UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS SOBRAL

CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

WILLIAN WITHI ALVES DE SOUSA

ANÁLISE E IMPLEMENTAÇÃO DE PRÁTICAS DE OBSERVABILIDADE ESPECÍFICA PARA MICROSSERVIÇOS

SOBRAL

2024

WILLIAN WITHI ALVES DE SOUSA

ANÁLISE E IMPLEMENTAÇÃO DE PRÁTICAS DE OBSERVABILIDADE ESPECÍFICA PARA MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do CAMPUS SOBRAL da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia da Computação.

Orientador: Prof. Erick Aguiar Donato

SOBRAL

2024

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S698a Sousa, Willian Withi Alves de.
Análise e implementação de práticas de observabilidade específica para microsserviços / Willian Withi Alves de Sousa. – 2024.
42 f. : il.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Sobral, Curso de Engenharia da Computação, Sobral, 2024.
Orientação: Prof. Me. Erick Aguiar Donato.

1. microsserviços. 2. observabilidade. 3. logs. 4. rastreamento distribuído. 5. métricas. I. Título.

CDD 621.39

WILLIAN WITHI ALVES DE SOUSA

ANÁLISE E IMPLEMENTAÇÃO DE PRÁTICAS DE OBSERVABILIDADE ESPECÍFICA PARA MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do CAMPUS SOBRAL da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia da Computação.

Aprovada em: 29 de Outubro de 2024

BANCA EXAMINADORA

Prof. Erick Aguiar Donato (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Ialis Cavalcente de Paula Junior
Universidade Federal do Ceará (UFC)

Prof. Dr. Wendley Souza da Silva
Universidade Federal do Ceará (UFC)

AGRADECIMENTOS

Primeiramente agradeço a Deus, pela vida e saúde.

Aos meus pais por sempre me apoiar em todas as minhas conquistas e pelo incentivo para continuar perseguindo meus sonhos. Eles são minha base de motivação, tudo que sou hoje é graça a eles.

A minha esposa, Ana Lyvia por ser minha parceira de vida, pelo amor dado, companheirismo, inspiração, por todo apoio e entusiamo a mim concebido para sempre seguir na minha trajetória.

Aos meus professores, que durante a minha jornada acadêmica me ensinaram a ter perseverança, pela confiança nos projetos desenvolvidos e pelos ensinamentos ao longo dos anos.

Agradeço a instituição de ensino Universidade Federal do Ceará - Campus Sobral, por proporcionar caminhos de aprendizagem com excelentes profissionais e recursos de ensino.

RESUMO

Com o crescimento da internet e a evolução das arquiteturas de desenvolvimento de *software*, foi necessário adotar novas abordagens para garantir maior robustez. No entanto, mesmo com todos os benefícios das arquiteturas, o modelo arquitetural introduz um nível maior de complexidade quando se trata da observabilidade do sistema. Nessa abordagem, múltiplos serviços interagem entre si, tornando mais complexo acompanhar e mapear o fluxo completo das requisições dentro do sistema. Sendo assim, o objetivo deste projeto é analisar, implementar e avaliar práticas de observabilidade específicas para sistemas baseados em arquitetura de microsserviços. A intenção é melhorar o desempenho, a detecção de falhas e a eficiência operacional por meio de uma abordagem prática e detalhada. Adicionalmente, será criado um repositório contendo um exemplo prático que sirva de referência para desenvolvedores. Esse recurso permitirá a consulta de configurações e a comparação dos benefícios de cada ferramenta de observabilidade, facilitando a tomada de decisões e proporcionando um modelo prático de implementação.

Palavras-chave: microsserviços, observabilidade, logs, rastreamento distribuído.

ABSTRACT

With the growth of the internet and the evolution of software development architectures, it became necessary to adopt new approaches to ensure greater robustness. However, even with all the benefits of these architectures, the architectural model introduces a higher level of complexity when it comes to system observability. In this approach, multiple services interact with each other, making it more complex to track and map the complete flow of requests within the system. Therefore, the objective of this project is to analyze, implement, and evaluate specific observability practices for systems based on microservices architecture. The aim is to improve performance, failure detection, and operational efficiency through a practical and detailed approach. Additionally, a repository will be created containing a practical example that serves as a reference for developers. This resource will allow the consultation of configurations and the comparison of the benefits of each observability tool, facilitating decision-making and providing a practical implementation model. Keywords: microservices, observability, logs, distributed tracing.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Com o passar dos anos, o cenário de desenvolvimento vem crescendo bastante, e com isso vem surgindo novas tecnologias, como linguagens de programação, paradigmas e processos. Então, para acompanhar tais mudanças as empresas estão atualizando-se principalmente na área de arquitetura de aplicações *web* e migrando seus serviços (Di Francesco, 2017).

Segundo *Lewis and Fower (2014)* o estilo de arquitetura de microsserviços é uma abordagem para desenvolver um único aplicativo como um conjunto de pequenos serviços, cada um executando seu próprio processo e se comunicando com mecanismos leves, geralmente uma API de recurso HTTP. Esses serviços são construídos em torno de recursos de negócios e podem ser implantados de forma independente por meio de máquinas de implantação totalmente automatizadas.

A adoção da arquitetura de microsserviços em inúmeras aplicações proporcionou várias vantagens para as empresas devido às suas características de dividir os componentes, trazendo mais velocidade no ciclo de desenvolvimento, maior escalabilidade e uma manutenção simplificada. Como para a empresa Uber que por volta dos anos de 2012 a 2013 contava com dois serviços monolíticos, em que era encarados vários problemas operacionais devido o seu crescimento exponencial de usuários e dados, como riscos de indisponibilidade de todo o sistema devida uma única regressão dentro de uma base de código monolítica, assim como também a dificuldade de execução das funcionalidades do sistema de forma autônoma ou independente. A partir da migração para a arquitetura de microsserviços, os sistemas tornaram-se mais flexíveis e confiáveis, proporcionando escalar suas operações globalmente e diminuindo o tempo de desenvolvimento de novas aplicabilidades, permitindo que as funcionalidades, como cálculo de rotas, processo de pagamentos e gerenciamento de motorista fossem comandadas de maneira independentes. Assim como a empresa Uber, a Netflix também migrou de serviços monolíticos para microsserviços, proporcionando uma escala independente nas diferentes funcionalidades do sistema, como *streaming* de vídeo, gestão de contas e sugestões de conteúdo. Com a migração foi possível que as falhas de um serviço não afetassem toda a plataforma de *streaming*, como também a melhoria no desenvolvimento de implementação de novos recursos e testes.

No entanto, mesmo com todas as vantagens, essa arquitetura traz alguns desafios, como a complexidade adicional na coordenação dos microsserviços, o gerenciamento da comunicação entre eles e principalmente o monitoramento e a observabilidade.

Segundo Pete Hodgson (2019), a "observabilidade"tem um escopo amplo, desde

métricas técnicas de baixo nível até indicadores chave de desempenho de negócios (*Key Performance Indicator - (KPIs)*) de alto nível. Na extremidade técnica do espectro, podemos rastrear coisas como utilização de memória e CPU, E/S de rede e disco, contagens de *threads* e pausas de *garbage collection* (GC). No outro extremo do espectro, nossas métricas de negócios/domínio podem rastrear coisas como taxa de abandono do carrinho, duração da sessão ou taxa de falha no pagamento.

Nesse contexto, fica evidente que a arquitetura de microsserviços continua a moldar o cenário de desenvolvimento de *software*. E a observabilidade se torna um pilar essencial para garantir a confiabilidade e o desempenho dos sistemas.

A partir dos pontos anteriores apresentados, este trabalho visa ser um guia abrangente para enfrentar os desafios e aproveitar as oportunidades que essa abordagem oferece. Será implementado a observabilidade em uma aplicação de gerenciamento de cursos, apresentando o microsserviço de gerenciamento de curso e uma camada de cache, implementando com o redis, devido ser o microsserviço mais crítico da aplicação, por conta da sua elevada demanda.

