Universidade de Brasília - UnB
Faculdade de Ciência e Tecnologia em Engenharias - FCTE
Engenharia de Software

Observabilidade em Sistemas Distribuídos com Ferramentas Open Source: Uma Arquitetura Baseada em OpenTelemetry e LGTM

Autores: Fernando Vargas Teotônio de Oliveira
Professor Orientador: Doutor Fernando Willian Cruz

Brasília, DF
2025

Fernando Vargas Teotônio de Oliveira

Observabilidade em Sistemas Distribuídos com Ferramentas Open Source: Uma Arquitetura Baseada em OpenTelemetry e LGTM

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software

Universidade de Brasília - UnB

Faculdade de Ciência e Tecnologia em Engenharias - FCTE

Orientador: Doutor Fernando Willian Cruz

Brasília, DF

2025

Fernando Vargas Teotônio de Oliveira

Observabilidade em Sistemas Distribuídos com Ferramentas Open Source: Uma Arquitetura Baseada em OpenTelemetry e LGTM/ Fernando Vargas Teotônio de Oliveira. – Brasília, DF, 2025-

70 p. : il. (algumas color.) ; 30 cm.

Orientador: Doutor Fernando Willian Cruz

TCC – Universidade de Brasília - UnB

Faculdade de Ciência e Tecnologia em Engenharias - FCTE , 2025.

I. Doutor Fernando Willian Cruz. II. Universidade de Brasília. III. Faculdade de Ciência e Tecnologia em Engenharias. IV. Observabilidade em Sistemas Distribuídos com Ferramentas Open Source: Uma Arquitetura Baseada em OpenTelemetry e LGTM

Fernando Vargas Teotônio de Oliveira

Observabilidade em Sistemas Distribuídos com Ferramentas Open Source: Uma Arquitetura Baseada em OpenTelemetry e LGTM

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software

Trabalho em andamento. Brasília, DF, 11 de julho de 2025:

Doutor Fernando Willian Cruz
Orientador

Mauricio Serrano
Convidado 1

Renato Coral Sampaio
Convidado 2

Brasília, DF
2025

Dedico este trabalho
aos que caminharam antes de mim,
aos que me ensinaram com o silêncio e com o exemplo,
aos meus pais, pela base sólida,
aos meus ancestrais, pela força invisível que me guia,
e à Júlia, minha companheira,
pela presença serena e pelo amor.

(Fernando)

Agradecimentos

Agradeço, primeiramente, aos meus pais, especialmente a minha mãe, pelo amor incondicional, pela educação e por nunca medir esforços para que eu pudesse chegar até aqui.

Aos meus avós, pelo amor constante e pela segurança que sempre me transmitiram. Em cada gesto e em cada palavra, encontrei abrigo.

À Júlia, minha namorada, obrigada por estar ao meu lado em cada passo desta jornada. Pela paciência nos meus silêncios, pelas palavras nos meus cansaços e pela luz nos meus dias difíceis. Este trabalho também é teu.

Agradeço aos meus irmãos pelo companheirismo, pelas risadas e pelos momentos de descontração que deixam o meu dia a dia mais leve e feliz.

Ao meu orientador, Prof. Doutor Fernando Willian Cruz, pela orientação atenta, pela confiança depositada no meu trabalho e pelo incentivo constante à autonomia crítica e à profundidade técnica.

Aos colegas e amigos da graduação, que compartilharam comigo dúvidas, conquistas, códigos quebrados, cafés e noites em claro. Aprendi muito com cada um de vocês.

Aos professores da Universidade de Brasília, que contribuíram direta ou indiretamente com minha formação e ampliaram minha visão sobre a engenharia de software.

Às forças que me guiam espiritualmente, visíveis ou invisíveis, pelo axé, proteção e sabedoria que me sustentaram nos momentos mais desafiadores.

Por fim, agradeço a mim mesmo, por não desistir quando tudo parecia difícil. A jornada foi longa, mas valeu cada passo.

"O futuro só tem valor se nele podermos garantir
a perpetuação do passado."
(Pedro Bandeira)

Resumo

O avanço das arquiteturas distribuídas, impulsionado pela adoção de microsserviços e computação em nuvem, impôs novos desafios à visibilidade operacional dos sistemas de software. Neste contexto, conceitos como monitoração e observabilidade tornaram-se essenciais para garantir a confiabilidade e o desempenho de aplicações modernas. Este trabalho tem como objetivo principal investigar e aplicar, na prática, uma arquitetura de *observabilidade* baseada exclusivamente em ferramentas *open source*, com foco na coleta, correlação e visualização de métricas, *logs* e *traces*.

A pesquisa foi conduzida por meio de revisão bibliográfica, análise comparativa de soluções e implementação de uma arquitetura prática, validada em um microsserviço real em produção. A instrumentação foi realizada com o uso do *OpenTelemetry*, enquanto o backend da solução foi construído com a pilha LGTM — composta por *Loki*, *Grafana*, *Tempo* e *Mimir* — garantindo integração nativa com ambientes *cloud-native* e escalabilidade.

Os resultados obtidos demonstraram a viabilidade técnica da arquitetura proposta, permitindo correlação eficiente entre os sinais observáveis, diagnóstico rápido de falhas e maior visibilidade do comportamento interno do sistema. A monografia oferece ainda uma análise crítica sobre as decisões de projeto, limitações enfrentadas e caminhos para evoluções futuras.

Palavras-chave: Observabilidade. Monitoração. Sistemas distribuídos. Open source. OpenTelemetry. LGTM.

Abstract

The advancement of distributed architectures, driven by the adoption of microservices and cloud computing, has introduced new challenges to the operational visibility of software systems. In this context, concepts such as monitoring and observability have become essential to ensure the reliability and performance of modern applications. This work aims to investigate and apply, in practice, an *observability* architecture based exclusively on *open source* tools, focusing on the collection, correlation, and visualization of metrics, *logs*, and *traces*.

The research was conducted through a literature review, comparative analysis of solutions, and the implementation of a practical architecture, validated in a real production microservice. Instrumentation was performed using *OpenTelemetry*, while the backend of the solution was built using the LGTM stack — composed of *Loki*, *Grafana*, *Tempo*, and *Mimir* — ensuring native integration with *cloud-native* environments and scalability.

The results demonstrated the technical feasibility of the proposed architecture, enabling efficient correlation between observable signals, fast failure diagnosis, and greater visibility into the system's internal behavior. This monograph also presents a critical analysis of the design decisions, the limitations encountered, and potential paths for future development.

Keywords: Observability. Monitoring. Distributed systems. Open source. OpenTelemetry. LGTM.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

UI	Interface do Usuário
SGBDR	Sistema Gerenciador de Banco de Dados Relacional
CPU	Unidade Central de Processamento
CNFC	*Cloud Native Computing Foundation*
GCP	*Google Cloud Platform*
Azure	*Microsoft Azure*
API	Interface de Programação de Aplicações
SDK	Kit de Desenvolvimento de Software
LGTM	Acrônimo para Loki, Grafana, Tempo e Mimir

Sumário

1 Introdução

A evolução dos sistemas de software tem sido marcada por transformações significativas. Inicialmente, as aplicações eram construídas de forma monolítica, com todos os componentes integrados em uma única base de código e implantados como uma única unidade. Com o aumento da complexidade das aplicações, das demandas por escalabilidade e da necessidade de manter a alta disponibilidade, houve uma transição natural para arquiteturas distribuídas, especialmente baseadas em microsserviços. Esse modelo permite que diferentes partes do sistema sejam desenvolvidas, implantadas e escaladas de forma independente, promovendo agilidade e resiliência. No entanto, essa descentralização traz também novos desafios relacionados à visibilidade e ao controle sobre o comportamento do sistema como um todo.

Em ambientes distribuídos, nos quais múltiplos serviços interagem continuamente por meio de APIs e mensagens assíncronas, os métodos tradicionais de monitoração, centrados em métricas isoladas como uso de CPU, memória ou disponibilidade de serviços, mostram-se insuficientes. Tais abordagens geralmente detectam apenas os sintomas de falhas, sem oferecer mecanismos eficazes para identificar suas causas. Nesse contexto, ganha relevância o conceito de observabilidade, que propõe uma visão mais abrangente e integrada sobre o funcionamento dos sistemas, permitindo compreender seu estado interno a partir da análise de saídas externas como métricas, *logs* e *traces*.

Este trabalho tem como proposta a investigação das práticas contemporâneas de observabilidade em sistemas distribuídos, com foco na utilização de ferramentas *open source* voltadas à coleta, correlação e visualização dos sinais observáveis. A partir de uma revisão bibliográfica e de uma análise comparativa entre diferentes soluções adotadas na indústria, o estudo busca compreender os critérios de escolha dessas ferramentas e suas capacidades técnicas. O objetivo final é propor e validar, na prática, uma arquitetura de *backend* de observabilidade que seja escalável, interoperável e alinhada aos princípios da computação em nuvem.

A arquitetura proposta neste trabalho foi implementada diretamente em um *cluster* Kubernetes, sendo aplicada em um microsserviço real em produção. Ela é composta por um *pipeline* de instrumentação com OpenTelemetry e por uma infraestrutura de *backend* baseada na pilha LGTM: Loki para *logs*, Grafana para visualização, Tempo para *traces* distribuídos e Mimir para armazenamento de métricas. Através dessa integração, torna-se possível correlacionar os três pilares da observabilidade de forma eficiente, permitindo identificar gargalos, falhas e padrões de comportamento em tempo real.

Ao longo desta monografia, passaremos por uma trajetória que parte dos fun-

damentos teóricos da observabilidade, avança pela análise crítica de ferramentas *open source* amplamente utilizadas no mercado e culmina na demonstração de uma arquitetura prática. Com isso, o trabalho busca não apenas explorar os conceitos que sustentam a observabilidade moderna, mas também oferecer subsídios técnicos para sua aplicação efetiva em contextos produtivos.

Dessa forma, este estudo contribui com uma abordagem aplicada, que alia fundamentos conceituais e prática de engenharia, no intuito de fortalecer a visibilidade, o controle e a confiabilidade de sistemas distribuídos complexos. Ao documentar os desafios, decisões e resultados obtidos, o trabalho pretende oferecer uma base útil para desenvolvedores, engenheiros de observabilidade e pesquisadores interessados na construção de sistemas mais monitoráveis, resilientes e transparentes.

1.1 Motivação

A crescente adoção de arquiteturas distribuídas, especialmente baseadas em microsserviços e computação em nuvem, trouxe avanços significativos em escalabilidade, flexibilidade e resiliência. No entanto, também impôs novos desafios relacionados à visibilidade operacional e ao entendimento do comportamento dos sistemas. Em ambientes compostos por múltiplos serviços interconectados, torna-se cada vez mais difícil diagnosticar falhas, compreender o fluxo de requisições e antecipar degradações de desempenho utilizando apenas métodos tradicionais de monitoração. Tais métodos, muitas vezes limitados a métricas de infraestrutura como uso de CPU ou memória, são insuficientes para capturar a complexidade e a dinamicidade desses sistemas.

Nesse cenário, ganha destaque o conceito de observabilidade, que se consolida como uma evolução das práticas de monitoração. A observabilidade não se limita a detectar anomalias, mas busca compreender as causas por meio da coleta e correlação de sinais provenientes de diferentes camadas do sistema, como logs, métricas e *traces*. Essa abordagem permite análises mais profundas e decisões mais precisas, contribuindo diretamente para a estabilidade e confiabilidade das aplicações.

A motivação deste trabalho surge da necessidade de compreender como ferramentas *open source* podem ser utilizadas de forma integrada para oferecer observabilidade eficiente em sistemas distribuídos modernos. Em vez de se restringir à teoria, a pesquisa propõe-se a construir, aplicar e validar uma arquitetura prática que una diferentes soluções já consolidadas no ecossistema *cloud-native*. O foco recai na utilização do OpenTelemetry como camada de instrumentação unificada e na integração de um *backend* composto pelas ferramentas Loki, Grafana, Tempo e Mimir, estrutura conhecida como pilha LGTM.

O objetivo geral do estudo é analisar ferramentas *open source* voltadas à monitoração e observabilidade, com ênfase na integração dessas soluções em uma arquitetura

unificada e funcional. Para isso, foram definidos objetivos específicos que incluem a consolidação conceitual dos pilares da observabilidade, a comparação técnica entre ferramentas, a investigação do papel do OpenTelemetry na padronização da coleta de sinais, e a implementação de uma prova de conceito validada em um microsserviço real em produção. Ao longo do desenvolvimento, a arquitetura demonstrou-se eficiente na coleta, processamento e visualização de dados observáveis, possibilitando diagnósticos mais rápidos e melhor compreensão do comportamento do sistema.

Assim, a motivação deste trabalho está ancorada na busca por soluções acessíveis, escaláveis e aplicáveis, que possam ser replicadas em contextos reais sem depender de ferramentas proprietárias. Ao explorar o potencial das tecnologias *open source* e aplicá-las em um ambiente produtivo, este estudo contribui para o avanço das práticas de observabilidade e oferece um modelo técnico sólido para engenheiros de software, profissionais de *DevOps* e pesquisadores da área.

1.2 Cronograma do TCC

O cronograma de desenvolvimento do TCC está detalhado na Tabela 1, abrangendo o período de outubro de 2024 até julho de 2025.

1.3 Estrutura da Monografia

Esta monografia está organizada em sete capítulos, estruturados de forma a conduzir o leitor da fundamentação teórica até a aplicação prática da arquitetura de observabilidade em ambiente real.

O **Capítulo 1** apresenta a introdução ao tema, contextualizando a evolução das arquiteturas de software e os desafios associados à monitoração em sistemas distribuídos. Também são expostos a motivação do trabalho, seus objetivos, a metodologia adotada e a organização da monografia.

O **Capítulo 2** é dedicado à fundamentação teórica, abordando os principais conceitos relacionados à monitoração e observabilidade. São discutidas as limitações dos modelos tradicionais de monitoração e os requisitos de visibilidade em ambientes baseados em microsserviços e computação em nuvem.

O **Capítulo 3** apresenta uma análise comparativa de ferramentas *open source* voltadas à observabilidade. São investigadas soluções como Prometheus, Loki, Tempo, Mimir, Fluentd, ELK Stack, Jaeger e OpenTelemetry, destacando suas funcionalidades, pontos fortes, limitações e adequação a cenários *cloud-native*.

O **Capítulo 4** descreve a arquitetura proposta para coleta, processamento e visu-

alização dos sinais observáveis, detalhando a instrumentação realizada com o OpenTelemetry e a integração com a pilha LGTM. A arquitetura é validada em um microsserviço real em produção, demonstrando sua viabilidade, desempenho e capacidade de correlação entre *logs*, métricas e *traces*.

O **Capítulo 5** apresenta a fundamentação teórica que embasa a arquitetura de observabilidade proposta, estabelecendo a correspondência entre os conceitos estudados e os componentes técnicos implementados. São discutidas referências como *Observability Engineering*, *Cloud-Native Observability with OpenTelemetry* e *O11Y Explained*, conectando teoria e prática.

O **Capítulo 6** apresenta a análise dos resultados obtidos com a implementação prática, relacionando-os com os objetivos do trabalho. São discutidos os ganhos observados, os desafios enfrentados durante o processo de instrumentação e configuração, bem como as limitações identificadas e sua superação.

O **Capítulo 7** traz as considerações finais, destacando as contribuições do trabalho para a área de observabilidade em sistemas distribuídos e apontando caminhos para estudos futuros, com ênfase em avaliação de desempenho, automação da instrumentação e expansão para múltiplos serviços.

