**Chronos**

**Documento do Produto**

Sumário

[1 - Visão geral do produto: 3](#_Toc199063845)

[1.1 - Problema do cliente: 3](#_Toc199063846)

[1.2 - Necessidades: 3](#_Toc199063847)

[1.3 - Objetivo do produto: 3](#_Toc199063848)

[2 - Arquitetura Global: 4](#_Toc199063849)

[2.1 - Aplicações: 4](#_Toc199063850)

[2.2 - Serviços externos: 4](#_Toc199063851)

[2.3 - Design de Sistema: 4](#_Toc199063852)

[2.4 - Ferramentas para desenvolvimento: 5](#_Toc199063853)

[3 - Arquitetura de Aplicação: 5](#_Toc199063854)

[3.1 - Camadas: 6](#_Toc199063855)

[3.2 - Camada de API Rest(*rest*): 7](#_Toc199063856)

[3.2.1 - Controller: 7](#_Toc199063857)

[3.2.2 - Http: 7](#_Toc199063858)

[3.3 - Camada de RPC (rpc): 7](#_Toc199063859)

[3.3.1 - Action: 7](#_Toc199063860)

[3.3.2 - Call: 7](#_Toc199063861)

[3.4 - Camada de Fila (*queue*): 8](#_Toc199063862)

[3.4.1 - Job: 8](#_Toc199063863)

[3.4.2 - Amqp: 8](#_Toc199063864)

[3.5 - Camada de Provision (*prov*): 8](#_Toc199063865)

[3.5.1 - Provider: 8](#_Toc199063866)

[3.6 - Camada de Banco de Dados (*db*): 8](#_Toc199063867)

[3.7 - Camada de Interface de Usuário (ui): 9](#_Toc199063868)

[3.7.1 - Widget: 9](#_Toc199063869)

[3.7.2 - Notificações: 9](#_Toc199063870)

[3.8 - Portas e Adaptadores: 10](#_Toc199063871)

[4 - Arquitetura Web: 10](#_Toc199063872)

[4.1 - Processo de Deploy: 11](#_Toc199063873)

[4.2 - Desenvolvimento: 11](#_Toc199063874)

[4.2.1 - Tecnologias e bibliotecas: 11](#_Toc199063875)

# 1 - Visão geral do produto:

Chronos é uma aplicação web que simplifica o gerenciamento de ponto online, oferecendo funcionalidades completas para controle de jornada de trabalho. Permite o registro de ponto online, cálculo automático de horas, gestão de ausências e geração de relatórios detalhados. A interface intuitiva do Chronos facilita o acompanhamento da jornada de trabalho.

## 1.1 - Problema do cliente:

* Controle manual de ponto, sujeito a erros e fraudes.
* Dificuldade no cálculo preciso de horas trabalhadas e extras.
* Falta de visibilidade sobre a jornada de trabalho dos funcionários.
* Processos burocráticos para gestão de ausências.
* Dificuldade na geração de relatórios para análise de dados.
* Não conformidade com a legislação trabalhista.

## 1.2 - Necessidades:

Automatizar o controle de ponto para aumentar a precisão e reduzir erros. Obter cálculos precisos e automáticos de horas trabalhadas. Melhorar a visibilidade da jornada de trabalho dos funcionários. Simplifica a gestão de ausências com fluxos de aprovação. Gerar relalatórios personalizados para análise de dados. Garantir a conformidade com a legislação trabalhista.

## 1.3 - Objetivo do produto:

* Desenvolver uma aplicação web intuitiva e eficiente para o gerenciamento de ponto online.
* Permitir o registro preciso de ponto com diferentes opções.
* Oferecer cálculos automáticos de horas trabalhadas e extras.
* Simplificar a gestão de ausências com fluxos de aprovação.
* Fornecer relatórios detalhados.
* Garantir a segurança dos dados e a conformidade com a legislação.

Permitir o acesso remoto para funcionários e gestores.

# 2 - Arquitetura Global:

O Chronos é estruturado como um projeto multi-repo, usando dois principais repositórios, o front end e o back end. Essa abordagem permite maior integração, organização e reutilização de código.