1.1 Objetivo Geral

Nesse contexto, o principal objetivo deste projeto é analisar e implementar práticas de observabilidade específicas para sistemas baseados em arquitetura de microsserviços, visando melhorar o desempenho, a detecção de falhas e a eficiência operacional por meio de uma abordagem prática.

Adicionalmente, este projeto tem como propósito uma análise das ferramentas de observabilidade, bem como a criação de um repositório que contenha um exemplo prático. Esse exemplo servirá como um recurso de referência para desenvolvedores. Consequentemente, essa abordagem facilitará a tomada de decisões, tornando mais simples a consulta de configurações e benefícios oferecidos por cada ferramenta.

1.2 Objetivos Específicos

São objetivos específicos deste trabalho:

1. Identificar e selecionar um conjunto de práticas de observabilidade adequadas para microsserviços, levando em consideração métricas, rastreamento de solicitações, análise de logs e outros fatores relevantes;

2. Desenvolver um ambiente de teste que represente um cenário realista de microsserviços, permitindo a implementação das práticas de observabilidade selecionadas;
3. Coletar dados de métricas de desempenho, logs e rastreamento de solicitações dos microserviços após a implementação das práticas de observabilidade.

1.3 Organização do trabalho

A estrutura deste trabalho segue a seguinte organização: O Capitulo 2 aborda a evolução da arquitetura de *software* e ferramentas e tecnologias para observabilidade. O Capítulo 3 apresenta a metodologia utilizada na aplicação. O Capítulo 4 apresenta os resultados obtidos a partir da aplicação e o Capítulo 5 aborda as considerações finais do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão abordados os conceitos que servem como base para este projeto. Inicialmente, buscaremos explorar a evolução da arquitetura de *software* para adquirirmos uma compreensão do momento atual do desenvolvimento de *software*. Posteriormente, será aprofundado o estudo nos conceitos de microsserviços e observabilidade, uma vez que esses tópicos são de importância crucial para a compreensão dos desafios relacionados ao rastreamento distribuído.

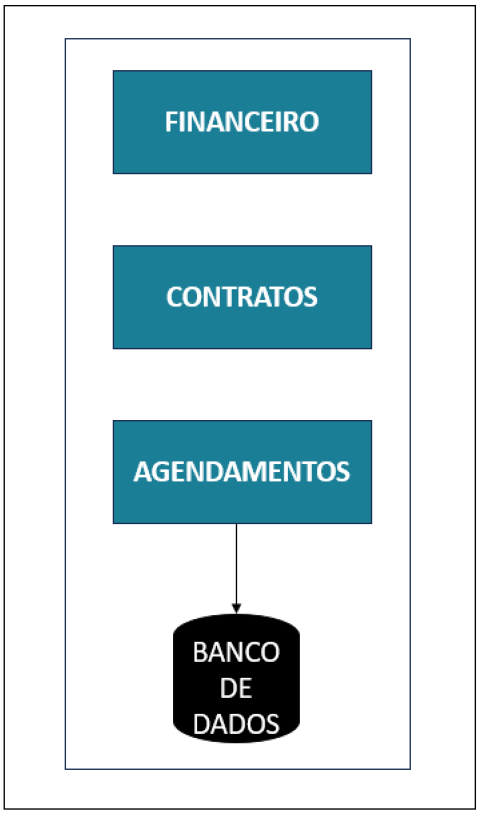
2.1 A evolução da Arquitetura de *software*

Ao longo do tempo, a arquitetura de *software* vem moldando a forma como construímos e escalamos aplicações. Três abordagens notáveis têm marcado essa jornada: as arquiteturas monolíticas, distribuídas e de microsserviços. Cada uma delas reflete uma resposta única aos desafios enfrentados pelo desenvolvimento de *software* em diferentes períodos e contextos. Nas próximas seções, exploraremos detalhadamente as características distintas de cada arquitetura, destacando como elas influenciaram e continuam a moldar o panorama do desenvolvimento de *software*.

2.1.1 Arquitetura Monolítica

De modo geral, monolíticos são aplicativos de *software* únicos e coesos que combinam todas as funcionalidades em uma única aplicação. Isso significa que todos os serviços são executados em um único contexto. O desenvolvimento e implantação de um aplicativo monolítico é geralmente mais simples, pois tudo está em um único lugar e pode ser testado e implantado de uma só vez. Além disso, monolíticos geralmente têm uma curva de aprendizado mais suave para novos desenvolvedores. A Figura 1 apresenta a arquitetura monolítica de um sistema, onde a interface de usuário, a lógica de negócios e a camada de dados compartilham a mesma base de código, banco de dados e recursos de infraestrutura.

Figura 1 – Arquitetura Monolítica ERP Caelum

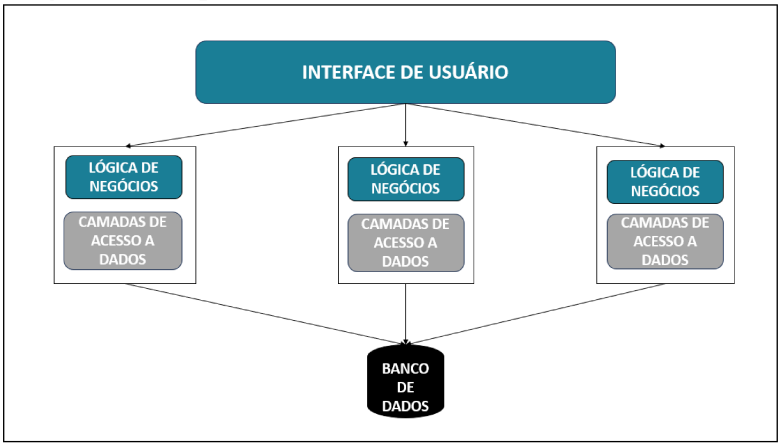


Fonte: Adaptado de (ALMEIDA,2014).

2.1.2 Arquitetura monolítica distribuída

A arquitetura monolítica distribuída é caracterizada pelo funcionamento da aplicação em diferentes processos ou serviços, cada um deles sendo executado em máquinas separadas ou instâncias distintas. Esses componentes têm a capacidade de interagir entre si, o que permite uma comunicação entre os serviços. Apesar disso, essa arquitetura mantém a centralização dos dados em um único ponto de armazenamento, como representado na Figura 2. Isso pode simplificar o gerenciamento e a manutenção da integridade dos dados, tornando mais fácil o controle de informações críticas. Entretanto, essa unificação dos dados em um único banco de dados central pode levar a gargalos de desempenho e questões de escalabilidade. Isso ocorre porque vários serviços precisam acessar o mesmo banco de dados, o que pode resultar em conflitos de acesso e limitações de escalabilidade do sistema, especialmente em situações de alta demanda. A Figura 2 ilustra essa configuração de uma arquitetura monolítica distribuída.

Figura 2 – Arquitetura monolítica distribuída.



Fonte: Adaptado de (RedHat,2023).

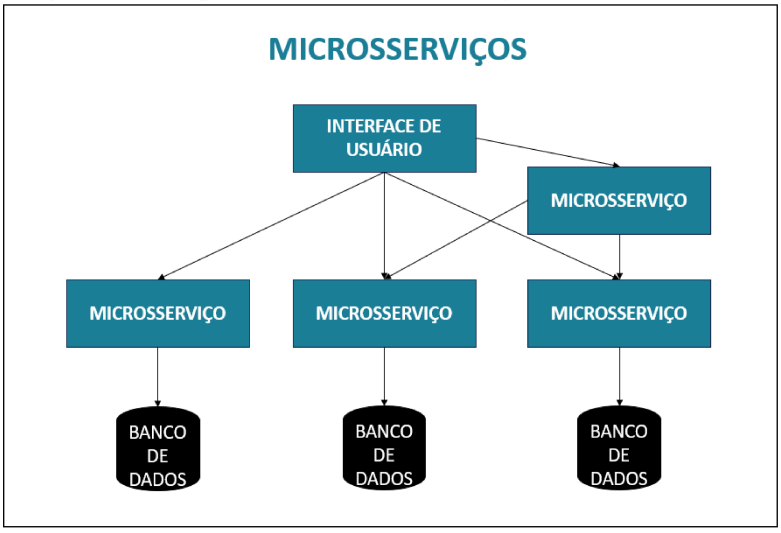
2.1.3 Arquitetura de microsserviços

Já a arquitetura de microsserviços é uma abordagem de desenvolvimento de software que se concentra na divisão de uma aplicação em componentes menores e independentes, conhecidos como microsserviços. Cada microsserviço é responsável por uma funcionalidade específica da aplicação e opera de forma autônoma, muitas vezes com sua própria base de código, banco de dados e ambiente de execução. Essa modularidade facilita o desenvolvimento, a implantação e a manutenção da aplicação, pois as equipes podem trabalhar de maneira mais ágil e iterativa, concentrando-se em partes individuais do sistema.