Tabela 1 – Cronograma de atividades para o TCC

Período	**Atividade**
14/10/2024 – 20/10/2024	Escolha e definição do tema
21/10/2024 – 27/10/2024	Lapidação do tema com o orientador e elaboração do pré-projeto
28/10/2024 – 10/11/2024	Início da pesquisa bibliográfica
11/11/2024 – 01/12/2024	Fichamento e organização das referências principais
02/12/2024 – 22/12/2024	Escrita da revisão bibliográfica
23/12/2024 – 05/01/2025	Pausa ou revisão preliminar do texto
06/01/2025 – 19/01/2025	Escolha e análise comparativa das ferramentas
20/01/2025 – 02/02/2025	Escrita da análise das ferramentas e consolidação teórica
03/02/2025 – 16/02/2025	Definição da arquitetura prática
17/02/2025 – 09/03/2025	Instrumentação com OpenTelemetry e testes iniciais
10/03/2025 – 30/03/2025	Implantação da arquitetura no ambiente de teste
31/03/2025 – 13/04/2025	Coleta de dados, ajustes e documentação da arquitetura
14/04/2025 – 27/04/2025	Análise e interpretação dos resultados
28/04/2025 – 11/05/2025	Escrita do capítulo de resultados
12/05/2025 – 25/05/2025	Escrita da fundamentação da arquitetura prática
26/05/2025 – 08/06/2025	Escrita da análise crítica e considerações finais
09/06/2025 – 15/06/2025	Revisão geral do texto e normalização conforme ABNT
16/06/2025 – 22/06/2025	Revisão com orientador e ajustes finais
23/06/2025 – 06/07/2025	Preparação para entrega e defesa
07/07/2025 – 11/07/2025	Entrega final e apresentação do TCC

2 Metodologia

A metodologia adotada neste trabalho é de natureza aplicada e qualitativa, com o objetivo de investigar, avaliar e implementar uma arquitetura de observabilidade baseada em ferramentas *open source*. A pesquisa foi conduzida em três frentes principais: a revisão teórica e conceitual sobre observabilidade, a análise comparativa entre ferramentas de código aberto e a implementação prática de uma arquitetura completa em um microsserviço real em produção.

Inicialmente, foi realizada uma revisão bibliográfica para fundamentar os conceitos de monitoração, observabilidade e os pilares técnicos que sustentam esses domínios, como *logs*, métricas e *traces*. Essa etapa envolveu a consulta de artigos acadêmicos, *whitepapers* de organizações como a CNCF (Cloud Native Computing Foundation), documentação oficial das ferramentas e literatura técnica especializada. O objetivo dessa revisão foi compreender as abordagens adotadas em sistemas modernos, bem como identificar padrões de arquitetura e melhores práticas amplamente aceitas.

A segunda etapa consistiu na análise e comparação de ferramentas open source que atendem aos requisitos de coleta e visualização dos três pilares da observabilidade. As ferramentas avaliadas incluíram Prometheus, Grafana Mimir, Loki, Fluentd, ELK Stack, Jaeger, Zipkin, Tempo e o próprio OpenTelemetry. Os critérios considerados para a comparação envolveram aspectos como escalabilidade, compatibilidade com Kubernetes, facilidade de integração com SDKs e *backends*, suporte à correlação entre sinais observáveis, custo operacional e aderência a padrões abertos. Essa análise guiou a definição da pilha tecnológica adotada na implementação da arquitetura.

A terceira etapa foi prática e consistiu na elaboração e implantação de uma arquitetura de *backend* de observabilidade baseada na integração entre o OpenTelemetry Collector e os componentes da pilha LGTM (Loki, Grafana, Tempo e Mimir). A arquitetura foi aplicada diretamente em um cluster Kubernetes, em um microsserviço real em produção, permitindo avaliar sua viabilidade em um cenário concreto e operacional. O processo de instrumentação foi realizado manualmente, utilizando o SDK do OpenTelemetry em uma aplicação Python, possibilitando a coleta de métricas customizadas, geração de spans representando transações distribuídas e exportação desses dados para o *backend*.

A coleta de *logs* foi feita diretamente nos pods, com envio para o Loki por meio de integração com o OpenTelemetry Collector. As métricas foram exportadas no formato OTLP para o Prometheus e armazenadas no Mimir como backend escalável. Os traces, por sua vez, foram enviados ao Tempo, viabilizando a visualização e análise distribuída. A configuração do OpenTelemetry Collector foi central na arquitetura, definindo pipelines

distintos para cada tipo de dado, com a utilização de *receivers* OTLP/gRPC, *processors* de enriquecimento e *batching*, e *exporters* dedicados para cada *backend*.

Toda a infraestrutura foi implantada nativamente em Kubernetes, com uso de arquivos de manifesto YAML e operadores específicos para o gerenciamento dos componentes. Essa abordagem permitiu orquestrar os serviços de forma controlada, escalável e alinhada às práticas modernas de DevOps. As visualizações foram construídas no Grafana, possibilitando a navegação correlacionada entre logs, métricas e traces.

Durante a implantação, foram realizados testes contínuos e avaliações qualitativas com base nos dados coletados pelo sistema em produção. A análise da eficácia da arquitetura considerou aspectos como confiabilidade dos dados, latência na propagação das informações, facilidade de diagnóstico de falhas e clareza na visualização dos sinais observáveis. A execução prática permitiu validar a arquitetura como uma solução funcional, estável e replicável em ambientes distribuídos reais.

3 Revisão Bibliográfica

Neste capítulo, realizaremos uma análise teórica dos conceitos essenciais que fundamentam a proposta deste trabalho. Iniciaremos com uma breve discussão sobre sistemas monolíticos e sistemas em microsserviços, destacando suas diferenças e como essas arquiteturas influenciam a maneira como as práticas de monitoração e observabilidade são aplicadas.

A seguir, abordaremos os conceitos fundamentais de monitoração e observabilidade, explicando suas definições, a inter-relação entre ambos e as nuances que os distinguem. Em particular, exploraremos os Quatro Sinais de Ouro (*Golden Signals*) (Latência, Tráfego, Erros e Saturação) que são métricas essenciais para a monitoração de sistemas e ajudam a identificar pontos críticos que podem impactar a experiência do cliente.

Por fim, revisaremos algumas das ferramentas mais utilizadas no mercado para coleta e análise de dados de monitoração, proporcionando uma base teórica que sustentará a análise e a proposição da arquitetura discutida nos capítulos seguintes.

3.1 Arquitetura Monolítica

A arquitetura monolítica é um modelo clássico de desenvolvimento de software, no qual uma única base de código é responsável por executar diversas funções de negócios. Em um sistema monolítico, o *kernel* gerencia todas as funcionalidades (POWELL; SMALLEY, 2024).

Na arquitetura monolítica, três componentes principais podem ser destacados, como se pode observar na Figura 1. O primeiro é a interface de usuário do lado do cliente (UI), que gerencia o que o usuário vê, incluindo imagens, textos e qualquer outra informação exibida na tela da interface, como dados relacionados às ações no navegador. O segundo é o banco de dados, que geralmente utiliza um sistema de gerenciamento de banco de dados relacional (SGBDR), composto por diversas tabelas. Por fim, a aplicação do lado do servidor lida diretamente com os recursos computacionais, como memória, CPU e armazenamento (POWELL; SMALLEY, 2024).

3.2 Arquitetura de Microsserviços

É uma abordagem nativa da nuvem em que um único aplicativo é dividido em vários componentes ou serviços menores, que são independentes, desacoplados e podem ser implantados de forma autônoma.(IBM, 2024)

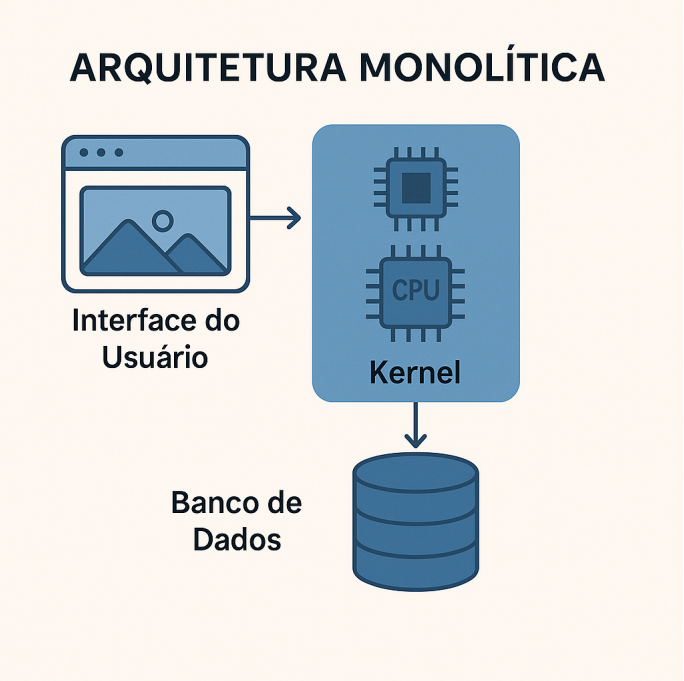

Figura 1 – Representação da arquitetura monolítica.

Essa arquitetura apresenta algumas características fundamentais. Cada microsserviço possui sua própria pilha de tecnologia, incluindo banco de dados e modelo de gerenciamento de dados. A comunicação entre os serviços é realizada por meio de APIs REST, (*streaming*) de eventos e corretores de mensagens. Além disso, os microsserviços são organizados de acordo com a capacidade de negócio, sendo essa divisão frequentemente chamada de contexto delimitado (IBM, 2024).

O valor dos microsserviços pode ser compreendido não apenas por suas características técnicas, mas também pelos benefícios comerciais e organizacionais que proporcionam. Uma das principais vantagens dessa abordagem é a facilidade na atualização do código, permitindo a adição de novos recursos ou funcionalidades sem a necessidade de modificar todo o aplicativo. Outro benefício relevante é a flexibilidade das equipes de desenvolvimento, que podem utilizar diferentes pilhas e linguagens de programação para componentes distintos. Além disso, a escalabilidade dos componentes de forma independente reduz desperdícios e custos, evitando a necessidade de escalar toda a aplicação quando apenas um serviço específico enfrenta alta carga (IBM, 2024).

3.3 Arquitetura Monolítica versus Arquitetura de Microsserviços

No contexto de desenvolvimento, começar com aplicações monolíticas é mais fácil, pois exige menos planejamento inicial, permitindo que você adicione módulos conforme necessário. No entanto, à medida que a aplicação cresce, ela pode se tornar complexa e difícil de atualizar. Já uma arquitetura de microsserviços demanda mais planejamento desde

o início, pois é necessário identificar funções independentes e criar APIs consistentes, mas esse esforço inicial torna a manutenção mais eficiente. Com microsserviços, as mudanças e correções de *bugs* são mais rápidas, e a reutilização de código tende a aumentar ao longo do tempo.(AWS, 2024)

Em relação a depuração, em uma arquitetura monolítica, o desenvolvedor pode rastrear os dados e examinar o comportamento do código dentro do mesmo ambiente de programação. Já em uma arquitetura de microsserviços, identificar problemas exige a análise de múltiplos serviços independentes. Depurar aplicações de microsserviços pode ser mais desafiador, pois envolve a colaboração de vários desenvolvedores responsáveis por diferentes microsserviços, o que frequentemente demanda testes coordenados, discussões e *feedback* entre os membros da equipe, aumentando o tempo e os recursos necessários.(AWS, 2024)

Já em modificações, numa aplicação monolítica, uma pequena alteração em uma parte do sistema pode afetar várias funções devido ao acoplamento forte do código. Além disso, ao introduzir novas mudanças, é necessário testar e reimplantar todo o sistema no servidor. Já a abordagem de microsserviços oferece mais flexibilidade, permitindo que os desenvolvedores façam alterações de forma mais isolada, modificando apenas funções específicas. Eles também podem implantar serviços de forma independente, o que é vantajoso em um fluxo de trabalho de implantação contínua, permitindo mudanças frequentes e pequenas sem comprometer a estabilidade do sistema. (AWS, 2024)

Na escalabilidade, as aplicações monolíticas enfrentam desafios à medida que crescem, pois todas as funcionalidades estão em uma única base de código, o que exige que a aplicação inteira seja escalada conforme as necessidades mudam. Por exemplo, se a performance cair devido ao aumento de tráfego em uma função de comunicação, será necessário aumentar os recursos de toda a aplicação, o que leva ao desperdício de recursos, já que nem todas as partes do sistema estão sobrecarregadas. Por outro lado, a arquitetura de microsserviços permite escalabilidade mais eficiente, pois cada componente tem seus próprios recursos em um sistema distribuído. Esses recursos podem ser escalados independentemente, ajustando-se às demandas específicas de cada serviço, como alocar mais recursos para um serviço de localização geográfica, sem afetar o restante do sistema.(AWS, 2024)

Para concluir, os microsserviços aceleram a inovação, reduzem riscos, aumentam a velocidade de entrada no mercado e diminuem o custo total de propriedade.(AWS, 2024)

3.4 Monitoração e Observabilidade

Com o aumento da dependência de arquiteturas distribuídas para a entrega de serviços de aplicativos, as práticas de monitoração e observabilidade tem se tornado cada

vez mais relevantes. Ambas desempenham papéis fundamentais na identificação e resolução de problemas que podem impactar os negócios, mas possuem diferenças significativas em suas abordagens e finalidades.

O monitoramento se concentra em fornecer consciência situacional, acompanhando métricas e eventos conhecidos para alertar sobre possíveis anomalias. Já a observabilidade vai além, permitindo uma análise mais profunda das causas dos problemas. Ela se baseia em dados emitidos pelos sistemas como *logs*, métricas e *traces* para deduzir seu estado interno, mesmo diante de comportamentos inesperados. Em outras palavras, enquanto a monitoração responde à pergunta "o que está errado?", a observabilidade ajuda a descobrir "por que está errado?" e "como corrigir?". (IBM Education, 2025)

Quando utilizadas de forma complementar, monitoração e observabilidade oferecem uma abordagem robusta para gerenciar sistemas modernos, melhorando a capacidade de detectar, diagnosticar e resolver problemas com agilidade e, consequentemente, minimizando impactos nos negócios.

3.4.1 Monitoração

A monitoração pode ser definida como o processo sistemático de coletar, analisar e utilizar informações para acompanhar o desempenho de um sistema ou programa em relação aos seus objetivos estabelecidos. Sua principal função é fornecer uma visão clara sobre métricas específicas, permitindo identificar tendências, padrões ou possíveis anomalias.(LIVENS, 2025)

Embora a monitoração seja um mecanismo poderoso para fornecer dados relevantes sobre o comportamento do sistema, ela geralmente se limita a analisar informações de maneira isolada, sem considerar o contexto mais amplo em que o sistema está inserido. Essa característica faz com que o monitoramento seja eficaz para detectar problemas, mas menos robusto na compreensão de suas causas e impactos em níveis mais profundos.(LIVENS, 2025)

O monitoração desempenha um papel crucial ao permitir que as equipes acompanhem o desempenho do sistema e identifiquem falhas previamente conhecidas. No entanto, sua eficácia depende diretamente da capacidade de definir, antecipadamente, quais métricas devem ser observadas. Quando problemas não previstos surgem, existe o risco de que falhas críticas em produção passem despercebidas, evidenciando a limitação do monitoramento em lidar com situações fora do escopo inicialmente planejado e comprometendo a capacidade de resposta proativa das equipes.(IBM Education, 2025)

3.4.1.1 Os quatro sinais dourados

No contexto de sistemas voltados para o usuário, o Google definiu quatro métricas fundamentais que devem ser priorizadas na monitoração: latência, tráfego, erros e saturação. Esses indicadores, conhecidos como os "quatro sinais dourados", fornecem uma visão abrangente sobre o desempenho e a saúde de um sistema.

