Licenciatura em Engenharia de Telecomunicações e Informática

Sistemas Distribuídos

Documento Arquitetural

Filipe Parafita (1231283)

André Silva (1211624)

José Perfeito (1231081)

11/01/26

Índice

Introdução

Este documento tem o objetivo de apresentar a arquitetura do sistema HAP (Health Appointment Platform). O sistema adota uma arquitetura de microserviços, organizada por domínio, com isolamento de dados e comunicação controlada entre serviços.

Este documento descreve a arquitetura atual do sistema e as principais decisões técnicas adotadas, servindo como referência para compreensão, avaliação e defesa da solução.

C4/C4+1

Nível 1

O Diagrama de Contexto (Nível 1) apresenta uma visão de alto nível do Sistema da Clínica Hospitalar. O objetivo deste diagrama é definir as fronteiras do sistema, identificando quem interage com ele (atores humanos) e quais as principais responsabilidades funcionais da plataforma, abstraindo-se de detalhes técnicos ou de implementação.

Nesta visão, o sistema é tratado como uma "caixa negra", focando-se no valor que entrega aos seus utilizadores.

Atores Externos

O diagrama identifica quatro tipos de atores humanos que interagem com o sistema, cada um com responsabilidades e permissões distintas:

- Utilizador Anónimo:
  Representa um visitante sem autenticação. A sua interação está limitada ao processo de registo inicial, que dá início à criação de uma identidade no sistema e ao perfil de paciente.
- Paciente:
  Utilizador autenticado que utiliza a plataforma para gerir as suas consultas. Pode pesquisar médicos, agendar e cancelar consultas, bem como consultar informação relacionada com o seu histórico clínico e agendamentos.
- Médico:
  Profissional de saúde que utiliza o sistema no contexto da prestação de cuidados. A sua principal interação é o registo de informação clínica associada às consultas realizadas, como diagnósticos, prescrições e notas médicas.
- Administrador:
  Responsável pela gestão operacional do sistema. Tem permissões alargadas para criar e gerir perfis de médicos e pacientes, bem como para aceder a informação agregada e relatórios estatísticos que apoiam a tomada de decisão.

Sistema em Estudo

- Sistema da Clínica Hospitalar
  Corresponde à plataforma de software desenvolvida no âmbito do projeto. Este sistema centraliza toda a lógica de negócio relacionada com autenticação,

agendamento de consultas, gestão de registos clínicos e disponibilização de informação aos diferentes perfis de utilizador.

No diagrama, o sistema é tratado como uma “caixa negra”, sendo relevantes apenas as suas responsabilidades externas e não a sua decomposição interna.

Interações Principais

As ligações representadas no diagrama correspondem aos principais fluxos de interação entre os atores e o sistema:

1. Registo de utilizadores – o utilizador anónimo inicia o processo de criação de conta.
2. Gestão de consultas – o paciente interage com o sistema para criar, consultar ou cancelar consultas.
3. Registo clínico – o médico introduz informação clínica associada às consultas realizadas.
4. Gestão e análise – o administrador gere dados mestres e consulta informação estatística do sistema.

Estas interações representam os fluxos de valor essenciais suportados pela plataforma.

Relação com a Arquitetura Interna

Embora o Diagrama de Contexto apresente uma visão simplificada, as interações representadas correspondem, internamente, a fluxos distribuídos mais complexos. Operações como o agendamento de consultas ou o registo clínico ativam mecanismos internos baseados em microsserviços, comunicação por eventos, Sagas e CQRS, garantindo escalabilidade, consistência eventual e tolerância a falhas.

Este diagrama serve, assim, como ponto de entrada conceptual para a compreensão do sistema, sendo complementado nos níveis seguintes do modelo C4 pelos diagramas de containers, componentes e código.

Nível 2

O Diagrama de Containers (C4 – Nível 2) detalha a decomposição interna do sistema em unidades de execução independentes, evidenciando uma arquitetura baseada em microsserviços orientada a eventos. Cada container representa um serviço executável ou um componente de infraestrutura essencial ao funcionamento da plataforma.

A arquitetura foi desenhada com foco em desacoplamento, escalabilidade horizontal e tolerância a falhas, recorrendo a padrões como CQRS e Sagas coreografadas. A comunicação segue um modelo híbrido, combinando chamadas síncronas para interação externa com comunicação assíncrona para coordenação interna entre serviços.

Componentes de Entrada e Infraestrutura

- API Gateway:
  O API Gateway funciona como o ponto único de entrada no sistema. É responsável por encaminhar pedidos externos para os microsserviços adequados, aplicar regras de segurança e garantir observabilidade transversal. Neste nível, o gateway também assume a geração e propagação de identificadores de rastreio (trace-id), permitindo correlacionar pedidos ao longo de múltiplos serviços.
- Identity Service:
  O serviço de identidade é responsável pela autenticação e autorização dos utilizadores. Centraliza a emissão e validação de tokens JWT, permitindo que os restantes microsserviços se foquem exclusivamente na lógica de negócio. A sua presença como container isolado reduz o acoplamento e reforça a segurança global do sistema.

- Message Broker:
  O broker de mensagens constitui o elemento central da comunicação assíncrona. É através deste componente que os serviços publicam e consomem eventos de domínio, permitindo coordenação distribuída sem dependências diretas entre produtores e consumidores.

Microsserviços de Negócio

Os microsserviços representam os principais domínios funcionais do sistema. Cada serviço é autónomo, possui responsabilidades bem definidas e mantém a sua própria persistência, seguindo o princípio de database-per-service.

- Scheduling Service:
  O serviço de agendamento é o núcleo funcional da plataforma. Gere o ciclo de vida das consultas e coordena fluxos distribuídos através de eventos. Este serviço aplica CQRS de forma explícita, separando operações de escrita e leitura, e recorre a Event Sourcing para garantir rastreabilidade e consistência nas transações distribuídas.
- Patient Service e Physician Service:
  Estes serviços gerem os domínios de paciente e médico, respetivamente. Atuam como participantes em Sagas, reagindo a eventos publicados pelo serviço de agendamento e emitindo respostas de validação. A interação com o restante sistema é feita maioritariamente de forma assíncrona.
- Clinical Records Service:
  O serviço de registos clínicos é responsável pela gestão do prontuário médico. Reage a eventos relacionados com consultas para permitir a criação e atualização de informação clínica, mantendo-se desacoplado do fluxo de agendamento.

Persistência de Dados

A arquitetura adota uma estratégia de persistência segregada, alinhada com os padrões CQRS e Event Sourcing.

- Event Store:
  Utilizado pelo serviço de agendamento para a vertente de escrita. Em vez de armazenar apenas o estado atual, o sistema persiste uma sequência imutável de eventos, permitindo reconstruir o estado e garantir auditoria completa.
- Read Models:
  Para as operações de leitura, são utilizados modelos de dados separados, otimizados para consultas e relatórios. Estes modelos são atualizados de forma assíncrona através do consumo de eventos publicados no broker.
- Bases de Dados de Domínio:
  Cada um dos restantes serviços mantém a sua própria base de dados isolada, assegurando independência e evitando acessos diretos entre domínios.

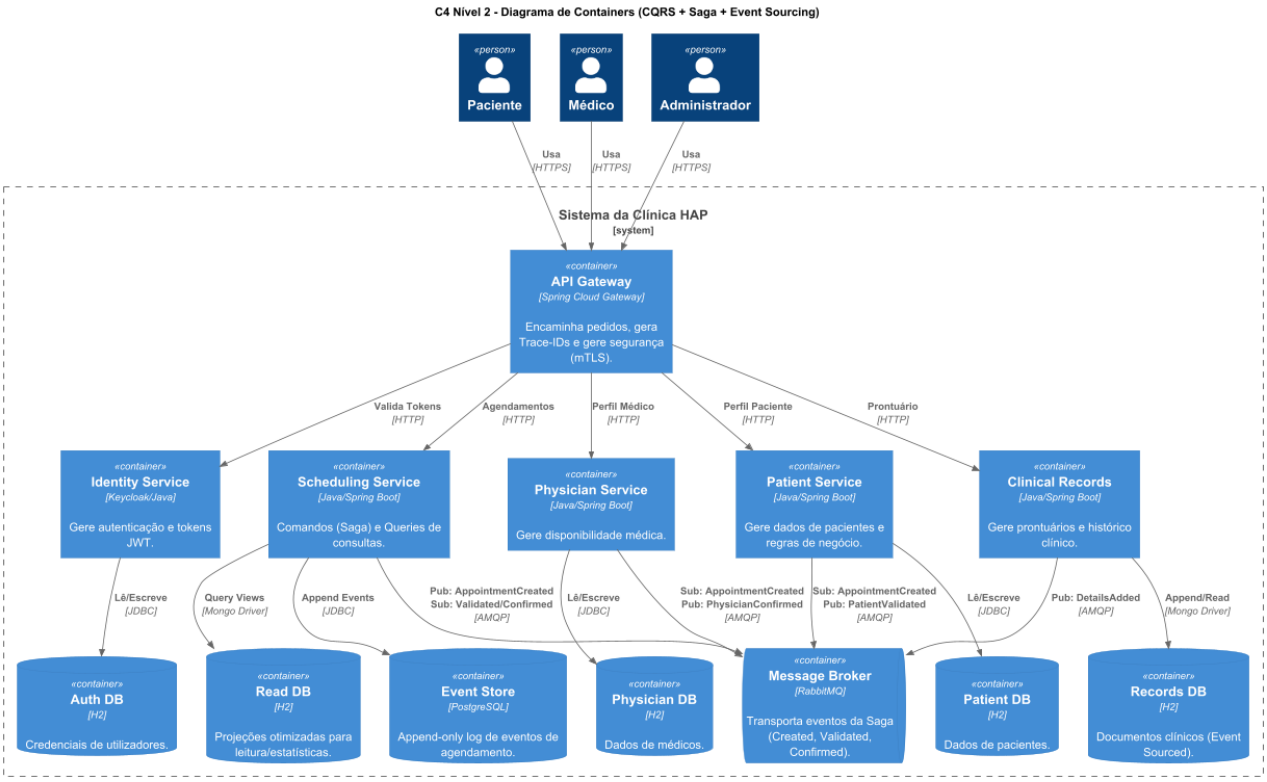
Padrões de Comunicação

O diagrama distingue claramente dois tipos de comunicação:

- Comunicação Síncrona (HTTP/REST):
  Utilizada sobretudo na interação entre cliente, API Gateway e microsserviços, bem como na validação de identidade e permissões.
- Comunicação Assíncrona (AMQP):
  Utilizada para a coordenação interna entre microsserviços. A troca de eventos através do broker elimina dependências diretas e chamadas bloqueantes, aumentando a resiliência do sistema perante falhas parciais.

Enquadramento Arquitetural

Este diagrama demonstra como o sistema passa de uma visão conceptual (Nível 1) para uma visão estrutural concreta, identificando claramente os containers responsáveis por execução, persistência e comunicação. Serve como base para os níveis seguintes do modelo C4, onde são detalhados os componentes internos de cada microsserviço e os principais fluxos de interação implementados.



Nível 3

Este nível descreve a estrutura interna de um microserviço, evidenciando como os componentes colaboram para implementar os requisitos da Parte 3: separação de responsabilidades (CQRS), comunicação por eventos (AMQP), consistência eventual e resiliência.

Para evitar repetição, apresenta-se detalhadamente o Scheduling Service, por concentrar a maior complexidade arquitetural (CQRS + Saga + Event Sourcing). Os serviços Patient e Physician seguem a mesma estrutura base, variando apenas as regras de domínio e os eventos publicados/consumidos.

Componentes principais

Entrada e Segurança:

- Security Filter (JWT + mTLS):
  Garante que pedidos externos chegam autenticados e que a comunicação interna service-to-service respeita o modelo zero-trust.
- Schedule Controller (HTTP/REST):
  Recebe comandos do utilizador (ex.: criar ou cancelar consulta) e encaminha para o fluxo de escrita. Queries GET seguem para o fluxo de leitura.
- Saga Event Listener (AMQP/RabbitMQ):
  Consome eventos de validação (ex.: respostas do Patient e Physician) e alimenta a evolução assíncrona do processo de agendamento.

