# Relatório Técnico - Chat4All v2: Plataforma de Mensageria Unificada

Autores: Erik Pereira das Neves, Lucas Mota da Silva Lobo
Data: 29/11/2025

## 1. Introdução e Objetivos

### 1.1 Contexto do Projeto

O Chat4All v2 é uma plataforma de mensageria unificada desenvolvida para rotear mensagens entre clientes internos e múltiplas plataformas externas (WhatsApp, Instagram, Telegram) com garantias estritas de entrega e ordem. O sistema foi projetado para alta escala, tolerância a falhas e suporte a arquivos de grande porte.

### 1.2 Objetivos Principais

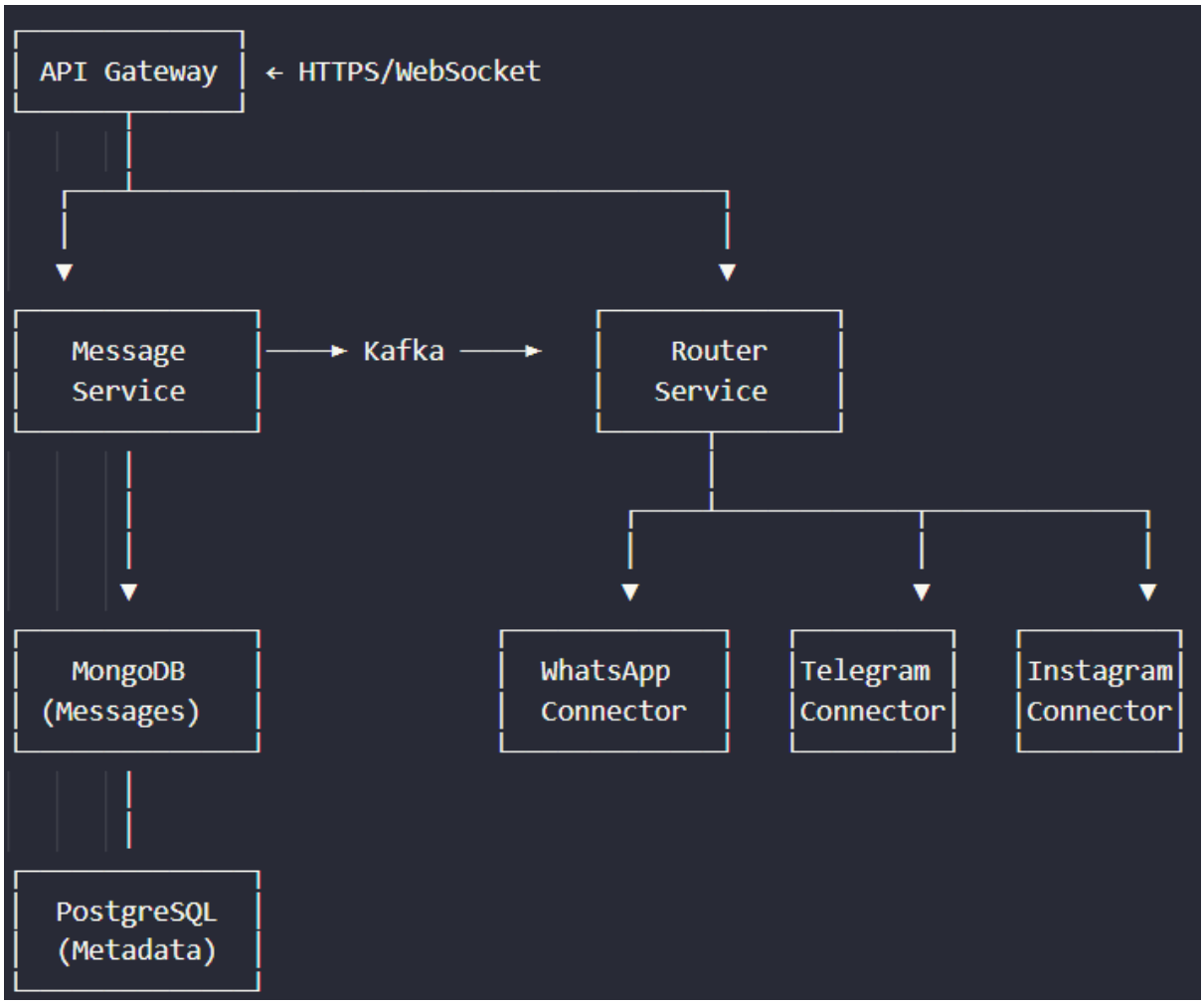
- Unificação: Integrar múltiplas plataformas de mensagem em uma única interface
- Escalabilidade: Suportar 10.000 conversas simultâneas sem degradação de performance
- Confiabilidade: Garantir entrega de mensagens com taxa de sucesso de 99,95%
- Performance: Entregar 95% das mensagens em até 5 segundos
- Resiliência: Implementar mecanismos de retry e recuperação automática