1. **Latência**: é o tempo necessário para que um sistema atenda a uma solicitação, desde o momento em que ela é recebida até a entrega da resposta. Na monitoração, é crucial diferenciar entre a latência de solicitações bem-sucedidas e a de solicitações que resultam em falha. Por exemplo, um erro HTTP 500 causado por uma falha na conexão com um banco de dados pode ser processado rapidamente, mas ainda representa uma falha no sistema. Ignorar essas solicitações no cálculo da latência geral pode levar a interpretações equivocadas. Além disso, erros que demoram para ser processados são ainda mais prejudiciais do que aqueles resolvidos rapidamente. Por esse motivo, é essencial monitorar a latência associada tanto a solicitações bem sucedidas quanto a falhas, para garantir uma avaliação precisa do desempenho e da saúde do sistema.(Google, 2025)

2. **Tráfego**: refere-se à quantidade de demanda que está sendo colocada em um sistema, geralmente expressa por uma métrica específica e de alto nível, adaptada ao tipo de aplicação. Em serviços web, por exemplo, o tráfego costuma ser medido em solicitações HTTP por segundo, podendo ser categorizado por tipo de solicitação, como conteúdo estático ou dinâmico. Em sistemas de *streaming* de áudio, o tráfego pode ser representado pela taxa de entrada e saída de dados na rede ou pelo número de sessões simultâneas ativas. Já em sistemas de armazenamento de valores-chave, métricas como transações ou recuperações por segundo são frequentemente utilizadas para medir o tráfego. Monitorar esse indicador é essencial para entender a carga de trabalho imposta ao sistema e identificar padrões que possam impactar seu desempenho.(Google, 2025)

3. **Erros**: representam a taxa de solicitações que falham, seja de forma explícita, como no caso de códigos HTTP 500, de forma implícita, como respostas HTTP 200 que retornam conteúdo incorreto, ou com base em políticas definidas, como uma solicitação que excede o tempo de resposta acordado (por exemplo, mais de um segundo). Nem sempre os códigos de resposta do protocolo são suficientes para capturar todas as condições de falha; nesse caso, pode ser necessário implementar protocolos internos adicionais para monitorar falhas parciais. A abordagem para monitorar erros varia conforme o tipo de falha: enquanto capturar códigos HTTP 500 em um balanceador de carga pode ser suficiente para identificar falhas completas, apenas testes de ponta a ponta do sistema são capazes de detectar problemas mais sutis, como

o envio de conteúdo incorreto. Monitorar esses aspectos é essencial para garantir a confiabilidade do sistema.(Google, 2025)

4. **Saturação**: A saturação mede o quão próximo um sistema está de sua capacidade máxima, destacando os recursos mais restritos. Por exemplo, em sistemas limitados pela memória, a saturação seria medida com base na utilização de memória; em sistemas com restrição de entrada e saída (I/O), a métrica seria focada na utilização de I/O. É importante observar que muitos sistemas começam a apresentar degradação no desempenho antes mesmo de atingir 100% de utilização. Por isso, é fundamental definir um limite de utilização ideal para cada recurso crítico. (Google, 2025)

   Em sistemas mais complexos, a saturação pode ser complementada por métricas de carga em níveis mais altos, como a capacidade do serviço de lidar com aumentos de tráfego. O sistema pode operar adequadamente no momento, mas seria capaz de suportar 10% a mais de tráfego? Ou já está no limite? Para serviços simples e estáticos, um teste de carga único pode fornecer uma estimativa confiável. Entretanto, para a maioria dos serviços, é necessário monitorar sinais indiretos, como a utilização da CPU ou a largura de banda da rede, que possuem limites conhecidos. (Google, 2025)

   Os aumentos de latência frequentemente indicam que o sistema está se aproximando da saturação. Medir o tempo de resposta do percentil 99 em uma janela de tempo curta (por exemplo, um minuto) pode fornecer alertas precoces sobre a saturação. Além disso, a saturação também abrange previsões de esgotamento de recursos, como alertas indicando que um banco de dados atingirá o limite de capacidade do disco rígido em poucas horas. (Google, 2025)

3.4.2 Observabilidade

A observabilidade, no contexto de sistemas de *software*, tem suas raízes na teoria do controle. Kálmán define a observabilidade como a capacidade de inferir os estados internos de um sistema a partir de suas saídas externas (KALMAN, 1960). Ao adaptar esse conceito para sistemas de *software* modernos, a observabilidade vai além de simplesmente entender o funcionamento interno de um aplicativo. Ela envolve a capacidade de diagnosticar o estado do sistema, mesmo quando esse estado é desconhecido ou não previsível.

Entretanto, é importante destacar que a definição popular de "observabilidade", frequentemente promovida por desenvolvedores de Software como Serviço (SaaS), limita o termo a um sinônimo de telemetria ou monitoramento, equiparando-o a métricas, logs e rastreamentos. Essa visão reduz a observabilidade a um conceito mais simples, tratando-a apenas como dados coletados de maneira isolada. Como argumentado no livro Observability Engineering: Achieving Production Excellence (2022), essa abordagem falha em

capturar a verdadeira essência da observabilidade, que envolve não apenas a coleta, mas a interpretação e correlação inteligente desses dados (AUSTIN; JONES; CUSTER, 2022).

Para que um sistema de software seja considerado observável, deve ser possível entender seu comportamento interno sem a necessidade de adicionar código personalizado ou adotar soluções específicas para cada nova situação. Isso implica em monitorar e analisar as saídas do sistema, como *logs*, métricas e *traces*, para obter uma visão clara sobre seu estado operacional. Em ambientes dinâmicos e multinuvem, essa capacidade de observação torna-se essencial para identificar e corrigir problemas de forma eficiente, sem comprometer o desempenho global do sistema.

Quando implementada corretamente, a observabilidade vai além do simples monitoramento. Ela permite identificar a causa raiz de falhas e anomalias, ajudando as equipes a responder rapidamente a incidentes e minimizar os impactos nos negócios.

3.4.2.1 Os três pilares da observabilidade

logs, métricas e *traces* são frequentemente referidos como os três pilares da observabilidade. No entanto, é importante entender que simplesmente coletar esses dados não transforma um sistema em observável. A verdadeira observabilidade surge quando essas informações são corretamente interpretadas e analisadas, permitindo uma compreensão profunda do comportamento do sistema e uma resposta eficaz a eventuais problemas.

1. ***logs***: são registros imutáveis e carimbados com data/hora que documentam eventos discretos ao longo do tempo. Esses *logs* podem se apresentar de três formas principais: como texto simples, no formato livre mais comum; como *logs* estruturados, tipicamente em formato JSON, que são altamente recomendados por sua clareza e legibilidade; ou como *logs* binários, como os usados por sistemas como MySQL para replicação ou por formatos como Protobuf.(OREILLY, 2025)

   Esses *logs* desempenham um papel crucial na depuração de falhas raras ou pouco frequentes, especialmente em sistemas distribuídos. Eles fornecem *insights* detalhados e contexto sobre comportamentos inesperados, ajudando as equipes a entender problemas que não podem ser facilmente detectados por outras métricas, como médias ou percentis. Contudo, em sistemas distribuídos, falhas raramente são causadas por um único evento isolado, mas sim por uma interação complexa entre vários componentes. Para diagnosticar adequadamente o problema, é necessário analisar essas interações e traçar o ciclo de vida de uma solicitação em toda a arquitetura distribuída.(OREILLY, 2025)

   Além de seu uso no diagnóstico de falhas, os *logs* também são valiosos para análises de negócios, oferecendo *insights* sobre o comportamento dos usuários e o desempenho

de produtos. Isso torna os *logs* essenciais tanto para a engenharia de confiabilidade quanto para decisões estratégicas de negócios.(OREILLY, 2025)

2. **Métricas**: são representações numéricas de dados mensurados ao longo de intervalos de tempo. Elas permitem modelar matematicamente e prever o comportamento de um sistema, tanto no presente quanto no futuro. Como as métricas são compostas apenas por números, elas são extremamente eficientes para armazenamento, processamento, compressão e recuperação de dados. Esse formato facilita a retenção de dados por períodos mais longos e também a realização de consultas mais rápidas, o que as torna ideais para a construção de dashboards que ilustram tendências históricas. Além disso, as métricas permitem a redução gradual da resolução dos dados, com agregações realizadas para períodos mais longos, como diário ou semanal, por exemplo.(OREILLY, 2025)

   Nos sistemas de monitoramento modernos, uma métrica é identificada por um nome e por pares chave-valor chamados *labels* (rótulos), o que permite alta dimensionalidade e facilita a análise dos dados. Cada amostra de métrica é composta por um valor numérico e um carimbo de tempo com precisão de milissegundos. Uma característica importante das métricas é que elas são imutáveis. Ou seja, a alteração do nome da métrica ou a adição e remoção de rótulos resultam na criação de uma nova série temporal.(OREILLY, 2025)

   As métricas oferecem vantagens significativas sobre os *logs*, especialmente no que diz respeito à eficiência no armazenamento e no processamento de dados. A principal vantagem é que, ao contrário dos *logs*, onde o custo de armazenamento e a complexidade aumentam à medida que o tráfego de usuários cresce, as métricas mantêm um overhead constante. Isso significa que, mesmo com um aumento no tráfego da aplicação, não há um aumento significativo no uso de disco ou na complexidade operacional, como ocorre com os *logs*. Com as métricas, é possível agregar dados de maneira eficiente e garantir que o tráfego de métricas não aumente proporcionalmente ao tráfego da aplicação.(OREILLY, 2025)

   Outro ponto importante é que as métricas são mais flexíveis para transformações matemáticas e estatísticas, como amostragem, agregação, sumarização e correlação. Isso as torna mais adequadas para relatar a saúde geral de um sistema. Além disso, as métricas são extremamente úteis para disparar alertas, já que consultas em bancos de dados de séries temporais em memória são muito mais rápidas e confiáveis.(OREILLY, 2025)

   No entanto, as métricas também têm suas limitações. Como elas são específicas a um único sistema, pode ser difícil entender o comportamento de um sistema distribuído como um todo, ou acompanhar a jornada completa de uma requisição que atravessa múltiplos sistemas. Embora seja possível correlacionar métricas com *logs*

para compreender esse comportamento, essa abordagem pode resultar em um aumento no armazenamento de métricas, especialmente se forem usados valores de alta cardinalidade, como identificadores de usuário. Além disso, ao lidar com uma grande quantidade de séries temporais, consultas de longo prazo podem se tornar lentas e menos eficientes.(OREILLY, 2025)

Apesar dessas limitações, quando usadas de forma otimizada, as métricas e os *logs* oferecem uma visão detalhada e completa dentro de um único sistema. No entanto, para entender a vida útil de uma requisição que passa por múltiplos sistemas, é necessário recorrer ao *traceamento* distribuído.(OREILLY, 2025)

3. **Trace**: é uma técnica utilizada para representar uma série de eventos distribuídos causalmente relacionados, que registram o fluxo de uma requisição através de um sistema distribuído. Os *traces* podem ser vistos como uma evolução dos *logs*, já que suas estruturas de dados se assemelham bastante às dos *logs* de eventos. No entanto, enquanto os *logs* geralmente registram informações pontuais de cada evento, os *traces* oferecem uma visão mais detalhada do caminho percorrido por uma requisição, permitindo entender melhor a execução e os pontos de interação entre diferentes componentes do sistema.(OREILLY, 2025)

   O *trace* se torna especialmente útil em arquiteturas baseadas em microserviços, mas qualquer aplicação complexa que envolva múltiplos componentes compartilhando recursos como rede, disco ou mutexes pode se beneficiar de sua implementação. O objetivo básico do *trace* é identificar pontos específicos no caminho de uma requisição, como chamadas de funções, limites de RPC *(Remote Procedure Calls)* ou segmentos de concorrência, como *threads*, *continuations* ou filas. Esses pontos representam, entre outras coisas:

   - *Forks* na execução, como a criação de novas *threads*.
   - *Hops* ou *fanouts*, que ocorrem quando a requisição atravessa limites de rede ou de processos.

   Os *traces* preservam a causalidade entre eventos, utilizando semânticas de "aconteceu antes"(*happens-before*). Isso significa que o *trace* pode fornecer uma visão clara de como o fluxo de uma requisição se desenvolve ao longo do tempo, mesmo em sistemas distribuídos. Esse fluxo é representado por um grafo acíclico dirigido (DAG), onde os nós são chamados de *spans*, e as conexões entre eles são chamadas de referências. Cada *span* representa um trecho específico da execução de uma requisição, e o fluxo total de uma requisição é reconstruído pela coleção desses *spans*.(OREILLY, 2025)

   Cada requisição é associada a um ID único global, que é propagado ao longo do caminho da requisição, permitindo que cada ponto de instrumentação registre ou enriqueça metadados. Esses metadados são enviados, então, para um coletor, que

reconstrói o fluxo de execução com base nas informações coletadas. Como ilustrado na Figura 2, a interação entre os serviços C e D pode ser destacada como o ponto de maior latência, proporcionando insights cruciais sobre a origem de um problema de desempenho.(OREILLY, 2025)

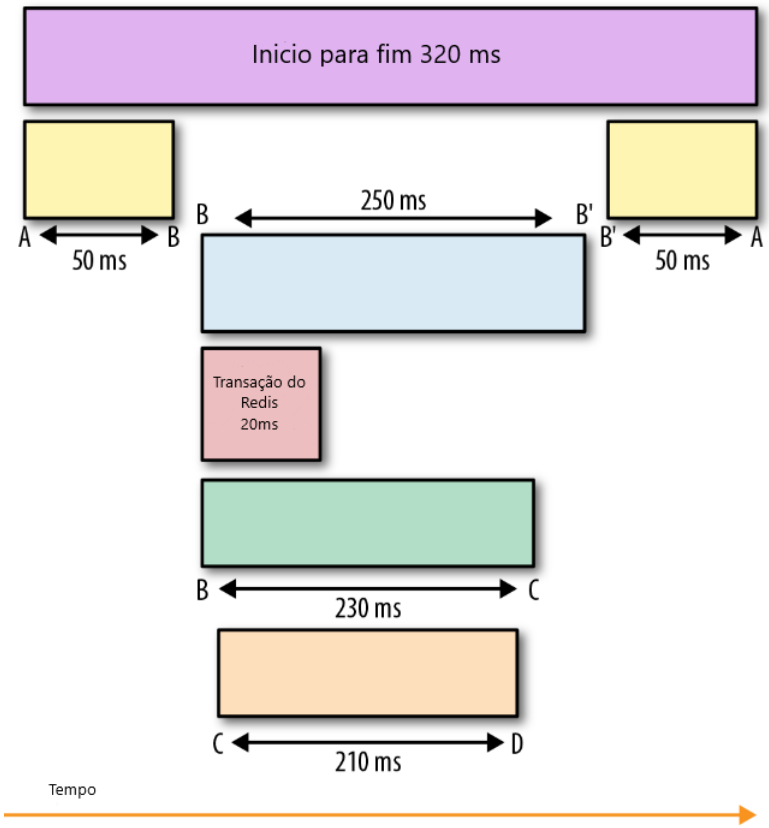

Figura 2 – *Trace* representado como intervalos: o intervalo A é o intervalo raiz, o intervalo B é um filho do intervalo A.

No entanto, o *trace* enfrenta desafios significativos, especialmente ao ser introduzido em infraestruturas existentes. Para que seja eficaz, cada componente no caminho de uma requisição precisa ser modificado para propagar as informações de *trace*, o que pode ser complexo, principalmente em arquiteturas poliglóticas (usando múltiplas linguagens e *frameworks*). Outro desafio é que, muitas vezes, o código de terceiros, como *frameworks* ou bibliotecas *open source*, também precisa ser instrumentado, o que aumenta a complexidade.(OREILLY, 2025)

Além disso, o custo do *trace* pode ser um obstáculo, embora seja consideravelmente menor do que o dos *logs*. Para reduzir a sobrecarga de *runtime* e os custos de armazenamento, os *traces* geralmente são amostrados. Isso significa que nem todas as requisições são *traceadas*, mas uma amostra representativa é coletada. A amostragem pode ocorrer de diferentes maneiras: no início de uma requisição, antes que qualquer *trace* seja gerado; no final do processo, depois que todos os sistemas participantes

registraram os *traces*; ou no meio do caminho, quando apenas os serviços *downstream* reportam o *trace*.

Cada abordagem de amostragem tem suas vantagens e desvantagens, e, em muitos casos, uma combinação dessas técnicas pode ser usada para otimizar a coleta de *traces*.(OREILLY, 2025)

3.4.3 Diferença entre Monitoração e Observabilidade

Embora monitoração e observabilidade compartilhem o objetivo de melhorar a confiabilidade e o desempenho dos sistemas, elas se distinguem significativamente em suas abordagens, objetivos e aplicações. Enquanto a monitoração é frequentemente associada à supervisão de métricas previamente definidas para detectar anomalias, a observabilidade se concentra na capacidade de compreender os estados internos de um sistema, mesmo diante de eventos inesperados.