As aplicações são os projetos finais e executáveis, como um site em Next.js, uma API em Java utilizando serviços. Utilizando arquitetura cliente e servidor, sendo que o cliente (front-end) faz uma requisição para o servidor (back-end), o servidor irá receber e processar a requisição, retornando assim uma resposta para o cliente, logo em seguida será mostrado para o cliente de alguma forma.

## 2.1 - Aplicações:

Front-End: Aplicação web principal do Chronos, onde os usuários interagem via navegador. O principal framework utilizado é o Next.js em TypeScript.

Back-End: Aplicação back, que se resume em uma aplicação consumida pelo front end para a recuperação de dados e alteração de dados do sistema. O principal framework utilizado é o SpringBoot em Java.

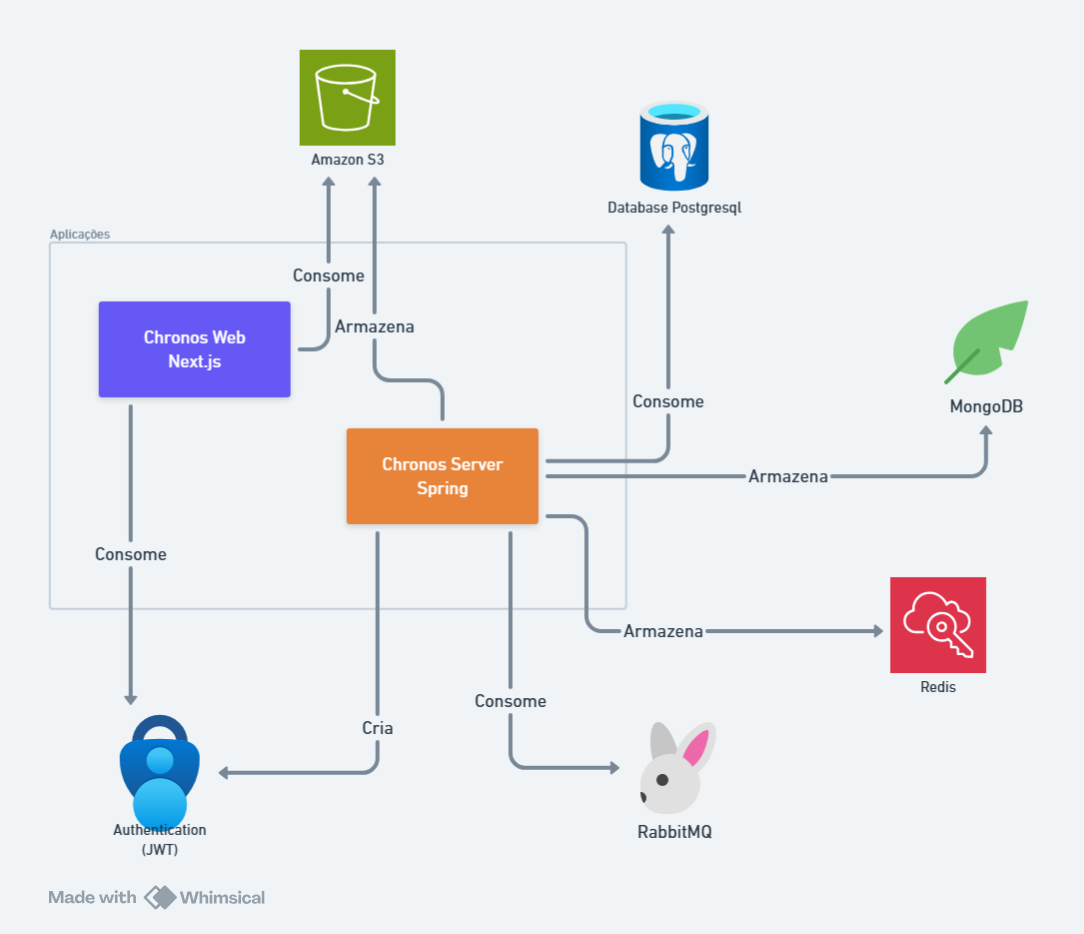
## 2.2 - Serviços externos:

Serviços externos são sistemas, APIs ou plataformas de terceiros com os quais alguma aplicação a Chronos se comunica para realizar alguma funcionalidade que ela mesma não faz sozinha e que não está no repositório principal.