Command Side – Escrita com Event Sourcing

- Scheduling Command Service:
  Aplica as regras de negócio e decide que evento deve ser emitido (criar, confirmar, cancelar).
- Saga Aggregator (State Machine + Reidratação):
  Reconstrói o estado da consulta a partir do histórico de eventos (reidratação) e determina se a Saga está completa ou se ainda faltam validações.
- Event Store Repository (Append-Only)
  Persiste eventos no Event Store DB (PostgreSQL). O estado atual é derivado do histórico, não armazenado diretamente como fonte única.
- Event Publisher (AMQP):
  Publica eventos no broker (ex.: AppointmentCreated, AppointmentBooked, AppointmentCancelled) para notificar outros serviços e para alimentar projeções.

3.3 Query Side – Leitura por Projeções (Read Model)

- Scheduling Query Service
  Responde a operações de leitura (agenda, histórico e listagens) sem depender de reidratação.
- Read Model Repository (Views/Projeções)
  Acede ao Read DB (H2) com tabelas otimizadas para consultas rápidas e relatórios.
- Scheduling Projector
  Consome os eventos do lado da escrita e atualiza as projeções no Read DB, garantindo consistência eventual entre escrita e leitura.

Resiliência Transversal

- Resilience Aspect (Resilience4j):
  Envolve o acesso às bases de dados e dependências críticas, aplicando padrões como:
  - TimeLimiter para evitar bloqueios prolongados (fail-fast);
  - Circuit Breaker para impedir falhas em cascata;
  - Retry apenas quando aplicável (falhas transitórias), mantendo idempotência no consumo de eventos.

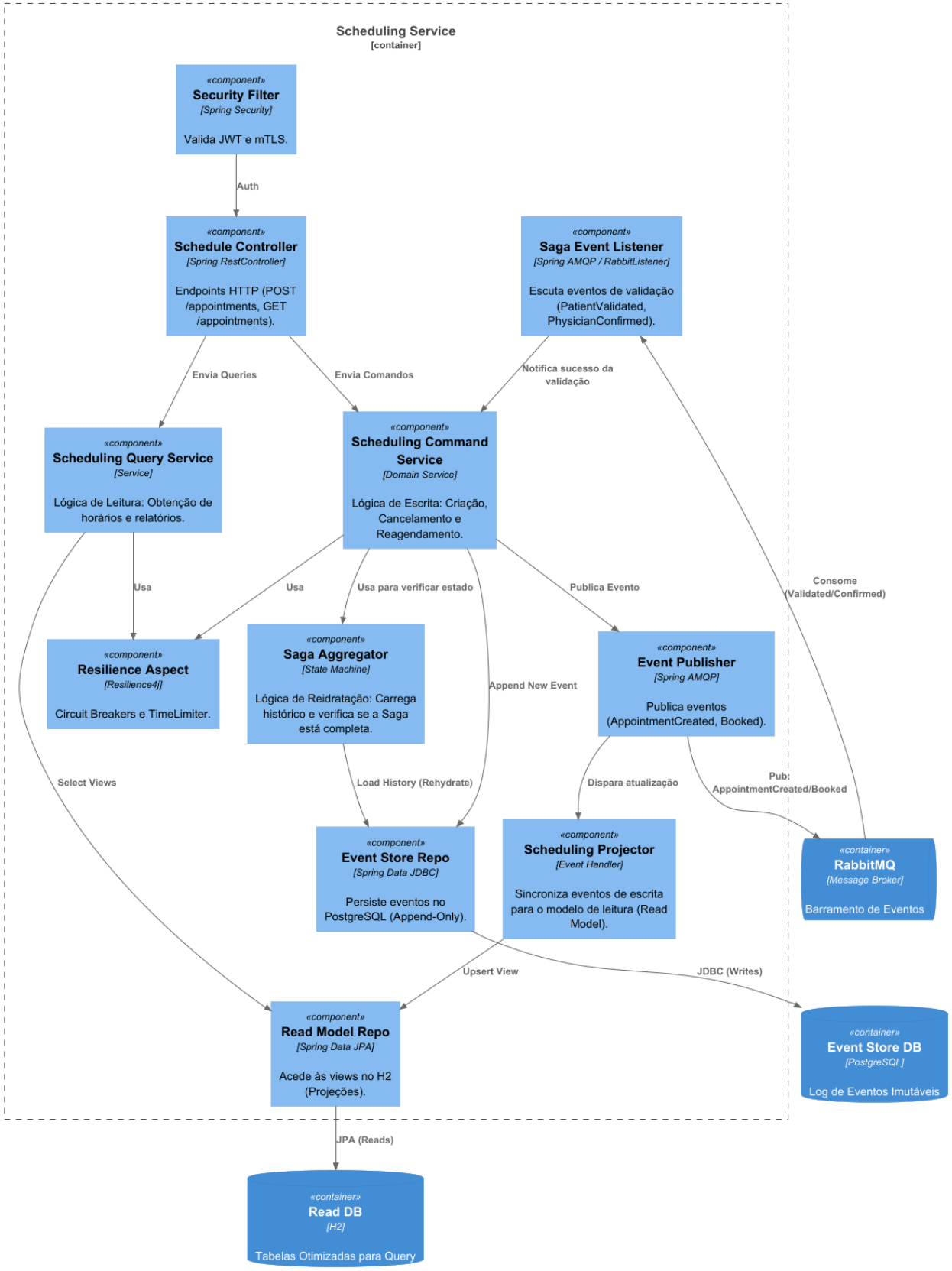
Generalização para Patient e Physician

Os serviços Patient e Physician seguem o mesmo padrão estrutural interno (Security Filter, Controller, AMQP Listener, Command Side, Query Side, Projector e Resilience). A diferença está no domínio e na participação na Saga:

- Patient Service recebe eventos de agendamento, valida condições do paciente e publica eventos de resposta (ex.: PatientValidated / rejeição).
- Physician Service recebe eventos de agendamento, valida disponibilidade e publica eventos de resposta (ex.: PhysicianAvailabilityConfirmed / rejeição).

Ambos mantêm separação lógica entre escrita e leitura (CQRS) e atualizam read models por eventos, suportando consistência eventual e execução multi-instância.

Scheduling Service
[Container - C4 Level 3: Component Diagram - CQRS, Saga & ES]



Nível 4+1

Fluxo de Agendamento de Consulta

Este diagrama representa o processo "Core" do sistema HAP: a criação de uma consulta médica.

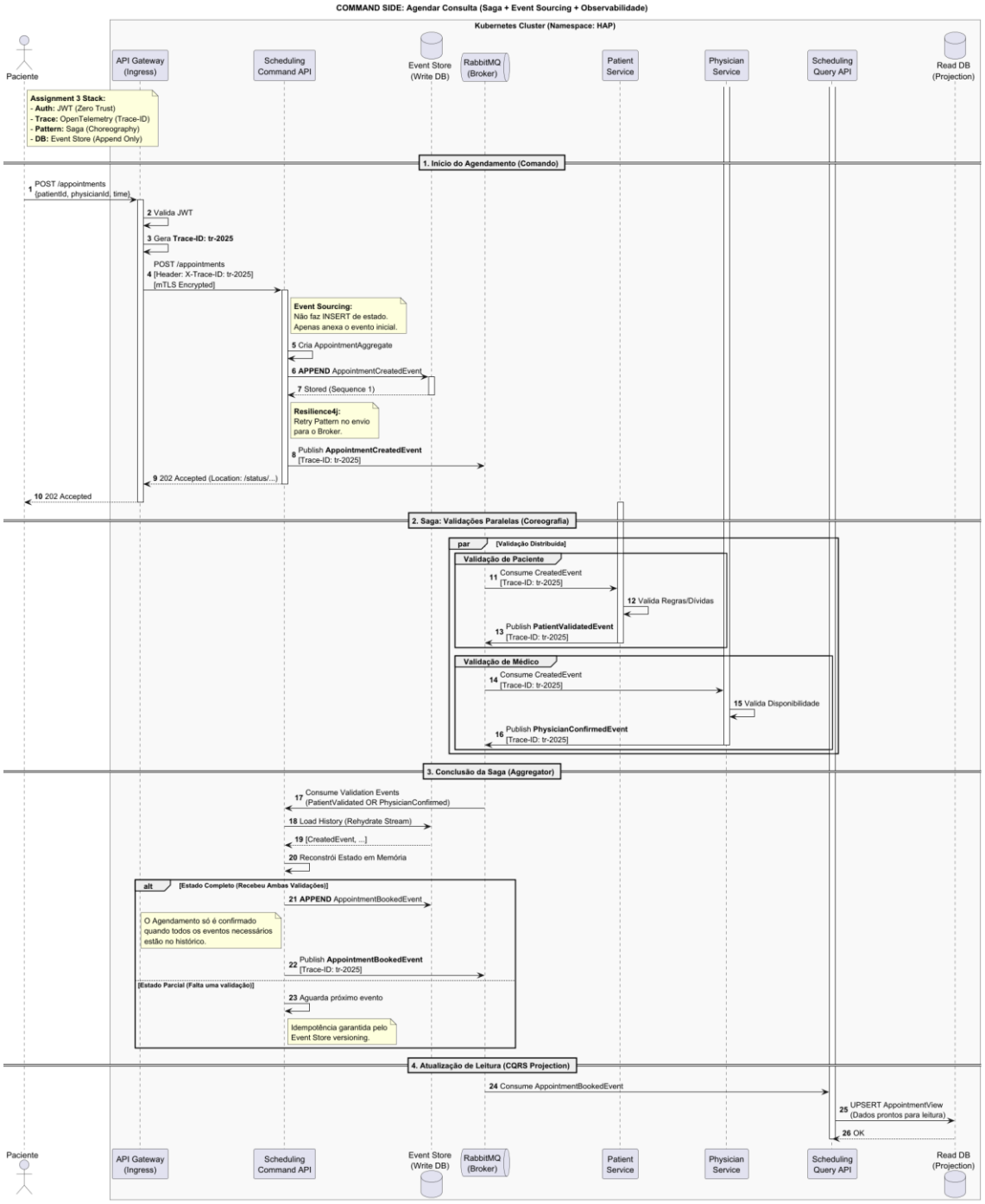
Conceitos Arquiteturais Fundamentais

1. Event Sourcing (Append-Only):
   • A base de dados de escrita (Event Store) nunca sofre atualizações (UPDATE). O estado de uma consulta não é uma linha numa tabela, mas sim a soma de todos os eventos associados a um ID (Created, Validated, Confirmed, Booked).
   • Isto garante um histórico auditável perfeito e facilita a gestão de concorrência.
2. Saga Coreografada (Paralelismo):
   • Não existe um orquestrador central a "mandar" nos serviços. O Scheduling Service emite um evento e os outros serviços (Patient e Physician) reagem a ele autonomamente e em paralelo.
   • Isto reduz a latência total do processo, pois as validações do paciente e do médico ocorrem ao mesmo tempo.
3. O Papel do Agregador (Passo 3):
   • O Scheduling Command API atua como um "Agregador Stateful". Sempre que recebe um evento de sucesso (PatientValidated ou PhysicianConfirmed), ele executa a Reidratação (lê o histórico de eventos).
   • A consulta só transita para o estado Booked quando o histórico confirma que ambas as validações necessárias foram recebidas.
4. Observabilidade (Trace-ID):
   • Dado que o processo salta entre 4 serviços e um Message Broker, o Trace-ID (tr-2025) é vital. Ele permite visualizar no Jaeger o "caminho crítico" do pedido e identificar qual dos microsserviços está a causar lentidão.

Descrição do Fluxo

1. Início Assíncrono:
   • O Paciente envia o pedido. O Gateway valida a segurança (JWT/mTLS) e gera o rasto (Trace-ID).
   • A API grava o evento AppointmentCreatedEvent e responde imediatamente com 202 Accepted. O utilizador não fica bloqueado à espera das validações complexas.
   • O envio para o Broker inclui padrões de resiliência (Retry) para garantir que a mensagem não se perde em caso de falha de rede momentânea.
2. Validação Distribuída:
   • O Message Broker distribui o evento.
   • O Patient Service verifica se o paciente tem dívidas ou seguros ativos.
   • O Physician Service verifica se o médico tem disponibilidade na agenda.
   • Ambos publicam o resultado no Broker, mantendo o Trace-ID original.