A tabela a seguir apresenta uma comparação detalhada entre os dois conceitos, destacando suas definições, objetivos, focos, abordagens e aplicações. Essa distinção ajuda a evidenciar como essas práticas, quando utilizadas de forma complementar, podem oferecer uma visão abrangente sobre a operação de sistemas modernos.

Tabela 2 – Comparação entre Monitoração e Observabilidade

Aspecto	**Monitoração**	**Observabilidade**
Definição	Processo de coletar, analisar e exibir métricas e eventos conhecidos para identificar anomalias e alertar sobre problemas.	Capacidade de entender o estado interno de um sistema por meio de dados emitidos (*logs*, métricas e *traces*), mesmo em situações inesperadas.
Objetivo	Responder à pergunta: "O que está errado?" e fornecer consciência situacional sobre falhas previamente conhecidas.	Responder às perguntas: "Por que está errado?" e "Como corrigir?", oferecendo uma visão mais profunda das causas dos problemas.
Foco	Análise de métricas específicas e eventos previamente definidos.	Análise de dados variados e não estruturados para compreender o comportamento do sistema em diferentes contextos.
Abordagem	Proativa, com foco em tendências, padrões e alertas préconfigurados. Limitada ao escopo de dados previamente configurados.	Exploratória e adaptativa, utilizando dados de telemetria para deduzir o comportamento interno de sistemas complexos e distribuídos.
Aplicação	Monitoramento de sistemas com comportamento mais estático ou previsível. Detecta falhas conhecidas de maneira eficaz.	Ideal para arquiteturas modernas, como microserviços e ambientes dinâmicos (*multinuvem*). Facilita a detecção de causas-raiz e solução de problemas complexos.

4 Ferramentas Open Source no Ecossistema de Observabilidade

A monitoração e a observabilidade tornaram-se aspectos essenciais para garantir a estabilidade e o desempenho de sistemas modernos, especialmente em arquiteturas distribuídas e baseadas em nuvem. No contexto *open source*, diversas ferramentas surgiram para atender a essas necessidades, oferecendo suporte à coleta e análise de métricas, *logs* e *traces*.

Este capítulo tem como objetivo explorar e analisar as principais ferramentas *open source* utilizadas no ecossistema de observabilidade, destacando aquelas com ampla adoção e suporte ativo da comunidade. A abordagem será focada em soluções que integram métricas, *logs* e *traces*, considerando critérios como adoção e maturidade das ferramentas, por exemplo, o Prometheus para métricas, o ELK Stack e o Loki para *logs*, e o Jaeger para *trace*. Também será considerada a integração com tecnologias *cloud-native*, enfatizando a compatibilidade com ecossistemas como Kubernetes, garantindo fácil implantação e escalabilidade. Outro critério importante será a flexibilidade e extensibilidade das ferramentas, ou seja, sua capacidade de adaptação a diferentes cenários por meio de *pluginsplugins*[1] e integrações com outras soluções do ecossistema. Além disso, será avaliada a eficiência e o desempenho das soluções, considerando o impacto no consumo de recursos e a escalabilidade em ambientes de produção.

A análise será conduzida a partir de categorias distintas, começando com ferramentas de métricas, com ênfase no Prometheus, amplamente adotado no mercado. Em seguida, serão discutidas soluções voltadas para *logs*, como ELK Stack, Fluentd e Loki, abordando seus casos de uso e limitações. Por fim, serão exploradas ferramentas para rastreamento distribuído (*tracing*), fundamentais para a observabilidade em arquiteturas complexas.

Com essa abordagem, espera-se fornecer uma visão comparativa e aprofundada sobre as principais soluções disponíveis, auxiliando profissionais e organizações na escolha das ferramentas que melhor atendem às suas necessidades de monitoração e observabilidade.

[1] São módulos ou extensões que adicionam funcionalidades a um sistema principal sem a necessidade de modificar seu código-fonte.

4.1 Ferramentas de Métricas: Foco no Prometheus

Dentre as diversas ferramentas de métricas disponíveis no ecossistema *open source*, o Prometheus se destaca como a solução mais amplamente adotada e recomendada para coleta, armazenamento e análise de métricas em tempo real. Sua popularidade e relevância são atestadas não apenas pela sua adoção massiva pela comunidade, mas também pelo seu status de projeto graduado na Cloud Native Computing Foundation [2] (CNCF) , o que reforça sua maturidade e confiabilidade.(Cloud Native Computing Foundation, 2018)

4.1.1 O Prometheus como Padrão para Métricas

O Prometheus[3] consolidou-se como solução predominante para monitoração de métricas em ambientes cloud-native, fato que se deve a características técnicas distintas e ampla adoção no mercado. Desenvolvido originalmente pela *SoundCloud* e atualmente um projeto graduado da CNCF, seu modelo de séries temporais demonstra eficiência comprovada tanto no armazenamento quanto na consulta de dados métricos (PROMETHEUS, 2023a).

Diferente de sistemas tradicionais baseados em push, o mecanismo de coleta via pull adotado pelo Prometheus oferece vantagens operacionais significativas. Essa arquitetura permite: (1) controle granular sobre a frequência de coleta, (2) descoberta automática de alvos em ambientes dinâmicos, e (3) maior resiliência frente a falhas temporárias nos sistemas monitorados (PROMETHEUS, 2023b).

A extensibilidade do sistema merece destaque particular. Através de exporters - atualmente mais de 200 disponíveis oficialmente - a ferramenta coleta métricas de sistemas heterogêneos, desde bancos de dados tradicionais até aplicações modernas em microsserviços. O PromQL, linguagem de consulta nativa, possibilita não apenas análises temporais complexas, mas também a definição de regras de alerta sofisticadas, fundamentais para operações em produção.

A integração com o ecossistema CNCF completa o quadro de vantagens. A compatibilidade nativa com Grafana[4] para visualização e com Kubernetes para orquestração de containers faz do Prometheus peça central em arquiteturas observáveis modernas. Essa interoperabilidade será especialmente relevante na proposta arquitetural discutida no Capítulo **??**, onde atuará como fonte primária de métricas.

[2] <https://www.cncf.io/>
[3] <https://prometheus.io/>
[4] <https://grafana.com/>

4.1.2 Limitações de Outras Ferramentas de Métricas

Embora existam outras ferramentas de métricas no mercado, como Graphite, InfluxDB e OpenTSDB, elas foram deixadas de lado nesta análise por várias razões. Primeiramente, ferramentas como Graphite e OpenTSDB exigem configurações mais complexas e manutenção de infraestrutura adicional, como bancos de dados dedicados. Isso aumenta a sobrecarga operacional e dificulta a adoção em ambientes dinâmicos.

Além disso, algumas ferramentas, como InfluxDB, enfrentam desafios de escalabilidade quando lidam com grandes volumes de dados ou ambientes altamente distribuídos. O Prometheus, por outro lado, foi projetado desde o início para ser escalável e eficiente.

Outro ponto importante é que, enquanto o Prometheus se integra perfeitamente com outras ferramentas do ecossistema CNCF, como Kubernetes e Grafana, outras soluções podem exigir configurações adicionais ou *plugins* para alcançar o mesmo nível de integração.

4.1.3 Integrações com Cortex, Thanos

Uma das principais vantagens do Prometheus é sua capacidade de se integrar com ferramentas como Cortex e Thanos, que estendem suas funcionalidades para cenários de longa duração e alta escalabilidade. O Cortex, por exemplo, é uma solução que permite o armazenamento de longo prazo e a agregação de métricas coletadas pelo Prometheus. Ele é especialmente útil em ambientes onde é necessário armazenar grandes volumes de dados históricos e realizar consultas complexas em múltiplos *clusters*. Além disso, o Cortex oferece suporte a *multi-tenancy*, o que o torna ideal para organizações que precisam compartilhar infraestrutura entre diferentes equipes ou projetos.(AUTHORS, 2023a)

O Thanos, por sua vez, é outra extensão popular para o Prometheus, projetada para resolver desafios de escalabilidade e longevidade de dados. Ele permite a agregação de métricas de múltiplos *clusters* de Prometheus em um único ponto de consulta, além de oferecer armazenamento de longo prazo em sistemas como o Amazon S3. O Thanos também introduz funcionalidades como compactação de dados e *downsampling*, que ajudam a reduzir o custo de armazenamento sem comprometer a qualidade das métricas.(AUTHORS, 2023b)

Essas integrações tornam o Prometheus uma escolha ainda mais poderosa para organizações que precisam de uma solução escalável e eficiente para monitoração de métricas em ambientes complexos e distribuídos.

4.1.4 Conclusão

O Prometheus se consolida como a ferramenta de métricas mais relevante no ecossistema *open source*, graças à sua maturidade, eficiência e integração com outras soluções modernas. Sua capacidade de se estender por meio de integrações com Cortex e Thanos o torna adequado para uma ampla gama de cenários, desde pequenas aplicações até grandes ambientes distribuídos. Embora outras ferramentas de métricas existam, o Prometheus se destaca como a escolha preferencial para a maioria dos casos de uso, especialmente em ambientes nativos da nuvem.

4.2 Ferramentas de *Logs* em Ambientes Cloud-Native

A gestão eficiente de *logs* em ambientes distribuídos modernos exige soluções que combinem escalabilidade, integração e custo-efetividade. Entre as opções *open source*, destacam-se três abordagens complementares: o consolidado ELK Stack, o flexível Fluentd e o emergente Loki, cada qual com características distintas que as tornam adequadas para diferentes cenários de implantação.

4.2.1 ELK Stack: Solução Completa para Análise de *Logs*

O ELK Stack[5] constitui a solução mais abrangente para gestão de *logs*, integrando três componentes especializados. O Elasticsearch oferece capacidades avançadas de indexação e busca distribuída, enquanto o Logstash provê um pipeline robusto para coleta, transformação e enriquecimento de dados através de sua extensa biblioteca de *plugins*. Complementando o ecossistema, o Kibana disponibiliza uma interface analítica poderosa com recursos de visualização interativa e detecção de anomalias.

Esta arquitetura madura, documentada em (ELASTIC, 2025), permite o tratamento de volumes massivos de dados com replicação automática e consultas complexas. No entanto, tal completude implica em desafios operacionais notáveis. A configuração em ambientes distribuídos frequentemente requer expertise especializada, e os custos de armazenamento crescem exponencialmente com a retenção de dados históricos. Estas limitações têm impulsionado a adoção de versões gerenciadas como o Elastic Cloud, que simplifica a implantação em plataformas AWS, GCP e Azure, embora com custos operacionais adicionais.

Apesar desses desafios, o ELK Stack mantém sua posição como referência para análise corporativa de *logs*, particularmente em cenários que demandam recursos avançados de segurança, conformidade e análise preditiva, frequentemente encontrados em seu pacote X-Pack. Sua capacidade de correlacionar *logs* com métricas e eventos de segu-

[5] <https://www.elastic.co/pt/elastic-stack>

rança o torna particularmente valioso em ambientes com requisitos rigorosos de auditoria e monitoração.

4.3 Fluentd: Coleta Flexível de *Logs* em Ambientes Distribuídos

O Fluentd[6] estabeleceu-se como uma solução fundamental para unificação de fluxos de dados em arquiteturas modernas. Desenvolvido originalmente na Treasure Data e agora projeto graduado da CNCF, este coletor de *logs* open source destaca-se por sua abordagem universal para coleta e roteamento de dados, particularmente em ambientes baseados em containers como Kubernetes (FLUENTD, 2025).

Sua arquitetura plugável constitui um dos principais diferenciais, com mais de 500 *plugins* oficiais que permitem integração direta com sistemas diversos, desde armazenamentos como Elasticsearch e S3 até plataformas analíticas como BigQuery. O tratamento nativo de *logs* estruturados em JSON simplifica significativamente o processamento e análise posterior, enquanto sua implementação leve em Ruby garante baixo overhead em sistemas de produção.

Apesar dessas vantagens, a flexibilidade do Fluentd traz certos desafios operacionais. Configurações avançadas em ambientes complexos frequentemente demandam ajustes manuais detalhados, e a qualidade dos *plugins* pode variar consideravelmente, exigindo avaliação criteriosa antes da implantação. Outra limitação relevante é a ausência de interface de visualização integrada, necessitando acoplamento com ferramentas como Kibana ou Grafana para análise completa dos dados.

Na prática, o Fluentd tem se mostrado especialmente valioso em três cenários principais: como agente de coleta centralizada em clusters Kubernetes, como componente de transformação em pipelines de dados em tempo real, e como ponte entre sistemas legados e plataformas cloud modernas. Suas integrações nativas com serviços como AWS CloudWatch, Google Cloud Logging e Azure Monitor facilitam sua adoção em estratégias multicloud, consolidando sua posição como ferramenta essencial para observabilidade em ambientes distribuídos.

4.4 Loki: Eficiência no Gerenciamento de *Logs* Cloud-Native

Desenvolvido pela Grafana Labs, o Loki[7] representa uma abordagem inovadora para o gerenciamento de *logs* em ambientes distribuídos modernos. Diferentemente de soluções tradicionais, o Loki foi concebido especificamente para operar com máxima efi-

[6] <https://www.fluentd.org/>
[7] <https://grafana.com/docs/loki/latest/get-started/overview/>

ciência em ambientes Kubernetes e *cloud-native*, adotando uma filosofia de design que prioriza simplicidade e baixo custo operacional (LOKI, 2025).

A arquitetura do Loki baseia-se em dois princípios fundamentais: indexação mínima e armazenamento econômico. Ao invés de indexar todo o conteúdo dos *logs*, o sistema armazena apenas metadados críticos, mantendo os dados brutos em sistemas de armazenamento de objetos como S3 ou sistemas de arquivos distribuídos. Esta abordagem, combinada com a linguagem de consulta LogQL (inspirada no PromQL do Prometheus), permite consultas poderosas com um fração do custo de soluções alternativas.

A integração nativa com o Grafana constitui um dos principais diferenciais do Loki, possibilitando a correlação direta entre *logs*, métricas e *traces* em uma única interface. Esta capacidade torna-o particularmente valioso para implementação de observabilidade unificada, especialmente quando combinado com outras ferramentas do ecossistema Grafana como Prometheus e Tempo.

Como ferramenta relativamente recente, o Loki apresenta algumas limitações dignas de nota. Sua focalização exclusiva em *logs* significa que não oferece capacidades nativas de processamento ou transformação de dados, exigindo a integração com ferramentas adicionais para esses fins. Além disso, sua comunidade e ecossistema, embora em crescimento, ainda não atingiram a maturidade de soluções mais estabelecidas como o ELK Stack.

Na prática, o Loki tem se mostrado ideal para três cenários principais: monitoração de *logs* em clusters Kubernetes, onde sua integração com Prometheus oferece vantagens significativas; análise correlacionada de telemetria através do Grafana; e armazenamento de longo prazo de *logs* com requisitos rigorosos de custo-efetividade. Sua compatibilidade com armazenamentos cloud como S3 e GCS, combinada com a capacidade de operar em modo *serverless*, posiciona-o como uma solução particularmente adequada para arquiteturas cloud-native modernas.ações com outras ferramentas do ecossistema CNCF, como o Prometheus e o Grafana.

4.4.1 Comparação das Ferramentas

A Tabela 3 resume as principais características das ferramentas de *logs* discutidas, destacando aspectos como escalabilidade, facilidade de uso, custo, visualização, casos de uso e integração com ambientes *cloud*. Essa comparação oferece uma visão geral das vantagens e limitações de cada solução.