Uma das principais vantagens da arquitetura de microsserviços é a escalabilidade e a flexibilidade que ela oferece. Os microsserviços podem ser dimensionados independentemente, o que significa que os recursos podem ser alocados de acordo com as necessidades específicas de cada componente, resultando em um melhor desempenho e eficiência global. Além disso, os microsserviços permitem a escolha de tecnologias adequadas para cada funcionalidade, o que é particularmente útil em ambientes heterogêneos ou quando se deseja adotar inovações tecnológicas sem afetar toda a aplicação. No entanto, a complexidade aumenta à medida que o número de microsserviços cresce, exigindo uma sólida gestão de implantação, monitoramento e orquestração.

A comunicação entre microsserviços, garantindo a segurança e a confiabilidade das transações, também é um desafio a ser enfrentado. A Figura 3 ilustra essa configuração de uma arquitetura de microsserviços.

Figura 3 – Arquitetura de microsserviços



Fonte: Adaptado de (RedHat,2023).

2.2 Observabilidade em sistemas distribuídos

Observabilidade é um conceito que foca em interpretar os dados recebidos de um sistema externo de forma que consiga monitorar e identificar falhas antes que elas causem impactos críticos no sistema. O objetivo de uma plataforma de observabilidade é encontrar o ponto de desempenho normal de um sistema e depois melhorá-lo. Claro que isso é em um caso ideal, mas a observabilidade permite descartar possibilidades quando um erro está ocorrendo. Para conseguir atingir este objetivo são necessários a relação de três pilares, que são métricas, *logs* e *traces*. [3]

Aplicar observabilidade em sistemas distribuídos é gerenciar de forma eficaz a complexidade inerente a sistemas que envolvem múltiplos componentes distribuídos e interconectados. Ela desempenha um papel fundamental na manutenção da confiabilidade, desempenho e escalabilidade desses sistemas.

2.3 Métricas, *logs* e rastreamento em microsserviços

As métricas são a base do monitoramento em microsserviços. Elas fornecem informações quantitativas sobre o comportamento do sistema, como a utilização de recursos, o tempo de resposta e o número de requisições por segundo. Com métricas adequadas, as equipes de operações e desenvolvimento podem identificar algum impedimento de algum processo, otimizar o desempenho e tomar decisões informadas para melhorar a qualidade do serviço.

Os *logs* são registros detalhados das atividades e eventos que ocorrem em um sistema

de microsserviços. Eles são essenciais para solucionar problemas, rastrear erros e investigar incidentes. Os *logs* podem conter informações sobre quem fez o quê, quando e por quê. Uma estratégia eficaz de gerenciamento de *logs* permite que as equipes rastreiem eventos de interesse, criem alertas e auditem as operações.

O rastreamento é a capacidade de acompanhar o fluxo de uma solicitação através dos vários componentes de um sistema de microsserviços. Isso é fundamental para entender como as solicitações se propagam pela arquitetura e identificar possíveis gargalos ou pontos de falha. O rastreamento também é valioso para a resolução de problemas e a otimização do desempenho, pois permite que as equipes identifiquem as interações entre os serviços e compreendam a latência de cada etapa.

Com o uso de ferramentas que possibilitam a entrega desses três pilares métricas, *logs* e rastreamento as equipes de desenvolvimento pode manter a confiabilidade dos sistemas, analisar e resolver problemas mais rápido e eficazes, garantindo assim uma melhor experiência e qualidade do sistema.

2.4 Ferramentas e tecnologias para observabilidade

Nesse contexto, as métricas, *logs* e rastreamento de microsserviços desempenham um papel fundamental na coleta de dados e análise de sistemas distribuídos. Existe uma variedade de ferramentas e tecnologias estão disponíveis para auxiliar na coleta, armazenamento, visualização de métricas, *logs* e rastreamento, tornando-se fundamentais para manter a confiabilidade e o desempenho desses sistemas. Neste estudo, investigaremos as principais opções disponíveis *open source* em cada uma dessas categorias e avaliaremos suas características, benefícios e integrações em ambientes de microsserviços.

2.4.1 Prometheus

Segundo a documentação oficial do Prometheus (PROMETHEUS, 2024), ele foi desenvolvido pela *SoundCloud* em 2012, pois as tecnologias existentes eram insuficientes para suas necessidades de observabilidade, destaca-se como uma solução de monitoramento de código aberto projetada para coleta e armazenamento de métricas em tempo real. Suas principais características são:

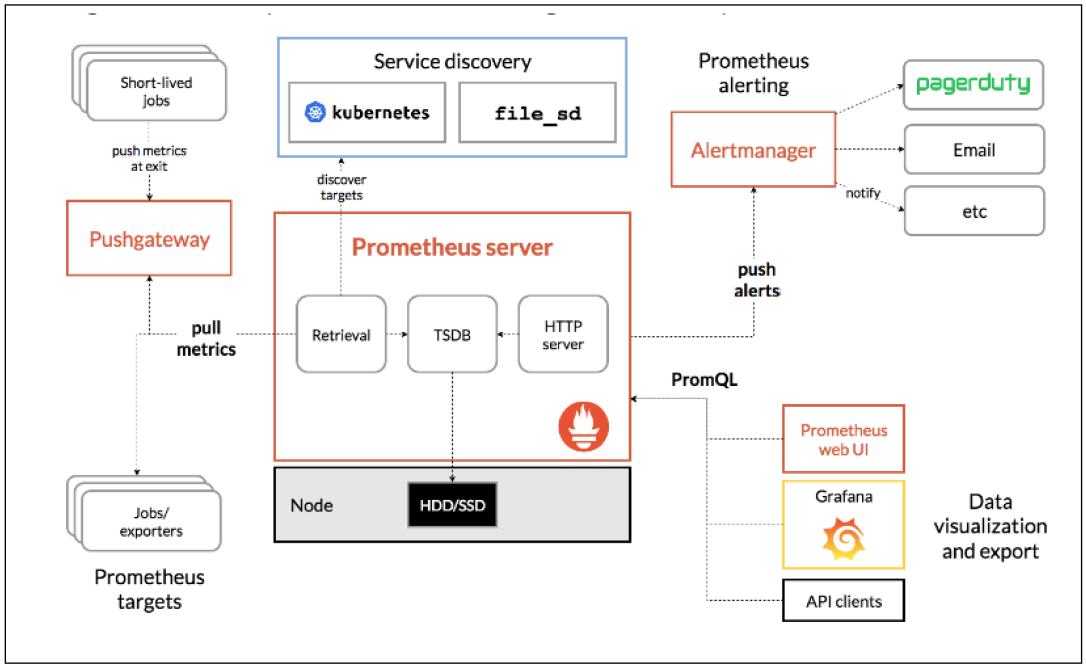
- Um modelo de dados multidimensional com dados de série temporal identificados por nome de métrica e pares chave/valor;
- PromQL, uma linguagem de consulta flexível para aproveitar essa dimensionalidade;
- Nenhuma dependência de armazenamento distribuído;
- A coleta de série temporal acontece por meio de um modelo pull sobre HTTP;
- O envio de séries temporais é suportado por meio de um *gateway* intermediário;
- Os alvos são descobertos por meio de descoberta de serviço ou configuração estática;
- Vários modos de suporte a gráficos e painéis.