3. Conclusão da Transação:
   • O Scheduling Service consome as respostas. Graças ao Event Sourcing, ele reconstrói o estado em memória para saber se já tem tudo o que precisa.
   • Se ambas as validações forem positivas, grava o evento definitivo AppointmentBookedEvent.
4. Consistência Eventual (CQRS):
   • O evento final é consumido pelo Query Side, que atualiza a base de dados de leitura para que a consulta apareça finalmente na lista do utilizador como "Confirmada".



Fluxo de Leitura de Estatísticas

Este diagrama ilustra o processo de obtenção de dados analíticos (estatísticas de duração de consultas) no lado de Leitura (Query Side) da arquitetura CQRS. O foco principal deste fluxo não é a lógica de negócio complexa, mas sim a Performance, Segurança e Resiliência do sistema distribuído.

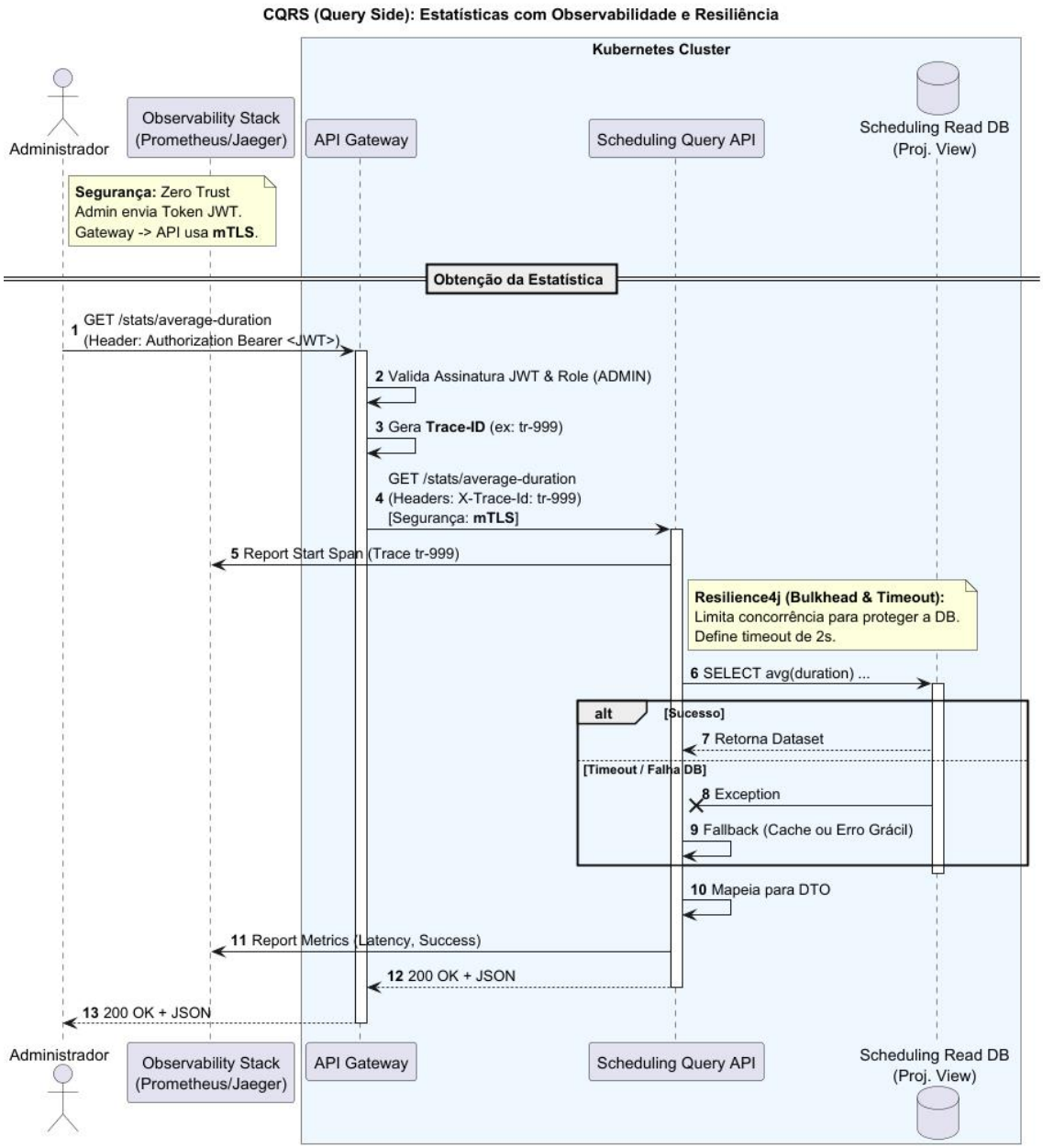
Principais Componentes e Padrões

1. Segurança (Zero Trust e mTLS):
   • A autenticação externa é feita via JWT (validado no API Gateway).
   • A comunicação interna entre o Gateway e a Scheduling Query API é protegida por mTLS (Mutual TLS), garantindo que apenas serviços autorizados dentro do cluster Kubernetes podem invocar a API.
2. Observabilidade (Distributed Tracing):
   • Assim que o pedido entra no sistema, o API Gateway gera um Trace-ID único (ex: tr-999).
   • Este ID é propagado nos cabeçalhos HTTP (X-Trace-Id) para todos os serviços subsequentes.
   • Isto permite monitorizar o tempo total da transação e identificar gargalos (bottlenecks) através de ferramentas como Jaeger ou Zipkin.
3. Resiliência (Resilience4j):
   • Como as consultas estatísticas podem ser pesadas para a base de dados, implementámos padrões de proteção na Query API:
   • TimeLimiter: Define um tempo máximo (ex: 2s) para a base de dados responder. Se demorar mais, a thread é libertada imediatamente para não bloquear o servidor.
   • Bulkhead: Limita o número de pedidos concorrentes simultâneos a esta rota, impedindo que um pico de tráfego de leitura afete outras partes do sistema.
   • Fallback: Em caso de erro ou timeout, o sistema não "crasha"; retorna uma resposta degradada de forma controlada (ex: dados em cache ou mensagem de erro amigável).

Descrição do Fluxo

1. Pedido Inicial: O Administrador solicita a média de duração das consultas. O API Gateway valida as credenciais e injeta o contexto de rastreio (Trace-ID).
2. Monitorização: A Query API reporta o início do processamento ("Start Span") à stack de observabilidade.
3. Proteção de Recursos: Antes de contactar a base de dados, o framework Resilience4j verifica se há "slot" disponível (Bulkhead) e inicia o temporizador (Timeout).
4. Consulta à Base de Leitura (Read DB): A API executa a query SQL/NoSQL na base de dados de leitura (uma Projection otimizada para queries rápidas, separada da escrita).

5. Resposta e Métricas:
   • Se a BD responder a tempo: Os dados são mapeados para DTO e retornados.
   • Se ocorrer Timeout ou Erro: O mecanismo de Fallback é ativado.
   • No final, são enviadas métricas (latência, sucesso ou erro) para o Prometheus para monitorização em tempo real.



Fluxo de Atualização e Cancelamento

Este diagrama detalha como o sistema gere modificações de estado complexas (Reagendamento e Cancelamento) utilizando Event Sourcing e o padrão Saga (Coreografia). Ao contrário de uma arquitetura tradicional baseada em CRUD, aqui

nenhuma informação é sobrescrita; todas as mudanças são preservadas como uma sequência de eventos imutáveis.

Conceitos Arquiteturais Chave

1. Event Sourcing e Reidratação:
   • A base de dados de escrita (Write DB) não guarda o estado atual da consulta. Guarda apenas o histórico de eventos.
   • Sempre que é necessário validar uma regra de negócio (ex: "Posso cancelar esta consulta?"), o serviço executa o processo de Reidratação: carrega todos os eventos passados desse ID (Load History) e reconstrói o estado em memória.
   • As alterações de estado são feitas através de APPEND de novos eventos (ex: RescheduleRequestedEvent, AppointmentCancelledEvent).
2. Saga Coreografada (Cenário A - Reagendar):
   • O reagendamento não é atómico. Requer confirmação externa (Disponibilidade do Médico).
   • O Scheduling Service inicia o processo e "pausa" logicamente, retornando 202 Accepted ao utilizador.
   • O Physician Service reage ao evento, valida o novo horário e emite uma confirmação. A Saga só termina quando o Scheduling Service processa essa confirmação e emite o evento final AppointmentRescheduledEvent.
3. Processamento Orientado a Eventos (Cenário B - Cancelar):
   • O cancelamento é um evento destrutivo que deve propagar-se por todo o sistema.
   • Ao emitir AppointmentCancelledEvent, múltiplos serviços reagem em paralelo: a Read DB remove a consulta da vista do utilizador e o Physician Service liberta o horário na agenda do médico.
4. Observabilidade e Segurança:
   • Trace-ID: IDs únicos (ex: tr-456, tr-789) acompanham o pedido desde o Gateway até ao processamento assíncrono no Message Broker, permitindo correlacionar logs distribuídos.
   • mTLS: Garante que apenas o Gateway pode invocar comandos de modificação na API.

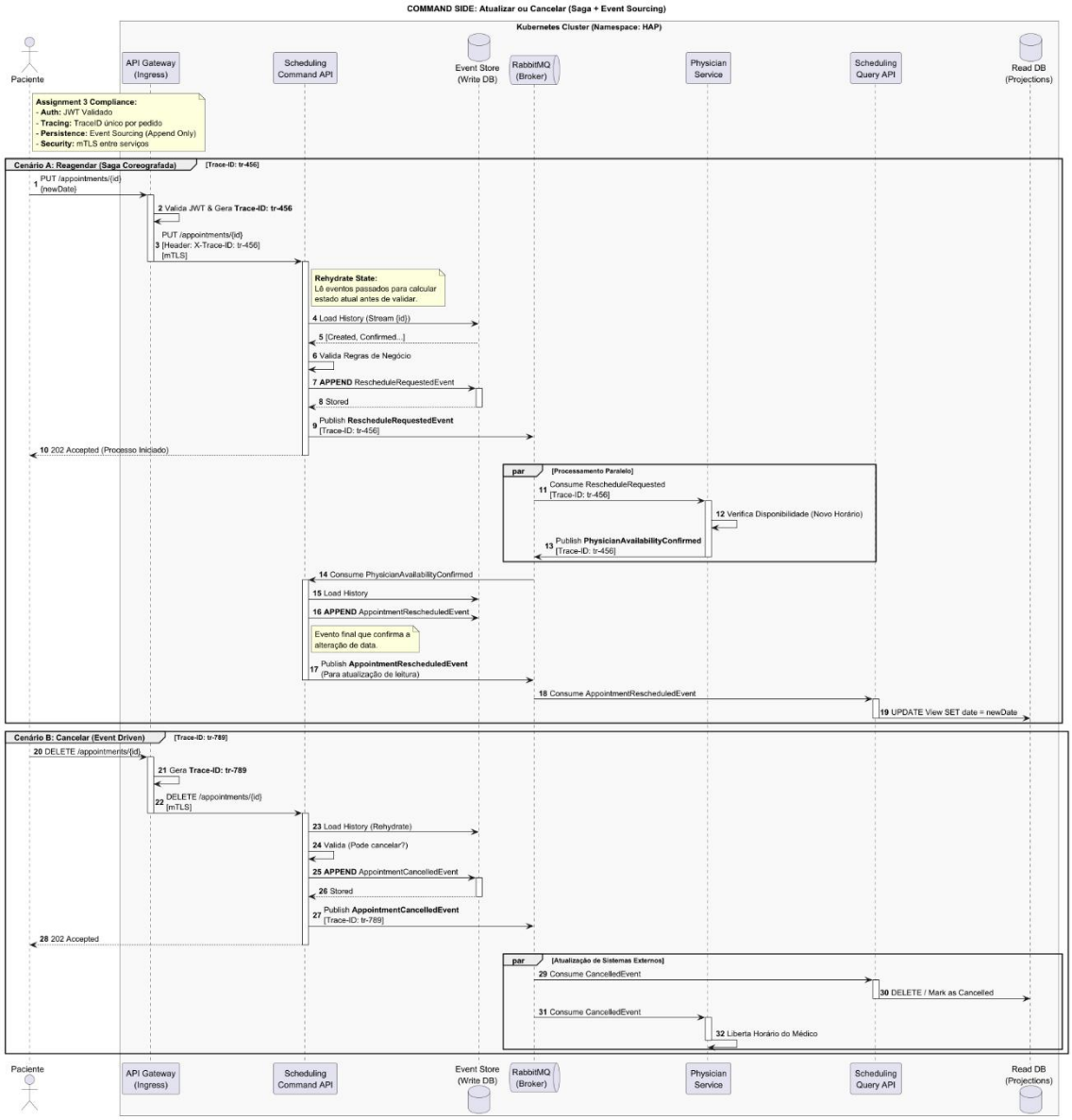
Descrição dos Cenários

Cenário A: Reagendar (Complexo)

1. Pedido: O Paciente solicita a alteração da data via PUT. O Gateway gera o Trace-ID tr-456.
2. Validação: A Command API reconstrói o estado (Rehydrate) para garantir que a consulta existe e não está concluída.
3. Início da Saga: É gravado e publicado o evento RescheduleRequestedEvent. O utilizador recebe resposta imediata (202).
4. Coreografia: O Physician Service verifica a agenda. Se houver vaga, publica PhysicianAvailabilityConfirmed.
5. Conclusão: A Command API consome a confirmação e grava o evento final AppointmentRescheduledEvent. A base de leitura é atualizada assincronamente.

