INF INSTITUTO DE INFORMÁTICA

UFG UNIVERSIDADE FEDERAL DE GOIÁS

Sistemas Distribuídos - 2025/2

CHAT4ALL
PLATAFORMA DE COMUNICAÇÃO UBÍQUA

Arthur Faria Peixoto
Geovanna Cunha Andrade Silva
Guilherme Ferreira de Oliveira
Sergio Natan Costa Barbosa

Prof. Vagner Sacramento
arthurfpeixoto, geovanna.cunha, ferreiraguilherme, sergionatan
@discente.ufg.br

Sumário

Arquitetura

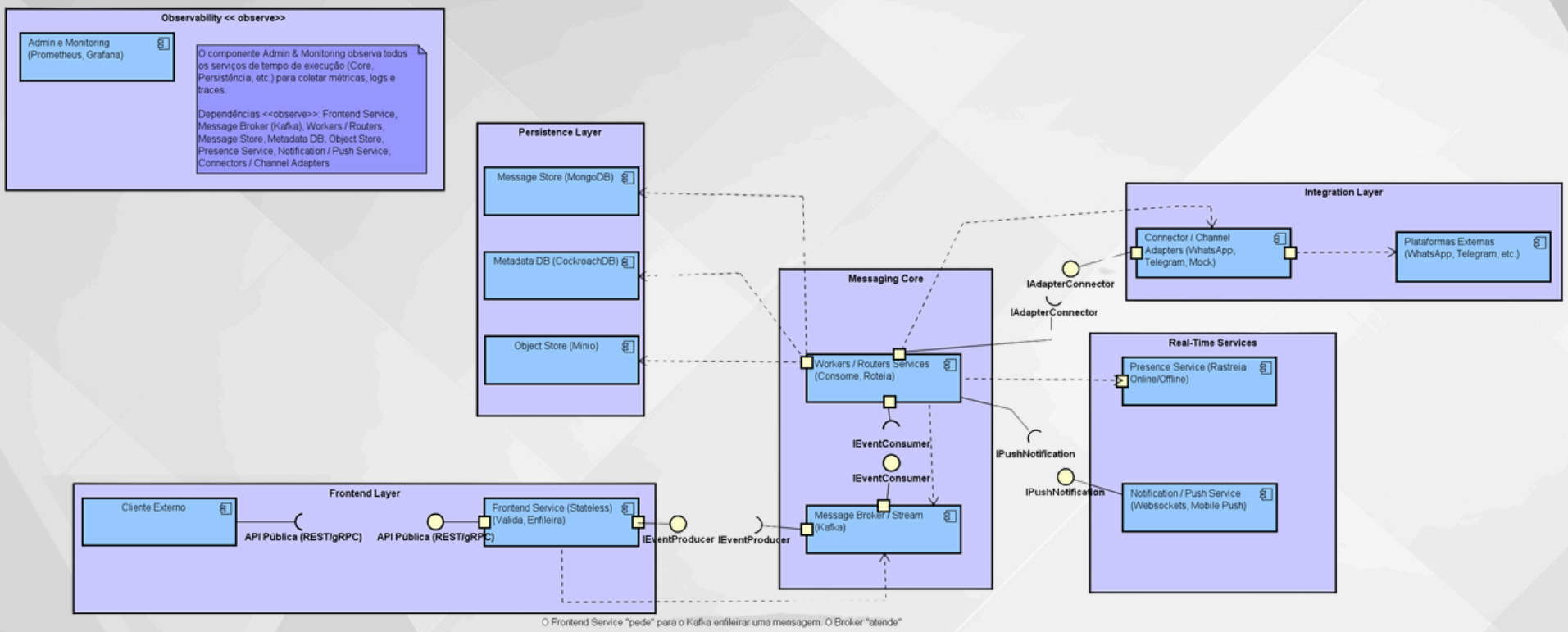


Diagrama de Componentes

O Chat4All é uma API ubíqua que permite que um usuário interaja (envie e receba) com todos os seus contatos, não importando a plataforma que eles usem, a partir de um único ponto.

Arquitetura

Diagrama de Contexto

Modelagem de Dados - Arquitetura de Persistencia

CockroachDB

O CockroachDB é o banco relacional distribuído responsável por armazenar os dados estruturados e críticos do sistema, como usuários, conversas, membros, permissões e configurações globais. Ele foi escolhido por combinar a familiaridade do SQL com a escalabilidade e a tolerância a falhas típicas de bancos distribuídos. Sua arquitetura baseada em Raft garante consistência forte e failover automático, tornando-o ideal para informações que exigem integridade e transações ACID.