O ecossistema Prometheus é composto por vários componentes, muitos dos quais são opcionais. Inclui o servidor principal do Prometheus, responsável por coletar e armazenar dados de séries temporais, bibliotecas de cliente para instrumentação de código de aplicativo, um *gateway push* para suportar empregos de curta duração, exportadores para serviços especiais como HAProxy, StatsD, Graphite, entre outros. Além disso, integra um gerenciador de alertas para lidar com alertas e diversas ferramentas de suporte. A Figura 4 ilustra a arquitetura do Prometheus e alguns de seus componentes.

O design do Prometheus enfatiza confiabilidade, proporcionando um sistema independente e acessível durante interrupções para diagnóstico rápido. Sua autonomia em relação a armazenamento de rede ou serviços remotos o torna uma escolha confiável, especialmente em situações onde outras partes da infraestrutura estão comprometidas, sendo adaptável tanto a ambientes de monitoramento centrados na máquina quanto em arquiteturas de microsserviços altamente dinâmicas. Sua capacidade de lidar com dados multidimensionais é particularmente robusta em cenários de microsserviços.

Embora confiável, pode não ser a escolha ideal em situações que demandam alta precisão. Os dados coletados podem não ser detalhados e completos o suficiente para garantir essa precisão. Em tais casos, é recomendável utilizar outro sistema específico para coletar e analisar dados, enquanto o Prometheus continua a ser uma solução valiosa para o restante do

Figura 4 – Arquitetura do Prometheus.



Fonte: PROMETHEUS, 2024

monitoramento.

2.4.2 Grafana

Uma outra ferramenta bastante utilizada em observabilidade é o Grafana, que oferece painéis interativos e personalizáveis que permitem aos usuários criar representações visuais de seus dados operacionais. O usuário pode integrar essa ferramenta com diversas fonte de dados incluindo o Prometheus, InfluxDB, Graphite, Elasticsearch, entre outras, permitindo uma visão unificada dos dados. Segundo a documentação oficial do Prometheus (PROMETHEUS, 2024) A combinação de Prometheus para extrair as métricas e o Grafana para visualização é uma prática comum em ambientes de monitoramento de código aberto.

A visualização do Grafana é bem completa podendo ser utilizada para métricas, *logs* e rastreamento devido ao seu ponto forte que é a facilidade de integração com diversas ferramentas, é utilizado nas métricas pois permite a criação de gráficos dinâmicos que exibem métricas essenciais, como taxa de requisições, uso de CPU, e latência, conforme a Figura 5.

Figura 5 – Dashboard Grafana

Fonte: (GRAFANA, 2024)

Apesar de todas essas vantagens, a adoção do Grafana pode apresentar desafios para novos usuários, devido a uma curva de aprendizado inicial que pode ser desafiadora. Além disso, a configuração inicial do Grafana pode ser intrincada, especialmente ao integrá-lo com diferentes fontes de dados e sistemas, demandando uma atenção cuidadosa durante todo o processo. Outro aspecto a ser considerado são as possíveis limitações em recursos gratuitos, e as versões gratuitas podem ter restrições em termos de escalabilidade e recursos mais avançados.

A eficácia do Grafana muitas vezes está vinculada à sua integração com outros sistemas de monitoramento e armazenamento de dados, o que, por sua vez, pode adicionar complexidade à configuração geral. Esses desafios, embora presentes, podem ser mitigados com tempo, prática e familiaridade contínua com a plataforma.

2.4.2.1 Grafana Loki

Juntamente com o Grafana, podemos o utilizar o Grafana Loki que é uma ferramenta de agregação e consulta de logs, desenvolvida para oferecer uma solução escalável e eficiente para o gerenciamento de logs, similar ao conceito de logs em tempo real oferecido por sistemas como o Elasticsearch. Ao contrário de outras soluções de monitoramento de logs que indexam todo o conteúdo dos logs, o Loki se diferencia ao indexar apenas os metadados, como rótulos, o que reduz significativamente o custo de armazenamento e o tempo de consulta.

2.4.3 InfluxDB

O InfluxDB é um sistema de gerenciamento de banco de dados desenvolvido para dados de séries temporais. Projetado para eficiência no armazenamento e recuperação de informações temporais, o InfluxDB é otimizado para lidar com a natureza contínua e sequencial desses dados. Ele oferece desempenho e escalabilidade ideais, sendo amplamente utilizado em ambientes que demandam armazenamento eficiente e análise de dados provenientes de sensores, métricas de desempenho, registros de eventos e outras fontes que evoluem ao longo do tempo.

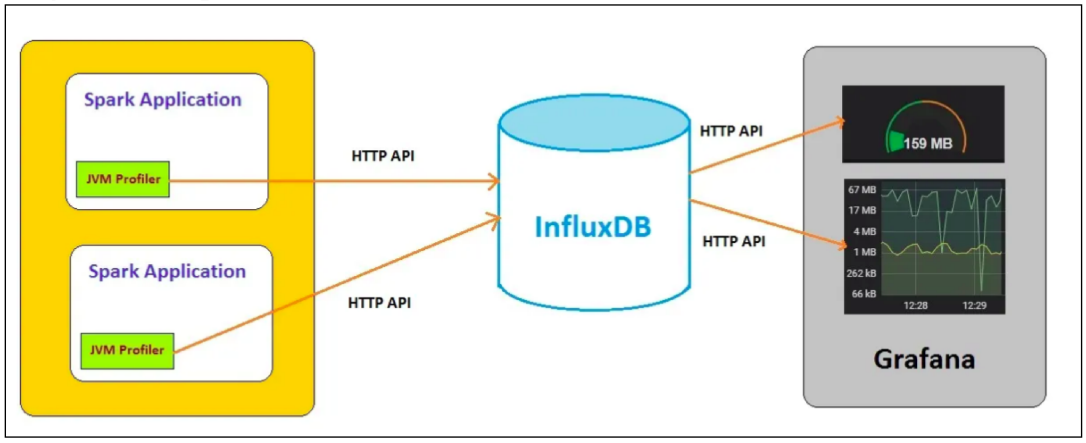
Essa ferramenta possui uma linguagem de consulta a InfluxQL que é uma característica distintiva do InfluxDB, sendo especialmente desenvolvida para simplificar a análise de dados temporais. Essa linguagem oferece recursos específicos para operações com séries temporais, tornando a consulta e interpretação de dados mais acessíveis e eficientes para usuários lidando com informações que evoluem ao longo do tempo.

A facilidade de integração é outro ponto forte dessa ferramenta. Com suporte a uma variedade de protocolos de comunicação, como HTTP e UDP, e a disponibilidade de bibliotecas e plugins para diversas linguagens de programação, o banco de dados se destaca pela capacidade de se integrar facilmente a diferentes ecossistemas, simplificando a implementação em ambientes variados. A Figura 6 ilustra a arquitetura do InfluxDB.

As políticas de retenção de dados oferecem uma gestão eficiente do armazenamento, permitindo a configuração para exclusão automática de dados mais antigos. Essa funcionalidade é essencial para manter a base de dados gerenciável ao longo do tempo, garantindo que apenas as informações relevantes sejam retidas.

A escalabilidade é uma característica crucial e proeminente. Projetado para oferecer opções de clustering, o banco de dados possibilita a distribuição eficiente da carga de trabalho.

Figura 6 – Arquitetura InfluxDB



Fonte: (Medium, 2018)

Isso assegura um desempenho consistente em ambientes que demandam escalabilidade horizontal, tornando-o adequado para cenários de crescimento e expansão.

Por fim, o InfluxDB vai além de ser apenas um banco de dados, sendo comumente combinado com ferramentas de visualização, como o Grafana. Essa integração possibilita a criação de painéis interativos e gráficos, simplificando a interpretação dos dados temporais armazenados. A sua comunidade ativa assim como no Grafana, contribui para a acessibilidade de suporte, recursos adicionais e atualizações frequentes.

Mesmo o InfluxDB sendo uma ótima ferramenta para registro de métricas, neste trabalho será utilizado o Prometheus, devido a curva de aprendizagem ser mais rápida em comparação com o InfluxDB.