Cenário B: Cancelar (Direto)

1. Pedido: O Paciente solicita o cancelamento via DELETE (Trace-ID tr-789).
2. Persistência: Após validar que o cancelamento é permitido (via histórico), a API grava o evento AppointmentCancelledEvent.
3. Propagação: O evento é publicado no Broker.
4. Efeitos Colaterais: O Scheduling Query API marca a consulta como cancelada (ou remove-a) e o Physician Service torna o horário disponível novamente para outros pacientes.



Fluxo de Registo de Detalhes Clínicos

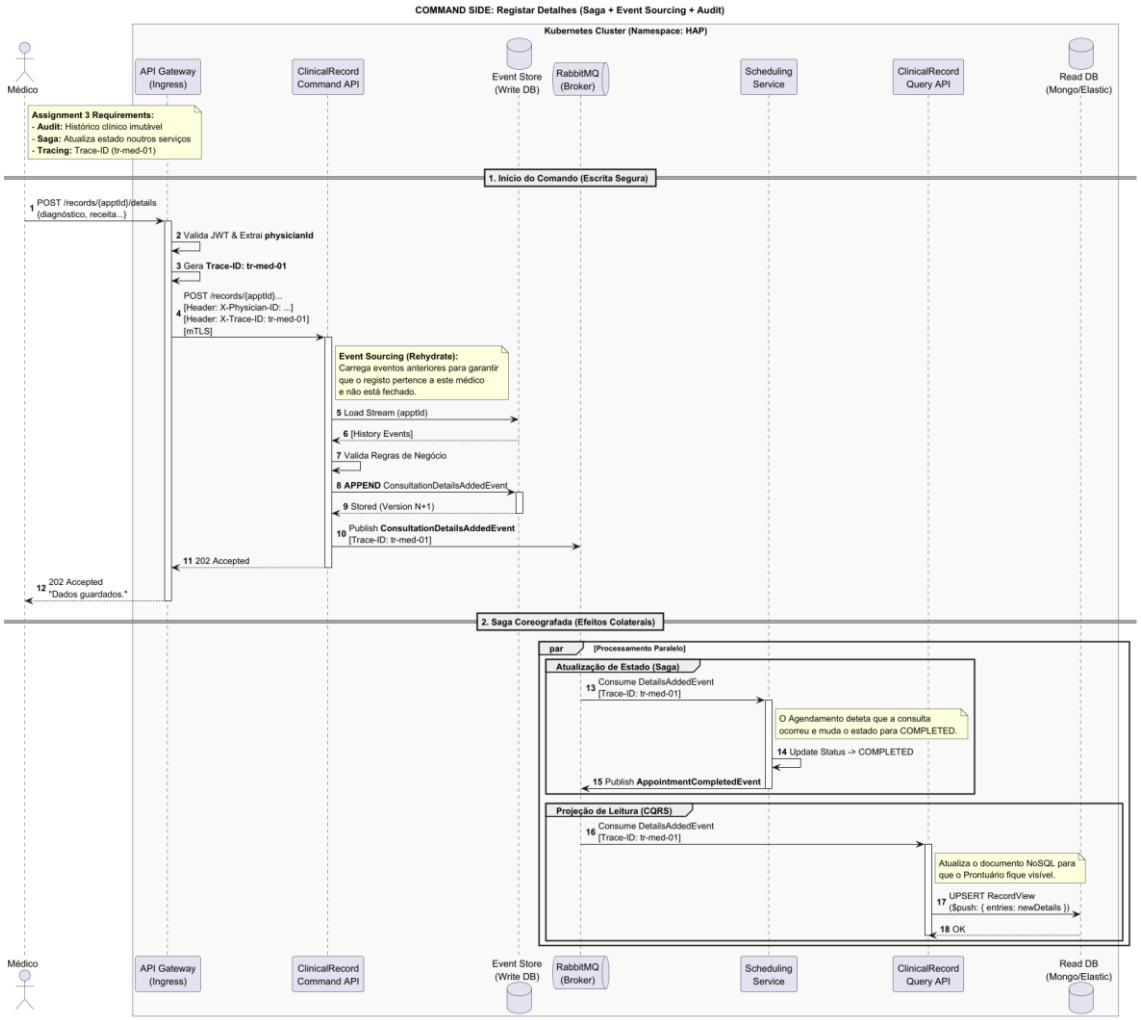
Este diagrama ilustra o processo de gravação de dados clínicos (diagnósticos, receitas, notas) associados a uma consulta. Devido à natureza sensível e legal destes dados, a arquitetura utiliza Event Sourcing para garantir um histórico inalterável e uma Saga Coreografada para propagar o estado de conclusão da consulta.

Decisões Arquiteturais Chave

1. Event Sourcing (Imutabilidade Legal):
   • Em sistemas médicos, nunca se deve sobrescrever um diagnóstico. Se um médico comete um erro, deve criar uma nova entrada de correção.
   • O sistema implementa isto nativamente ao usar um Event Store. Em vez de um UPDATE na tabela, fazemos um APPEND do evento ConsultationDetailsAddedEvent. Isto cria um rasto de auditoria perfeito e imutável (Audit Log) de todas as interações.
2. Saga Coreografada (Side-Effects):
   • O registo de detalhes clínicos tem um efeito colateral noutro domínio: significa que a consulta ocorreu efetivamente.
   • Através do padrão Coreografia, o Scheduling Service escuta o evento de detalhes (DetailsAddedEvent) e transita automaticamente o estado da consulta para COMPLETED. Isto mantém os serviços de Registos Clínicos e Agendamento totalmente desacoplados.
3. Observabilidade Transversal:
   • Como um único clique do médico despoleta ações em três serviços diferentes (Command API, Scheduling Service, Query API), o Trace-ID (tr-med-01) é fundamental para monitorizar a saúde de toda a transação distribuída e detetar falhas de sincronização.
4. Segurança (Contexto do Médico):
   • O Gateway extrai o physicianId do Token JWT e injeta-o no cabeçalho.
   • A Command API usa este ID durante a fase de Reidratação para garantir que apenas o médico responsável pela consulta pode adicionar notas à mesma.

Descrição do Fluxo

1. Comando Seguro:
   • O médico submete os detalhes.
   • O Gateway valida a identidade, gera o rasto (Trace-ID) e encaminha via mTLS.
2. Persistência (Event Sourcing):
   • A API carrega os eventos passados (Load Stream) para validar o estado atual.
   • Grava o novo evento no Event Store.
   • Retorna 202 Accepted imediatamente, libertando o utilizador.
3. Processamento Paralelo (Async):
   • Atualização de Estado (Saga): O Scheduling Service consome o evento e marca a consulta como Concluída, emitindo o seu próprio evento AppointmentCompletedEvent.
   • Projeção (CQRS): O Query API atualiza a base de dados de leitura (NoSQL), adicionando os novos detalhes ao documento do prontuário para que fiquem visíveis na interface do médico.



Fluxo de Pesquisa de Pacientes

Este diagrama representa a arquitetura de leitura (Query Side) para a pesquisa de pacientes. Sendo uma operação frequente e que envolve dados sensíveis, o desenho da solução prioriza a Privacidade (GDPR), a Performance e a Proteção do Sistema contra sobrecargas.

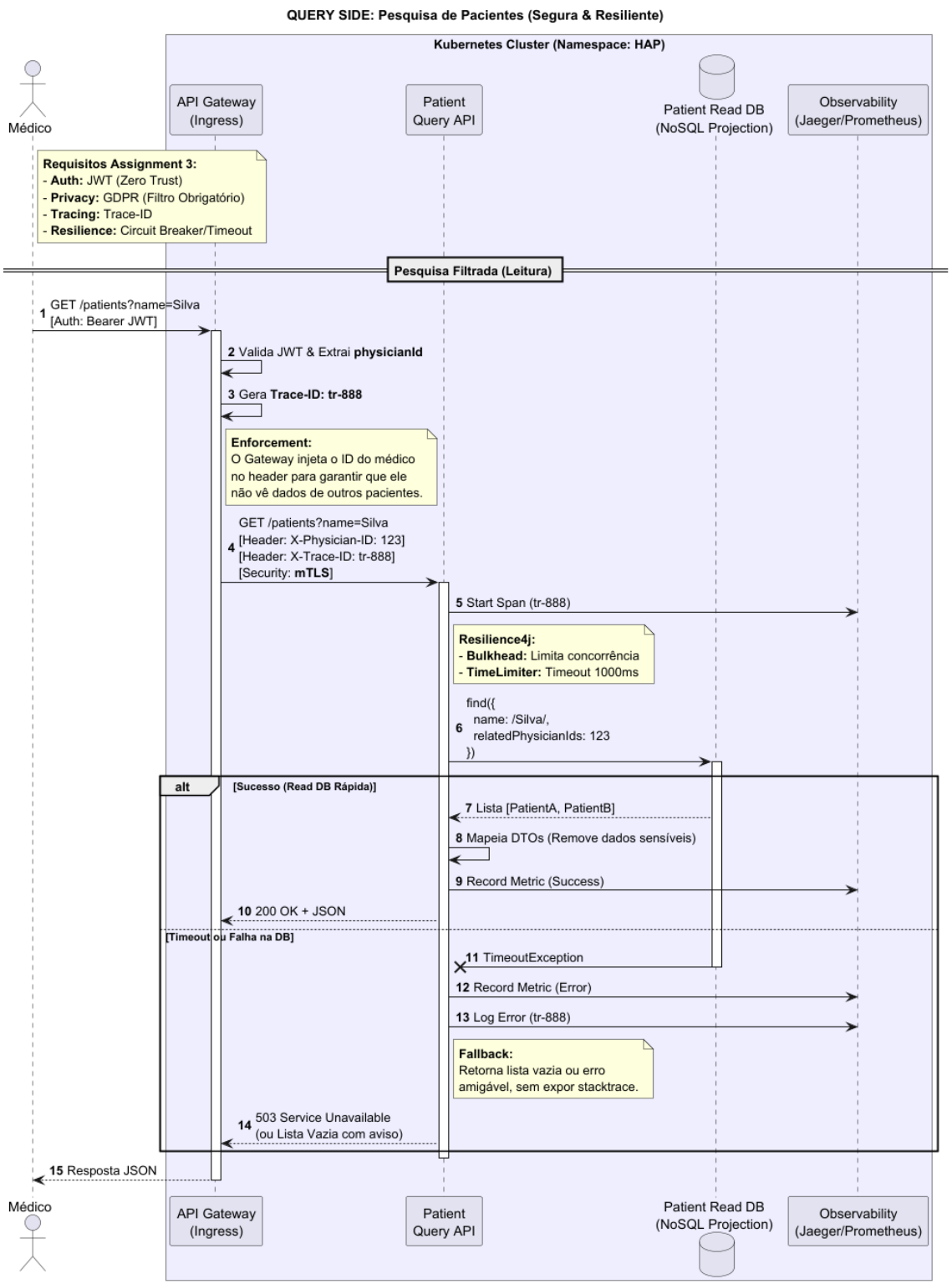
Destaques da Implementação

1. Segurança e Privacidade (GDPR by Design):
   • Enforcement no Gateway: A segurança não depende apenas da API interna. O API Gateway extrai o physicianId do Token JWT e injeta-o obrigatoriamente no cabeçalho do pedido.
   • Filtro Obrigatório: A query à base de dados inclui sempre { relatedPhysicianIds: 123 }. Isto garante que um médico nunca consegue visualizar pacientes com os quais não tem relação clínica, mesmo que tente manipular os parâmetros da URL.
   • mTLS: A comunicação entre o Gateway e a Query API é encriptada e autenticada mutuamente.

