou detalhes de implementação (como *stack traces*, mensagens de exceções de banco de dados) deve ser exposta ao utilizador final.
3. **Separação de Responsabilidades:** Cada camada da arquitetura tem uma responsabilidade distinta no tratamento de erros, garantindo que detalhes de infraestrutura não vazem para o núcleo da aplicação.

2. Hierarquia de Exceções da Aplicação

A hierarquia de exceções customizadas é desenhada para ser explícita e orientada à intenção, comunicando claramente o que deu errado do ponto de vista do negócio. Todas as exceções controladas da aplicação herdam de uma classe base `AxonAIException`.

- `AxonAIException` (Classe Base): A raiz de todas as exceções controladas.
- **Exceções Baseadas na Intenção:**
  - `NotFoundException`: Lançada quando um recurso específico não pode ser encontrado (ex: projeto, tarefa). Mapeia para **HTTP 404 Not Found**.
  - `ValidationException`: Lançada para erros de validação de input ou violações de regras de negócio que podem ser corrigidas pelo utilizador (ex: título de tarefa em branco). Mapeia para **HTTP 400 Bad Request**.
  - `ForbiddenException`: Lançada quando um utilizador autenticado tenta executar uma ação para a qual não tem permissão. Mapeia para **HTTP 403 Forbidden**.

- `ConflictException`: Lançada quando uma ação não pode ser concluída devido a um conflito com o estado atual do recurso (ex: tentar criar um recurso que já existe com um identificador único). Mapeia para **HTTP 409 Conflict**.

3. Estratégia de Tratamento por Camada

- **Camada de Aplicação (`application.service`):**
  - É a principal fonte das exceções da aplicação (`NotFoundException`, `ForbiddenException`, etc.). Os serviços validam as condições de negócio e de autorização e lançam a exceção apropriada.
- **Adaptadores de Saída (`adapter.out.*`):**
  - **Responsabilidade Crítica:** Esta camada é a fronteira que protege o núcleo da aplicação de exceções específicas da infraestrutura.
  - **Implementação:** Os `Adapters` (ex: `PersistenceAdapter`, `GeminiAdapter`) devem capturar exceções da tecnologia que utilizam (ex: `SQLException`, `DataIntegrityViolationException` do Spring Data, `HttpTimeoutException` do cliente HTTP da IA) e **traduzi-las** para uma exceção da aplicação mais apropriada (ex: `ConflictException` ou uma `InfrastructureException` genérica que resultará num erro 500). Isso impede que o núcleo da aplicação tenha conhecimento sobre `JPA`, `HTTP`, etc.
- **Adaptador de Entrada (`adapter.in.web`):**
  - **Responsabilidade:** Atua como a fronteira final que traduz todas as exceções (tanto as da aplicação quanto as inesperadas) em respostas HTTP padronizadas.
  - **Implementação:** Um `GlobalExceptionHandler` (`@RestControllerAdvice`) centraliza toda a lógica de mapeamento de exceções para respostas HTTP.

4. Formato Padrão da Resposta de Erro da API (Enriquecido)

Toda resposta de erro da API seguirá um formato JSON padronizado e rico em informações, agora incluindo um `traceId` para correlação com os logs.

JSON

```
{
"timestamp": "2025-09-02T18:30:00.123Z",
"status": 404,
"error": "NOT_FOUND",
"message": "O projeto com o ID '123e4567-e89b-12d3-a456-426614174000' não foi encontrado.",
```

```
  "path": "/api/v1/projects/123e4567-e89b-12d3-a456-426614174000",
  "traceId": "a7b3c1d9-e8f0-4a2b-8c7d-6e5f4g3h2i1j"
}
```

- `traceId`: Um identificador único gerado para cada requisição. Este ID será incluído em **todos os logs** gerados durante o processamento daquela requisição, permitindo rastrear o fluxo completo de uma operação e correlacionar um erro específico reportado pelo cliente com os logs detalhados no backend.

5. Mapeamento no `GlobalExceptionHandler`

O `GlobalExceptionHandler` irá mapear a hierarquia de exceções para os códigos de status HTTP apropriados:

- `@ExceptionHandler(NotFoundException.class)` → **HTTP 404 Not Found**
- `@ExceptionHandler(ForbiddenException.class)` → **HTTP 403 Forbidden**
- `@ExceptionHandler(ValidationException.class)` → **HTTP 400 Bad Request**
- `@ExceptionHandler(ConflictException.class)` → **HTTP 409 Conflict**
- `@ExceptionHandler({MethodArgumentNotValidException.class, ConstraintViolationException.class})` → **HTTP 400 Bad Request**
- `@ExceptionHandler(AIError.class)` → Mapeia o código de erro interno do `AIError` para o status HTTP apropriado.
- `@ExceptionHandler(Exception.class)` → **HTTP 500 Internal Server Error** (catch-all para erros inesperados).

6. Estratégia de Logging

A estratégia de logging é fundamental para a depuração e monitorização.

- **Formato Mandatório:** Todos os logs devem ser emitidos em **JSON estruturado**. Isso torna os logs pesquisáveis e fáceis de analisar por ferramentas de observabilidade (Datadog, ELK Stack, etc.). Cada entrada de log **deve** conter o `traceId`.
- **Nível** `WARN`: Para exceções controladas que resultam em respostas `4xx` (erros do cliente). Estas indicam um problema com a requisição do cliente, não uma falha no servidor. O log deve conter a mensagem de erro e o contexto da requisição.

- **Nível** `ERROR` **:** Para exceções inesperadas que resultam em respostas `5xx`. Estas indicam uma falha no servidor. O log deve **sempre** incluir o *stack trace* completo para facilitar a depuração.

Especificação da Camada de Segurança

1. Princípios Fundamentais

A segurança no AxonAI é um pilar fundamental da arquitetura, não uma camada adicional. Guiamo-nos pelos seguintes princípios:

- **Defesa em Profundidade:** A segurança é aplicada em múltiplas camadas. A falha de um único controle de segurança não deve comprometer o sistema como um todo.
- **Princípio do Menor Privilégio:** Cada componente e utilizador do sistema deve ter apenas as permissões estritamente necessárias para realizar as suas funções.
- **Segurança por Design (*Security by Design*):** A segurança é considerada desde o início do ciclo de vida do desenvolvimento, com padrões e práticas seguras integradas em todas as camadas da aplicação.
- ***Zero Trust* (Confiança Zero):** Nenhuma requisição, seja interna ou externa, é confiável por padrão. Toda interação deve ser autenticada e autorizada.