2.4.4 OpenTelemetry

De acordo com a documentação oficial do OpenTelemetry (OpenTelemetry, 2024), O OpenTelemetry é um projeto de código aberto da *CNCF (Cloud Native Computing Foundation)* com o objetivo de criar uma maneira padrão e independente de fornecedor para coletar dados de telemetria para aplicativos, infraestrutura e serviços. Além disso, promove a interoperabilidade, facilitando a comunicação entre diferentes serviços e componentes, proporcionando uma visão abrangente do desempenho do sistema.

A ferramenta fornece recursos robustos de rastreamento, permitindo a análise detalhada do caminho que uma solicitação percorre, identificando gargalos e otimizando o desempenho. Além disso, a comunidade ativa de desenvolvedores que mantém o OpenTelemetry e contribui para a evolução constante da ferramenta, mantendo-a alinhada com as melhores

práticas e requisitos emergentes.

A capacidade de rastrear transações através de serviços distribuídos é crucial para entender o fluxo de execução de uma solicitação. Os principais componentes da arquitetura do openTelemetry são instrumentação, APIs e SDKs, coletores de telemetria, exportadores, tracing e métricas.

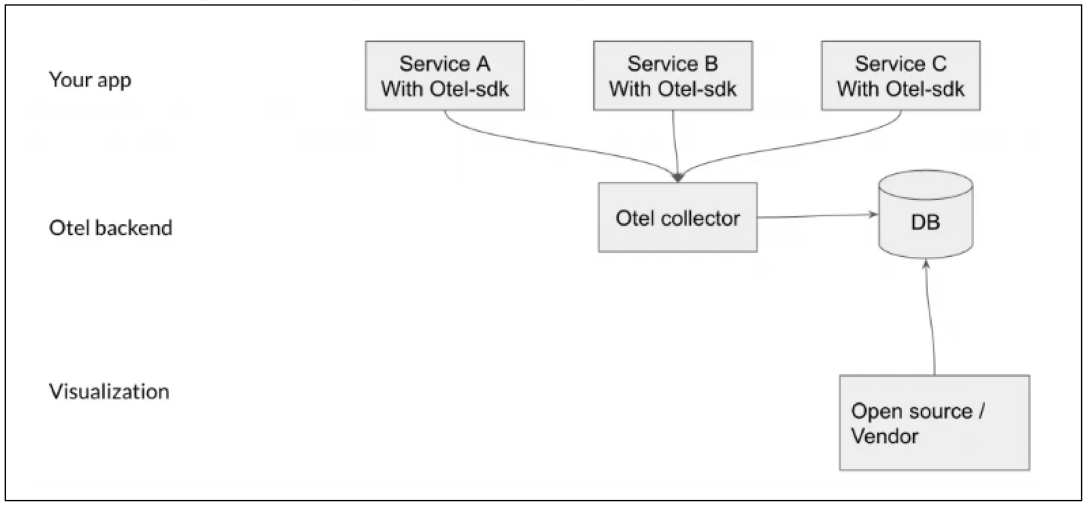
No processo de instrumentação, os desenvolvedores incorporam código aos seus aplicativos para coletar dados de telemetria, utilizando as bibliotecas e SDKs para várias linguagens de programação fornecidas, facilitando a integração e permitindo uma instrumentação uniforme. As APIs do OpenTelemetry definem contratos e métodos para instrumentação, enquanto os SDKs implementam essas APIs, proporcionando funcionalidades prontas para uso. Essa abordagem é flexível e suporta diversas linguagens, garantindo uma experiência consistente em ambientes diversos.

Os coletores de telemetria são responsáveis por receber e armazenar os dados coletados pelos instrumentos. Esses coletores podem ser implantados centralmente ou distribuídos, dependendo dos requisitos do ambiente. Eles também suportam diferentes formatos de exportação, garantindo interoperabilidade com diversas ferramentas de observabilidade.

Os exportadores convertem os dados de telemetria coletados pelos instrumentos para os formatos necessários e os enviam para sistemas externos ou ferramentas de observabilidade. Essa funcionalidade é vital para visualização e análise em plataformas como Grafana, Prometheus ou Jaeger. Além disso, o OpenTelemetry suporta tanto o rastreamento de transações quanto a coleta de métricas. As métricas ajudam a monitorar estatísticas de desempenho, enquanto o rastreamento fornece uma visão detalhada do percurso de execução, destacando o fluxo de operações e transições ao longo do sistema distribuído.

Como qualquer sistema de monitoramento, o OpenTelemetry pode introduzir um certo overhead de desempenho. A coleta extensiva de dados de telemetria pode impactar o desempenho do sistema, especialmente em ambientes de produção. O equilíbrio entre a quantidade de dados coletados e o impacto no desempenho deve ser cuidadosamente considerado. Além disso, apesar de seus benefícios, a curva de aprendizado inicial para implementar e configurar o OpenTelemetry pode ser desafiadora para equipes menos familiarizadas com conceitos de observabilidade e instrumentação de código.

Figura 7 – Arquitetura de pilha básica do OpenTelemetry



Fonte: (Equipe Aspecto, 2022)

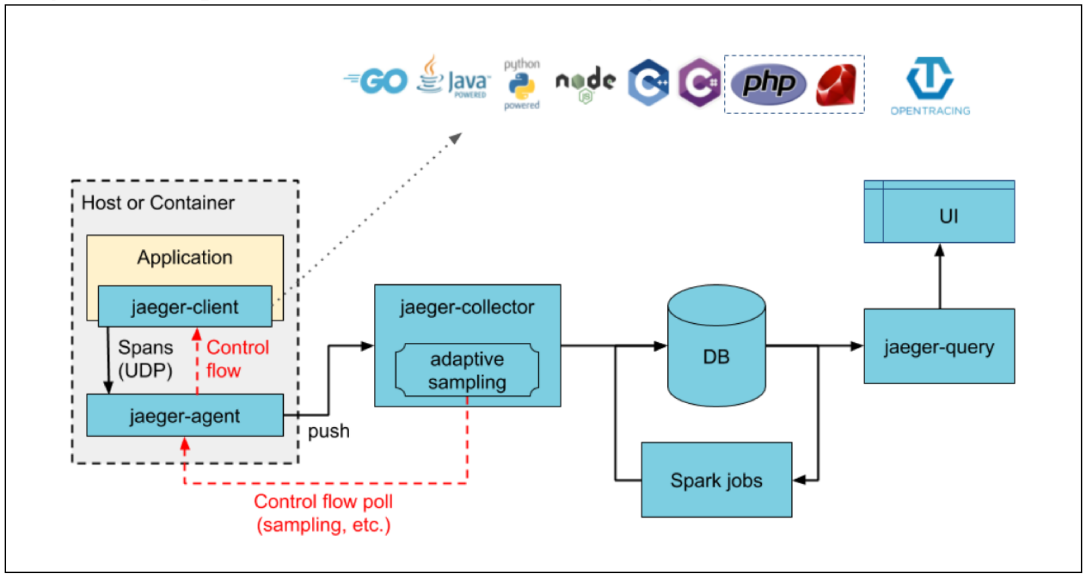
2.4.5 Jaeger

Segundo a documentação oficial do Jaeger (Jaeger, 2024), O Jaeger é baseado na instrumentação e APIs OpenTracing, independentes de fornecedor. A Uber, uma empresa de transporte privado, desenvolveu o Jaeger como um projeto open source em 2015. É uma ferramenta de código aberto para rastreamento distribuído, projetada para monitorar e solucionar problemas em ambientes complexos e distribuídos. Ele é frequentemente utilizado em conjunto com o OpenTelemetry para fornecer recursos avançados de rastreamento de transações em sistemas distribuídos.

Essa ferramenta apresenta as solicitações de execução como traces, que mostram os dados/caminhos de execução através de um sistema. Um trace é formado por um ou mais spans, que são as unidades de trabalho lógicas do Jaeger. Cada span inclui o nome da operação, a data/hora de início e a duração. Os spans podem estar aninhados e ordenados. Os principais componentes dessa ferramenta são: Agentes, servidor (collector), Armazenamento (Storage), Consulta (query).