2. Resiliência (Resilience4j):
   • TimeLimiter: Foi configurado um timeout rigoroso de 1000ms. Se a base de dados (NoSQL Projection) estiver degradada, o sistema aborta o pedido imediatamente em vez de manter a thread bloqueada.
   • Bulkhead: Limita o número de pesquisas simultâneas. Se houver um pico de tráfego, o padrão Bulkhead impede que a funcionalidade de pesquisa consuma todos os recursos do cluster, protegendo outras funcionalidades críticas.
3. Observabilidade:
   • O Trace-ID (tr-888) é gerado na entrada e acompanha todo o ciclo de vida do pedido.
   • São registadas métricas de negócio (sucesso vs. erro) e métricas técnicas (latência), permitindo a criação de alertas no Grafana se a taxa de erros na pesquisa aumentar.

Descrição do Fluxo

1. Pedido e Contexto:
   • O médico inicia a pesquisa.
   • O Gateway valida a identidade e gera o contexto de rastreio (tr-888).
2. Proteção:
   • Antes de executar a lógica, a Query API verifica as políticas de resiliência (concorrência e tempo).
3. Execução da Query:
   • A pesquisa é feita numa Projeção de Leitura (NoSQL), otimizada para velocidade, desacoplada da base de escrita.
   • O filtro de segurança (physicianId) é aplicado nativamente na query.
4. Tratamento de Falhas:
   • Caminho Feliz: Retorna a lista de pacientes (DTOs limpos de dados técnicos).
   • Timeout ou Erro: Se o limite de 1000ms for excedido, a exceção é capturada. O sistema regista o erro nos logs (com o Trace-ID para debugging) e retorna uma resposta de erro controlada (503 ou lista vazia), evitando expor stacktraces ao cliente.


QUERY SIDE: Pesquisa de Pacientes (Segura & Resiliente)
Kubernetes Cluster (Namespace: HAP)
Médico
API Gateway (Ingress)
Patient Query API
Patient Read DB (NoSQL Projection)
Observability (Jaeger/Prometheus)
Requisitos Assignment 3:
- Auth: JWT (Zero Trust)
- Privacy: GDPR (Filtro Obrigatório)
- Tracing: Trace-ID
- Resilience: Circuit Breaker/Timeout
Pesquisa Filtrada (Leitura)
1 GET /patients?name=Silva [Auth: Bearer JWT]
2 Valida JWT & Extrai physicianId
3 Gera Trace-ID: tr-888
Enforcement:
O Gateway injeta o ID do médico no header para garantir que ele não vê dados de outros pacientes.
4 GET /patients?name=Silva [Header: X-Physician-ID: 123] [Header: X-Trace-ID: tr-888] [Security: mTLS]
5 Start Span (tr-888)
Resilience4j:
- Bulkhead: Limita concorrência
- TimeLimiter: Timeout 1000ms
6 find({ name: /Silva/, relatedPhysicianIds: 123 })
alt
[Sucesso (Read DB Rápida)]
7 Lista [PatientA, PatientB]
8 Mapeia DTOs (Remove dados sensíveis)
9 Record Metric (Success)
10 200 OK + JSON
[Timeout ou Falha na DB]
11 TimeoutException
12 Record Metric (Error)
13 Log Error (tr-888)
Fallback:
Retorna lista vazia ou erro amigável, sem expor stacktrace.
14 503 Service Unavailable (ou Lista Vazia com aviso)
15 Resposta JSON

Fluxo de Registo de Médico

Este diagrama ilustra o processo distribuído de registo de um novo profissional clínico. A complexidade deste caso de uso reside na necessidade de sincronizar dois domínios distintos: o Domínio Clínico (dados do médico, licença, especialidade) e o Domínio de Identidade (credenciais de acesso, roles, autenticação).

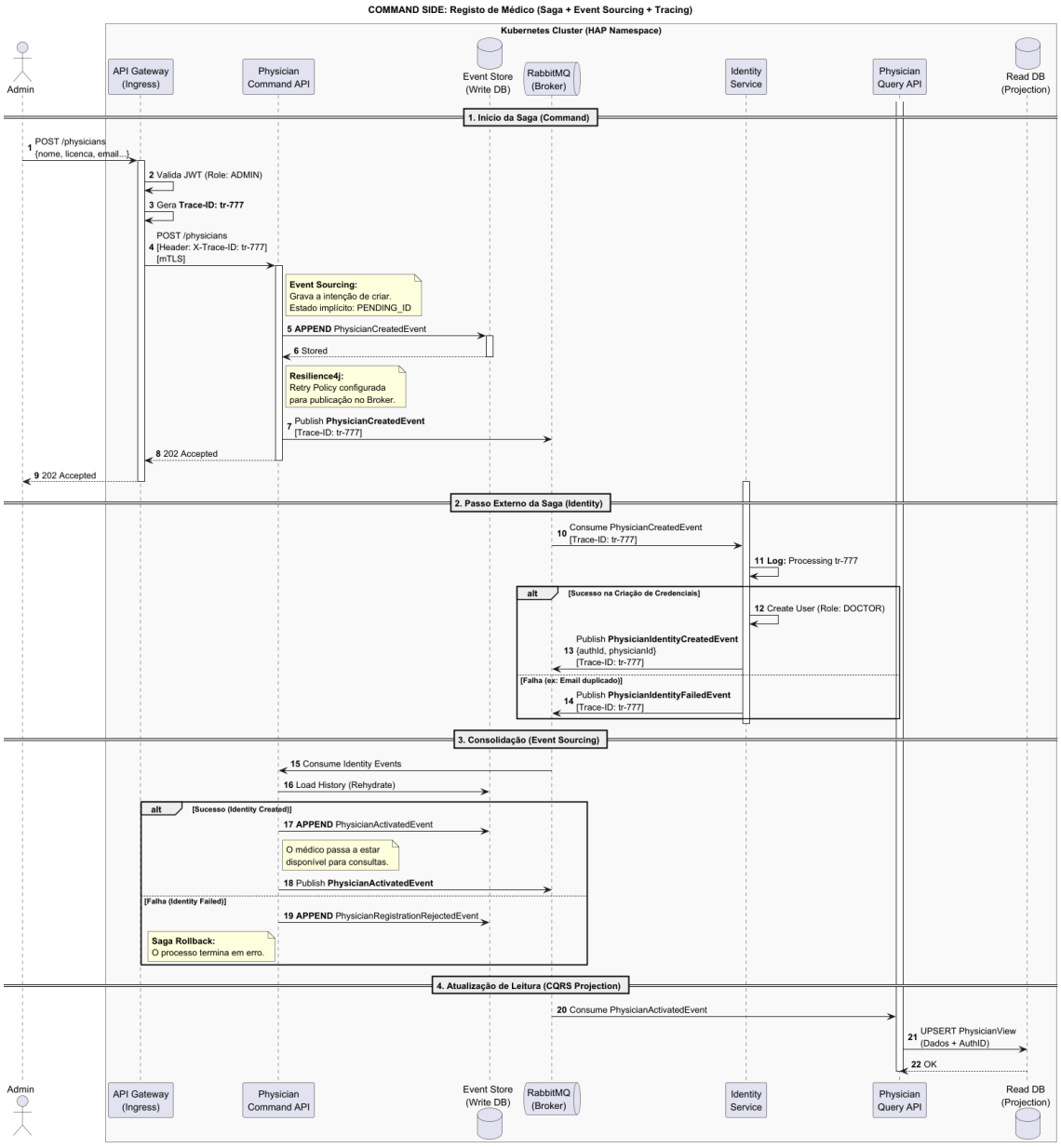
Arquitetura e Decisões de Design

1. Saga Distribuída (Integração com Identity Service):
   • A criação de um médico não é uma operação atómica local.
   • É necessário criar o registo na base de dados do hospital e, apenas se isso for válido, criar as credenciais no serviço de Identidade.
   • O Physician Command API não comunica diretamente com o Identity Service.
   • Utiliza-se o Message Broker para desacoplar os serviços, garantindo que o sistema clínico não falha se o sistema de identidade estiver temporariamente indisponível.
2. Event Sourcing (Auditabilidade Imutável):
   • Em vez de guardar apenas o estado final (Médico Ativo), o sistema grava a sequência de eventos.
   • PhysicianCreatedEvent representa a intenção.
   • PhysicianActivatedEvent representa a conclusão.
   • PhysicianRegistrationRejectedEvent representa a falha.
   • Isto permite auditoria completa do processo.
3. Observabilidade (Rastreio Cross-Service):
   • O Trace-ID (tr-777) é gerado no Gateway e propagado para o Identity Service via RabbitMQ.
   • Isto permite correlacionar logs de serviços diferentes e identificar a origem de falhas.
4. Resiliência:
   • A publicação no Broker inclui uma Retry Policy.
   • Se o RabbitMQ estiver temporariamente indisponível, o serviço tenta reenviar o evento antes de desistir.

Descrição das Etapas

1. Início (Intenção):
   • O Admin envia os dados.
   • O Gateway valida a segurança e inicia o rastreio.
   • O Command API persiste o evento PhysicianCreatedEvent no Event Store.
   • Retorna 202 Accepted para não bloquear o Admin.
2. Processamento Externo:
   • O Identity Service consome o evento.
   • Tenta criar o utilizador com a role DOCTOR.
   • Emite PhysicianIdentityCreatedEvent em caso de sucesso.
   • Emite PhysicianIdentityFailedEvent em caso de erro.

3. Consolidação:
   • O Command API escuta a resposta da Identidade.
   • Faz o Rehydrate do estado (lê eventos anteriores).
   • Grava o evento final Activated ou Rejected.
4. Projeção:
   • O Query Side atualiza a vista de leitura.
   • O médico passa a estar visível nas pesquisas e associado ao seu AuthID.



Fluxo de Registo de Paciente

Este diagrama detalha o processo de criação de um novo paciente no sistema HAP. Devido à necessidade de criar simultaneamente o registo clínico e as credenciais de acesso no serviço de identidade, o processo é gerido como uma Saga Distribuída. O uso