2. Autenticação e Gestão de Sessão

A gestão de sessão será implementada utilizando o padrão de **Refresh Token Rotation** para maximizar a segurança contra o roubo de tokens.

- **Estratégia:**
    1. **Login:** Após uma autenticação bem-sucedida, o servidor gera e retorna dois tokens:
        - **Access Token:** Um JSON Web Token (JWT) de curta duração (15 minutos), assinado com uma chave assimétrica (RS256). Contém as permissões (`claims`) do utilizador.
        - **Refresh Token:** Um token opaco, criptograficamente seguro e de longa duração (7 dias). Este token é armazenado numa tabela `refresh_tokens` no banco de dados, associado ao `userId`, e marcado como ativo.

2. **Acesso a Recursos:** O cliente envia o `Access Token` no cabeçalho `Authorization: Bearer <token>` para aceder a endpoints protegidos.
3. **Renovação de Sessão (Rotação):**
    - Quando o `Access Token` expira, o cliente envia o `Refresh Token` para um endpoint específico (`/api/v1/auth/refresh`).
    - O servidor valida o `Refresh Token` contra o banco de dados.
    - Se válido, o servidor **invalida** o `Refresh Token` antigo e gera um **novo par** de `Access Token` e `Refresh Token` , retornando-os ao cliente.
4. **Detecção de Roubo:** Se o servidor receber um `Refresh Token` que já foi invalidado (reutilizado), isso indica um potencial roubo. O sistema irá invalidar imediatamente **toda a família de tokens** para aquele utilizador, forçando um novo login em todos os dispositivos.

- **Justificativa:** A curta duração do `Access Token` limita a janela de exposição em caso de roubo. A rotação e a detecção de reutilização do `Refresh Token` fornecem um mecanismo robusto para revogar sessões e proteger contra ataques de longo prazo.

3. Autorização

A autorização é a responsabilidade da Camada de Aplicação (`application.service`).

- **Estratégia:** Antes de executar qualquer lógica de negócio, o serviço de aplicação deve:
    1. Carregar o recurso solicitado (ex: `Workspace` , `Project`) a partir da camada de persistência.
    2. Verificar se o `userId` (extraído do JWT pelo `Controller`) tem permissão para aceder ou modificar aquele recurso (ex: é membro do `Workspace`).
    3. Se a verificação falhar, o serviço deve lançar uma `ForbiddenException` , que será traduzida para uma resposta **HTTP 403 Forbidden**.

4. Segurança de Credenciais e Segredos

- **Hashing de Senha:**
    - **Padrão:** O algoritmo **Bcrypt** é mandatório para o hashing de senhas de utilizadores com autenticação local. Nenhuma senha deve ser armazenada em texto plano ou com algoritmos de hash fracos.

- **Segurança das Chaves BYOK (Bring Your Own Key):**
  - **Padrão:** Será implementado o padrão **Envelope Encryption** com um KMS (Key Management Service) dedicado (ex: AWS KMS, Google Cloud KMS).
  - **Fluxo:**
    1. Quando um utilizador submete uma chave de API, o `ApiKeyVault` solicita uma nova Chave de Encriptação de Dados (DEK) ao KMS.
    2. O KMS retorna a DEK em texto plano e a mesma DEK encriptada com uma Chave de Encriptação de Chaves (KEK) mestra.
    3. A chave de API do utilizador é encriptada com a DEK em texto plano.
    4. A chave de API encriptada e a **DEK encriptada** são armazenadas no banco de dados. A DEK em texto plano é descartada da memória.
    5. Para desencriptar, o processo é invertido: a DEK encriptada é enviada ao KMS, que a desencripta usando a KEK mestra, e a DEK em texto plano resultante é usada para desencriptar a chave de API na memória da aplicação.
  - **Justificativa:** A aplicação nunca tem acesso à chave mestra (KEK), e as chaves de dados (DEKs) são armazenadas de forma segura, oferecendo um nível de segurança de padrão empresarial para os segredos dos utilizadores.
- **Segredos da Aplicação:** Todas as credenciais da aplicação (senhas de banco de dados, chaves de assinatura de JWT, credenciais do KMS) devem ser fornecidas à aplicação através de variáveis de ambiente seguras ou um serviço de gestão de segredos da plataforma de implantação (ex: Render, AWS Secrets Manager).

5. Segurança da Aplicação (Essenciais)

- **Validação de Input:** Todas as entradas de dados provenientes de fontes não confiáveis (utilizadores, APIs externas) devem ser rigorosamente validadas na fronteira da aplicação (DTOs, Controllers) para prevenir ataques de injeção (XSS, SQL Injection).
- **Proteção contra SSRF (Server-Side Request Forgery):** As funcionalidades que interagem com URLs fornecidas pelo utilizador devem validar e

restringir rigorosamente os endereços de destino para prevenir que o servidor seja usado como um proxy para atacar sistemas internos ou externos.

- **Cabeçalhos de Segurança HTTP:** A aplicação deve configurar cabeçalhos de segurança HTTP essenciais, como `Content-Security-Policy`, `Strict-Transport-Security` e `X-Content-Type-Options`, para proteger o frontend contra ataques comuns.

Ensino e Análise Final

Esta especificação eleva a segurança do AxonAI de um modelo "suficiente para o MVP" para um modelo de "defesa em profundidade" adequado para produção. A mudança mais significativa é a adoção de padrões que limitam o "raio de explosão" de um incidente de segurança. A rotação de refresh tokens garante que um token roubado tenha uma utilidade limitada no tempo. O Envelope Encryption com KMS garante que uma violação do banco de dados não exponha imediatamente as chaves de API dos utilizadores, pois o atacante ainda precisaria comprometer o KMS. Esta abordagem em camadas é a essência da engenharia de segurança moderna: assumir que falhas podem e vão ocorrer, e construir sistemas que são resilientes a essas falhas.