As principais vantagens são Permitir monitorar e rastrear transações em sistemas distribuídos complexos, Facilitar a identificação e solução de problemas de desempenho em ambientes distribuídos, Assim como as outras ferramentas são explanadas aqui nesse projeto ele também pode ser integrado com outros sistemas de monitoramento e registro, como *Prometheus* e *Grafana*, suporta várias linguagens de programação e *frameworks* e é um projeto de código aberto.

Figura 8 – Arquitetura de armazenamento do Jaeger



Fonte: (Jaegertracing.io, 2023)

2.4.6 Ferramentas de apoio

2.4.6.1 Docker

O Docker é uma plataforma de código aberto que automatiza o empacotamento, distribuição e execução de aplicações em contêineres. Os contêineres são unidades leves e portáteis que encapsulam uma aplicação e todas as suas dependências, garantindo que ela funcione de forma consistente em qualquer ambiente, desde desenvolvimento até produção. Com o Docker, os desenvolvedores podem criar ambientes isolados para suas aplicações, evitando conflitos de dependências e problemas de compatibilidade. Além disso, o Docker facilita o gerenciamento e a escalabilidade de aplicações, permitindo a orquestração de múltiplos contêineres e a integração com ferramentas de automação e gerenciamento.

2.4.6.2 Docker Compose

O *Docker Compose* é uma ferramenta que permite a definição e execução de aplicações compostas por múltiplos contêineres *Docker*. Usando um arquivo de configuração *YAML*, você pode especificar todos os serviços, redes e volumes necessários para a aplicação em um único local. Com um comando, você pode iniciar, parar e gerenciar todos os contêineres simultaneamente, simplificando o processo de configuração e a coordenação de ambientes complexos. Essa abordagem facilita o desenvolvimento, a testagem e a implantação de aplicações que dependem de vários serviços interconectados.

2.4.6.3 Spring Boot

O *Spring Boot* é uma ferramenta criada para simplificar o desenvolvimento e configuração de aplicações Java. Ele reduz a complexidade ao fornecer componentes pré-configurados, permitindo que uma aplicação esteja pronta para produção rapidamente, com o mínimo de esforço na configuração e implantação. Apresentando um modelo de desenvolvimento mais direto e eficiente, o *Spring Boot* facilita a criação de projetos Java, oferecendo uma abordagem prática e simplificada para o desenvolvimento de aplicações.

No projeto, utilizaremos o *Spring Boot* para facilitar a criação e configuração do microsserviço Java que compõe a aplicação de exemplo. Serão apenas um microsserviço com uma função específica dentro do sistema que é cadastrar cursos.

2.4.6.4 Redis

O gerenciamento de cache foi feito com o Redis. Ao usar o Redis como cache, é possível reduzir significativamente o tempo de resposta de consultas a bancos de dados mais lentos ou evitar chamadas repetidas a *APIs* externas. O Redis permite a configuração de políticas de expiração de chaves, para garantir que os dados sejam removidos da memória quando não forem mais necessários ou quando o espaço se esgotar. Essa abordagem melhora a escalabilidade do sistema, diminuindo a carga nos servidores de banco de dados e aumentando a eficiência geral da aplicação.

2.4.6.5 PostgreSQL

O *PostgreSQL* é um sistema de gerenciamento de banco de dados relacional de código aberto, amplamente reconhecido por sua robustez, extensibilidade e conformidade com padrões SQL. Ele é projetado para lidar com grandes volumes de dados e suporta uma ampla gama de tipos de dados, incluindo JSON, permitindo fácil integração com aplicações modernas. Sua arquitetura avançada permite extensões e funcionalidades personalizadas, como indexação avançada, replicação e partição de tabelas, tornando-o uma escolha popular para sistemas que exigem alta performance e complexidade.

2.4.6.6 Nginx

O Nginx é um servidor web de código aberto amplamente utilizado, conhecido por sua alta performance e capacidade de lidar com um grande número de conexões simultâneas. Sua arquitetura assíncrona e orientada a eventos permite uma utilização eficiente dos recursos do sistema, tornando-o ideal para sites de alto tráfego. Além de funcionar como um servidor web, o Nginx atua como um proxy reverso e balanceador de carga, possibilitando a distribuição de requisições entre múltiplos servidores de backend. Essa funcionalidade melhora a disponibilidade e a escalabilidade das aplicações, além de oferecer suporte a cache HTTP, que reduz a carga nos servidores e acelera o tempo de resposta.

Outro aspecto importante do Nginx é sua flexibilidade e facilidade de configuração. Os arquivos de configuração em texto simples permitem personalizar o comportamento do servidor de acordo com as necessidades específicas de cada projeto. Com suporte a SSL/TLS, autenticação e controle de acesso, o Nginx também desempenha um papel fundamental na segurança de aplicações web. Sua grande comunidade e extensa documentação facilitam a implementação e a resolução de problemas, solidificando sua posição como uma solução robusta e confiável para gerenciar tráfego de rede e servir aplicações web de forma eficaz.

3 METODOLOGIA E DESENVOLVIMENTO

O levantamento e seleção de práticas de observabilidade tornaram-se uma parte crucial do ciclo de vida do desenvolvimento, promovendo a resiliência e a capacidade de resposta dos aplicativos em ambientes complexos. A observabilidade refere-se à capacidade de entender o que está acontecendo dentro de um sistema, medindo e analisando seus estados internos por meio de métricas, logs e rastreamentos. Essa abordagem vai além do simples monitoramento, buscando proporcionar uma melhor identificação e resolução de problemas.

O primeiro passo no estabelecimento de um ambiente observável é o levantamento de práticas existentes. Isso envolve uma análise detalhada das características específicas do sistema, identificando suas necessidades, requisitos de negócios e os pontos críticos que exigem monitoramento constante.

3.1 Práticas comuns de observabilidade

Uma estratégia fundamental para a observabilidade é a instrumentação adequada, que implica a introdução de código e bibliotecas para coletar dados essenciais, como métricas de desempenho e eventos. Esse processo estabelece uma base sólida para análise e otimização, oferecendo informações essenciais sobre o desempenho do sistema.

Além da instrumentação, a coleta de logs é uma prática comum que registra eventos significativos, oferecendo a capacidade de realizar análises detalhadas posteriormente. Esses logs desempenham um papel crucial na identificação proativa de problemas e comportamentos fora do comum, possibilitando uma resposta eficaz para resolver questões antes que impactem o usuário final.

Outro elemento chave da observabilidade é o rastreamento distribuído, que monitora o fluxo de dados entre os diferentes componentes do sistema. Essa prática proporciona uma visão detalhada das transações, permitindo uma compreensão mais profunda do funcionamento interno do sistema e facilitando a identificação de possíveis locais dentro do sistema que está causando lentidão ou falha.

A implementação de alertas e notificações é essencial para a detecção proativa de problemas. Ao configurar alertas, a equipe pode responder rapidamente às questões identificadas, evitando potenciais impactos no usuário final e garantindo a integridade contínua do sistema.

A análise de métricas desempenha um papel vital na avaliação do sistema. Utilizando

métricas relevantes, as equipes podem medir o desempenho, a eficiência e a confiabilidade do sistema, permitindo ajustes e melhorias contínuas. Essas práticas comuns de observabilidade são essenciais para garantir o bom funcionamento e a evolução do sistema.

Por fim, a escolha das práticas de observabilidade deve ser personalizada de acordo com as características específicas de cada sistema. Aspectos como a arquitetura da aplicação, requisitos de negócios e contexto operacional influenciam na seleção das melhores práticas.

3.2 Avaliação das práticas em relação aos requisitos do sistema

Para realizar uma avaliação adequada das práticas de observabilidade, é fundamental identificar os componentes críticos do sistema. Esses componentes incluem serviços principais, APIs, bancos de dados, e outros elementos essenciais que garantem o funcionamento do ecossistema de microsserviços. A escolha dos componentes que devem ser monitorados baseia-se em uma análise detalhada dos requisitos funcionais e não funcionais do sistema.

Os requisitos funcionais referem-se às funcionalidades que o sistema deve fornecer, como operações de criação, leitura, atualização e exclusão de dados (CRUD), enquanto os requisitos não funcionais abrangem aspectos como desempenho, escalabilidade, confiabilidade e segurança. A correta identificação dos componentes críticos permite priorizar quais elementos precisam de maior atenção em termos de monitoramento e análise de desempenho.