AWS: Na AWS, o S3 é um serviço de armazenamento de objetos da AWS altamente escalável, durável, seguro e de alta performance, para armazenar principalmente imagens e PDFs.

## 2.3 - Design de Sistema:

System Design é o processo de planejar e arquitetar a estrutura de um sistema de software — ou seja, como diferentes partes de um sistema vão funcionar juntas para resolver um problema real de forma eficiente, escalável e confiável. A seguir segue o diagrama do System Design atual do Chronos.



## 2.4 - Ferramentas para desenvolvimento:

Ferramentas de desenvolvimento são recursos — como programas, bibliotecas ou serviços — que ajudam desenvolvedores a criar, depurar, manter e melhorar softwares com mais eficiência. As ferramentas utilizadas para ajudar no desenvolvimento das aplicações e pacotes do Chronos são:

* Editor de código: [**Visual Studio Code**](https://code.visualstudio.com/)
* Controle de versionamento: [**Git**](https://git-scm.com/)

# 3 - Arquitetura de Aplicação:

A arquitetura de uma aplicação Chronos segue os princípios da arquitetura limpa, que consiste em dividir a aplicação em camadas, de forma a proteger as regras negócio de serviços e ferramentas externas.

Uma vez que todas as regras de negócio do Chronos estão concentradas no pacote core, as aplicações servem como uma casca para que usuários e serviços possam interagir com essas regras, garantindo que elas sejam o coração do Chronos e que tudo ao redor seja facilmente substituível. Logo, todas as aplicações dependem do core, mas o contrário nunca pode acontecer.

## 3.1 - Camadas:

Cada aplicação é dividida em camadas, que é uma parte da estrutura de um sistema que tem uma responsabilidade bem definida e que consegue comunica com outras camadas de forma organizada e padronizada. Cada camada é especializada em uma tarefa e se comunica com as outras de forma organizada e padronizada. A quantidade e o tipo de camadas variam conforme a complexidade e as necessidades da aplicação.

Uma camada pode conter elementos, como *gateways*, *handlers* e *protocol*, que são definidos por interfaces dentro do domínio.

*Handler*: é uma função ou classe responsável por receber uma requisição, processá-la e retornar uma resposta. É como um operador que executa uma tarefa com base em uma solicitação.

*Protocol*: é um objeto que define como o *handler* se comunica, oferecendo métodos e estruturas que ajudam o *handler* a entender a requisição e formatar a resposta da forma correta — tudo de acordo com o contexto da camada onde estão inseridos.

Por exemplo: Na camada de API REST, o *handler* é chamado de *controller*, e o protocolo usado é o HTTP. Ambos são geralmente definidos por interfaces para garantir desacoplamento e facilitar testes e manutenções.

*Gateway*: é uma função ou classe responsável por intermediar a comunicação com serviços externos. Normalmente, utiliza um objeto chamado *client* para realizar as requisições.

Por exemplo, um *gateway* que precisa realizar requisições HTTP pode usar a biblioteca [axios](https://axios-http.com/ptbr/docs/intro) como *client* para enviar chamadas à API. Já um *gateway* de banco de dados pode utilizar o ORM [prisma](https://www.prisma.io/) como *client* para executar consultas no banco.

### 3.2 - Camada de API Rest(*rest*):

A camada rest é responsável por receber requisições web via protocolo HTTP. Ela atua como a porta de entrada da aplicação, permitindo que usuários (ou outros sistemas) interajam com o sistema.

### 3.2.1 - Controller:

É o handler que recebe e trata de requisições HTTP.

### 3.2.2 - Http:

É o protocol que fornece acesso aos dados da requisição (headers, body, query, etc.) e métodos para retornar a resposta adequada (status code, JSON, etc.), ou ainda redirecionar o usuário para outra rota ou alterar *cookies*.