Como trade-off, ele apresenta maior latência em operações de escrita em comparação a bancos NoSQL, pois cada operação precisa passar por consenso entre réplicas. Além disso, o custo operacional e a curva de aprendizado são mais altos do que os de um PostgreSQL tradicional, especialmente em clusters grandes.

Mongo DG

O MongoDB é usado para armazenar mensagens, estados de entrega e leitura, cursores e metadados de arquivos. Ele foi escolhido por sua capacidade de lidar com altíssimos volumes de escrita e leitura de forma escalável e por sua flexibilidade de schema — essencial em sistemas de mensageria, onde o formato das mensagens pode evoluir com o tempo.

O ponto forte do MongoDB é o desempenho e a facilidade para sharding horizontal, mas ele sacrifica consistência imediata: a replicação é eventual. Isso é aceitável em um cenário de mensagens, onde uma leitura ligeiramente desatualizada não compromete a experiência. Por outro lado, ele é menos eficiente para consultas complexas ou relacionamentos e exige cuidado no desenho dos índices e shards para evitar gargalos.

Modelagem de Dados - Arquitetura de Persistencia

Redis

O Redis atua como armazenamento em memória para dados transitórios, como status de presença, caches, locks distribuídos e informações temporárias de roteamento. Ele foi escolhido por oferecer latência extremamente baixa e suportar padrões nativos de pub/sub, expiração automática (TTL) e estruturas de dados leves.

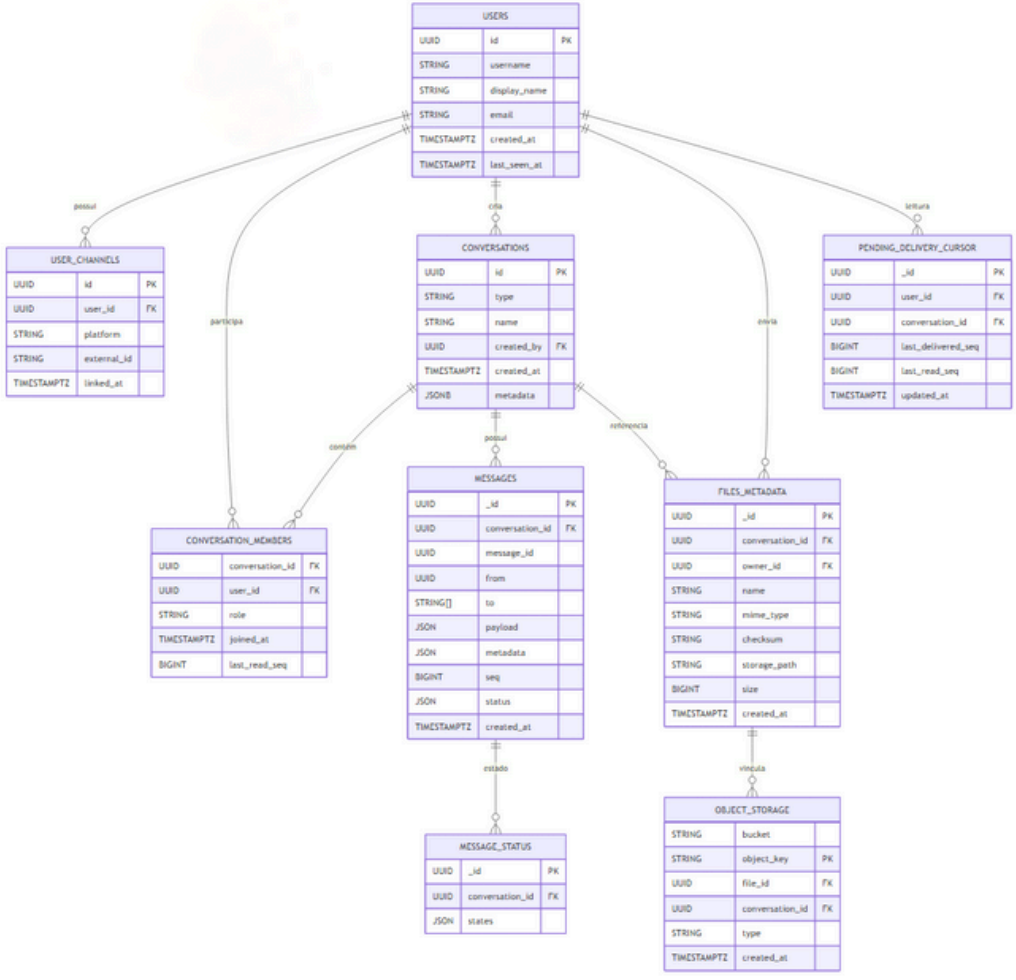
Seu principal trade-off é a volatilidade: por ser in-memory, qualquer dado não persistido pode ser perdido em caso de falha. Isso é aceitável porque as informações armazenadas nele (como “usuário online” ou “lock ativo”) são de natureza efêmera. Contudo, ele não serve para dados relacionais nem históricos e requer atenção ao uso de memória e políticas de expiração para não crescer indefinidamente.