Especificação de Ports e Adapters

1. Princípios e Estratégia

Esta especificação detalha a implementação da Arquitetura Hexagonal (Ports and Adapters), que cria uma separação rigorosa entre o núcleo da aplicação (domínio e casos de uso) e os detalhes de infraestrutura (banco de dados, API REST, serviços de IA). Os princípios são:

- **Separação de Responsabilidades:** O núcleo da aplicação não deve ter conhecimento de como os dados são apresentados ao utilizador (HTTP/JSON) nem de como são armazenados (PostgreSQL/JPA).
- **Contratos Explícitos (Ports):** A comunicação entre o núcleo e a infraestrutura é definida por interfaces explícitas (Ports). Os `Input Ports` representam os casos de uso, e os `Output Ports` representam os requisitos de infraestrutura do núcleo (ex: "preciso de persistir este objeto").
- **Implementações Intercambiáveis (Adapters):** Os `Adapters` são as implementações concretas dos `Ports`. Eles traduzem os sinais da infraestrutura para chamadas no núcleo da aplicação (`Driving Adapters`) ou implementam os requisitos do núcleo usando uma tecnologia específica (`Driven Adapters`).

2. Estrutura de Pacotes

A estrutura de pacotes é organizada para refletir a arquitetura:

- `com.axonal.domain` : Contém o modelo de domínio puro.
- `com.axonal.application.port.in` : Contém as interfaces dos `Input Ports` (Casos de Uso) e seus `Commands` e `DTOs` de retorno.
- `com.axonal.application.port.out` : Contém as interfaces dos `Output Ports` para infraestrutura externa (ex: `ProjectRepositoryPort` , `AiModelPort` , `ApiKeyVault`).
- `com.axonal.application.service` : Contém as implementações dos Casos de Uso (dos `Input Ports`).
- `com.axonal.adapter.in.web` : Contém o `Driving Adapter` para a API REST (Controllers, DTOs da Web e Mappers).

- `com.axonal.adapter.out.persistence` : Contém o `Driven Adapter` para a persistência (Entidades JPA, Repositórios Spring Data, Mappers de persistência e a implementação do `RepositoryPort`).
- `com.axonal.adapter.out.ai` : Contém o `Driven Adapter` para a interação com LLMs (ex: `GeminiAdapter`).
- `com.axonal.adapter.out.security` : Contém `Driven Adapters` para componentes de segurança (ex: implementação do `ApiKeyVault`).

3. Detalhamento dos Ports (Contratos)

- **Input Ports (Casos de Uso):**
  - Definidos como interfaces em `...application.port.in`. Eles formam a API da camada de aplicação, definindo todas as funcionalidades que o sistema oferece.
  - Exemplos: `RegisterUserUseCase`, `CreateProjectUseCase`, `ProposeTasksForProjectUseCase`.
- **Output Ports (Requisitos de Infraestrutura):**
  - Definidos como interfaces em `...application.port.out`. A camada de aplicação depende destas interfaces, não das suas implementações.
  - **Port de Persistência:** `ProjectRepositoryPort` define métodos que operam sobre objetos de domínio puros, como `save(Project project)` e `findById(ProjectId id): Optional<Project>`.
  - **Port de IA:** `AiModelPort` define o contrato para interagir com um modelo de IA (`execute` e `executeStream`).
  - **Port de Segurança:** `ApiKeyVault` (anteriormente `ApiKeyStore`) define o contrato para armazenar e recuperar chaves de API de forma segura.

4. Detalhamento dos Adapters (Implementações)

- **Driving Adapter (Web - `...adapter.in.web`):**
  - **Componentes:** `@RestController`, `Request/Response DTOs` e `Mappers`.
  - **Responsabilidade:** Mapear requisições HTTP para `Commands` da camada de aplicação, invocar o `UseCase` correspondente e mapear os `DTOs` da

aplicação para `Responses HTTP`. Esta camada é responsável por toda a interação com o protocolo HTTP e a serialização JSON.

- **Driven Adapters (Saída):**
  - **Adaptador de Persistência (`...adapter.out.persistence`):**
    - **Componentes:** `ProjectPersistenceAdapter`, `Entidades JPA`, `Mappers de Persistência` e as interfaces `JpaRepository` do Spring Data.
    - **Responsabilidade:** A classe `ProjectPersistenceAdapter` implementa a interface `ProjectRepositoryPort`. Internamente, ela usa um `Mapper` para converter o objeto de domínio `Project` numa `ProjectEntity` JPA, e depois usa o `ProjectJpaRepository` para persistir a entidade no banco de dados.
  - **Adaptador de IA (`...adapter.out.ai`):**
    - **Componentes:** `GeminiAdapter`, `OpenAIAdapter`.
    - **Responsabilidade:** Implementar a interface `AiModelPort`, traduzindo o `AiRequest` genérico para a API específica do provedor e mapeando a resposta de volta.
  - **Adaptador de Segurança (`...adapter.out.security`):**
    - **Componentes:** `KmsApiKeyVault`.
    - **Responsabilidade:** Implementar a interface `ApiKeyVault`, orquestrando a encriptação e desencriptação das chaves de API com um KMS externo.

5. Fluxo de Exemplo Revisado: "Criar um Projeto"

Este fluxo ilustra a arquitetura com o desacoplamento completo:

1. **Requisição HTTP:** O cliente envia `POST /api/v1/projects` com um corpo JSON correspondente ao `CreateProjectRequest` DTO da camada web.
2. **Controller (`ProjectController`):** Recebe o `CreateProjectRequest`. O Spring Security valida o JWT e a identidade do utilizador é extraída.
3. **Mapeamento (Web):** O `WebMapper` é invocado para converter o `CreateProjectRequest` num `CreateProjectCommand` (o objeto esperado pela camada de aplicação).