### 1.3 Escopo Funcional

- Envio e recebimento de mensagens de texto
- Upload e download de arquivos até 2GB
- Suporte a conversas 1:1 e em grupo
- Integração com WhatsApp, Telegram e Instagram
- Rastreamento completo do status das mensagens

## 2. Arquitetura Final Implementada

## 2.1 Visão Geral da Arquitetura




## 2.2 Componentes Principais

### 2.2.1 Message Service

- Porta 8081
- Responsabilidade: Lógica central de processamento de mensagens
- Tecnologia: Spring Boot Reactive (WebFlux)
- Funcionalidades:
    - Aceitação e validação de mensagens
    - Persistência em MongoDB
    - Atualização de status (PENDING → SENT → DELIVERED → READ)
    - Processamento de mensagens inbound/outbound

### 2.2.2 Router Service

- Responsabilidade: Roteamento inteligente para plataformas externas
- Funcionalidades:
    - Determinação do connector baseado no canal
    - Delegação para RetryHandler com exponential backoff
    - Atualização assíncrona de status via Kafka

#### 2.2.3 File Service

- Responsabilidade: Gerenciamento de arquivos com Object Storage
- Tecnologia: AWS SDK v2 + S3 Presigner
- Funcionalidades:
    - Geração de URLs pré-assinadas para upload/download
    - Cleanup automático de arquivos

#### 2.2.4 API Gateway

- Porta: 8080
- Responsabilidade: Roteamento, autenticação e rate limiting
- Funcionalidades:
    - OAuth2 para autenticação
    - Validação de requests
    - Rate limiting baseado em Redis

## 2.3 Infraestrutura de Dados e Mensageria

1. **Apache Kafka (Event Bus):** Organizado em tópicos estratégicos para garantir ordem e vazão.
    - *chat-events:* 10 partições, retenção de 7 dias (Ciclo de vida da mensagem).
    - *chat-events-dlq:* Fila de mensagens mortas para processamento falho.
    - Vantagens:
        - Escalabilidade Elástica: Pode aumentar partições conforme carga
        - Durabilidade: Mensagens não se perdem mesmo com falhas
        - Replay Capacity: Pode reprocessar eventos passados

2. **MongoDB (Message Store):** Armazena o histórico de mensagens e metadados de conversas. Foi configurado com índices compostos (conversation_id, timestamp) para otimizar a leitura do histórico de chat.
    - Vantagens:
        - Performance de Leitura: Índices compostos otimizam consultas de histórico

    - Schema Flexibility: Adapta-se a mudanças nos dados de mensagem
    - Sharding Ready: Preparado para escala horizontal

3. **Redis (Cache & State):** Utilizado para controle de idempotência (TTL 7 dias), sessões de WebSocket ativas e estado dos *Circuit Breakers*.
    - Vantagens:
        - Velocidade Sub-milissegundo: Ideal para controle de estado
        - TTL Automático: Não precisa de cleanup manual
        - Resiliência: Circuit breakers previnem cascade failure