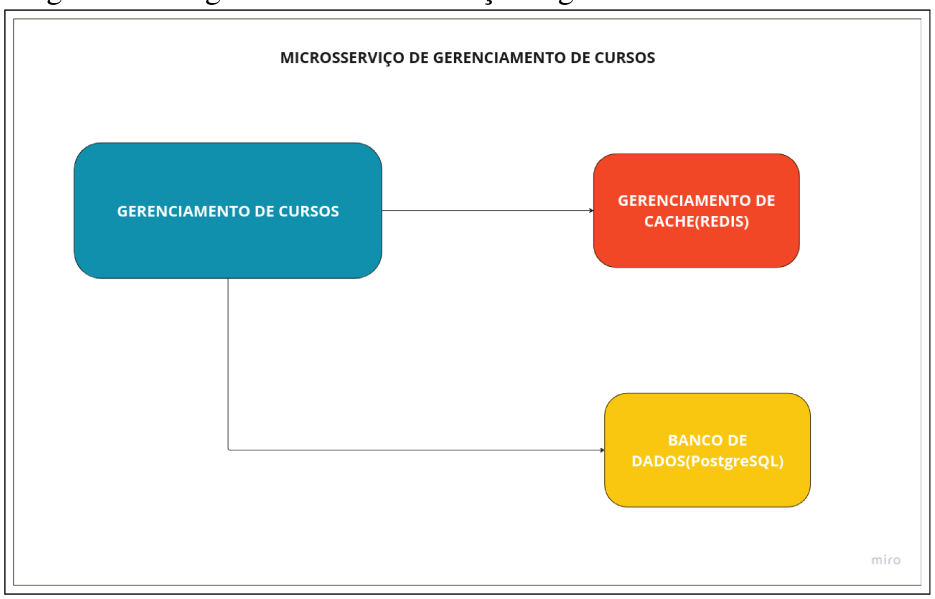
Durante a avaliação, será analisado como as práticas de observabilidade podem ser aplicadas para atender às necessidades específicas do sistema. Isso envolve a implementação de logs estruturados, métricas de desempenho e rastreamento distribuído (tracing), visando garantir que as operações realizadas pelos microsserviços sejam monitoradas de forma eficiente e que os problemas possam ser rapidamente identificados e corrigidos.

A análise também deve considerar a integração dessas práticas com ferramentas como Prometheus, Grafana, Loki e Jaeger, que oferecem suporte ao monitoramento contínuo dos serviços e à visualização centralizada de métricas e logs, assegurando que os requisitos de confiabilidade e desempenho do sistema sejam atendidos.

Neste trabalho, a implementação da observabilidade será realizada em um microsserviço de exemplo, que tem a função de realizar o gerenciamento de cursos, desenvolvido em Spring Boot, que até o momento não possui nenhum mecanismo de monitoramento. O microsserviço se comunica com um sistema de cache Redis e um banco de dados, executando operações como criação, atualização e remoção de registros de cursos. A ausência de ferramentas

de observabilidade limita a capacidade de monitorar e avaliar o desempenho dos microsserviços, o que torna essencial a adoção de um conjunto robusto de ferramentas para logs, métricas e rastreamento distribuído, visando melhorar a visibilidade sobre o comportamento da aplicação e sua confiabilidade operacional. A Figura 9 ilustra os principais serviços envolvidos no sistema de gerenciamento de cursos.

Figura 9 – Diagrama do microsserviço de gerenciamento de curso



Fonte: Elaborado pelo próprio autor.

O microsserviço de gerenciamento de cursos foi desenvolvido com o framework Spring Boot, conhecido por sua simplicidade e robustez na construção de microsserviços em Java. Este microsserviço possui conexões diretas tanto com o banco de dados quanto com o sistema Redis, que gerencia o cache da aplicação. Entretanto, um dos desafios significativos enfrentados durante o desenvolvimento foi a falta de uma camada de observabilidade. Essa ausência torna difícil a identificação de limitações de desempenho, problemas de latência e falhas que possam surgir em produção, comprometendo a eficiência e a confiabilidade do sistema.

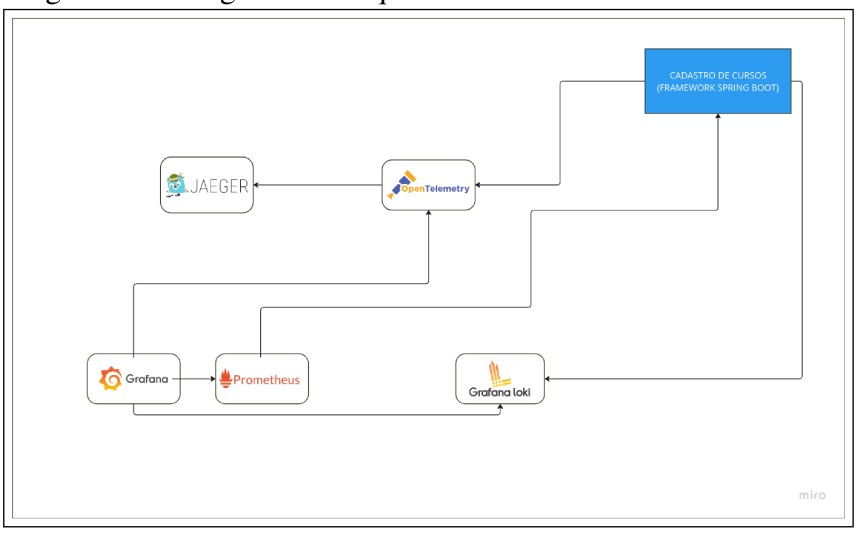
3.3 Definição geral da arquitetura de observabilidade

O primeiro passo envolveu uma análise detalhada dos requisitos do sistema, com o objetivo de identificar as funcionalidades mais críticas do microsserviço de gerenciamento de cursos e garantir que ele fosse adequadamente priorizado para implementação e monitoramento. Durante essa análise, foram destacados como essenciais os processos de criação, atualização e remoção de registros de cursos, assegurando a integridade e a precisão dos dados. Além disso, a

necessidade de garantir que o sistema possa se integrar eficientemente com o cache Redis e o banco de dados foi considerada fundamental para o desempenho do microsserviço.

Visando assegurar a qualidade operacional do sistema e promover a integração eficaz das ferramentas selecionadas para aprimorar a observabilidade, foram escolhidas as seguintes ferramentas, levando em consideração a compatibilidade com microsserviços: Prometheus para métricas, Grafana para visualização, Jaeger para traces e OpenTelemetry para correlação e Grafana Loki para gerenciamento de logs. A Figura 10 mostra o digrama dessa arquitetura.

Figura 10 – Diagrama da arquitetura de monitoramento



Fonte: autor, 2024

O microsserviço foi instrumentado utilizando o Spring Boot Actuator em conjunto com o Micrometer, que são responsáveis por exportar métricas cruciais do sistema, como latência das requisições, taxas de erro, e uso de recursos, como CPU e memória. O Actuator fornece os endpoints para acessar essas métricas, enquanto o Micrometer facilita a integração com diversas plataformas de monitoramento, permitindo uma observabilidade completa e eficiente dos microsserviços. Exemplos práticos vão ser implementados, destacando a importância de cada métrica para avaliar e otimizar o desempenho do sistema.