4. **Invocação do Caso de Uso:** O controller invoca o método do `CreateProjectUseCase`, passando o `Command` e o `userId`.
5. **Serviço de Aplicação (`CreateProjectService`):**
    - Recebe o `Command`.
    - Executa a lógica de autorização.
    - Cria um novo objeto de domínio `Project` (puro, sem anotações JPA).
    - Invoca o port de persistência: `projectRepositoryPort.save(newProject)`.
    - Mapeia o objeto `Project` retornado para um `ProjectDTO` e o retorna.
6. **Adaptador de Persistência (`ProjectPersistenceAdapter`):**
    - Recebe a chamada do `save(newProject)`.
    - Usa o `PersistenceMapper` para converter o objeto de domínio `Project` numa `ProjectEntity` JPA.
    - Chama `projectJpaRepository.save(projectEntity)` para persistir no banco de dados.
    - Mapeia a entidade salva de volta para um objeto de domínio e o retorna.
7. **Mapeamento (Resposta):** De volta ao `Controller`, o `ProjectDTO` recebido do serviço é mapeado para um `ProjectResponse` DTO pelo `WebMapper`.
8. **Resposta HTTP:** O controller retorna o `ProjectResponse` no corpo da resposta JSON com o status `201 Created`.

Especificação da Camada de Aplicação

1. Princípios e Estratégia

A Camada de Aplicação é o coração da orquestração dos casos de uso de negócio. Os seus princípios fundamentais são:

1. **Orquestração, Não Lógica de Negócio:** Os serviços de aplicação coordenam o fluxo de trabalho. Eles invocam os `Output Ports` para interagir com a infraestrutura e utilizam os objetos de Domínio para executar a lógica de negócio. A lógica de negócio crítica e as regras de estado residem exclusivamente no Modelo de Domínio.
2. **Agnóstica à Entrega e à Persistência:** A camada de aplicação não tem conhecimento de HTTP, JSON, JPA ou SQL. Ela opera com `Commands` e `DTOs` puros e interage com o mundo exterior através das interfaces dos `Ports`, garantindo a sua total independência da infraestrutura.
3. **Implementação dos Input Ports:** Os serviços de aplicação são as implementações concretas dos `Input Ports` (Casos de Uso), definindo as funcionalidades que o sistema oferece e formando a sua API interna.

2. Estrutura de Implementação (Mantida)

A estrutura de pacotes é mantida por sua clareza e excelente separação de responsabilidades:

- `...application.port.in`: Contém as interfaces dos `Input Ports` (Casos de Uso) e seus `Commands` e `DTOs`.
- `...application.service`: Contém as implementações concretas dos casos de uso (anotadas com `@Service`).

3. Fluxo de Trabalho de um Caso de Uso

O fluxo de trabalho de um caso de uso agora reflete a interação com os `Ports`, desacoplando completamente a aplicação da infraestrutura. Usando o caso de uso "Criar um Novo Projeto" como exemplo:

1. **Contrato do Caso de Uso (Input Port):** Uma interface `CreateProjectUseCase` define o contrato, aceitando um `CreateProjectCommand` e retornando um

`ProjectDTO`.

2. **Comando (Command):** A classe `CreateProjectCommand` encapsula os dados de entrada e inclui validações (`jakarta.validation`).
3. **Serviço de Aplicação (`CreateProjectService`):** A implementação concreta (`@Service`) orquestra o fluxo:
    - **Passo 1: Demarcação da Transação:** O método público do serviço é anotado com `@Transactional`. Esta é a fronteira que garante a atomicidade da operação.
    - **Passo 2: Receber o Command:** O serviço recebe o `CreateProjectCommand`.
    - **Passo 3: Carregar Agregados e Validar Permissões:** O serviço utiliza os `Output Ports` (injetados via construtor, ex: `WorkspaceRepositoryPort`) para carregar os agregados de domínio necessários (ex: `workspaceRepositoryPort.findById(...)`). Em seguida, realiza as verificações de autorização. Se a permissão for negada, lança uma `ForbiddenException`.
    - **Passo 4: Executar a Lógica de Negócio:** O serviço invoca os métodos do agregado de domínio (ex: `workspace.createProject(...)`) para aplicar as regras de negócio. O agregado retorna um novo objeto de domínio `Project`.
    - **Passo 5: Persistir o Novo Estado:** O serviço chama o `Output Port` de persistência correspondente (ex: `projectRepositoryPort.save(newProject)`).
    - **Passo 6: Mapear e Retornar o DTO:** O serviço mapeia o objeto de domínio `Project` retornado para um `ProjectDTO` e o retorna.

4. Responsabilidades Fundamentais da Camada de Aplicação

Com a arquitetura limpa, as responsabilidades da Camada de Aplicação são refinadas e focadas:

- **Gerenciamento de Transações:** É a **única** camada que deve conter a anotação `@Transactional`, definindo as fronteiras das unidades de trabalho do sistema.
- **Tratamento de Autorização:** É a responsabilidade fundamental da Camada de Aplicação garantir que um utilizador tenha as permissões necessárias para executar uma ação. Ela orquestra o carregamento dos dados necessários para a decisão de autorização.

- **Orquestração de Casos de Uso:** Coordena a interação entre o domínio e a infraestrutura (através dos `Ports`), garantindo que a lógica de negócio seja executada na ordem correta e dentro de uma transação consistente.

Ensino e Análise Final

A camada de aplicação, numa arquitetura limpa, é como um maestro de uma orquestra. O maestro (`Application Service`) não toca nenhum instrumento (`lógica de negócio`), nem construiu o teatro (`infraestrutura`). Ele lê a partitura (`UseCase`) e coordena os músicos (`Domain Objects`) e a equipa de palco (`Adapters` via `Ports`) para produzir a sinfonia. Esta reestruturação posiciona a Camada de Aplicação exatamente nesse papel: um orquestrador puro. Ao depender apenas de interfaces (`Ports`), ela se torna extremamente fácil de testar. Podemos "simular" (`mock`) os `Ports` para verificar a lógica de orquestração do serviço de forma isolada, sem precisar de um banco de dados ou de qualquer outra infraestrutura real. Esta testabilidade e clareza de propósito são os principais benefícios desta abordagem.