4.4.2 Conclusão

Cada uma das ferramentas analisadas possui pontos fortes e é mais adequada para diferentes cenários. O ELK Stack se destaca como uma solução completa e poderosa para análise de logs, oferecendo suporte a dashboards e alertas avançados. No entanto, sua

Característica	**ELK Stack**	**Fluentd**	**Loki**
Escalabilidade	Alta (Elasticsearch)	Alta	Alta
Facilidade de Uso	Complexo (config. inicial)	Moderado (plugins necessários)	Leve e simples
Custo	Alto (armazenamento e infra)	Baixo	Baixo (armazenamento eficiente)
Visualização	Kibana (poderosa e flexível)	Depende de integração (Kibana)	Grafana (integrado)
Casos de Uso	Análise corporativa, dashboards	Coleta centralizada, pipelines	*Logs* em Kubernetes, Grafana
Integração Cloud	Elastic Cloud (AWS, GCP, Azure)	Multi-cloud (plugins)	Multi-cloud (S3, GCS)

Tabela 3 – Comparação entre ELK Stack, Fluentd e Loki

complexidade e custo podem ser um desafio para ambientes menores ou com restrições orçamentárias. O Fluentd, por sua vez, é uma excelente escolha para coleta e roteamento de logs em ambientes distribuídos, especialmente quando integrado com Kubernetes. Sua flexibilidade e leveza o tornam uma opção atraente para pipelines de dados. Já o Loki é a melhor opção para ambientes *cloud-native* e Kubernetes, particularmente quando a integração com Grafana e o baixo custo de armazenamento são prioridades. É uma ferramenta moderna e eficiente, embora ainda em evolução.

A escolha entre essas ferramentas deve levar em consideração as necessidades específicas do ambiente, como o volume de logs, a integração com outras ferramentas de observabilidade e o orçamento disponível. Em muitos casos, combinar duas ou mais ferramentas, como o Fluentd para coleta e o Loki para armazenamento, pode resultar na solução mais eficaz.

4.5 Ferramentas de *Tracing* em Sistemas Distribuídos

O rastreamento distribuído tornou-se componente essencial para observabilidade em arquiteturas modernas, permitindo acompanhar o fluxo de requisições através de múltiplos serviços e identificar pontos de latência ou falha. Entre as soluções *open source*, Jaeger e Zipkin destacam-se como as principais alternativas para implementação de sistemas completos de *tracing*, cada qual com características distintas que as tornam adequadas para diferentes cenários.

4.5.1 Jaeger

O Jaeger[8] é uma ferramenta de rastreamento distribuído desenvolvida originalmente pela Uber e agora mantida pela Cloud Native Computing Foundation (CNCF). Ele é amplamente utilizado em ambientes *cloud-native* e *Kubernetes*.(JAEGER, 2025)

Entre suas vantagens, destaca-se a integração nativa com *Kubernetes*, permitindo descoberta de serviços e escalabilidade automática. O suporte a múltiplos *backends* de armazenamento, como Cassandra, Elasticsearch e Kafka, garante flexibilidade na escolha da infraestrutura. A interface gráfica do Jaeger facilita a visualização de *traces* e a análise de desempenho, além de ser compatível com o OpenTelemetry, permitindo integração com outras ferramentas de observabilidade.

No entanto, sua configuração pode ser complexa, especialmente em ambientes de grande escala, e a necessidade de um *backend* de armazenamento pode aumentar a carga operacional. O Jaeger é amplamente utilizado para rastreamento de requisições em *microservices*, identificação de gargalos em sistemas distribuídos e integração com *Kubernetes* e ambientes *cloud-native*. Ele suporta armazenamento em sistemas como Elasticsearch e Cassandra, que podem ser hospedados em nuvens como AWS, GCP e Azure. Além disso, conta com versões gerenciadas, como o Jaeger Operator para Kubernetes.

4.5.2 Zipkin

O Zipkin[9] é uma das primeiras ferramentas de rastreamento distribuído *open source*, desenvolvida pelo Twitter. Ele é conhecido por sua simplicidade e facilidade de uso.

Sua principal vantagem é a configuração simplificada, tornando-o ideal para pequenos e médios ambientes. Suporta múltiplos backends de armazenamento, como MySQL, Cassandra e Elasticsearch, e possui uma comunidade ativa, garantindo ampla documentação e suporte.

Entretanto, suas funcionalidades são mais limitadas em comparação ao Jaeger, não possuindo suporte nativo avançado para Kubernetes. Além disso, sua interface gráfica é mais básica e menos poderosa. O Zipkin é recomendado para rastreamento de requisições em sistemas menores ou menos complexos e em cenários onde a simplicidade e a facilidade de uso são prioridade. Pode ser integrado com backends como Elasticsearch e Cassandra, que podem ser hospedados em nuvens como AWS, GCP e Azure.

[8] <jaegertracing.io>
[9] <zipkin.io>

4.5.3 Comparação das Ferramentas

A Tabela 4 apresenta uma comparação entre três ferramentas amplamente utilizadas para rastreamento distribuído.

Característica	**Jaeger**	**Zipkin**
Escalabilidade	Alta (grandes sistemas)	Moderada (pequenos/médios sistemas)
Facilidade de Uso	Moderada (config. complexa)	Alta (simples e fácil)
Backend	Cassandra, Elasticsearch, Kafka	MySQL, Cassandra, Elasticsearch
UI	Poderosa e intuitiva	Simples e básica d
Integração Cloud	Alta (Kubernetes, multi-cloud)	Moderada (backends em nuvem)
Casos de Uso	Microsserviços, Kubernetes	Sistemas menores

Tabela 4 – Comparação entre Jaeger, Zipkin

4.5.4 Conclusão

Cada uma das ferramentas analisadas possui suas particularidades, sendo mais adequada para diferentes cenários. O Jaeger, por exemplo, é ideal para ambientes *cloud-native* e Kubernetes, especialmente em sistemas de grande escala que exigem funcionalidades avançadas e uma interface de usuário poderosa. Já o Zipkin se destaca como uma excelente escolha para sistemas menores ou menos complexos, onde a simplicidade e a facilidade de uso são prioritárias.

A escolha entre essas ferramentas deve ser feita com base nas necessidades específicas do ambiente, considerando aspectos como a complexidade do sistema, o volume de dados e a integração com outras soluções de observabilidade.

5 Fundamentação Teórica para a Arquitetura de Observabilidade

A seleção das referências teóricas seguiu critérios rigorosos de relevância temática, atualidade e aplicabilidade prática à arquitetura de observabilidade proposta neste trabalho. Durante o levantamento bibliográfico, observou-se uma escassez significativa de literatura acadêmica acessível que tratasse de maneira aprofundada as abordagens modernas de observabilidade. Buscas realizadas nas bases CAPES Periódicos, IEEE Xplore e ACM Digital Library, com os termos *observability AND lgtm* e *observability AND opentelemetry*, resultaram em poucos trabalhos disponíveis em acesso aberto e com aplicabilidade direta ao contexto do projeto.

Diante disso, optou-se por fundamentar a pesquisa em três obras técnicas amplamente reconhecidas na engenharia de software contemporânea, que consolidam práticas, padrões e ferramentas adotadas na indústria para construção de sistemas observáveis.

5.1 Referências Fundamentais

5.1.1 Observability Engineering (Majors et al., 2022)

A obra de Majors, Benson e Schuyler oferece uma base conceitual sólida para a arquitetura modular adotada neste trabalho. Os autores exploram a separação entre geração, coleta e armazenamento de telemetria, destacando a importância de instrumentação não intrusiva. Além disso, o livro apresenta técnicas de correlação entre métricas, *logs* e *traces*, fundamentais para análises mais eficientes e diagnósticos precisos em sistemas distribuídos.

5.1.2 Cloud-Native Observability with OpenTelemetry (Boten, 2021)

O livro de Boten fornece diretrizes práticas para implementação de observabilidade baseada em OpenTelemetry. Seus conteúdos foram particularmente relevantes para a configuração do *collector*, definição de estratégias de bufferização utilizando Kafka e padronização do transporte de dados por meio do protocolo OTLP. A obra também discute a interoperabilidade entre diferentes fontes e destinos de telemetria, alinhando-se com os princípios de portabilidade e modularidade adotados na arquitetura.

5.1.3 O11Y Explained (Mao, 2021)

Já o trabalho de Mao foi essencial para a avaliação e seleção dos backends de observabilidade, especialmente no que se refere à stack LGTM (Loki, Grafana, Tempo, Mimir). O autor discute critérios de comparação entre soluções, padrões de consulta unificada e os principais *trade-offs* de cada abordagem, oferecendo insumos valiosos para decisões arquiteturais informadas.

5.2 Correspondência com a Arquitetura Proposta

A Tabela 5 sintetiza como cada decisão arquitetural está respaldada pelas obras analisadas, evidenciando a relação entre teoria e prática no desenvolvimento da solução.

Tabela 5 – Correspondência entre fundamentação teórica e componentes arquiteturais

Componente	**Referência**	**Contribuição**
OpenTelemetry Collector	Boten (2021)	Arquitetura modular de processamento
Buffer Kafka	Majors et al. (2022)	Padrão de desacoplamento
Stack LGTM	Mao (2021)	Avaliação comparativa

As obras referenciais utilizadas neste trabalho, embora não façam parte do circuito acadêmico tradicional, oferecem uma base teórica robusta, amplamente validada na prática profissional. Elas cobrem desde os fundamentos conceituais da observabilidade moderna até aspectos técnicos e operacionais cruciais para a implementação eficaz da arquitetura proposta. Sua adoção se mostrou adequada frente à escassez de literatura científica acessível e permitiu a construção de uma solução alinhada com os padrões atuais da engenharia de software distribuído.

6 Arquitetura *Backend* de Observabilidade

A arquitetura proposta tem como principal objetivo oferecer um *backend* de observabilidade moderno, modular e extensível, capaz de atender às necessidades de sistemas distribuídos contemporâneos. A solução visa centralizar e padronizar a coleta, o processamento e o armazenamento de sinais observáveis: *logs*, métricas e *traces* em uma infraestrutura resiliente, eficiente e escalável.

Um dos pilares fundamentais da proposta é a capacidade de operar como um serviço de observabilidade, oferecendo interfaces padronizadas para ingestão de dados a partir de aplicações instrumentadas. Com isso, busca-se reduzir o acoplamento entre os sistemas de origem e os mecanismos de observabilidade, promovendo maior flexibilidade, portabilidade e manutenibilidade.

A arquitetura também foi concebida para suportar escalabilidade horizontal, de modo que componentes críticos, como *collectors*, armazenamentos e processadores, possam ser replicados ou distribuídos conforme a demanda. Isso é desejável para lidar com grandes volumes de dados gerados por aplicações em produção, especialmente em ambientes baseados em microsserviços e infraestrutura elástica.

Além disso, o projeto busca garantir suporte nativo aos três principais sinais de observabilidade: *logs*, métricas e *traces*. A integração desses sinais é projetada de forma coesa, com correlação entre eventos, agregação eficiente de dados e interoperabilidade com ferramentas e padrões abertos. Essa integração unificada é essencial para fornecer uma visão abrangente do comportamento do sistema e acelerar a detecção e resolução de falhas.

O desacoplamento entre as camadas de coleta, processamento e exportação dos dados é um aspecto central da arquitetura. Isso permite que cada componente seja atualizado, escalado ou substituído de forma independente, reduzindo a complexidade operacional e aumentando a confiabilidade geral da solução.

O backend de observabilidade, baseado na pilha LGTM (Loki, Grafana, Tempo e Mimir), foi implantado em um único *cluster Kubernetes*. Para validação da pipeline de telemetria, a aplicação observada foi executada em outro *cluster*, de onde os dados (logs, traces e métricas) foram coletados e enviados ao backend central. Essa separação evidencia que a arquitetura proposta suporta cenários *multi-cluster*, mantendo um backend unificado capaz de receber dados de diferentes ambientes.

Nos tópicos a seguir, são descritos os componentes do backend de observabilidade e a forma como eles foram integrados para garantir a coleta e correlação dos sinais obser-

váveis.

6.1 Componentes da Arquitetura

A arquitetura de *backend* de observabilidade proposta é composta por múltiplos componentes especializados, organizados em camadas para garantir flexibilidade, escalabilidade e desacoplamento. Esta seção descreve os principais blocos que integram a solução, conforme ilustrado na Figura 3.

Os SDKs do OpenTelemetry são responsáveis por instrumentar aplicações, capturando métricas, *logs* e *traces* diretamente do código-fonte. Esses dados são então encaminhados ao *Collector*, que atua como um componente centralizado para recepção, processamento e exportação dos sinais de observabilidade. O *Collector* pode ser implantado tanto como um agente local em cada *cluster* e sua arquitetura permite a utilização de múltiplos *receivers*, *processors* e *exporters* (OPENTELEMETRY, 2025).

6.1.1 Visão Geral da Aplicação Observada

Para validar a arquitetura de observabilidade proposta, foi utilizada uma aplicação real implementada como um microsserviço em *Python*, baseado no framework *FastAPI*. Essa API REST é responsável por criar, editar e remover monitores na plataforma *AppDynamics*, atuando como intermediária entre clientes consumidores e a API oficial do AppDynamics.

A aplicação foi implantada em um *cluster Kubernetes* e instrumentada com *OpenTelemetry* para coleta de logs estruturados, geração de traces distribuídos e métricas básicas, como latência de requisições e contagem de erros. Os dados coletados foram enviados para a pilha LGTM (Loki, Grafana, Tempo e Mimir), permitindo correlacionar eventos e visualizar o comportamento da aplicação em tempo real.

O foco principal foi avaliar a eficácia da arquitetura de observabilidade na coleta e análise dos sinais, ao invés de realizar estudos de desempenho da aplicação em si.

6.1.2 Apache Kafka: Pipeline de Dados em Tempo Real

O Apache Kafka é uma plataforma distribuída de streaming de eventos projetada para alta performance, tolerância a falhas e escalabilidade horizontal. Ele opera como um sistema de mensagens baseado no modelo *publish-subscribe*, permitindo que diferentes sistemas produtores enviem eventos para tópicos que são consumidos de forma independente por múltiplos consumidores.

No contexto deste trabalho, o Kafka foi utilizado como mecanismo de ingestão assíncrona de logs e traces no pipeline de observabilidade, conectando o OpenTelemetry

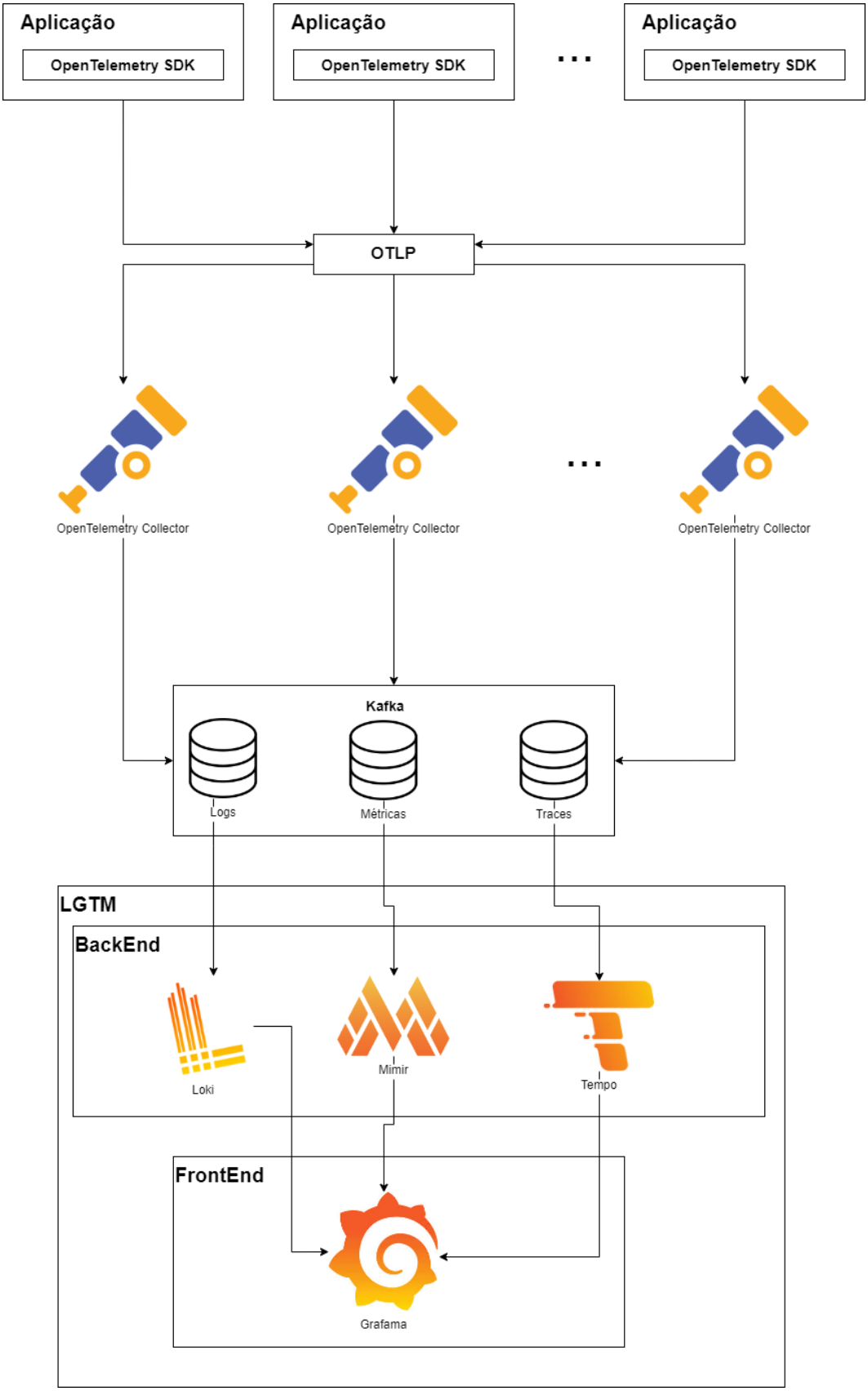

Figura 3 – Componentes principais do *backend* de observabilidade.