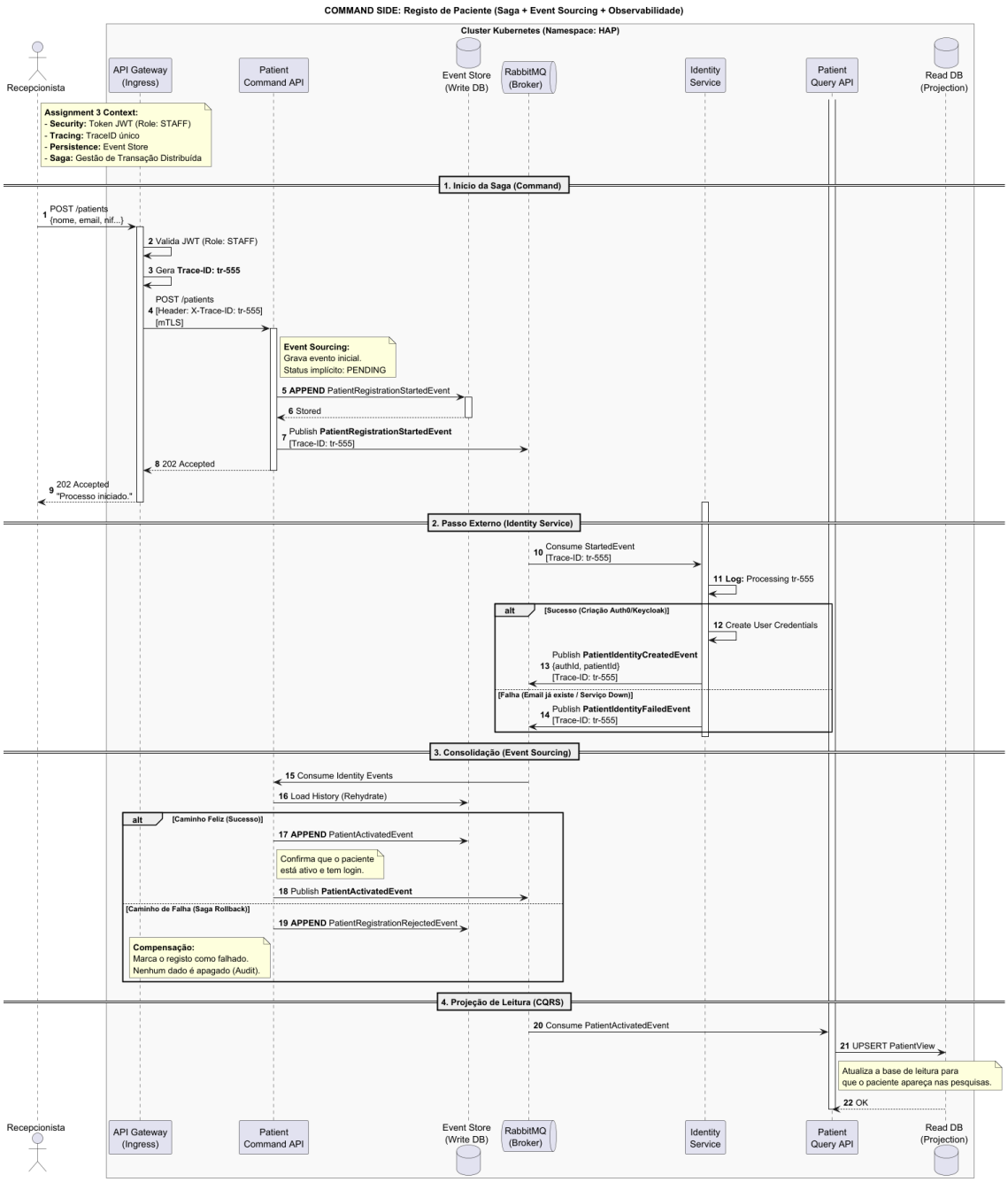
de Event Sourcing garante que todo o histórico de tentativas de registo fica preservado para auditoria.

Decisões Arquiteturais e Padrões

1. Event Sourcing e Auditoria:
   • O sistema utiliza um modelo append-only, evitando remoção ou sobrescrita de dados.
   • Em caso de falha no registo, é gravado um evento de rejeição, mantendo o histórico completo da tentativa.
   • Este modelo permite saber quem tentou registar o paciente, quando e por que motivo falhou.
2. Saga com Compensação:
   • O registo envolve dois domínios distintos: Paciente e Identidade.
   • O Patient Command API inicia o processo e aguarda a resposta assíncrona do Identity Service.
   • Dependendo do resultado, o processo avança para ativação ou executa uma compensação lógica.
3. Observabilidade:
   • O Trace-ID acompanha o pedido desde o Gateway até às filas de mensagens.
   • Este mecanismo permite identificar facilmente onde o processo falhou.
4. Segurança:
   • Apenas utilizadores com permissões adequadas podem iniciar o registo.
   • A comunicação interna entre serviços é protegida por mTLS.

Descrição do Fluxo

1. Início:
   • A rececionista envia os dados do paciente.
   • O Gateway valida o token, gera o Trace-ID e grava o evento PatientRegistrationStartedEvent.
   • O sistema responde com 202 Accepted.
2. Identidade:
   • O Identity Service consome o evento.
   • Cria as credenciais do paciente ou publica um evento de erro.
3. Consolidação:
   • O Patient Command API reidrata o estado.
   • Em caso de sucesso grava PatientActivatedEvent.
   • Em caso de falha grava PatientRegistrationRejectedEvent.
4. Atualização de Leitura:
   • O Query API atualiza a base de dados de leitura, garantindo consistência eventual.



Fluxo de Relatório Mensal

Este diagrama ilustra o processo de geração de relatórios estatísticos mensais na arquitetura CQRS. Trata-se de uma operação intensiva em recursos, pelo que a arquitetura privilegia a proteção do sistema e a monitorização do desempenho.

Destaques da Implementação

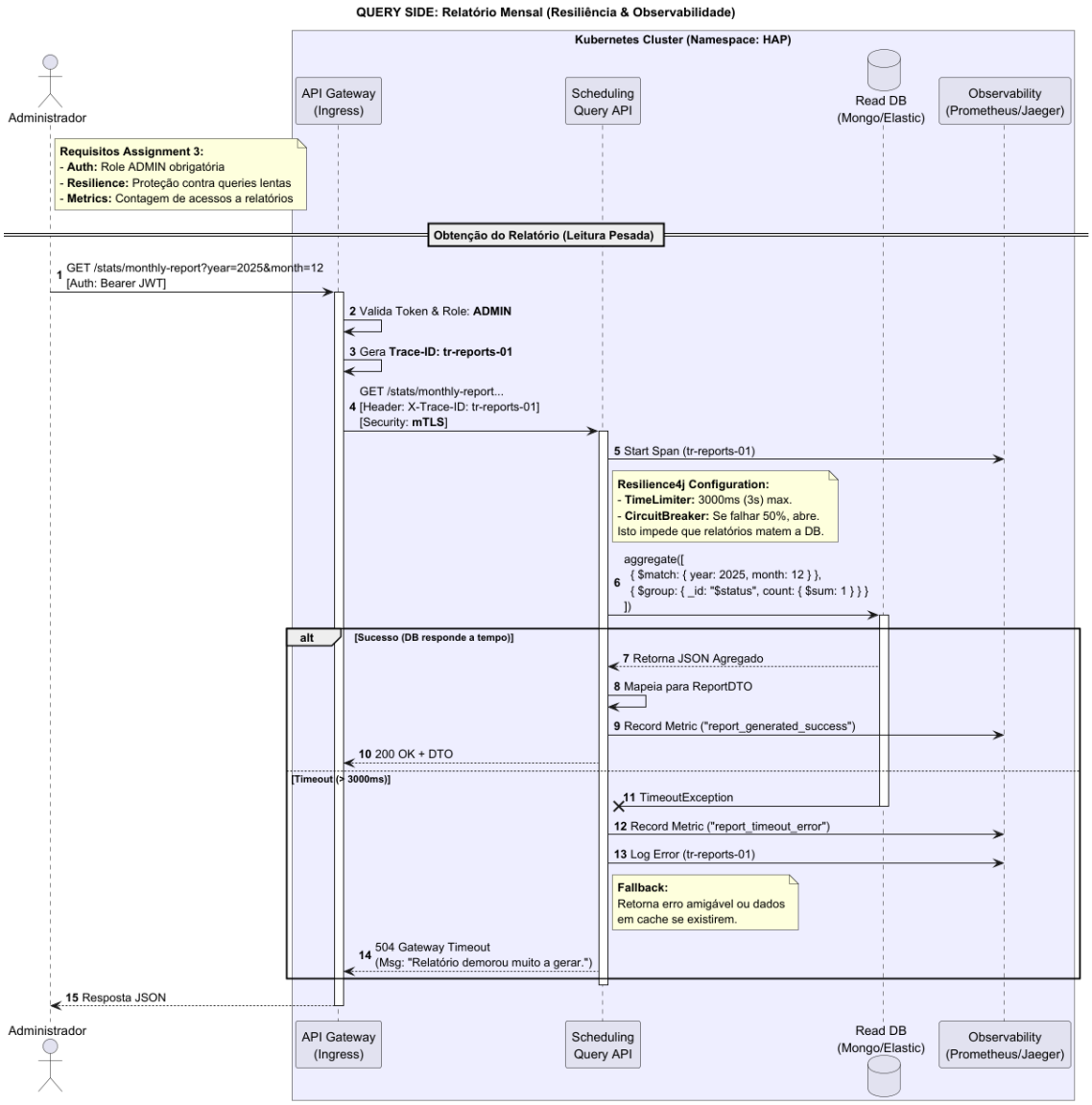
1. Proteção e Resiliência:
   • Foi configurado um TimeLimiter de 3000ms para evitar bloqueio prolongado de threads.

• O Circuit Breaker interrompe temporariamente pedidos se a taxa de falhas ultrapassar o limiar definido.
• O sistema aplica o padrão Fail Fast para preservar a estabilidade global.

2. Base de Dados de Leitura:
   • As agregações são executadas numa base de dados orientada a documentos.
   • Esta base está otimizada para operações de leitura e agregação.
3. Observabilidade:
   • São registadas métricas específicas como sucesso e falha na geração de relatórios.
   • O Trace-ID permite identificar consultas que causam latência elevada.
4. Segurança:
   • Apenas utilizadores com role ADMIN podem solicitar relatórios.
   • A comunicação interna é protegida por mTLS.

Descrição do Fluxo

1. Pedido:
   • O administrador solicita o relatório.
   • O Gateway valida permissões e gera o Trace-ID.
2. Proteção:
   • A Query API aplica as políticas de resiliência antes de aceder à base de dados.
3. Agregação:
   • A query de agregação é executada na base de leitura.
4. Resposta:
   • Em caso de sucesso, o relatório é devolvido.
   • Em caso de timeout, é retornada uma resposta controlada e registada nas métricas.



Fluxo de Detalhes do Paciente

Este diagrama detalha o processo de leitura de dados sensíveis de um paciente específico. A arquitetura prioriza a segurança, a conformidade com o GDPR e a auditoria de acessos, garantindo simultaneamente performance.

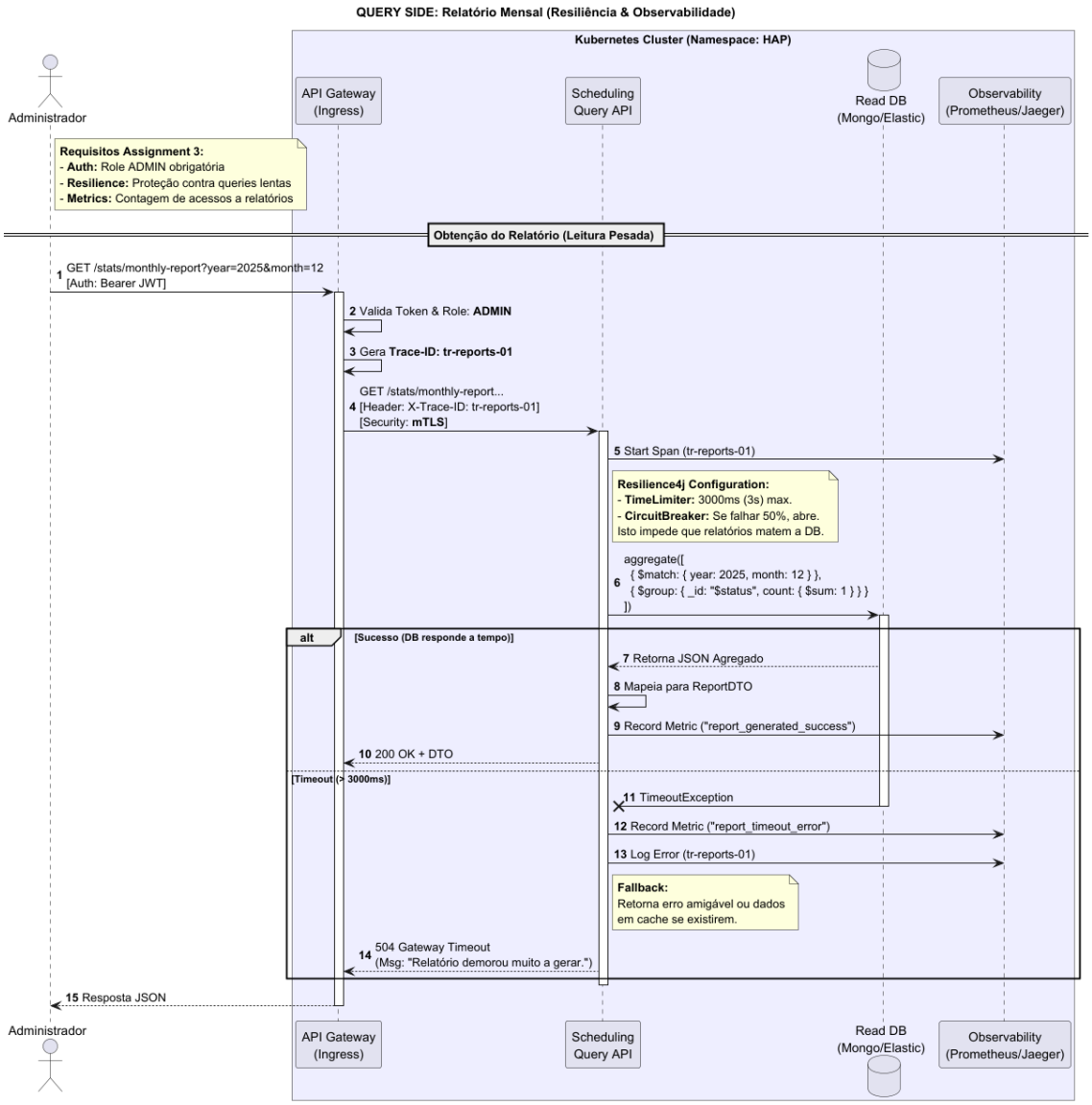
Decisões de Arquitetura e Segurança

1. Segurança e GDPR:
   • O API Gateway extrai o physicianId do token JWT e injeta-o no cabeçalho do pedido.
   • A consulta exige correspondência entre o paciente e o médico associado.
   • Em acessos não autorizados, o sistema retorna 404 para evitar enumeração de dados.