Especificação da Camada de Interação com IA

1. Objetivo Estratégico e Princípios

O objetivo estratégico desta camada é fornecer uma interface segura, resiliente, agnóstica a provedores e observável para a interação com modelos de IA (Inteligência Artificial). Os princípios que guiam esta arquitetura são:

- **Abstração e Desacoplamento:** O núcleo da aplicação deve ser completamente agnóstico a qual provedor de LLM (Large Language Model) está a ser utilizado. A troca de um provedor (ex: Google Gemini por OpenAI GPT) deve ser possível com alterações mínimas, confinadas a esta camada.
- **Segurança por Design:** A proteção das chaves de API do utilizador (BYOK - Bring Your Own Key) e o controle rigoroso sobre o uso e os custos são pilares fundamentais, não funcionalidades adicionais.
- **Resiliência e Fiabilidade:** A interação com serviços externos é inerentemente instável. A arquitetura deve incluir, por padrão, mecanismos de resiliência (Retries, Circuit Breakers) para lidar com falhas transitórias de rede ou do provedor.
- **Governança e Observabilidade:** Cada chamada à IA deve ser auditável, metrificada e registada. O sistema deve ter a capacidade de aplicar políticas complexas de uso e custo para prevenir abusos e gerir a economia do serviço.

2. Arquitetura de Pipeline de IA

A requisição de IA segue um pipeline de componentes especializados, cada um com uma responsabilidade única. Isto substitui o `AiIntegrationService` monolítico por uma arquitetura mais manutenível e extensível.

- **Componentes Principais:**
    1. `AiFacade` : O ponto de entrada único (façade) para a camada de aplicação. Orquestra a chamada através do pipeline.
    2. `PolicyEngine` : Intercetado primeiro, valida se a requisição está em conformidade com as políticas de negócio (limites de uso, permissões

de funcionalidade, controle de custos).

3. `ApiKeyVault`: Recupera e desencripta de forma segura a chave de API necessária para a requisição.
4. `ProviderRouter`: Seleciona o `Adapter` do provedor de IA apropriado com base na configuração do utilizador ou em regras de negócio.
5. `AIModelPort` **(Adapter):** A implementação concreta que traduz o `AIRequest` genérico para a chamada específica da API do provedor (ex: `GeminiAdapter`, `OpenAIAdapter`).

- **Fluxo de Execução:**
  1. Um Caso de Uso da camada de aplicação invoca o `AIFacade` com um `AIRequest`.
  2. O `AIFacade` primeiro chama o `PolicyEngine` para validar a requisição. Se a validação falhar, o processo é interrompido.
  3. O `AIFacade` invoca o `ApiKeyVault` para obter a chave de API desencriptada do utilizador.
  4. O `AIFacade` passa a requisição para o `ProviderRouter`, que retorna o `Adapter` correto.
  5. O `AIFacade` invoca o método `execute` do `Adapter` selecionado, passando a chave e a requisição.
  6. O `Adapter` lida com a lógica de resiliência (Retry/Circuit Breaker) e retorna um `AIResponse` ou lança um `AIError`.
  7. O `AIFacade` regista a utilização e os custos (através de um `UsageMeter`) antes de retornar a resposta ao Caso de Uso.

3. Contratos de Domínio (Ports)

As interfaces que definem a comunicação entre a camada de aplicação e a camada de IA são mantidas e enriquecidas.

- `AIModelPort` **(Interface):** O contrato central que define a interação com um modelo de IA (`execute` e `executeStream`). Cada provedor terá uma implementação.
- `AIRequest` **(Objeto de Comando):** Encapsula todos os detalhes de uma requisição.

- **Campos:** `taskType` (Enum), `input` (multimodal, suportando texto e imagem), `config` (`ModelConfig`), `userContext`, e `tools` (para suportar function calling).

- `AIResponse` **e** `AIError`**:** A estrutura de resposta padronizada e a taxonomia de erros são mantidas para garantir um tratamento de erros consistente e resiliente.

4. Segurança e Governança

- **Armazenamento de Chaves (** `ApiKeyVault` **):**
  - **Estratégia:** Será implementado um cofre de chaves utilizando **Envelope Encryption**. As chaves de API dos utilizadores são encriptadas com uma Chave de Encriptação de Dados (DEK) única. Cada DEK, por sua vez, é encriptada com uma Chave de Encriptação de Chaves (KEK) mestra, gerida por um serviço de KMS (Key Management Service) dedicado (ex: AWS KMS, Google Cloud KMS).
  - **Justificativa:** Esta abordagem garante que os dados em repouso são seguros e que a chave mestra nunca é exposta à aplicação, oferecendo um padrão de segurança de nível empresarial.
- **Controle de Uso (** `PolicyEngine` **):**
  - **Estratégia:** Um motor de regras que permite a definição de políticas de governança complexas. Ele consultará o plano de subscrição do utilizador e o seu uso atual para:
    1. Aplicar limites de taxa (rate limiting).
    2. Verificar o saldo de "unidades de IA" (ver seção 4 da resolução anterior).
    3. Restringir o acesso a modelos ou funcionalidades específicas com base no plano.

5. Operações e Resiliência

- **Resiliência:** A lógica de Retry e Circuit Breaker será implementada dentro de cada `Adapter` utilizando uma biblioteca padrão como Resilience4j. As políticas serão configuráveis e acionadas com base nos `AIError.Code` retornados (ex: tentar novamente em `PROVIDER_UNAVAILABLE`, mas abrir o circuito em falhas repetidas).

- **Observabilidade:**
  - **Logs Estruturados (JSON):** Mandatório para todas as operações, contendo `traceId`, `tenantId`, `provider`, `taskType`, duração e custos. **Segredos nunca serão registados.**
  - **Métricas (Micrometer):** Exportação de métricas essenciais como latência por provedor, contagem de tokens (entrada/saída), taxas de erro e atividade do Circuit Breaker.
  - **Tracing Distribuído (OpenTelemetry):** O `traceId` será propagado desde a requisição HTTP inicial até a chamada final à API externa, permitindo uma depuração completa do ciclo de vida da requisição.

Especificação da Camada de Persistência

1. Princípios e Estratégia

A camada de persistência no AxonAI atua como um **Adaptador de Saída (Driven Adapter)**. A sua única responsabilidade é implementar os `Output Ports` de repositório definidos pela Camada de Aplicação, traduzindo as operações sobre objetos de domínio puros em interações com o banco de dados.