Collector aos backends Loki e Tempo. Essa escolha se justifica pela necessidade de garantir resiliência no tráfego de dados de telemetria e desacoplamento entre componentes da arquitetura.

Além disso, o Kafka facilita a escalabilidade do sistema e reduz a perda de dados durante picos de carga, ao permitir o armazenamento intermediário dos eventos capturados antes da persistência final. Sua arquitetura distribuída e seu ecossistema de conectores também favorecem a integração com sistemas como o Kubernetes, Prometheus, Grafana e ferramentas de análise em tempo real.

6.2 Instrumentação para coleta de Traces

Para habilitar a geração e exportação de *traces*, é necessário configurar corretamente o SDK do OpenTelemetry. A implementação prática apresentada na Listing 6.1 demonstra a instalação das bibliotecas necessárias para o ambiente Python.

Listing 6.1 – Instalação dos pacotes OpenTelemetry.

```
pip install opentelemetry-api
pip install opentelemetry-sdk
pip install opentelemetry-exporter-otlp
```

O primeiro passo no código consiste em importar os módulos necessários, conforme exemplificado na Listing 6.2.

Listing 6.2 – Importação dos módulos e variáveis de ambiente.

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    BatchSpanProcessor,
    ConsoleSpanExporter,
    SimpleSpanProcessor
)
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
    OTLPSpanExporter
from opentelemetry.sdk.resources import Resource
from app.envs import (
    OTEL_EXPORTER_OTLP_ENDPOINT,
    OTEL_EXPORTER_OTLP_INSECURE,
    OTEL_SERVICE_NAME
)
```

A configuração do *TracerProvider*, que define os atributos do serviço monitorado, está descrita na Listing 6.3.

Listing 6.3 – Configuração do TracerProvider.

```
provider = TracerProvider(
    resource=Resource.create({
        "service.name": OTEL_SERVICE_NAME,
    })
)
```

A configuração do exportador OTLP (OpenTelemetry Protocol), responsável por enviar os dados ao OpenTelemetry Collector, é apresentada na Listing 6.4.

Listing 6.4 – Configuração do exportador OTLP.

```
otlp_exporter = OTLPSpanExporter(
    endpoint=OTEL_EXPORTER_OTLP_ENDPOINT,
    insecure=OTEL_EXPORTER_OTLP_INSECURE
)
batch_processor = BatchSpanProcessor(otlp_exporter)
provider.add_span_processor(batch_processor)
```

A Listing 6.5 mostra o registro do provedor como padrão global da aplicação, garantindo consistência no contexto de rastreamento.

Listing 6.5 – Registro do provedor como padrão global.

```
trace.set_tracer_provider(provider)
tracer = trace.get_tracer("global")
```

Essa configuração permite que as informações de rastreamento fluam de maneira estruturada desde a aplicação até o *backend* de observabilidade, possibilitando análises detalhadas sobre a execução dos serviços.

6.2.1 Visualização no Grafana

Com a instrumentação ativa, os *traces* gerados pela aplicação são encaminhados ao *Collector* e armazenados no *backend* Tempo. A Figura 4 apresenta a interface do Grafana exibindo esses *traces*, permitindo a análise detalhada das requisições, bem como a identificação de gargalos e anomalias no fluxo de execução dos serviços.

A Figura 5 apresenta o detalhamento de um *trace* individual, permitindo a inspeção de cada operação realizada ao longo de uma requisição. Os *span* ilustram etapas como a conexão ao banco de dados e a execução de consultas SQL, com atributos contextuais que facilitam o diagnóstico e a análise de desempenho. Além disso, é possível correlacionar cada *span* com seus respectivos *logs*, viabilizando uma observabilidade completa e integrada.

A Figura 6 mostra o *trace* representado como um grafo de spans, onde cada nó corresponde a uma operação monitorada. Essa visualização facilita a compreensão do

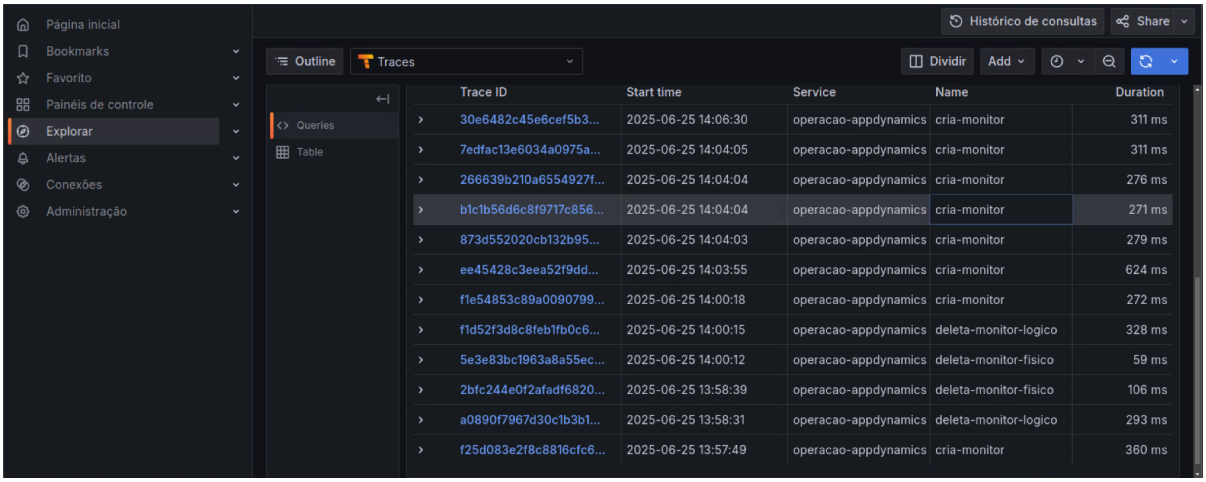

Figura 4 – Visualização dos *traces* capturados pelo OpenTelemetry no painel do Grafana, integrando-se com o *backend* Tempo.

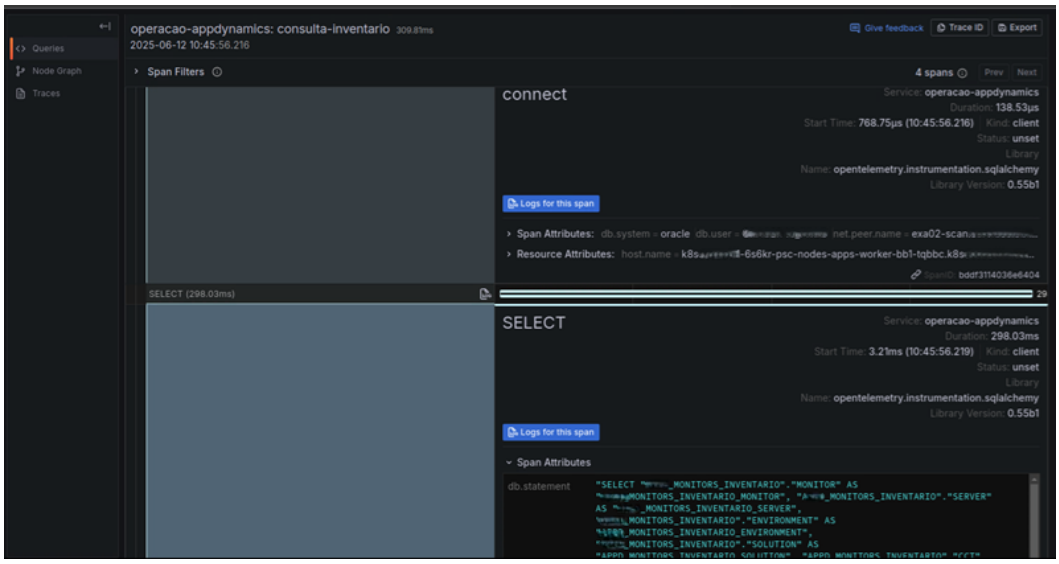

Figura 5 – Detalhamento de um trace individual no Tempo. Cada bloco representa um *span*, contendo informações como duração, tipo de operação (por exemplo, conexão ou consulta ao banco de dados), atributos do recurso e do span, além de permitir a correlação com logs.

fluxo de execução dentro do sistema, permitindo identificar visualmente quais operações consomem mais tempo ou representam gargalos de desempenho. Assim, a integração da instrumentação com o *backend* Tempo e a visualização via Grafana fornece uma base sólida para análise de comportamento em ambientes distribuídos.

6.3 Instrumentação para coleta de Métricas

Para habilitar a coleta e exportação de *métricas*, apresenta-se uma implementação prática em Python, demonstrando a criação do provedor de métricas, a definição dos

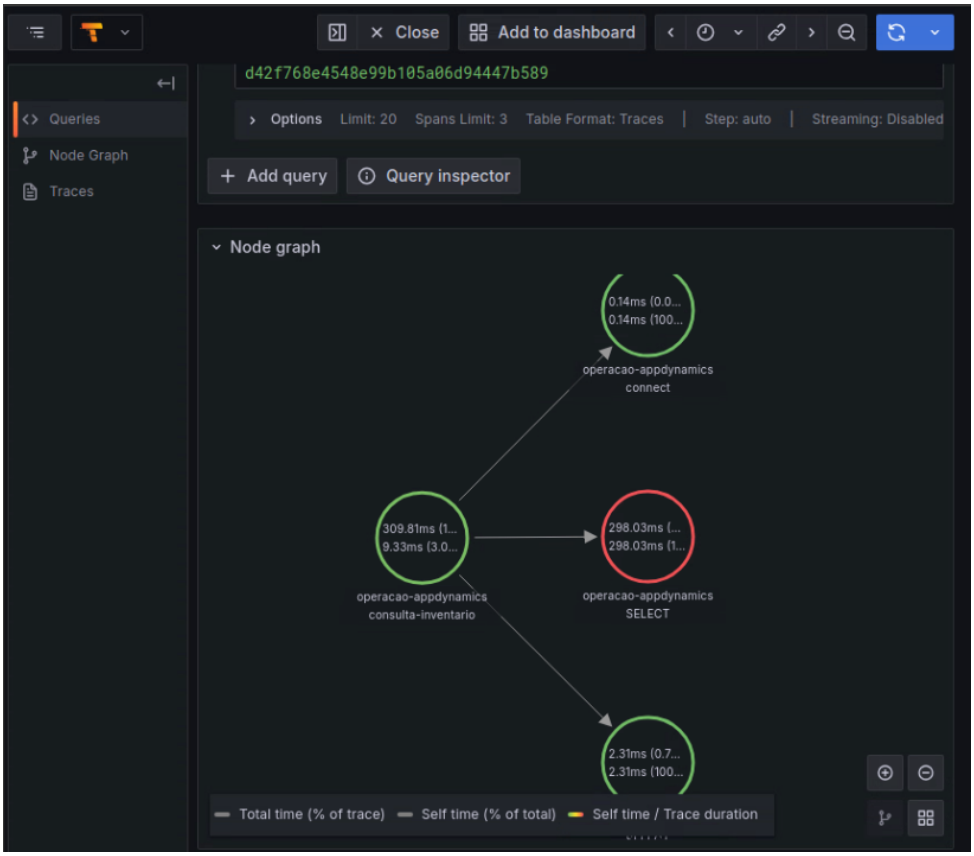

Figura 6 – Visualização em grafo de um trace no Grafana. Os nós representam spans individuais e as arestas indicam a relação entre operações encadeadas. As cores ajudam a identificar o tempo gasto em cada operação, destacando eventuais gargalos.

leitores de exportação e a configuração do exportador OTLP.

As bibliotecas necessárias são as mesmas utilizadas na instrumentação de *traces*, com a adição do módulo de métricas, conforme ilustrado na Listing 6.6.

Listing 6.6 – Instalação dos pacotes OpenTelemetry.

```
pip install opentelemetry-api
pip install opentelemetry-sdk
pip install opentelemetry-exporter-otlp
```

A configuração tem início com a importação dos módulos responsáveis pela instrumentação de métricas, conforme apresentado na Listing 6.7.

Listing 6.7 – Importação dos módulos para instrumentação de métricas.

```
from opentelemetry import metrics
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.sdk.metrics.export import
    PeriodicExportingMetricReader
```

```
from opentelemetry.exporter.otlp.proto.grpc.metric_exporter
   import OTLPMetricExporter
from opentelemetry.sdk.resources import Resource
from app.envs import (
    OTEL_EXPORTER_OTLP_ENDPOINT,
    OTEL_EXPORTER_OTLP_INSECURE,
    OTEL_SERVICE_NAME
)
```

O próximo passo consiste em configurar o recurso associado ao serviço, permitindo identificar a origem das métricas no *backend* de observabilidade, conforme demonstrado na Listing 6.8.

Listing 6.8 – Configuração do recurso para identificação do serviço.

```
resource = Resource(attributes={
    "service.name": OTEL_SERVICE_NAME
})
```

Em seguida, é criado o exportador OTLP, que envia as métricas ao OpenTelemetry *Collector* por meio do protocolo OTLP via gRPC, como mostrado na Listing 6.9.

Listing 6.9 – Configuração do exportador OTLP para métricas.

```
otlp_exporter = OTLPMetricExporter(
    endpoint=OTEL_EXPORTER_OTLP_ENDPOINT,
    insecure=OTEL_EXPORTER_OTLP_INSECURE
)
```

O leitor de métricas (*metric reader*) é configurado para exportar os dados periodicamente para o Collector, conforme exemplificado na Listing 6.10.

Listing 6.10 – Criação do leitor periódico de métricas.

```
reader = PeriodicExportingMetricReader(otlp_exporter)
```

Por fim, o `MeterProvider` é criado e registrado como provedor global, permitindo que toda a instrumentação da aplicação compartilhe o mesmo contexto de coleta de métricas, como exibido na Listing 6.11.

Listing 6.11 – Registro do provedor global de métricas.