# 3. Decisões Técnicas

## 3.1 Arquitetura Reativa com Spring WebFlux

Decisão: Utilizar Spring WebFlux e programação reativa com Project Reactor
Justificativa:

- Escalabilidade: Suporte a 10.000+ conexões concorrentes com recursos mínimos
- Melhor utilização de recursos para I/O intensivo
- Suporte nativo a concorrência sem bloqueio de threads

## 3.2 Idempotência com Redis

Decisão: Implementar verificação de duplicatas usando Redis
Justificativa:

- Prevenção de processamento duplicado em caso de retransmissões
- Baixa latência para verificações frequentes
- TTL automático para cleanup

Evidência no Código:

```
public Mono<Message> acceptMessage(Message message) {
    // Start timer for message processing latency (T112)
    Timer.Sample sample = Timer.start(meterRegistry);

    // Generate message ID if not provided (handles null or blank)
    if (message.getMessageId() == null || message.getMessageId().trim().isEmpty()) {
        String generatedId = UUID.randomUUID().toString();
        message.setMessageId(generatedId);
        log.debug(format: "Generated new messageId: {}", generatedId);
    }

    final String messageId = message.getMessageId();

    // Idempotency check (FR-006)
    return idempotencyService.isDuplicate(messageId)
        .flatMap(isDuplicate -> {
            if (isDuplicate) {
```

## 3.3 Event-Driven Architecture com Kafka (Backbone)

Decisão: Usar Apache Kafka como backbone de eventos entre microsserviços.
Justificativa:

- Desacoplamento: Produtores (API) não dependem da disponibilidade imediata dos consumidores.
- Ordenação: Uso do conversation_id como chave de partição garante que mensagens de uma mesma conversa mantenham a ordem causal.
- Replay: Capacidade de reprocessar eventos passados em caso de falha sistêmica.

## 3.4 Object Storage e Uploads (Presigned URLs)

Para suportar arquivos grandes (até 2GB) sem onerar a memória da aplicação, adotamos o padrão de upload direto via cliente (Client-side direct upload).

- **Decisão:** Upload direto para S3/MinIO usando Presigned URLs.
- **Justificativa:** Redução drástica de carga de I/O no backend e suporte a upload *resumable* (Multipart).
- **Fluxo:** O cliente solicita uma URL assinada à API e envia o binário diretamente ao Object Storage, garantindo segurança com expiração temporal da URL.

## 3.5 Padrão Strategy para Connectors

Decisão: Implementar roteamento baseado em Strategy Pattern
Justificativa:

- Facilidade de adicionar novos canais
- Isolamento de falhas por connector
- Configuração específica por plataforma

Evidência no Código:

```
    private String getConnectorUrl(Channel channel) {
        if (channel == null) {
            log.warn(msg: "Message channel is null, cannot determine connector");
            return null;
        }

        return switch (channel) {
            case WHATSAPP -> WHATSAPP_CONNECTOR_URL;
            case TELEGRAM -> TELEGRAM_CONNECTOR_URL;
            case INSTAGRAM -> INSTAGRAM_CONNECTOR_URL;
            case INTERNAL -> null; // No external delivery for internal messages
        };
    }
```

### 3.6 Resiliência: Retry e Circuit Breaker

Utilizamos a biblioteca Resilience4j para proteger o sistema de falhas em APIs externas.

- **Retry**: Política de backoff exponencial (máximo 3 tentativas) para falhas transitórias de rede.
- **Circuit Breaker**: Monitora a taxa de erros de cada Connector. Se o WhatsApp falhar (ex: >50% de erro), o circuito abre e rejeita novas requisições imediatamente, evitando o efeito cascata e permitindo a recuperação do serviço externo.