## 3.3 - Camada de RPC (rpc):

A camada rpc é responsável por lidar com chamadas de funções que são executadas em um ambiente diferente daquele em que foram invocadas — geralmente o servidor. No contexto do *Next.js*, por exemplo, há funções que rodam no cliente e outras que rodam exclusivamente no servidor. Essas últimas se comportam como uma API RPC, onde o cliente chama diretamente uma função que é executada do outro lado (no *server*).

### 3.3.1 - Action:

Action é o handler responsável por processar essa função remota. Assim como qualquer função comum, ela pode ou não receber parâmetros e pode ou não retornar algo.

### 3.3.2 - Call:

*Call* é o protocol que fornece à *action* acesso ao contexto do servidor, incluindo leitura e escrita de *cookies*, obtenção de dados do usuário autenticado, redirecionamentos e outras interações típicas do lado *server*.

## 3.4 - Camada de Fila (*queue*):

A camada *queue* é responsável por todo o processamento assíncrono da aplicação, como execução de *jobs* em *background*, *cron jobs*, *webhooks*, filas de mensagens e fluxos automatizados (*workflows*).

### 3.4.1 - Job:

*Job* é o *handler* responsável em processar as mensagens recebidas pela fila. Um *job* nem sempre precisa retornar algo, mas pode retornar dependendo da necessidade ou da ferramenta utilizada.

### 3.4.2 - Amqp:

*Amqp* é o *protocol* que fornece acesso aos dados do *job* e métodos para controlar o fluxo de execução, como aplicar *delay*, *retry*, falha intencional, etc. Também pode publicar eventos de domínio, embora essa funcionalidade ainda esteja sendo avaliada para manter ou substituir.

## 3.5 - Camada de Provision (*prov*):

A camada de *prov* em prover recursos que dependem necessariamente de um serviço ou uma biblioteca externa para realizar uma funcionalidade específica sem necessariamente com o protocolo HTTP, como envio de e-mail, armazenar/recuperar arquivos estáticos em *storage*, processar pagamentos, lidar com arquivos PDF, CSV etc. Portanto, essa *camada* só possui *gateways*.

### 3.5.1 - Provider:

Um provider é um gateway que encapsula a biblioteca ou SDK responsável por prover determinada funcionalidade externa.

Exemplos: StorageProvider, PaymentProvider, PdfProvider, EmailProvider.

## 3.6 - Camada de Banco de Dados (*db*):

A camada de database db cuida da persistência de dados. Ela não possui *handlers*, mas sim *gateways*, chamados aqui de *repositories*.

*Repository*: é responsável por salvar, buscar e atualizar dados relacionados a uma única entidade ou objeto de valor, mantendo as operações de banco *desacopladas* do restante do sistema.

## 3.7 - Camada de Interface de Usuário (ui):

A camada ui é a responsável por fornecer a interface gráfica e a lógica de interação com o usuário, seja em uma página web ou em uma tela mobile. Ela é diferente das demais porque não possui *handlers* nem *protocols*, mas sim *widgets* — blocos reutilizáveis que combinam visual com lógica de interface.

### 3.7.1 - Widget:

Um widget é composto por:

* View: a parte visual do componente, sendo um template html ou [jsx](https://react.dev/learn/writing-markup-with-jsx).
* Lógica: uma função ou classe que controla o estado e comportamento da interface. Em projetos com React, convenciona-se usar um *hook* (useAlgo) para representar essa lógica.

Exemplo: o *widget LoginPage* poderia ser composto por *LoginPageView* (interface) e useLoginPage (lógica).

Esse padrão segue a arquitetura [MVVM (Model-View-ViewModel)](https://learn.microsoft.com/pt-br/dotnet/architecture/maui/mvvm), muito comum em apps modernos.

### 3.7.2 - Notificações:

É enviada uma notificação para o colaborador quando:

 Esquece-se de bater o ponto no dia.

 Quando uma das suas solicitações é recusada ou aprovada pelo gestor.