```
provider = MeterProvider(
    resource=resource,
    metric_readers=[reader]
)

metrics.set_meter_provider(provider)
```

```
meter = metrics.get_meter("global")
```

Com essa configuração, a aplicação está apta a coletar e exportar métricas customizadas e automáticas para o OpenTelemetry Collector.

6.3.1 Visualização no Grafana

A Figura 7 apresenta uma métrica coletada automaticamente pela instrumentação HTTP com OpenTelemetry, representando o total de *bytes* recebidos por requisições de clientes. Esses dados são expostos como séries temporais e visualizados no Grafana, com *backend* de armazenamento configurado via Prometheus ou Mimir. O uso de métricas permite acompanhar o comportamento da aplicação ao longo do tempo, identificar padrões de uso e detectar anomalias de forma proativa.

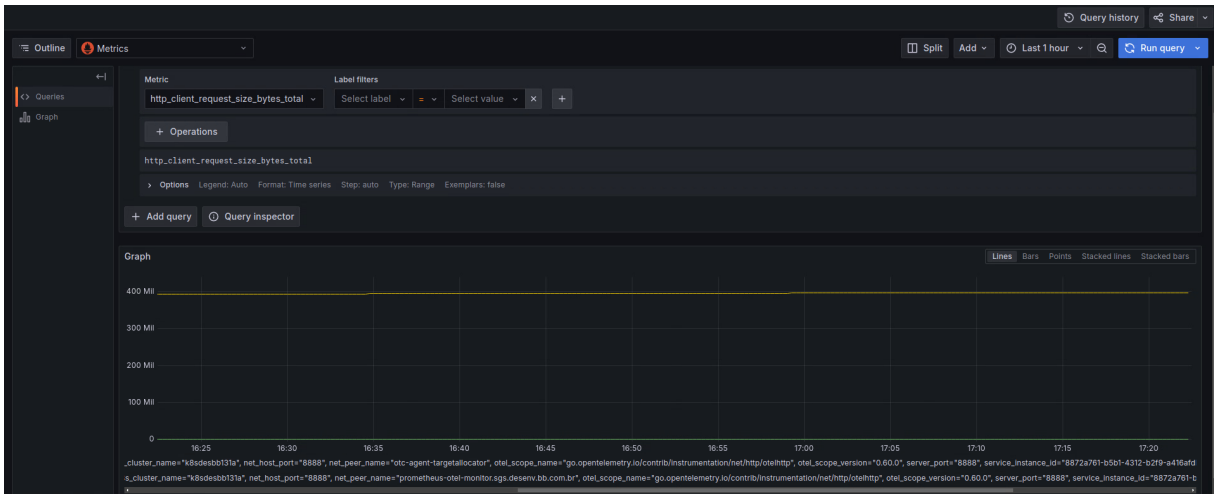

Figura 7 – Painel de métricas exibido no Grafana. A consulta mostra a evolução do total de bytes recebidos por requisições HTTP.

6.4 Instrumentação para coleta de *Logs*

Diferentemente das métricas e dos *traces*, a coleta de logs no Kubernetes não requer, em geral, instrumentação explícita no código-fonte da aplicação. Em conformidade com as boas práticas do ecossistema *cloud-native*, os contêineres escrevem os logs diretamente para a saída padrão (`stdout` e `stderr`), e esses dados são gerenciados pelo runtime de contêineres.

No contexto da arquitetura proposta, os logs são coletados diretamente no nível do nó, por meio de um agente do OpenTelemetry Collector implantado como *DaemonSet* no *cluster* Kubernetes. Esse agente utiliza o receptor `filelog` para monitorar os arquivos de *log* localizados em `/var/log/containers/`, onde o runtime do Kubernetes armazena as saídas dos *pods* em execução, conforme exemplificado na Listing 6.12.

Listing 6.12 – Trecho de configuração do receiver filelog no OpenTelemetry Collector.

```
receivers:
  filelog:
    include: [ /var/log/containers/*.log ]
    start_at: beginning
    operators:
      - type: json_parser
        parse_from: body
```

6.4.1 Visualização no Grafana

A Figura 8 apresenta a visualização dos *logs* coletados diretamente dos *pods* do Kubernetes por meio do Loki. Os registros são enriquecidos com metadados como nome do *pod*, UID, sistema operacional e *timestamp* observado. No painel, é possível aplicar filtros por nível de severidade (*info*, *error*), explorar padrões de ocorrência e buscar mensagens específicas. Essa abordagem facilita a correlação de logs com métricas e *traces*, completando a estratégia de observabilidade baseada em três pilares.

Figura 8 – Exibição de *logs* estruturados no Grafana, com base nos dados armazenados no Loki. É possível aplicar filtros por nível de severidade, realizar buscas por atributos e analisar o volume de *logs* ao longo do tempo. A coleta é realizada diretamente dos *pods* do Kubernetes, com enriquecimento automático de metadados.

6.5 Configuração do OpenTelemetry Collector

O OpenTelemetry Collector exerce um papel central na arquitetura de observabilidade proposta, atuando como ponto de coleta, transformação e roteamento dos três sinais observáveis. Implantado no ecossistema Kubernetes, sua configuração foi realizada

por meio de arquivos `values.yaml`, permitindo a definição de pipelines específicas para cada tipo de sinal.

Além da exportação direta aos *backends* (Grafana Tempo, Mimir e Loki), a arquitetura utiliza uma fila Kafka como camada intermediária para o processamento de eventos, promovendo maior desacoplamento, tolerância a falhas e elasticidade no fluxo de dados.

6.5.1 Pipeline de Traces

A recepção de *traces* ocorre via protocolo OTLP, com exportação intermediada por Kafka. O *backend* final é o Grafana Tempo, que consome os dados do tópico apropriado. A Listing 6.13 apresenta a configuração correspondente.

Listing 6.13 – Pipeline de traces com Kafka como intermediário.

```
receivers:
  otlp:
    protocols:
      grpc:

exporters:
  kafka/traces:
    brokers: ["kafka:9092"]
    topic: traces

service:
  pipelines:
    traces:
      receivers: [otlp]
      exporters: [kafka/traces]
```

6.5.2 Pipeline de Métricas

As métricas são recebidas via OTLP e também exportadas para um tópico Kafka, permitindo que múltiplos consumidores possam processá-las, inclusive o Grafana Mimir, por meio de um conector de Prometheus Remote Write. A Listing 6.14 mostra a configuração deste pipeline.

Listing 6.14 – Pipeline de métricas com Kafka.

```
receivers:
  otlp:
    protocols:
```

```
      grpc:

exporters:
  kafka/metrics:
    brokers: ["kafka:9092"]
    topic: metrics

service:
  pipelines:
    metrics:
      receivers: [otlp]
      exporters: [kafka/metrics]
```

6.5.3 Pipeline de Logs

Os logs são coletados diretamente dos nós do Kubernetes, por meio do *receiver* `filelog`, que monitora os arquivos de *log* dos *containers*. Após processamento, os eventos de *log* são enviados a um tópico Kafka dedicado, que será posteriormente consumido pelo *backend Loki*.

Listing 6.15 – Pipeline de logs com coleta de arquivos e exportação para Kafka.

```
receivers:
  filelog:
    include: [ /var/log/containers/*.log ]
    start_at: beginning
    operators:
      - type: json_parser
        parse_from: body

exporters:
  kafka/logs:
    brokers: ["kafka:9092"]
    topic: logs

service:
  pipelines:
    logs:
      receivers: [filelog]
      exporters: [kafka/logs]
```

6.5.4 Considerações Arquiteturais

Essa configuração permite desacoplar os fluxos de coleta e armazenamento por meio da utilização do Kafka como intermediário. Dessa forma, os dados observáveis não são enviados diretamente aos sistemas de backend, mas publicados em tópicos dedicados, permitindo escalabilidade horizontal, balanceamento de carga, e tolerância a falhas.

Além disso, cada pipeline é definida de forma independente, o que garante flexibilidade na composição de diferentes estratégias de exportação, filtragem e transformação dos dados.

Para desacoplar a coleta da persistência e possibilitar maior resiliência e escalabilidade, é utilizado um sistema de mensageria baseado no Apache Kafka. Esse componente atua como um *buffer* intermediário, absorvendo variações de carga e garantindo a entrega assíncrona e durável dos dados de observabilidade aos componentes, conforme ilustrado no Listing 6.15.

6.5.5 Backend LGTM: Loki, Mimir e Tempo

O *backend* da arquitetura proposta é estruturado com base na stack LGTM, composta pelas ferramentas Loki, Mimir e Tempo, desenvolvida e mantida pela Grafana Labs. Cada uma dessas soluções é especializada em um tipo de sinal de observabilidade, e juntas oferecem uma base unificada, escalável e eficiente para a ingestão, armazenamento e consulta de *logs*, métricas e *traces*.

6.5.5.1 Configuração do Grafana Loki

O Grafana Loki é o componente responsável pelo armazenamento e indexação dos eventos de *logs* coletados no ambiente Kubernetes. Sua principal característica é a eficiência no armazenamento, pois evita o uso de índices pesados, priorizando a organização dos dados por meio de rótulos (*labels*) configuráveis.

Na arquitetura proposta, os *logs* são coletados pelos agentes OpenTelemetry Collector diretamente nos nós do *cluster* e exportados para o Apache Kafka. O Grafana Loki atua como consumidor dos eventos presentes no tópico `logs`, utilizando o `loki.source.kafka` como mecanismo de ingestão.

6.5.5.2 Ingestão via Kafka

A configuração que define a fonte Kafka no arquivo `loki.yaml`, responsável por consumir os eventos do tópico e transformá-los em entradas válidas de log, está apresentada no Listing 6.16:

Listing 6.16 – Fonte Kafka no Loki.

```
loki:
  source:
    kafka:
      brokers:
        - kafka:9092
      topics:
        - logs
      group_id: loki-consumer-group
      labels:
        job: kubernetes-logs
        container: '{{ .container_name }}'
        namespace: '{{ .kubernetes_namespace_name }}'
        pod: '{{ .kubernetes_pod_name }}'
      json:
        expressions:
          ts: timestamp
          line: message
      timestamp:
        path: ts
        format: RFC3339
```

Neste exemplo, os *logs* consumidos são esperados no formato JSON e devem conter os campos `timestamp` e `message`. As expressões JSON definem o mapeamento dos dados brutos para o modelo utilizado pelo Loki.

6.5.5.3 Armazenamento dos *logs*

Para ambientes com requisitos de persistência, o Loki pode ser configurado para armazenar os blocos localmente (modo `filesystem`) ou em soluções como o Amazon S3. No exemplo apresentado no Listing 6.17, o armazenamento ocorre no sistema de arquivos:

Listing 6.17 – Configuração local de armazenamento no Loki.

```
storage_config:
  boltdb:
    directory: /var/loki/index
  filesystem:
    directory: /var/loki/chunks
```

Em cenários com múltiplos consumidores ou ambientes distribuídos, o Loki também pode operar no modo escalável com o Loki Stack *(ingester, distributor, querier, etc.)*, embora neste trabalho tenha-se optado por uma instância simplificada para foco na coleta e visualização.

6.5.5.4 Visualização no Grafana

Uma vez que os *logs* são armazenados pelo Loki, a visualização é realizada no Grafana por meio de painéis e consultas baseadas em `LogQL`. A correlação com métricas e *traces* também pode ser feita a partir de labels como `traceID`, permitindo rastrear execuções completas no sistema com base em um único *log*.

6.5.6 Configuração do Grafana Mimir

O Grafana Mimir é utilizado como *backend* de armazenamento e consulta de métricas na arquitetura de observabilidade proposta. Compatível com o protocolo Prometheus Remote Write, o Mimir permite escalar horizontalmente a coleta e análise de métricas, oferecendo alta disponibilidade e retenção eficiente para ambientes de grande volume, como clusters Kubernetes.

6.5.6.1 Ingestão via Prometheus Remote Write

A ingestão de métricas é realizada a partir do OpenTelemetry Collector, que exporta os dados para um tópico Kafka. Um conector específico ou um componente intermediário consome esse tópico e envia as métricas para o Mimir por meio do protocolo `remote_write`, conforme ilustrado no Listing 6.18:

Listing 6.18 – Exportador Prometheus Remote Write no Collector.

```
exporters:
  prometheusremotewrite:
    endpoint: http://mimir:9009/api/v1/push
```

Do lado do Mimir, a configuração dos blocos básicos é definida no arquivo `mimir.yaml`, contemplando os serviços de ingestão, armazenamento e consulta.

6.5.6.2 Configuração básica do Mimir

A seguir, um exemplo simplificado de configuração do Mimir em modo monolítico, com ingestão, compactação e armazenamento em disco local, apresentado no Listing 6.19:

Listing 6.19 – Configuração do Mimir em modo monolítico.

```
multitenancy_enabled: false

ingester:
  lifecycler:
    ring:
      kvstore:
        store: inmemory
```

```
      replication_factor: 1
  wal:
    enabled: true

blocks_storage:
  backend: filesystem
  filesystem:
    dir: /data/mimir/tsdb
  tsdb:
    retention_period: 7d
    dir: /data/mimir/tsdb
```

Essa configuração permite o armazenamento de métricas em arquivos locais por até 7 dias.

6.6 Configuração do Grafana Tempo

O Grafana Tempo é utilizado como *backend* para armazenamento e análise de *traces* distribuídos. Desenvolvido para ser altamente escalável, o Tempo é compatível com o protocolo OTLP e pode consumir dados diretamente de tópicos Kafka, tornando-se uma escolha ideal para ambientes observáveis baseados em mensageria.

6.6.1 Ingestão via Kafka

Na arquitetura proposta, os eventos de *tracing* são publicados no tópico `traces` do Apache Kafka pelo OpenTelemetry Collector. O Tempo, por sua vez, é configurado para consumir esse tópico utilizando o componente `tempo.receiver.kafka`, conforme apresentado no Listing 6.20.

Listing 6.20 – Ingestão de traces no Tempo via Kafka.

```
receivers:
  kafka:
    brokers:
      - kafka:9092
    topic: traces
    group_id: tempo-consumer
    encoding: otlp_proto

distributor:
  receivers:
    kafka: {}
```

A configuração descrita no Listing 6.20 define o consumo de mensagens OTLP no formato *Protobuf*. O grupo de consumo garante paralelismo controlado, e o *encoding* determina o formato esperado dos *span*.

6.6.2 Armazenamento de *Traces*

O Tempo armazena os dados de trace em sistemas de arquivos ou backends de objetos. Para ambientes locais e testes, o uso de disco local é suficiente, conforme ilustrado no Listing 6.21:

Listing 6.21 – Configuração de armazenamento local no Tempo.