MinIO

O MinIO é usado para armazenar arquivos grandes — como imagens, vídeos e áudios de até 2 GB. Ele segue o padrão S3, permitindo integração fácil com bibliotecas e SDKs existentes. Sua escolha foi motivada pela necessidade de lidar com grandes volumes de dados binários sem sobrecarregar os bancos transacionais, e por suportar uploads fragmentados e retomáveis, que reduzem falhas em transferências.

Como trade-off, o MinIO não permite consultas estruturadas nem relacionamentos; ele apenas guarda os objetos binários, exigindo que metadados (como tamanho, checksum ou dono do arquivo) sejam salvos em outro banco, como o MongoDB. Embora seja open-source e leve, sua maturidade operacional ainda é inferior a serviços de nuvem como o S3 da AWS em grandes implantações.

Relacionamento de Entidades



Integridade do Sistema

1. **Idempotência Global:** O message_id deve ser exclusivo, garantindo a deduplicação durante o processo de escrita.
2. **Ordem por Conversa:** O par (conversation_id, seq) deve ser único e seguir uma sequência crescente.
3. **Cursor de Leitura:** É fundamental assegurar que last_read_seq ≤ last_delivered_seq ≤ max(seq) para cada usuário, mantendo a coerência na leitura das mensagens.
4. **Integridade de Arquivo:** O campo files_metadata.checksum deve estar presente, e o storage_path deve ser validado para garantir a integridade dos arquivos.

Checks de Saúde

- **Gaps de Sequência:** Não deve haver lacunas na sequência de conversation_id; devem ser iguais a 0.
- **Duplicatas de message_id:** Não devem existir, mantendo este número em 0.
- **Consistência do Cursor de Entrega:** O pending_delivery_cursor deve estar alinhado com conversation_members.last_read_seq.

Chave de Partição Principal

- **conversation_id:** Utilizado em tópicos/partições do Kafka e shards do Mongo, garantindo ordem causal e paralelismo.

Por que Funciona

- Mensagens da mesma conversa são direcionadas para a mesma partição, assegurando uma sequência linear por conversa.
- Diferentes conversas são alocadas em partições distintas, permitindo escalabilidade horizontal**.**

Mitigação de Hot-Partitions

- Uso de round-robin para distribuir partições físicas, mantendo o roteamento lógico baseado em conversation_id.
- Implementação de auto-split em shards "quentes" no Mongo, com base em limites predefinidos.
- Aplicação de limites de taxa por conversa e controle de pressão nos workers.

Índices Essenciais

- messages: { conversation_id, seq } (único)
- pending_delivery_cursor: { user_id, conversation_id }
- files_metadata: { conversation_id, created_at }

Observabilidade

Ferramentas:

- OpenTelemetry Collector: ponto único de entrada; padroniza rótulos; aplica filtros/redação; roteia métricas → Prometheus e logs → Loki.
- Prometheus: coleta métricas do Collector e de exporters de infra (Kafka/Redis/MongoDB/MinIO); executa regras de alerta; fonte para painéis de desempenho e SLO/SLI.
- Loki: recebe somente logs estruturados via Collector; consulta por rótulos (service, env, connector, message_id); investigação rápida de incidentes.
- Grafana:
  - Dashboards: Executivo (TPS/erros/p95), API, Workers, Connectors, Infra.
  - Exploração: partir de uma série de métricas → abrir logs no Loki filtrando por service/env e refinando por trace_id/message_id.
  - Governança: datasources, dashboards e permissões versionados “as code”.