É enviada uma notificação para o gestor quando:

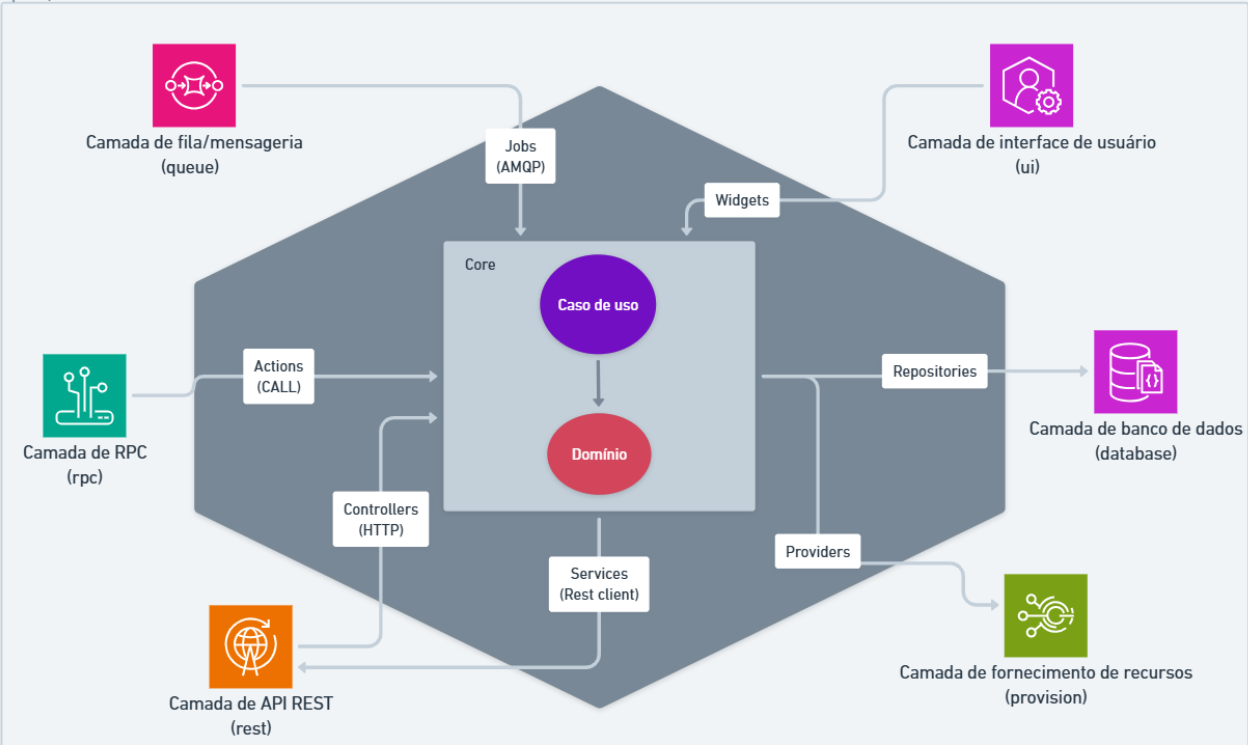
 Quando um dos colaboradores do mesmo setor do gestor abre uma solicitação.

## 3.8 - Portas e Adaptadores:

A Chronos segue os princípios da arquitetura hexagonal, o que significa que a lógica de negócio da aplicação (núcleo) está totalmente isolada dos detalhes de infraestrutura. O núcleo não sabe como os dados são armazenados, exibidos ou enviados — ele só sabe o que precisa ser feito.

Por isso, toda comunicação entre camadas acontece por meio de interfaces (portas). Os adaptadores (implementações) são conectados a essas interfaces para permitir que o sistema funcione com diferentes tecnologias.

Importante: as camadas podem se comunicar livremente desde que isso ocorra por meio de interfaces. Não há uma hierarquia rígida entre as camadas (exceto pelo core, que é o centro do domínio e nunca deve depender de nenhuma outra camada).