```
storage:
  trace:
    backend: local
    local:
      path: /var/tempo/traces
```

6.7 Visualização e Análise com Grafana

Na arquitetura proposta, o Grafana é adotado como ferramenta de visualização central, integrando-se nativamente com os componentes do *backend* de observabilidade baseados na stack LGTM. Essa integração facilita a análise unificada de métricas, *logs* e *traces* em ambientes distribuídos.

A conexão com o Loki permite a consulta e visualização de *logs* estruturados, utilizando uma linguagem de consulta inspirada no PromQL, o que facilita a investigação de falhas e o rastreamento de eventos em tempo real. Com o Mimir, o Grafana exibe métricas temporais compatíveis com a API do Prometheus, possibilitando a construção de painéis detalhados com gráficos, estatísticas agregadas e indicadores personalizados coletados por meio da instrumentação com OpenTelemetry. Já a integração com o Tempo permite a visualização de traces distribuídos, essencial para a identificação de gargalos, latências e o rastreamento de requisições em sistemas compostos por múltiplos serviços.

Além das visualizações, o Grafana permite a configuração de alertas baseados em regras, como variações no consumo de recursos, número de erros ou aumento de latência. Também oferece suporte a *dashboards* dinâmicos com variáveis, facilitando a reutilização de painéis em diferentes contextos, e permite o versionamento e compartilhamento de *dashboards*.

Assim, o Grafana desempenha um papel fundamental na arquitetura de observabilidade proposta, fornecendo uma interface única e interativa que consolida dados oriundos de múltiplas fontes.

6.8 Considerações de Escalabilidade e Resiliência

A arquitetura proposta foi concebida com foco em escalabilidade horizontal e resiliência a falhas, características essenciais para ambientes distribuídos e dinâmicos. Cada componente foi selecionado com base em sua capacidade de lidar com grandes volumes de dados de forma eficiente e confiável.

O uso de uma fila de mensageria, como Kafka, permite desacoplar a produção e o consumo de dados, amortecendo picos de carga e evitando a perda de informação em casos de falhas temporárias nos consumidores. Essa abordagem facilita o escalonamento independente dos coletores, processadores e mecanismos de persistência, permitindo que cada parte da arquitetura cresça conforme a demanda específica de *logs*, métricas ou *traces*.

A separação dos dados por tipo, direcionando *logs* ao Loki, métricas ao Mimir e *traces* ao Tempo, também contribui para a escalabilidade, pois permite aplicar políticas de retenção, replicação e compactação adaptadas a cada tipo de dado. Além disso, essas ferramentas foram escolhidas por sua compatibilidade com ambientes nativos em nuvem, suporte a clusterização e integração com o ecossistema Prometheus/Grafana, o que garante flexibilidade para operação em ambientes gerenciados ou autogerenciados.

A instrumentação com OpenTelemetry também favorece a resiliência, pois promove padronização e portabilidade dos dados, além de suportar diversos protocolos de exportação. Em caso de falha em um destino final, os dados podem ser reprocessados ou redirecionados, mitigando perdas e reduzindo o tempo de recuperação.

Essas decisões arquiteturais têm como objetivo garantir não apenas o funcionamento contínuo da solução, mas também sua capacidade de se adaptar a diferentes cenários de uso e crescimento sem comprometer a integridade e a performance do sistema.

7 Análise dos Resultados

7.1 Análise dos Resultados com Base nos Objetivos

Este trabalho teve como objetivo geral analisar ferramentas *open source* voltadas à monitoração e observabilidade em sistemas distribuídos, com ênfase na integração dessas soluções em uma arquitetura unificada. Esse objetivo foi alcançado por meio de uma abordagem que envolveu estudo teórico, comparação técnica e uma implementação prática com tecnologias amplamente utilizadas no ecossistema *cloud-native*.

O primeiro passo foi a consolidação conceitual sobre monitoração e observabilidade. O trabalho demonstrou com clareza a distinção entre os dois conceitos e o papel complementar que ambos exercem na manutenção de sistemas modernos. A partir da definição dos "quatro sinais dourados": latência, tráfego, erros e saturação; e dos três pilares da observabilidade: *logs*, métricas e *traces*; foi possível estabelecer critérios sólidos para avaliar ferramentas *open source*. Assim, o primeiro objetivo específico, de explorar os fundamentos conceituais do tema, foi atendido.

A análise comparativa entre as ferramentas *open source* consolidou o Prometheus como padrão de facto para coleta e consulta de métricas, devido à sua eficiência e integração com ambientes Kubernetes. Já para *logs*, o estudo mostrou que o Loki se destaca por seu modelo de indexação leve, baixo custo de armazenamento e integração nativa com Grafana. No domínio do *tracing*, o uso do OpenTelemetry associado ao *backend* Tempo apresentou-se como uma abordagem robusta para rastreamento distribuído. A comparação técnica entre essas ferramentas supos sua maturidade, escalabilidade e alinhamento com práticas modernas de *DevOps*, o que satisfaz o segundo objetivo específico do trabalho.

O terceiro e quarto objetivos específicos: investigar o papel do OpenTelemetry e projetar uma arquitetura prática; foram alcançados com a definição e implementação de um *backend* completo baseado na pilha LGTM (Loki, Grafana, Tempo e Mimir), orquestrado por meio do OpenTelemetry Collector. Essa arquitetura permitiu centralizar a coleta de sinais observáveis, desacoplando as aplicações instrumentadas dos *backends* de armazenamento e visualização. A configuração do *Collector* contemplou *pipelines* específicos para cada tipo de dado: *logs*, métricas e *traces* foram processados por *receivers* OTLP, enriquecidos e exportados para seus respectivos destinos, utilizando o protocolo gRPC e transporte assíncrono via Kafka. A padronização oferecida pelo OpenTelemetry permitiu manter consistência entre os dados coletados, viabilizando a correlação entre métricas, *logs* e *spans* com base no mesmo contexto distribuído.

O quinto objetivo, referente à implementação de uma prova de conceito, foi vali-

dado com sucesso. A aplicação foi instrumentada manualmente com o SDK do OpenTelemetry, utilizando Python para coleta de métricas e *traces*, e capturando *logs* diretamente nos *pods* do Kubernetes. As informações coletadas foram enviadas ao OpenTelemetry Collector, que as roteou para os componentes do *backend*. Métricas foram armazenadas e consultadas via Prometheus Remote Write integrado ao Grafana Mimir, enquanto os *logs* foram recebidos por meio do Loki com ingestão via Kafka. Os *traces* foram processados pelo*backend Tempo*, também com ingestão assíncrona, e visualizados no Grafana de forma correlacionada. Essa arquitetura demonstrou-se funcional, eficiente e modular, provando ser capaz de atender aos requisitos operacionais de observabilidade em sistemas distribuídos.

Por fim, o sexto objetivo: analisar os resultados obtidos; mostrou que a arquitetura baseada em ferramentas *open source* e integradas via OpenTelemetry não só é viável, como também escalável e extensível. A interoperabilidade entre os componentes da pilha LGTM e o Collector permite a construção de dashboards completos, com visualizações cruzadas de *logs*, métricas e *traces*. Além disso, o uso de identificadores compartilhados entre os sinais observáveis possibilitou a navegação contextualizada dentro do Grafana, fortalecendo a capacidade de análise causal e reduzindo o tempo de diagnóstico de falhas.

Discussão

A análise dos resultados confirma que o uso de ferramentas *open source* permite construir uma infraestrutura de observabilidade comparável a soluções comerciais, sem os altos custos associados. A arquitetura proposta baseou-se em princípios de desacoplamento, modularidade e interoperabilidade, e mostrou-se eficaz na coleta e correlação de dados em um ambiente distribuído.

Ao utilizar o OpenTelemetry como camada de instrumentação e coleta, o sistema ganhou flexibilidade, permitindo a substituição ou atualização dos *backends* sem necessidade de reconfiguração das aplicações monitoradas. A integração com a pilha LGTM ampliou essa capacidade ao proporcionar um ambiente visual coeso para análise dos três pilares da observabilidade. A sincronização entre Tempo, Loki e Mimir garantiu consistência temporal e semântica, fortalecendo a análise de causa-raiz.

A infraestrutura mostrou-se eficiente e responsiva frente a requisições reais do sistema, apresentando baixos tempos de ingestão e visualização dos dados nos painéis do Grafana. A correlação entre métricas, *logs* e *traces* se manteve estável ao longo do tempo, mesmo sob eventos inesperados e picos de tráfego, o que demonstra a maturidade e resiliência da solução proposta.

Dessa forma, os resultados obtidos sustentam não apenas a viabilidade técnica da arquitetura, mas também sua aplicabilidade prática em contextos reais de produção. A

observabilidade obtida com essa abordagem favorece diagnósticos mais rápidos, melhora o tempo de resposta da equipe e contribui diretamente para a confiabilidade e estabilidade do serviço observado.

Problemas e Limitações

Durante a execução do projeto, alguns desafios comprometeram a fluidez do processo. A configuração do OpenTelemetry Collector revelou-se mais complexa do que o inicialmente previsto, exigindo domínio sobre a estrutura de configuração YAML e sobre o funcionamento interno dos *pipelines* de dados. Além disso, a instrumentação manual das aplicações demandou um esforço considerável para garantir a correlação entre sinais, especialmente na ausência de auto-instrumentação compatível com todos os serviços utilizados.

Outro ponto que demandou atenção foi a necessidade de calibrar corretamente a ingestão assíncrona via Kafka, ajustando *buffers* e parâmetros de retenção para evitar perdas de dados em cenários de maior volume. No entanto, essas dificuldades foram superadas com ajustes técnicos, validações incrementais e testes contínuos ao longo do ciclo de implantação.

Apesar desses entraves, os objetivos do trabalho foram atingidos de forma satisfatória. A avaliação em ambiente de produção reforça a eficácia da arquitetura proposta, que demonstrou estabilidade, escalabilidade e compatibilidade com demandas reais de observabilidade. Os resultados obtidos abrem caminho para estudos futuros com foco em avaliação comparativa de desempenho, otimização de custos.

8 Considerações Finais

Este trabalho apresentou uma investigação aplicada sobre ferramentas open source de observabilidade em sistemas distribuídos, com foco na análise, seleção e implementação de uma arquitetura funcional baseada em OpenTelemetry e na pilha LGTM (Loki, Grafana, Tempo e Mimir). O estudo abordou desde os conceitos fundamentais de observabilidade e monitoração até a comparação técnica entre diferentes ferramentas, culminando na criação de um ambiente real de observação aplicado a um microsserviço em produção.

A execução do projeto permitiu compreender com profundidade os requisitos necessários para estabelecer uma infraestrutura observável eficiente e interoperável. Foi possível demonstrar que soluções baseadas em software livre, quando bem integradas, são capazes de atender às demandas de visibilidade, rastreabilidade e análise em sistemas modernos, mantendo um bom equilíbrio entre custo, escalabilidade e flexibilidade. A utilização do OpenTelemetry como núcleo de instrumentação padronizada mostrou-se estratégica, facilitando a correlação entre métricas, logs e traces, bem como a manutenção e evolução da arquitetura ao longo do tempo.

Os resultados obtidos reforçaram a viabilidade de adotar ferramentas open source em contextos reais de produção, desmistificando a ideia de que soluções robustas e eficientes precisam necessariamente ser proprietárias ou onerosas. A arquitetura proposta foi validada em um cenário prático, evidenciando sua estabilidade, desempenho e aderência às boas práticas de engenharia de observabilidade. A análise dos dados coletados indicou que a abordagem adotada é capaz de reduzir o tempo de diagnóstico de falhas, apoiar decisões técnicas mais informadas e aumentar a confiabilidade geral do sistema monitorado.

A realização deste trabalho também evidenciou a importância de se considerar aspectos como usabilidade das ferramentas, clareza na visualização de dados e padronização na coleta de informações. A experiência prática com a configuração dos pipelines de coleta, a integração com sistemas de mensageria como Kafka e a orquestração de visualizações no Grafana trouxe aprendizados técnicos valiosos e consolida a base para aplicações futuras mais complexas.

Para os próximos passos, considera-se expandir a arquitetura atual para múltiplos microsserviços, incorporando práticas de segurança e controle de acesso mais refinadas, bem como explorar o uso de *machine learning* para detecção de anomalias com base nos dados observáveis já estruturados. Também será relevante aprofundar a avaliação de desempenho em cenários com alta concorrência, além de evoluir os *dashboards* e alertas para diferentes perfis de usuários.

Com isso, este trabalho contribui tanto no campo acadêmico quanto no profissional, fornecendo um modelo prático, replicável de como implementar observabilidade completa em sistemas distribuídos, utilizando exclusivamente soluções open source.

Referências

AUSTIN, C.; JONES, B.; CUSTER, R. *Observability Engineering: Achieving Production Excellence*. 1st. ed. [S.l.]: O'Reilly Media, 2022. ISBN 978-1492076445. Citado na página 28.

AUTHORS, C. *Cortex Documentation*. 2023. Disponível em: <https://cortexmetrics.io/docs/>. Citado na página 36.

AUTHORS, T. *Thanos Documentation*. 2023. Disponível em: <https://thanos.io/>. Citado na página 36.

AWS. *Monolítico x Microsserviços — Diferença entre arquiteturas de desenvolvimento de software*. 2024. <https://aws.amazon.com/pt/compare/the-difference-between-monolithic-and-microservices-architecture/>. Acesso em: 13 dez. 2024. Citado na página 24.

Cloud Native Computing Foundation. *Cloud Native Computing Foundation announces Prometheus graduation*. 2018. Acesso em: 2025-01-30. Disponível em: <https://www.cncf.io/announcements/2018/08/09/prometheus-graduates/>. Citado na página 35.

ELASTIC. *Elastic Stack*. 2025. Acessado em: 10 fev. 2025. Disponível em: <https://www.elastic.co/pt/elastic-stack>. Citado na página 37.

FLUENTD. *Fluentd*. 2025. Acessado em: 10 fev. 2025. Disponível em: <https://www.fluentd.org/architecture>. Citado na página 38.

Google. *Site Reliability Engineering*. 2025. <https://sre.google/sre-book/monitoring-distributed-systems/>. Acesso em: 17 jan. 2025. Citado 2 vezes nas páginas 26 e 27.

IBM. *What are Microservices?* 2024. <https://www.ibm.com/topics/microservices>. Acesso em: 13 dez. 2024. Citado 2 vezes nas páginas 22 e 23.

IBM Education. *Observability vs monitoring*. 2025. <https://www.ibm.com/think/topics/observability-vs-monitoring>. Acesso em: 17 jan. 2025. Citado na página 25.

JAEGER. *Jaeger*. 2025. Acessado em: 10 fev. 2025. Disponível em: <https://www.jaegertracing.io/docs/>. Citado na página 41.

KALMAN, R. On the general theory of control systems. *IFAC Proceedings Volumes*, v. 1, n. 1, p. 491–502, 1960. ISSN 1474-6670. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1474667017700948>. Citado na página 27.

LIVENS, J. *Observability vs. monitoring: What's the difference?* 2025. <https://www.dynatrace.com/news/blog/observability-vs-monitoring/>. Acesso em: 17 jan. 2025. Citado na página 25.

LOKI. *Loki*. 2025. Acessado em: 10 fev. 2025. Disponível em: <https://grafana.com/docs/loki/latest/get-started/overview/>. Citado na página 39.

OPENTELEMETRY. *OpenTelemetry: Traces, Métricas, Logs e Baggage*. 2025. Acesso em: 28 mai. 2025. Disponível em: <https://dev.to/dellamas/opentelemetry-traces-metricas-logs-e-baggage-4foo>. Citado na página 46.

OREILLY. *Distributed Systems Observability*. 2025. <https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html#fig42>. Acesso em: 20 jan. 2025. Citado 5 vezes nas páginas 28, 29, 30, 31 e 32.

POWELL, P.; SMALLEY, I. *Monolithic architecture*. 2024. <https://www.ibm.com/think/topics/monolithic-architecture>. Acesso em: 13 dez. 2024. Citado na página 22.

PROMETHEUS. *Prometheus Documentation*. 2023. Acesso em: 20 jan. 2025. Disponível em: <https://prometheus.io/docs/>. Citado na página 35.

PROMETHEUS. *Prometheus Exporters and Integrations*. 2023. Acesso em: 20 jan. 2025. Disponível em: <https://prometheus.io/docs/instrumenting/exporters/>. Citado na página 35.

Apêndice A — Repositórios Oficiais e Requisitos da Arquitetura

Este apêndice apresenta os repositórios oficiais e versões das ferramentas utilizadas na implementação da arquitetura de observabilidade baseada na pilha LGTM (Loki, Grafana, Tempo, Mimir), OpenTelemetry e Apache Kafka. Todos os componentes utilizados são *open source* e foram implantados em ambiente Kubernetes por meio de Helm Charts.

A.1 Repositórios de Código-Fonte

- **Grafana Helm Charts:** <https://grafana.github.io/helm-charts>
- **Grafana OSS:** <https://github.com/grafana/grafana>
- **Loki:** <https://github.com/grafana/loki>
- **Mimir:** <https://github.com/grafana/mimir>
- **Tempo:** <https://github.com/grafana/tempo>
- **Apache Kafka (Bitnami Helm Chart):** <https://github.com/bitnami/charts/tree/main/bitnami/kafka>
- **Apache Kafka (Upstream):** <https://github.com/apache/kafka>

Tabela 6 – Helm Charts e Versões Utilizadas

Repositório	**Nome do Chart**	**Versão**
<https://grafana.github.io/helm-charts>	`grafana` (grafana)	`^7.3.9`
<https://grafana.github.io/helm-charts>	`loki` (loki-distributed)	`^0.79.0`
<https://grafana.github.io/helm-charts>	`mimir` (mimir-distributed)	`^5.3.0`
<https://grafana.github.io/helm-charts>	`oncall` (grafana-oncall)	`^1.3.114`
<https://grafana.github.io/helm-charts>	`tempo` (tempo-distributed)	`^1.9.2`
<https://charts.bitnami.com/bitnami>	`kafka` (bitnami/kafka)	`^26.3.2`

A.2 Tabela de Requisitos das Ferramentas Utilizadas

A.3 Observações

- Todas as ferramentas foram instaladas por meio de Helm Charts oficiais, com exceção do Apache Kafka, que utilizou o repositório mantido pela Bitnami.
- A escolha pelas versões estáveis visou garantir compatibilidade com o cluster Kubernetes e com os demais componentes da pilha LGTM.

As versões específicas das ferramentas utilizadas na arquitetura foram gerenciadas por meio de Helm Charts, conforme detalhado na Tabela 6. A inclusão do Apache Kafka foi essencial para compor os fluxos de ingestão assíncronos tanto de logs quanto de traces, promovendo desacoplamento e escalabilidade.