2. Auditoria:
   • Cada acesso incrementa a métrica sensitive_data_viewed.
   • O Trace-ID permite auditoria completa de acessos.
3. Resiliência:
   • Foi definido um timeout agressivo de 500ms.
   • Em caso de atraso, a ligação é interrompida imediatamente.

Descrição do Fluxo

1. Início:
   • O médico solicita os detalhes do paciente.
   • O Gateway valida a identidade e inicia o rastreio.
2. Proteção:
   • A Query API valida mTLS e políticas de resiliência.
3. Consulta:
   • É executada uma leitura por chave primária na projeção de leitura.
4. Resposta:
   • Sucesso: dados devolvidos e acesso auditado.
   • Não autorizado ou inexistente: retorno de 404.
   • Falha técnica: retorno de 503 Service Unavailable.



Fluxo de Ver Lista de Consultas de um Paciente

Este diagrama representa o processo de consulta da lista de consultas associadas a um paciente. O fluxo é implementado no Query Side da arquitetura CQRS e privilegia performance e segurança.

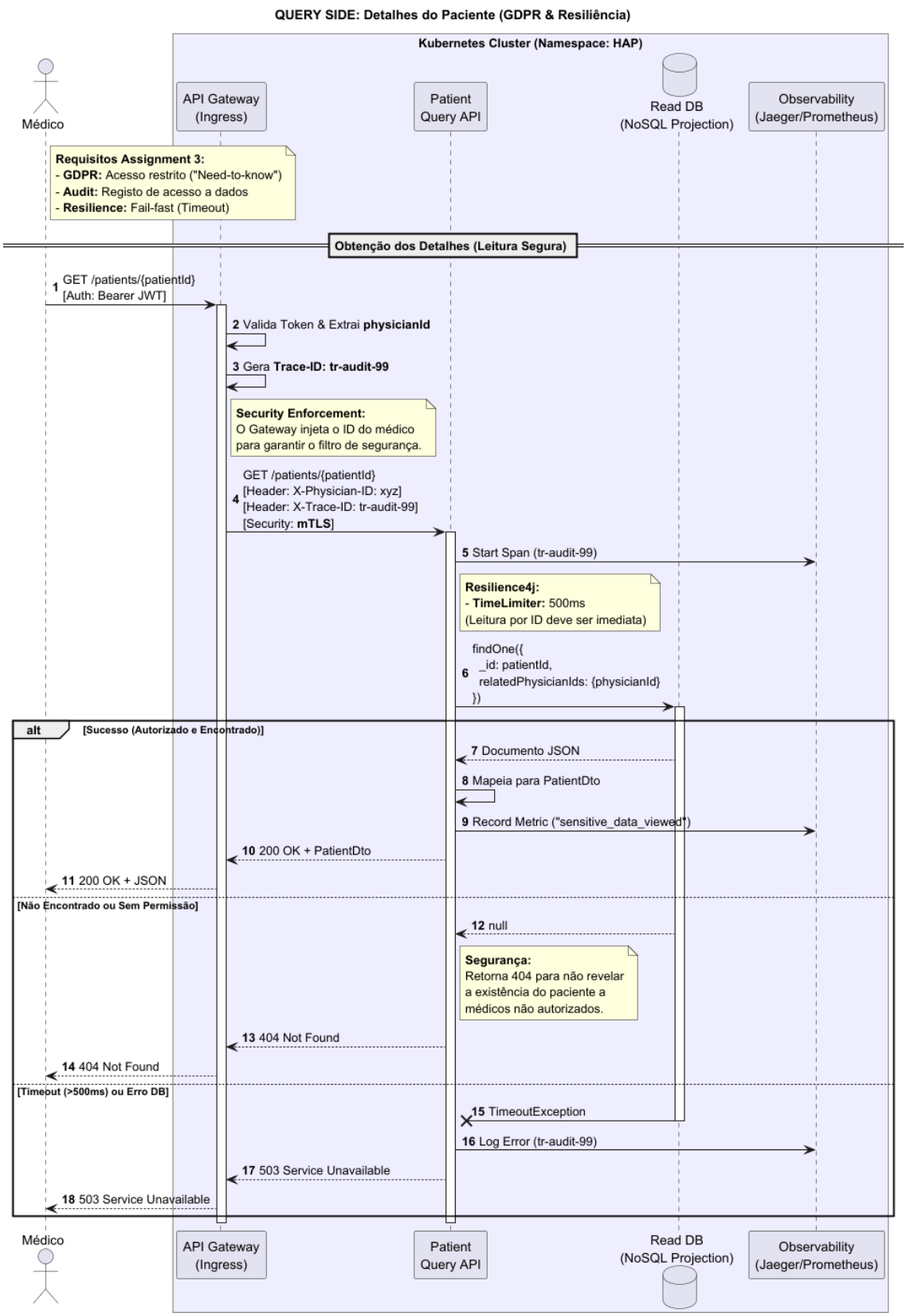
Decisões Arquiteturais

1. Separação CQRS:
   - As consultas são obtidas exclusivamente a partir da base de dados de leitura.
   - Não existe impacto direto na base de escrita.
2. Segurança:
   - O Gateway valida o token JWT.
   - Apenas o próprio paciente ou utilizadores autorizados podem aceder à lista.

3. Performance:
   • A projeção de leitura está otimizada para listagens.
   • São utilizadas queries simples e paginadas.

Descrição do Fluxo

1. Pedido:
   • O paciente solicita a lista de consultas.
   • O Gateway valida a identidade e gera o Trace-ID.
2. Consulta:
   • A Query API executa a pesquisa na base de dados de leitura.
3. Resposta:
   • A lista de consultas é devolvida ao cliente.
   • Métricas de latência e sucesso são registadas.


QUERY SIDE: Detalhes do Paciente (GDPR & Resiliência)
Kubernetes Cluster (Namespace: HAP)
Médico
API Gateway (Ingress)
Patient Query API
Read DB (NoSQL Projection)
Observability (Jaeger/Prometheus)
Requisitos Assignment 3:
- GDPR: Acesso restrito ("Need-to-know")
- Audit: Registo de acesso a dados
- Resilience: Fail-fast (Timeout)
Obtenção dos Detalhes (Leitura Segura)
1 GET /patients/{patientId}
[Auth: Bearer JWT]
2 Valida Token & Extrai physicianId
3 Gera Trace-ID: tr-audit-99
Security Enforcement:
O Gateway injeta o ID do médico para garantir o filtro de segurança.
4 GET /patients/{patientId}
[Header: X-Physician-ID: xyz]
[Header: X-Trace-ID: tr-audit-99]
[Security: mTLS]
5 Start Span (tr-audit-99)
Resilience4j:
- TimeLimiter: 500ms
(Leitura por ID deve ser imediata)
6 findOne({
_id: patientId,
relatedPhysicianIds: {physicianId}
})
alt
[Sucesso (Autorizado e Encontrado)]
7 Documento JSON
8 Mapeia para PatientDto
9 Record Metric ("sensitive_data_viewed")
10 200 OK + PatientDto
11 200 OK + JSON
[Não Encontrado ou Sem Permissão]
12 null
Segurança:
Retorna 404 para não revelar a existência do paciente a médicos não autorizados.
13 404 Not Found
14 404 Not Found
[Timeout (>500ms) ou Erro DB]
15 TimeoutException
16 Log Error (tr-audit-99)
17 503 Service Unavailable
18 503 Service Unavailable

CQRS Design

Visão geral do CQRS no projeto

A motivação para adotar CQRS resulta dos requisitos do trabalho, que exigem separação clara entre escrita e leitura para os recursos principais. Na arquitetura atual, os serviços expunham endpoints REST onde leitura e escrita partilhavam o mesmo modelo e a mesma base de dados. Na arquitetura adotada:

- commands alteram estado (criar, atualizar, cancelar)
- queries apenas leem, com modelos otimizados para pesquisa/listagem
- eventos publicam alterações após commands bem-sucedidos
- read models são atualizados por consumo de eventos, resultando em consistência eventual

Estratégia adotada

A abordagem usada é pragmática: não se criaram microserviços separados para command e query. Cada bounded context mantém-se num único microserviço, mas internamente existe separação de handlers/fluxos para commands e queries. A escrita mantém o modelo canónico e as invariantes; a leitura usa read models mais simples e otimizados.

Para cumprir o requisito de database-per-microservice-instance, cada instância mantém o seu read model local (base de dados ou schema separado). Isto permite que, mesmo com falha parcial de instâncias, outras continuem a responder com os seus read models.

Relação com eventos, broker e fan-out

Cada command bem-sucedido publica um evento de domínio num broker AMQP. Todas as instâncias consomem esses eventos e atualizam o seu read model local. O fan-out entre réplicas é usado como mecanismo de fallback em queries (quando o recurso não está no read model local, por lag ou distribuição), consultando peers por HTTP conforme a estratégia definida no projeto.

Commands e queries por serviço

Physician Service

Commands:

- register physician (POST /api/v1/physicians) publica physician registered
- update physician (PUT /api/v1/physicians/{id}) publica physician updated
- deactivate/delete physician (DELETE /api/v1/physicians/{id}) publica physician deactivated (preferencialmente soft-delete)

Queries:

- get physician details (GET /api/v1/physicians/{id}) lê do read model local
- search physicians (GET /api/v1/physicians?...) lê do read model local com índices por nome/departamento/especialização, com fan-out como fallback
- list departments/specializations (GET /api/v1/departments, GET /api/v1/specializations) lido de read model replicado

Patient Service

Commands:

- register patient (POST /api/v1/patients) publica patient registered
- update patient (PUT /api/v1/patients/{id}) publica patient updated
- deactivate patient (DELETE ou PATCH) publica patient deactivated

Queries:

- get patient details (GET /api/v1/patients/{id}) lê do read model local
- search patients by name (GET /api/v1/patients?name=...) lê de read model com índice de texto
- search by contact (GET /api/v1/patients?email=... ou phone=...) lê de read model com índices específicos

Scheduling Service

Commands:

- schedule consultation (POST /api/v1/consultations) cria em estado inicial e inicia saga; publica consultation scheduling started
- confirm consultation (POST /api/v1/consultations/{id}/confirm) publica consultation scheduled
- cancel consultation (POST /api/v1/consultations/{id}/cancel) publica consultation cancellation requested e consultation cancelled
- update consultation details (PUT /api/v1/consultations/{id}) publica consultation updated

Queries:

- get consultation by id (GET /api/v1/consultations/{id}) lê do read model local
- list patient consultations (GET /api/v1/consultations?patientId=...) lê do read model indexado por paciente e tempo
- list physician consultations (GET /api/v1/consultations?physicianId=...) lê do read model indexado por médico e tempo
- reports (GET /api/v1/consultations/reports/...) lê de read model analítico alimentado por eventos

Clinical Records Service

Commands:

- create record (POST /api/v1/records) publica consultation record created
- update record (PUT /api/v1/records/{id}) publica consultation record updated
- void records for cancelled consultation (evento-driven) publica consultation records voided

Queries:

- get record by id (GET /api/v1/records/{id}) lê do read model local
- search records (GET /api/v1/records?consultaId=&patientId=&physicianId=...) lê de read model com índices
- export patient records (GET /api/v1/records/export?patientId=...) lê de read model preparado para export

Write model vs read model

O write model mantém as entidades canónicas e invariantes de negócio, aplicando transições de estado e validações. Após persistência, publica eventos. O read model é atualizado por consumo de eventos e é otimizado para queries frequentes (listagens, filtros e relatórios). Cada instância mantém um read model local, suportando consistência eventual.