Métricas:

- API
  - Taxa de requisições
  - Latências p95/p99 por rota
  - Códigos de erro 4xx/5xx
- Connectors
  - Sucessos/erros por connector/canal
  - Tempo de entrega
- Workers
  - Processados por segundo
  - Duração do processamento
  - Retries/falhas
- Infraestrutura
  - Saúde, latência e uso de Kafka, Redis, MongoDB, MinIO e K8s (exporters)

SLO/SLI

- **Disponibilidade da API:** ≥ 99,95%
- **P95:** < 200 ms
- **Erros 5xx:** < 1%
- **Tempo de entrega por conector:** dentro do alvo do canal

Os dashboards mostram o cumprimento das metas; os alertas são acionados quando os objetivos são superados; e os post-mortems utilizam métricas junto com logs correlacionados.

Por Que NestJS?

Arquitetura & Escalabilidade

Modularidade (Plugins): Isola cada Conector (WhatsApp, Telegram) como um módulo, facilitando o desenvolvimento e a manutenção. {Módulos, Providers, Controllers}

Comunicação Distribuída

Suporte Nativo a Microserviços: Adaptadores prontos para comunicação assíncrona e resiliente. {Adaptadores para Kafka, gRPC, Redis, NATS}

Persistência de Dados

Integração Simplificada com as principais tecnologias do projeto. {@nestjs/mongoose (para MongoDB) e Prisma (para CockroachDB).}

Desenvolvimento (DX)

Código Previsível e Testável: Usa TypeScript, Injeção de Dependência e padrões de design bem definidos. {TypeScript, Pipes, Interceptors, Guards}

Armazenamento de Arquivos

Fácil Integração S3: Serviço dedicado para gerenciamento de arquivos grandes. {SDKs S3/MinIO em um módulo Nest.}

Esqueleto do Projeto

```
/
├── .git/
├── README.md                    # Instruções de Setup e Deploy
├── docker-compose.yml # Orquestração local (Kafka, Mongo, MinIO, CockroachDB, Prometheus/Grafana)
├── k8s/ # Manifestos para Deploy em Produção (se aplicável, para alta disponibilidade)
├── ops/ # Configurações de Operação e Observabilidade
│   ├── prometheus/
│   │   └── prometheus.yml          # Configuração de coleta de métricas
│   ├── grafana/
│   │   ├── dashboards/             # Definições de Dashboards (versionados "as code")
│   │   └── datasources/
│   ├── otel-collector/
│   │   └── config.yaml             # Configuração do OpenTelemetry Collector
│   └── database-init/
│       ├── 01-cockroach-init.sql   # Scripts de inicialização (Schema CockroachDB)
│       └── 02-mongo-init.js        # Scripts de inicialização (Índices MongoDB)
├── services/ # Microsserviços (Baseados em NestJS)
│   ├── gateway-api/ # 1. API Externa (Produtor Kafka, Ponto de entrada)
│   │   └── src/
│   │       ├── main.ts             # Usa Fastify Adapter para performance
│   │       ├── messages/ # Módulo para POST /v1/messages
│   │       ├── files/ # Módulo para Upload Resumable (POST /v1/files/initiate)
│   │       ├── auth/ # Módulo para autenticação
│   ├── router-worker/ # 2. Worker de Roteamento (Consumidor Kafka, Lógica principal)
│   │   └── src/
│   │       ├── main.ts             # Microservice Transport (Kafka, gRPC)
│   │       ├── message-processor/ # Lógica de persistência (MongoDB, CockroachDB) e deduplicação
│   │       ├── connector-router/ # Serviço que decide qual Connector chamar
│   ├── channel-connectors/ # 3. Adapters Plugáveis (Módulos/Serviços independentes)
│   │   └── src/
│   │       ├── whatsapp-connector/ # Módulo/Microserviço WhatsApp
│   │       ├── telegram-connector/ # Módulo/Microserviço Telegram (Integração real)
│   │       ├── mock-connector/ # Módulo/Microserviço Mock (Para testes cross-channel)
│   │       └── shared/ # Interface comum para adapters (sendFile, sendMessage, onWebhookEvent)
│   ├── storage-service/ # 4. Serviço dedicado ao MinIO/S3
│   │   └── src/
│   │       └── s3-client/ # Lógica de interação com MinIO para chunked upload/serving
└── presence-service/ # 5. Gerenciamento de status online/offline no Redis
    └── src/
        └── redis-manager/ # Interação com Redis para latência ultrabaixa
```

INF
INSTITUTO DE INFORMÁTICA

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

CHAT DISTRIBUÍDO
COM COMUNICAÇÃO VIA SOCKET

Sistemas Distribuídos - 2025/2