# 4 - Arquitetura Web:

É a aplicação que permite aos usuários acessarem as funcionalidades do *Chronos Core* por meio de um navegador web. Além disso, ela também processa requisições HTTP de outros sistemas ou aplicações via padrão REST, bem como executa tarefas assíncronas por meio de filas.

## 4.1 - Processo de Deploy:

A hospedagem da aplicação web ocorre no [Google Cloud Plataform (GCP)](https://console.cloud.google.com/) utilizando o serviço [Cloud Run](https://cloud.google.com/run), por meio do qual o *build* do projeto *Next.js* é executado dentro de um *container Docker*.

## 4.2 - Desenvolvimento:

### 4.2.1 - Tecnologias e bibliotecas:

#### BackEnd:

* [Java Spring](https://spring.io) – *Framework* robusto para construção de APIs REST com injeção de dependência, segurança e suporte a testes.
* [Docker](https://www.docker.com) – Plataforma para empacotar, distribuir e executar aplicações em contêineres isolados.
* [PostgreSQL](https://www.postgresql.org) – Banco de dados relacional poderoso e open-source.
* [MongoDB](https://www.mongodb.com) – Banco de dados NoSQL orientado a documentos, ideal para dados flexíveis.
* [Redis](https://www.mongodb.com) – Banco de dados em memória utilizado para cache e filas rápidas.
* [RabbitMQ](https://www.rabbitmq.com) – *Broker* de mensagens para comunicação assíncrona entre serviços.
* [Lombok](https://projectlombok.org) – Biblioteca Java que reduz o *boilerplate* ao gerar automaticamente *getters*, setters e construtores.
* [Java JWT](https://github.com/auth0/java-jwt) – Biblioteca para geração e validação de *tokens* JWT, usada em autenticação segura.
* [JasperReports](https://community.jaspersoft.com/download-jaspersoft/community-edition/) – Ferramenta para geração de relatórios em PDF ou outros formatos diretamente do *backend*.

#### FrontEnd:

* [TypeScript](https://www.typescriptlang.org) – *Superset* do JavaScript com tipagem estática, que ajuda a prevenir erros e melhorar a produtividade.
* [React](https://react.dev) – Biblioteca para criação de interfaces de usuário baseadas em componentes.
* [Next.js](https://nextjs.org) – *Framework full-stack* para React com suporte a SSR, SSG, rotas automáticas e API *Routes*.
* [TailwindCSS](https://tailwindcss.com) – *Framework* de CSS utilitário que permite estilizar com classes diretamente no HTML/JSX.
* [HeroUI](https://www.heroui.com) – Conjunto de componentes visuais (UI) estilizados com TailwindCSS.
* [Next Safe Action](https://next-safe-action.dev) – Biblioteca para criar actions seguras no servidor, com validação de input e tratamento automático de erros.
* [Use Hooks TS](https://usehooks-ts.com) – Coleção de *hooks* customizados em TypeScript para reutilização de lógica em componentes React.
* [Date-fns](https://date-fns.org) – Biblioteca leve para manipulação e formatação de datas em *JavaScript/TypeScript.*
* [Framer Motion](https://motion.dev) – Biblioteca para animações declarativas em aplicações React.
* [Lucide](https://lucide.dev) – Conjunto de ícones *open-source* baseado em SVG, ideal para interfaces modernas.
* [Nuqs](https://nuqs.47ng.com) – Biblioteca para sincronizar *query params* da URL com o estado da aplicação em *Next.js.*
* [React Hook Form](https://react-hook-form.com) – Biblioteca para gerenciamento de formulários com foco em performance e validação.
* [SWR](https://swr.vercel.app/pt-BR) – Biblioteca da Vercel para *fetching* de dados com cache e revalidação automática.
* [Zustand](https://zustand-demo.pmnd.rs) – Gerenciador de estado simples e eficiente para React.
* [Zod](https://zod.dev) – Biblioteca de validação de *schemas*, com integração nativa com TypeScript.
* [Biome.js](https://biomejs.dev/pt-br/) – Ferramenta all-in-one para lint, formatter e linter de código JavaScript/TypeScript.