Exemplos end-to-end

Agendar consulta e listar consultas:

- command schedule consultation publica evento
- saga valida physician e patient e publica consultation scheduled
- instâncias atualizam read models por consumo do evento
- query lista consultas lê do read model local (podendo estar temporariamente desatualizado)

Criar registo clínico e pesquisar por paciente:

- command create record valida estado e IDs e publica evento
- instâncias atualizam read models com índices por patientId/physicianId/consultaId
- queries lêem localmente; fan-out pode ser usado como fallback

Messaging and Broker (AMQP)

Broker e papel no sistema

O broker utilizado é RabbitMQ. Os serviços ligam-se como produtores e/ou consumidores. O broker é usado para:

- propagação de eventos de domínio (fire-and-forget)
- atualização de read models locais em cada instância
- coordenação de sagas por eventos
- desacoplamento entre serviços e eliminação de cascatas síncronas

Modelo lógico de exchanges, filas e routing

A estratégia usa exchanges topic por domínio:

- hap.patients, hap.physicians, hap.consultations, hap.records
  As routing keys seguem o padrão domínio.ação (ex.: patient.registered, consultation.scheduled). Cada instância mantém a sua fila, permitindo consumo independente e atualização do seu read model local.

Estrutura das mensagens

Mensagens em JSON com envelope comum:

- eventId, eventType, occurredAt, sourceService, payload
  Headers principais:
- x-correlation-id para correlação ponta-a-ponta
- x-saga-id para identificar instância de saga
- x-retry-count para debug e testes

Semântica:

- entrega at-least-once, exigindo consumidores idempotentes (ex.: registo de eventId processados, upsert, on conflict do nothing)

Eventos por serviço

Patient publica: patient registered, updated, deactivated
Physician publica: physician registered, updated, deactivated
Scheduling publica: scheduling started, scheduled, updated, cancellation requested, cancelled, done

Scheduling consome: eventos de validação (patient validated/failed, physician confirmed/rejected) e resposta de records voided
Clinical records publica: record created, updated, records voided
Clinical records consome: cancellation requested

Sagas suportadas

Agendar consulta:

- scheduling publica scheduling started
- patient valida e publica validação
- physician valida e publica confirmação/rejeição
- scheduling recebe ambos e publica scheduled ou falha equivalente

Cancelar consulta:

- scheduling publica cancellation requested
- clinical records voids e publica records voided
- scheduling finaliza e publica cancelled

CQRS e atualização de read models

Cada instância consome eventos e atualiza o seu read DB local. Isto introduz consistência eventual: existe atraso esperado entre write model e read model.

Fiabilidade e DLQ

ACK manual após consumo bem-sucedido. Em falhas recuperáveis, NACK com requeue; em falhas permanentes, encaminhamento para DLQ por fila (ex.: scheduling-service.instance-1.events.dlq), permitindo inspeção e resolução manual.

Testes e validação

- publicar eventos a partir de commands e verificar queues
- confirmar que múltiplas instâncias consomem e atualizam read models
- validar sagas em sucesso e falha
- validar idempotência reenviando a mesma mensagem

Deployment Multi-Instance

Estratégia

O ambiente de entrega privilegia Docker Compose. Cada instância tem:

- porta distinta

- fila distinta no broker
- read DB/schema distinto (database-per-instance para leitura)

Load balancing no gateway

O gateway mantém lista de instâncias por serviço e distribui requests com estratégia simples (round-robin/random/least-connections). Quando uma instância falha, o gateway encaminha para as restantes.

Arranque e shutdown

Ordem recomendada:

1. broker
2. read DBs
3. instâncias dos serviços
4. gateway
   Após reinício, as instâncias voltam a sincronizar o read model consumindo eventos.

Testes distribuídos

- balanceamento: múltiplos GET para verificar alternância de instâncias
- eventos: command gera evento e todas as instâncias consomem
- consistência eventual: atraso aceitável entre command e query
- falha parcial: desligar uma instância e manter funcionalidade
- sagas: sucesso e falha com validações e cancelamento com records

Observability

Logging centralizado

Arquitetura típica:
microserviços/gateway -> coletor (fluent bit/fluentd) -> storage (elastic/opensearch/loki) -> UI (kibana/grafana)

Os logs são estruturados e incluem:
timestamp, level, service, instanceId/pod, correlationId, traceId/spanId (quando disponível), message, e opcionalmente eventId/messageId/sagaId. Informação sensível não é registada.

Validação:

- executar um request (ex.: POST /consultas)
- pesquisar por correlationId e seguir o fluxo nos logs do gateway, serviço e consumers AMQP

- simular falha de instância e observar logs de erro/fallback

Métricas (Prometheus + Grafana)

Métricas recolhidas:

- HTTP: requests, latência p50/p95/p99, erros 4xx/5xx
- runtime/jvm: cpu, memória, threads, gc
- DB: conexões, tempos médios (quando disponível)
- RabbitMQ: publish/consume rate, queue depth, DLQ count

Dashboards:

- visão geral do sistema
- dashboards por microserviço
- dashboard de mensageria

Validação:

- targets UP no Prometheus
- gerar tráfego e confirmar contadores/latência
- induzir falha e observar aumento de erros/latência

Tracing distribuído (Jaeger/OpenTelemetry)

A instrumentação cria spans por request no gateway, propaga contexto por headers HTTP e metadata AMQP e regista spans por endpoints, publish/consume e DB calls.

Validação:

- executar POST /consultas
- encontrar trace em Jaeger e confirmar spans de gateway, serviços, broker e DB
- induzir falha e ver spans marcados como erro

Health checks

Liveness para verificar aplicação viva; readiness para verificar dependências (DB e broker). Em falha de dependências, a instância é retirada do tráfego.

Resilience and Fault Tolerance

Padrões aplicados

Timeout:

- aplicado em chamadas HTTP inter-serviços, publish/consume AMQP e operações de DB via pools/config
- objetivo: fail fast e evitar bloqueio de threads

Retry:

- usado em operações idempotentes (GET, validações, consumo de mensagens)
- retries limitados e backoff
- não aplicado em operações não idempotentes sem idempotência explícita

Circuit breaker:

- protege chamadas críticas e evita repetição de chamadas a serviço degradado
- estados closed/open/half-open e integração com logs/métricas/tracing

Bulkhead:

- separação de recursos por tipo de operação (command/query, HTTP/AMQP)
- limites explícitos de concorrência para impedir saturação global

Fallback:

- comportamento controlado em queries e fluxos onde faz sentido (ex.: resposta degradada, lista vazia, erro controlado)
- em sagas, fallback traduz-se em compensação/rollback lógico

Cenários de falha para demo

- desligar physician-service e observar timeout/retry/circuit breaker e recuperação
- falha durante saga e compensação com estado final consistente
- falha em consumer e DLQ sem bloquear o restante processamento

Security

User-to-service

O identity service emite JWT com roles/scopes/claims. O cliente envia bearer token. O gateway e serviços validam assinatura, expiração e claims.

Autorização:

- enforcement no gateway e reforço em cada serviço
- endpoints read requerem *.read; endpoints write requerem *.write; regras adicionais por role e ownership

Regras por serviço

Scheduling:

- criar consulta: patient/admin + consultations.write
- cancelar/confirmar: patient (própria) ou admin + consultations.write
- listar: patient só as suas; doctor as suas; admin todas + consultations.read

Clinical records:

- criar/editar: doctor/admin + records.write, com validação de consulta e relação médico-consulta
- ler: patient só seus; doctor só das suas consultas; admin todos + records.read

Patient:

- write: admin + patients.write
- read: patient só o próprio; doctor/service apenas quando necessário + patients.read

Physician:

- write: admin + physicians.write
- read: patient/doctor/admin + physicians.read

Service-to-service e mTLS

Chamadas internas incluem JWT com role SERVICE e scopes mínimos. mTLS garante autenticação mútua e impede chamadas internas não autorizadas.

RabbitMQ:

- credenciais e permissões por exchange/queue
- mensagens com correlationId/traceId e validação de formato e idempotência nos consumidores

Gestão de secrets

Sem hardcode e sem secrets em repo. Secrets via env vars e/ou Kubernetes Secrets (DB, RabbitMQ, chaves JWT, certificados mTLS). Nunca expostos em logs.

Validação

- sem token: 401
- token inválido/expirado: 401
- token válido sem permissões: 403

- mTLS inválido: falha handshake
- consumers rejeitam mensagens inválidas e duplicadas não causam efeitos colaterais

Deployment, CI/CD and Governance

Estratégia de Deployment

O deployment do sistema foi desenhado para suportar um ambiente distribuído com múltiplas instâncias por microserviço, comunicação assíncrona via broker e consistência eventual.

Os princípios adotados foram:

- cada microserviço é implantado de forma independente,
- o sistema continua funcional perante falhas parciais,
- atualizações não causam indisponibilidade global.

Dependendo do ambiente, são suportados dois modos principais:

- Docker Compose, usado para desenvolvimento, testes e demonstração académica,
- Kubernetes, usado como referência arquitetural para ambientes orquestrados.

Em ambos os casos, o comportamento esperado do sistema mantém-se equivalente.

Deployment com Docker

No contexto académico do projeto, Docker é utilizado como tecnologia base para empacotamento e execução dos serviços.

Cada microserviço é distribuído como:

- uma imagem Docker independente,
- configurada exclusivamente através de variáveis de ambiente,
- sem dependências de estado local não controlado.

A utilização de múltiplas instâncias por serviço permite:

- demonstrar escalabilidade horizontal,
- validar balanceamento de carga,
- testar cenários de falha controlada.

O API Gateway atua como ponto único de entrada, distribuindo tráfego entre instâncias disponíveis.

CI/CD Pipeline

O pipeline de CI/CD automatiza o ciclo de vida do software desde a alteração de código até à geração de artefactos prontos a executar.

O pipeline segue uma abordagem incremental e segura:

- cada commit desencadeia validações automáticas,
- erros são detetados o mais cedo possível,
- apenas código validado é empacotado e distribuído.

As principais responsabilidades do pipeline são:

- garantir que o código compila,
- garantir que testes essenciais passam,
- gerar imagens Docker versionadas,
- disponibilizar artefactos prontos para deploy.

Mesmo quando o deploy automático não é aplicado (por opção académica), o pipeline garante que o sistema é sempre reproduzível.

Governance do Deployment

As decisões de deployment são governadas por convenções explícitas, documentadas e aplicadas de forma consistente.

Estas convenções reduzem:

- ambiguidade na operação,
- erros humanos,
- divergência entre ambientes.

A governance assegura que:

- todos os serviços seguem o mesmo padrão,
- novas funcionalidades integram-se sem quebrar o sistema,
- a evolução do sistema é controlada e previsível.

Conclusão

Este Documento Arquitetural apresenta a solução desenvolvida para o sistema HAP, descrevendo as principais decisões técnicas e arquiteturais adotadas ao longo do projeto.

A arquitetura assenta em microserviços independentes, comunicação assíncrona por eventos e na aplicação de padrões como CQRS, Sagas, observability e resiliência. Estas decisões permitem suportar múltiplas instâncias, tolerar falhas parciais e garantir um funcionamento previsível em ambientes distribuídos.

Foram identificados e testados *edge cases* relevantes, como falhas de comunicação, indisponibilidade parcial de serviços, carga elevada e duplicação de mensagens. Os mecanismos de timeout, retry, circuit breaker, bulkhead e compensação demonstraram ser eficazes na contenção de falhas e na manutenção da consistência eventual do sistema.

As limitações conhecidas foram assumidas de forma consciente e documentadas, nomeadamente a visibilidade temporária da consistência eventual e a necessidade de intervenção manual em cenários extremos. As lições aprendidas reforçam a importância da observabilidade e do desenho arquitetural orientado à falha em sistemas distribuídos.

No seu conjunto, a solução cumpre os requisitos, sendo tecnicamente coerente, e alinhada com boas práticas de engenharia de software distribuído.