- **Separação de Modelos:** Existe uma separação rigorosa entre os objetos do **Modelo de Domínio** e as **Entidades de Persistência (JPA)**. A camada de persistência opera exclusivamente com Entidades JPA internamente, e expõe apenas objetos de Domínio através dos `Ports`.
- **Implementação de Contratos:** Esta camada implementa as interfaces `Repository Port` (ex: `ProjectRepositoryPort`). Ela é um detalhe de infraestrutura, e o núcleo da aplicação não tem conhecimento sobre JPA, Hibernate ou SQL.
- **Responsabilidade Única:** A responsabilidade da camada de persistência é exclusivamente a **gestão do estado** (salvar, carregar, atualizar, deletar) dos agregados no banco de dados.

2. Estrutura de Implementação

O código da camada de persistência residirá no pacote `com.axonai.adapter.out.persistence`.

- `...persistence.entity`: Contém as classes de entidade JPA (`@Entity`). Estas classes representam a estrutura das tabelas do banco de dados e **não contêm lógica de negócio**.
- `...persistence.repository`: Contém as interfaces que estendem o `JpaRepository` do Spring Data. Elas operam sobre as Entidades JPA (ex: `ProjectJpaRepository extends JpaRepository<ProjectEntity, UUID>`).
- `...persistence.mapper`: Contém as classes de `Mapper` (ex: `ProjectPersistenceMapper`). A sua responsabilidade é a tradução bidirecional entre objetos de Domínio (ex: `Project`) e Entidades JPA (ex: `ProjectEntity`).
- `...persistence.adapter`: Contém as implementações dos `Output Ports` (ex: `ProjectPersistenceAdapter`). Esta é a fachada da camada de persistência.

3. Estratégias Operacionais

- **Gerenciamento de Transações:**
  - **Localização:** A anotação `@Transactional` do Spring permanece aplicada exclusivamente nos métodos públicos dos **Serviços da Camada de Aplicação** (ex: `CreateProjectService`).
  - **Justificativa:** O caso de uso define a unidade de trabalho do negócio. A transação deve abranger todo o caso de uso, que pode envolver múltiplas chamadas a diferentes repositórios. O `Adapter` de persistência executa dentro desta fronteira transacional.
- **Estratégia de Fetching e Prevenção de `LazyInitializationException`:**
  - **Padrão:** Todas as associações (`@OneToMany`, `@ManyToOne`, etc.) nas Entidades JPA são configuradas com `FetchType.LAZY` por padrão.
  - **Regra de Ouro:** A responsabilidade de carregar os dados necessários para um caso de uso é do `Adapter de Persistência`.
  - **Implementação:** O `Serviço de Aplicação` solicita um agregado através do `Repository Port` (ex: `projectRepositoryPort.findProjectWithTasksById(projectId)`). A implementação no

    `ProjectPersistenceAdapter` delega a chamada para um método no `JpaRepository` que utiliza `JOIN FETCH` ou `@EntityGraph` para garantir que o agregado seja carregado completamente dentro da transação. O

    `Mapper` então traduz a `Entity` totalmente inicializada para um objeto de Domínio.
- **Mapeamento de Exceções:**
  - **Estratégia:** O `Adapter de Persistência` é a fronteira que impede o vazamento de exceções de infraestrutura.
  - **Implementação:** Os métodos no `PersistenceAdapter` devem capturar exceções específicas do Spring/JPA (ex: `DataIntegrityViolationException`) e relançá-las como uma exceção da aplicação mais apropriada (ex: `ConflictException`), conforme definido na "Estratégia de Tratamento de Exceções".

4. Estratégia de Testes e Evolução de Schema (Mantida e Reforçada)

A estratégia de testes é crucial para garantir a robustez desta camada.

- **Migração de Schema com Flyway:**
  - **Ferramenta:** Flyway é mandatório para gerir a evolução do schema do banco de dados através de scripts SQL versionados, localizados em `src/main/resources/db/migration`.
- **Estratégia de Testes:**
  1. **Testes de Slice (`@DataJpaTest`):** Focados em testar o **mapeamento** das Entidades JPA e a lógica de **queries customizadas** dos `JpaRepositories`. Utilizam um banco de dados em memória (H2) para velocidade.
  2. **Testes de Integração (Testcontainers):** Focados em testar o `PersistenceAdapter` de ponta a ponta. Estes testes invocam os métodos do `Adapter` (que recebem e retornam objetos de Domínio) e verificam se o estado é corretamente persistido e recuperado. Utilizam Testcontainers para instanciar um container Docker do PostgreSQL real, garantindo que os testes rodem contra um schema idêntico ao de produção.

Ensino e Análise Final

Com esta especificação, a camada de persistência assume seu papel correto na Arquitetura Hexagonal: um "detalhe" de implementação. O núcleo da aplicação simplesmente diz: "persista este objeto de domínio", através de um `Port`. Como isso acontece – se é com JPA, JDBC, ou mesmo escrevendo em ficheiros – é problema do `Adapter`. Esta separação rigorosa, validada por uma estratégia de testes robusta com Testcontainers, garante que o componente mais propenso a mudanças e otimizações (a interação com o banco de dados) possa evoluir de forma independente, sem nunca colocar em risco a integridade da lógica de negócio.

Especificação do Adaptador Web (API REST)

1. Princípios e Estratégia

A camada Web atua como o Adaptador de Entrada (Driving Adapter) principal do sistema. A sua responsabilidade é expor os casos de uso da Camada de Aplicação através de uma interface RESTful, seguindo os seguintes princípios:

1. **Tradução, Não Lógica:** Os Controllers são tradutores. A sua única função é validar, mapear e delegar. Nenhuma lógica de negócio reside nesta camada.
2. **API como Contrato Público Estável:** A estrutura dos endpoints, os corpos de requisição/resposta (DTOs da Web) e os códigos de status HTTP formam o contrato público da nossa API. Este contrato deve ser estável e evoluir de forma controlada para não quebrar clientes existentes.
3. **Stateless:** A API é completamente stateless. Cada requisição contém toda a informação necessária para ser processada, com a autenticação gerida via JWTs no cabeçalho `Authorization`.