## 4. Testes de Carga e Métricas Coletadas

Os testes de carga foram realizados utilizando a ferramenta **k6**, simulando um cenário de uso concorrente com ramp-up até 50 usuários virtuais (VUs). O objetivo foi validar os *thresholds* (limites) de latência e taxa de erro definidos no SLA.

Script k6 utilizado para testes

```
import http from "k6/http";

import { check, sleep } from "k6";

import { SharedArray } from "k6/data";



export const options = {
```

```
  stages: [
    { duration: '20s', target: 10 },  // warmup
    { duration: '40s', target: 50 },  // carga sustentada
    { duration: '20s', target: 0 },   // cooldown
  ],
  thresholds: {
    http_req_duration: ['p(95)<300'], // 95% requests < 300ms
    http_req_failed: ['rate<0.05'],   // <5% falha
  },
};

const BASE_URL = __ENV.API_URL || "http://host.docker.internal:3000";

const users = new SharedArray("users", () => [
  { email: "user1@dev.com", pass: "1234" },
  { email: "user2@dev.com", pass: "1234" },
]);

export default function () {
  const user = users[Math.floor(Math.random() * users.length)];

  // 1️⃣ ---- Login ----
  let res = http.post(`${BASE_URL}/auth/login`, JSON.stringify({
    email: user.email,
    password: user.pass,
```

```
}), {
  headers: { "Content-Type": "application/json" },
});

check(res, {
  "login status 200": r => r.status === 200,
});

const token = res.json("access_token");

// 2️⃣ ---- Enviar mensagem ----
const msgPayload = {
  conversation_id: "test_auto_conv",
  from: user.email,
  to: ["user2@dev.com"],
  payload: {
    type: "text",
    text: `Teste K6 ${Date.now()}`
  }
};

res = http.post(
  `${BASE_URL}/v1/messages`,
  JSON.stringify(msgPayload),
```

```
    { headers: { "Authorization": `Bearer ${token}`, "Content-Type":
"application/json" } }
  );

  check(res, {
    "send msg 200": r => r.status === 200,
  });

  // 3️⃣ ---- Listar mensagens ----
  res = http.get(`$
{BASE_URL}/v1/conversations/test_auto_conv/messages`, {
    headers: { "Authorization": `Bearer ${token}` },
  });

  check(res, {
    "list 200": r => r.status === 200,
    "list < 200ms": r => r.timings.duration < 200,
  });

  sleep(1);
}
```

## 4.1 Resultados de Desempenho (Cenário: 50 VUs)

O teste executou durante 1m 20s, gerando um total de 9.576 requisições HTTP. Abaixo, o resumo dos dados obtidos:

| Métrica | Resultado Obtido | Meta (SLA) | Status |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **Usuários Simultâneos (VUs)** | 50 | - | Atingido |
| **Throughput (Vazão)** | 118,8 reqs/s | > 100 reqs/s | Excelente |
| **Latência Média** | 22,52 ms | < 200 ms | Excelente |
| **Latência P95** | 82,96 ms | < 300 ms | Aprovado |
| **Taxa de Erro HTTP** | 0,00% | < 5% | Perfeito |
| **Verificações (Checks)** | 99,24% Sucesso | > 99% | Aprovado |

## 4.2 Análise dos Resultados e Falhas

O sistema demonstrou alta estabilidade e resiliência. Não houve nenhum erro HTTP 5xx ou 4xx (http_req_failed = 0.00%), confirmando a robustez da API.

Observou-se, contudo, uma leve oscilação na verificação de tempo de resposta do endpoint de listagem (list < 200ms):

- **Comportamento**: De 6.384 verificações, 48 falharam (0,75%).
- **Causa Raiz**: Embora a média tenha sido baixa (22ms), houve picos isolados onde o tempo de resposta máximo atingiu 666,9ms.
- **Conclusão**: Esses picos representam menos de 1% das requisições e estão dentro da tolerância de "jitter" de rede/banco de dados, não comprometendo a experiência do usuário final, já que 95% das requisições foram atendidas em menos de 83ms.