2. Estrutura de Implementação

O código da Camada Web residirá no pacote `com.axonal.adapter.in.web`. A estrutura é desenhada para criar uma fronteira clara (Anti-Corruption Layer) entre o mundo externo e a camada de aplicação.

- `...web.controller`: Contém os Controllers (`@RestController`). Cada controller é focado num recurso de domínio.
- `...web.dto.request`: Contém classes que modelam os corpos das requisições (ex: `CreateProjectRequest`). Incluem anotações de validação (`@NotBlank`, `@Email`, etc.).
- `...web.dto.response`: Contém classes que modelam os corpos das respostas (ex: `ProjectResponse`). Estes DTOs definem a estrutura exata dos dados retornados pela API.

- `...web.mapper`: Contém os Mappers (ex: `ProjectWebMapper`) responsáveis pela tradução bidirecional entre os DTOs da Web e os Commands / DTOs da Camada de Aplicação.
- `...web.exception`: Contém o `GlobalExceptionHandler` (`@RestControllerAdvice`), responsável por traduzir exceções em respostas HTTP padronizadas.

3. Convenções da API REST (Mantidas)

Para garantir consistência e previsibilidade, a API seguirá as seguintes convenções padrão da indústria:

- **Versioning:** Todas as rotas terão o prefixo `/api/v1`.
- **Nomenclatura de Recursos:** Substantivos no plural (ex: `/projects`, `/tasks`).
- **Métodos HTTP:** Utilização semântica dos verbos HTTP (`POST`, `GET`, `PUT` / `PATCH`, `DELETE`).
- **Códigos de Status HTTP:** Utilização correta dos códigos de status (`201 Created`, `200 OK`, `204 No Content`, `400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`).

> ## Endpoint Crítico: Ativação do "Modo Foco"
>
> A operação de focar em um item do checklist, central para a aplicação, é exposta da seguinte forma:
>
> - **Endpoint:** `PUT /api/v1/tasks/{taskId}/checklist-items/{checklistItemId}/focus`
> - **Método HTTP:** `PUT`. A escolha se deve à idempotência da operação: chamar a mesma requisição múltiplas vezes resulta no mesmo estado final (aquele item específico estará focado).
> - **Descrição:** Ativa o "Modo Foco" para um `ChecklistItem` específico dentro de uma `Task`. O `TaskAggregate` no domínio garantirá que qualquer outro item previamente focado seja desfocado.
> - **Corpo da Requisição:** Vazio.
> - **Resposta de Sucesso:** `200 OK` - Retorna o DTO (`ChecklistItemResponse`) do item que foi focado, refletindo seu novo estado.
> - **Respostas de Erro:**
>   - `403 Forbidden`: O usuário autenticado não tem permissão para modificar a tarefa.

- `404 Not Found`: A tarefa (`taskId`) ou o item de checklist (`checklistItemId`) não foram encontrados.

4. Exemplo de Fluxo Revisado: Criar um Projeto

Este fluxo ilustra a arquitetura desacoplada em ação:

1. **Requisição:** O cliente envia `POST /api/v1/projects` com um corpo JSON que corresponde à estrutura do DTO `CreateProjectRequest`. O `Access Token` JWT é enviado no cabeçalho `Authorization`.
2. **Controller:** O método no `ProjectController` é ativado. O Spring Security valida o JWT e a identidade do utilizador é extraída. O corpo da requisição é mapeado para o objeto `CreateProjectRequest`. A validação (`@Valid`) é acionada.
3. **Mapeamento para a Aplicação:** O `Controller` invoca o `WebMapper`, que traduz o DTO `CreateProjectRequest` para um `CreateProjectCommand`, o objeto esperado pela camada de aplicação.
4. **Delegação para o Caso de Uso:** O controller invoca o `CreateProjectUseCase` (o Input Port), passando o `CreateProjectCommand` preenchido e o `userId` extraído do Principal.
5. **Resposta da Aplicação:** O caso de uso processa a lógica e retorna um `ProjectDTO` da camada de aplicação.
6. **Mapeamento da Resposta:** O `Controller` recebe o `ProjectDTO` e invoca o `WebMapper` novamente para traduzi-lo num `ProjectResponse`, o DTO que define o contrato de resposta da API.
7. **Resposta HTTP:** O controller coloca o `ProjectResponse` dentro de uma `ResponseEntity` com status `201 Created` e o cabeçalho `Location`, e a retorna ao cliente.

5. Integração com a Camada de Segurança (Mantida)

- **Extração de Identidade:** Os `Controllers` são a única camada responsável por interagir com o `SecurityContext` do Spring para extrair o `userId` do `Principal` injetado.
- **Isolamento:** Nenhum artefacto do Spring Security (como o objeto `Principal`) é passado para a Camada de Aplicação. Apenas o

`userId` (um tipo primitivo ou UUID) é passado para os `Commands`, mantendo a camada de aplicação agnóstica ao framework de segurança.

6. Tratamento Global de Exceções (Mantido)

- **Centralização:** A classe `GlobalExceptionHandler` (`@RestControllerAdvice`) interceptará todas as exceções lançadas pelas camadas inferiores.
- **Tradução:** Ela conterá métodos (`@ExceptionHandler`) para mapear exceções específicas da aplicação para as respostas HTTP padronizadas e seguras, conforme a "Estratégia de Tratamento de Exceções".

Ensino e Análise Final

A reestruturação do Adaptador Web completa a implementação da Arquitetura Hexagonal no backend. A introdução da camada de DTOs e Mappers da Web pode parecer um boilerplate adicional, mas na verdade é a implementação de uma **Camada Anti-Corrupção (Anti-Corruption Layer)**. Esta camada protege o núcleo da sua aplicação das inevitáveis mudanças e da complexidade do mundo exterior (neste caso, o protocolo HTTP). Ela permite que o contrato da sua API evolua de forma independente da lógica de negócio interna, um fator crucial para a manutenibilidade e escalabilidade a longo prazo de qualquer sistema.

Especificação da Arquitetura Frontend

1. Princípios e Estratégia

O frontend do AxonAI será desenvolvido seguindo quatro princípios fundamentais:

1. **Arquitetura Baseada em Componentes:** A UI será decomposta em componentes reutilizáveis, com responsabilidades bem definidas, facilitando a manutenção e a consistência visual.
2. **Segurança de Tipos (*Type Safety*):** O uso de TypeScript é mandatório para garantir a robustez, reduzir bugs em tempo de execução e melhorar a experiência de desenvolvimento através da análise estática.
3. **Estado Reativo e Previsível:** A gestão de estado será claramente dividida entre estado do servidor (dados da API) e estado do cliente (UI), utilizando bibliotecas especializadas para cada um.
4. **Separação de Responsabilidades:** A lógica de UI (componentes), a lógica de negócio do cliente (hooks) e as chamadas de serviço (API) serão mantidas em ficheiros e pastas separadas.

2. Tecnologias e Bibliotecas

A stack tecnológica foi escolhida para maximizar a produtividade do desenvolvedor e a robustez da aplicação.

- **Framework:** React 19 com Vite
- **Linguagem:** TypeScript
- **Gestão de Estado do Servidor:** TanStack Query (React Query)
- **Gestão de Estado Global do Cliente:** Zustand
- **Biblioteca de Componentes:** Shadcn/ui
- **Chamadas HTTP:** Axios
- **Roteamento:** React Router

3. Estratégia de Gestão de Estado

Para garantir a previsibilidade e evitar complexidade, a gestão de estado seguirá uma hierarquia clara:

1. **Estado do Servidor (TanStack Query):** É a fonte da verdade para **qualquer dado que venha da API**. TanStack Query será responsável por fetching, caching, sincronização e atualização de dados do servidor. Deve-se evitar duplicar este estado em qualquer outro gestor de estado.
2. **Estado Global do Cliente (Zustand):** Para estado que **não vem do servidor**, mas precisa ser partilhado por toda a aplicação. Exemplos: estado de autenticação do utilizador (ex: `isAuthenticated`, `userProfile`), tema da UI (claro/escuro), estado de um tutorial interativo.
3. **Estado Local (Component State - `useState` / `useReducer`):** Para estado que é confinado a um único componente ou a uma pequena árvore de componentes que podem partilhá-lo via *props*. Exemplos: estado de um formulário antes de ser submetido, se um modal está aberto ou fechado, o valor de um campo de input.

4. Estrutura de Pastas (Mantida)

A estrutura de pastas organizada é fundamental para a escalabilidade.

```
/src
/api          # Módulos de chamada à API, organizados por recurso
/components     # Componentes de UI reutilizáveis (Shadcn/ui e customizados)
/features       # Módulos de funcionalidades (ex: autenticação, projeto)
/hooks          # Hooks customizados e reutilizáveis
/lib          # Configurações de bibliotecas (axios.ts, etc.)
/pages          # Componentes que representam as páginas da aplicação
/providers      # Provedores de contexto da aplicação (React Query, Router)
/store          # Lojas do Zustand para estado global
/types          # Definições de tipos TypeScript, geradas a partir da API
```

5. Sincronização de Tipos com a API (Type Generation)

Para garantir a consistência de tipos entre o frontend e o backend, os tipos do TypeScript serão gerados automaticamente a partir da especificação OpenAPI 3 do backend.

- **Ferramenta:** `openapi-typescript` ou similar.
- **Fluxo de Trabalho:**
    1. O backend expõe a especificação `openapi.json` através do `springdoc-openapi`.

2. Um script no `package.json` do frontend (`npm run generate-api-types`) irá buscar esta especificação e gerar um ficheiro `api-types.ts` no diretório `/src/types`.
3. Este script será executado durante o processo de build e pode ser executado manualmente sempre que o contrato da API for alterado, garantindo que o frontend esteja sempre sincronizado com a API.

6. Estratégia de Testes

A aplicação terá uma cobertura de testes robusta para garantir a fiabilidade e facilitar a refatoração.

- **Ferramentas:**
  - **Test Runner:** Vitest.
  - **Testes de Componentes:** React Testing Library.
  - **Mocking de API:** Mock Service Worker (MSW).
  - **Testes End-to-End (E2E):** Playwright.
- **Níveis de Teste:**
  1. **Testes de Integração (Foco Principal):** Testaremos componentes ou páginas inteiras renderizadas no DOM virtual. As interações do utilizador serão simuladas e as chamadas à API serão interceptadas e "mockadas" com o MSW. O objetivo é testar o comportamento da aplicação da perspetiva do utilizador, sem depender de um backend real.
  2. **Testes Unitários:** Para lógica de negócio complexa isolada em hooks ou funções utilitárias puras.
  3. **Testes End-to-End (E2E):** Um conjunto pequeno e crítico de testes que verificam os fluxos de trabalho mais importantes (ex: login completo, criação de um projeto, proposta de tarefas pela IA) contra um ambiente de *staging* com um backend real.

7. Qualidade de Código e Convenções

Ferramentas automatizadas serão usadas para manter a qualidade e a consistência do código.

- **Linting:** ESLint com plugins para React, TypeScript e acessibilidade (a11y) para detetar problemas de código estaticamente.

- **Formatação:** Prettier para garantir um estilo de código consistente em toda a base de código.
- **Git Hooks:** Husky e `lint-staged` serão configurados para executar o ESLint e o Prettier automaticamente antes de cada `commit`, prevenindo que código com problemas de qualidade seja enviado para o repositório.
- **Convenções de Commit:** A equipa adotará o padrão **Conventional Commits** para as mensagens de `commit` (`feat:`, `fix:`, `chore:`, etc.). Isto permite a geração automática de `CHANGELOGs` e facilita a compreensão do histórico do projeto.

Ensino e Análise Final

Uma arquitetura de frontend moderna e escalável é um ecossistema, não apenas uma biblioteca. Escolher React é o primeiro passo, mas o que garante o sucesso do projeto a longo prazo são as **regras e ferramentas** que governam como o código é escrito, testado e mantido. Ao definir estratégias claras para gestão de estado, testes automatizados e qualidade de código, criamos "guardrails" que permitem aos desenvolvedores construir novas funcionalidades com velocidade e confiança. A automação da geração de tipos a partir da especificação OpenAPI é a ponte que solidifica a colaboração entre frontend e backend, eliminando uma classe inteira de bugs de integração e garantindo que ambas as equipas falem a mesma "linguagem".