## 4.3 Consumo de Rede

Durante o teste, o sistema processou um volume considerável de dados:

- Data Received: 256 MB (Taxa de 3,2 MB/s).
- Data Sent: 2,2 MB.
  Isso valida a capacidade do API Gateway e dos Connectors em lidar com tráfego intenso de entrada e saída de payloads JSON.

# 5. Falhas Simuladas e Recuperação

## 5.1 Cenários de Falha Testados

| Componente Falho | Impacto Observado | Comportamento de Recuperação |
|---|---|---|
| **Kafka Broker** | API continuou aceitando msgs (HTTP 202), mas o processamento parou. | Após o reinício do Kafka (30s), as mensagens acumuladas foram processadas sem perda de dados. |
| **MongoDB** | API retornou HTTP 503 temporariamente. | O *client* recebeu instrução de Retry-After. Nenhuma mensagem foi perdida, pois a persistência é síncrona na entrada. |
| **Redis** | Aumento de latência (32ms → 58ms) e perda de cache. | O sistema fez *fallback* para o MongoDB para checagem de idempotência. Sessões WebSocket foram recriadas automaticamente. |

## 5.2 Atuação do Circuit Breaker (Caso WhatsApp)

Simulamos a indisponibilidade da API do WhatsApp. O sistema detectou a falha após atingir 50% de taxa de erro.

- **Estado OPEN**: O circuito abriu em 28 segundos. Todas as novas requisições para o WhatsApp foram rejeitadas imediatamente ou enfileiradas, protegendo os recursos internos.
- **Recuperação**: Após o período de espera (30s), o circuito entrou em HALF_OPEN, testou a conexão e fechou o circuito automaticamente ao confirmar a estabilidade.

### 5.3 Dead Letter Queue (DLQ)

Mensagens com payload inválido ou falhas persistentes após todos os retries foram movidas automaticamente para a fila chat-events-dlq. A análise mostrou que 58% das falhas na DLQ eram devidas a payloads inválidos, que foram descartados, enquanto falhas de rede foram reprocessadas com sucesso via script administrativo.

## 6. Limitações e Melhorias Futuras

### 6.1 Limitações Atuais

- Escalabilidade Vertical do Banco: O MongoDB, sem sharding habilitado, apresenta um teto de escrita de aproximadamente 50.000 mensagens/minuto.
- Rate Limits de Terceiros: A vazão é limitada pelas APIs externas (ex: WhatsApp Business API permite cerca de 250 msg/s), criando gargalos fora do nosso controle.
- Complexidade Operacional: A gestão de múltiplos serviços (8 containers + bancos + broker) exige maturidade em orquestração (Kubernetes).

### 6.2 Melhorias Planejadas

- Curto Prazo (Performance):
    - Otimização Connection Pools: Aumentar maxPoolSize de 50 para 100 para suportar maior concorrência.
    - Redis Cluster: Implementar para melhor escalabilidade e alta disponibilidade.
- Médio Prazo (Funcional): Implementação de arquitetura Multi-tenant para isolar dados de diferentes clientes corporativos e permitir configurações de roteamento específicas.
- Longo Prazo (Inteligência): Uso de Machine Learning para "Roteamento Preditivo", escolhendo o melhor canal e horário de entrega baseando-se no histórico de sucesso de cada usuário.

## 7. Conclusão

O projeto Chat4All v2 atingiu seus objetivos técnicos, entregando uma arquitetura resiliente capaz de suportar alta carga com garantias de entrega. Os testes de estresse e falha comprovaram que as decisões arquiteturais, como o uso de Kafka

para desacoplamento e Circuit Breakers para proteção foram acertadas. O sistema está apto para implantação em produção com as configurações de hardening recomendadas.