3.3.1 Implementação da observabilidade

3.3.1.1 Agente coletor

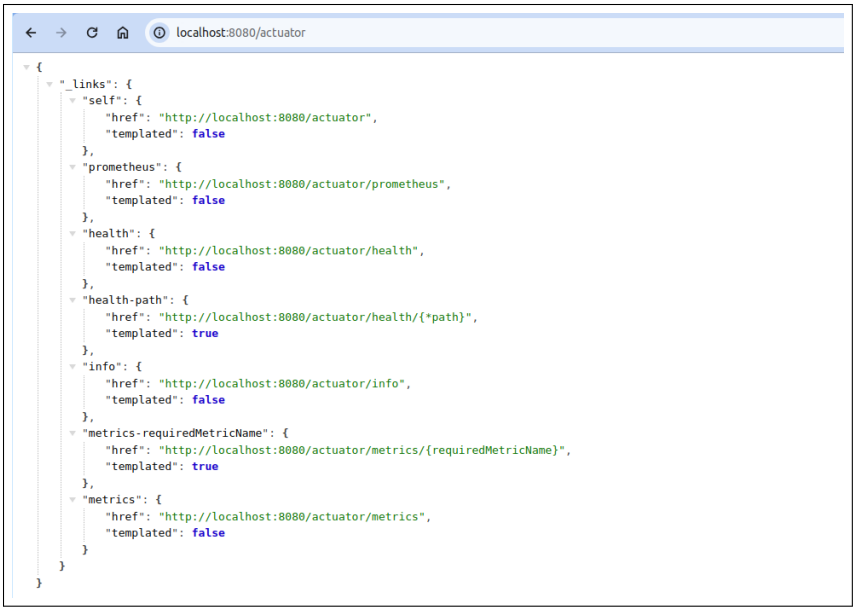
As informações são expostas por meio de um agente coletor. Como nosso sistema foi desenvolvido em Spring Boot, podemos utilizar um agente chamado Spring Boot Actuator, que fornece pontos de acesso prontos para monitorar e gerenciar aplicações Spring Boot em

produção. Ele oferece uma série de endpoints que permitem visualizar informações vitais, como métricas, saúde do sistema, detalhes de configuração, logs, e mais. Esses endpoints podem ser utilizados para realizar diagnósticos e entender o estado atual da aplicação sem a necessidade de adicionar código manualmente para expor esses dados.

Em conjunto foi utilizado o Micrometer, que é responsável por instrumentação que se integra perfeitamente ao Spring Boot Actuator para fornecer métricas aplicacionais robusta e extensível. Ele funciona como uma interface entre a aplicação e diversas plataformas de monitoramento, como Prometheus, Grafana, New Relic, entre outras. Com o Micrometer, é possível coletar métricas detalhadas sobre o desempenho da aplicação, como tempo de resposta, utilização de recursos, e contagem de eventos, além de criar métricas customizadas específicas para as necessidades do negócio. Em conjunto, Actuator e Micrometer fornecem uma solução completa para monitoramento, facilitando a observabilidade e a gestão eficiente de aplicações Spring Boot. Para habilita-lo podemos adicionar suas dependências no projeto maven.

Após o início da aplicação, é possível verificar a disponibilização automática de alguns endpoints padrão do Spring Boot Actuator. Esses endpoints podem ser acessados através da URL: http://localhost:8080/actuator, fornecendo acesso imediato a informações essenciais para o monitoramento e a gestão da aplicação.

Figura 11 – Métricas expostas pelo actuator



Fonte: Elaborado pelo próprio autor.

O Spring Boot utiliza o Micrometer, uma fachada que simplifica a integração de métricas com plataformas externas de monitoramento. Dentre os diversos sistemas suportados,

como Atlas e AWS CloudWatch, optamos por utilizar o Prometheus. Essa escolha se deve à sua alta capacidade de coleta e armazenamento de métricas em tempo real, integração nativa com o ecossistema Kubernetes, além de oferecer uma poderosa linguagem de consulta (PromQL) para análise detalhada dos dados, o que o torna uma solução robusta e amplamente utilizada em ambientes de microsserviços.

3.3.1.2 Configurando o Prometheus

O primeiro passo para configurar o Prometheus é criar e configurar o arquivo prometheus.yml, que define como o Prometheus coleta as métricas. Este arquivo pode ser criado dentro do próprio projeto para maior controle e personalização. Para executar o Prometheus, utilizamos o Docker com Docker Compose, o que simplifica significativamente o processo de configuração e execução do serviço, automatizando o gerenciamento do contêiner e garantindo um ambiente consistente e facilmente reproduzível.

O Docker empacota o Prometheus em um contêiner, garantindo portabilidade e consistência, enquanto o Docker Compose define e gerencia a configuração do contêiner e suas dependências através de um arquivo docker-compose.yml. Isso permite iniciar o Prometheus e outros serviços relacionados com um único comando, facilitando o gerenciamento e a integração com outros componentes do sistema.

3.3.1.3 Configurando o Grafana

O Grafana é uma das ferramentas mais completas para a exibição de gráficos de monitoramento, destacando-se pela sua alta configuração e flexibilidade. Ele permite a criação de diversos dashboards que facilitam a visualização de métricas de maneira intuitiva e com rápida compreensão. Sua interface amigável e a capacidade de personalização são grandes atrativos, possibilitando uma visualização detalhada e eficiente dos dados monitorados.

Para adicionar o Grafana ao projeto, utilizamos o Docker Compose. Essa ferramenta simplifica a configuração e a execução do Grafana, permitindo a criação de ambientes complexos com múltiplos serviços de forma simplificada. O Docker Compose gerencia dependências e configurações, facilitando a integração de componentes como o Grafana e suas fontes de dados sem a necessidade de configurações manuais extensivas.

Após adicionar o Grafana ao projeto, o próximo passo é integrar o Prometheus como fonte de dados. O Prometheus coleta e armazena métricas que podem ser visualizadas no Grafana.

Com o Prometheus configurado como fonte de dados, você pode criar dashboards personalizados no Grafana para exibir gráficos, tabelas e outros widgets baseados nas métricas coletadas. Isso permite uma análise detalhada e uma visualização abrangente das informações monitoradas. Na Figura 12 podemos observar as métricas extraídas da aplicação no *Dashboard* do Grafana.

Figura 12 – Dashboard Grafana

Fonte: Elaborado pelo próprio autor.

3.3.1.4 Configurando o Jaeger com OpenTelemetry

O Jaeger é uma ferramenta de rastreamento distribuído que permite monitorar e depurar a performance de sistemas complexos, identificando problemas de latência e entendendo o fluxo de chamadas entre diferentes serviços. Para uma configuração completa, é comum utilizar o Jaeger em conjunto com o OpenTelemetry, que fornece uma solução padronizada para a coleta de métricas e traces.

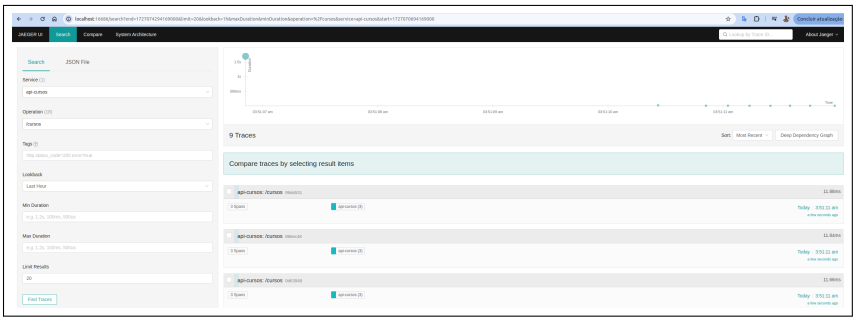
Para começar, você pode usar o Docker Compose para configurar o Jaeger. adicione ao arquivo docker-compose.yml os serviços do Jaeger, como o agente, o coletor e a interface de usuário. Além de configurar o Jaeger, você deve instrumentar seu código para enviar dados de rastreamento utilizando o OpenTelemetry. O OpenTelemetry permitem adicionar rastreamento e métricas de forma padronizada. Para isso, você precisará adicionar o agente ou biblioteca OpenTelemetry correspondente ao seu código, configurando-o para enviar os dados para o coletor Jaeger.

Finalmente, acesse a interface do usuário do Jaeger através do navegador, utilizando o endereço http://localhost:16686 como mostra na Figura 13. A partir da interface, você pode visualizar as traces coletadas, analisar o desempenho das operações e identificar possíveis gargalos ou problemas no sistema. O Jaeger, combinado com o OpenTelemetry, oferece uma visão detalhada da execução dos serviços, facilitando a análise e a otimização do desempenho.

Além disso, a integração do Jaeger com o OpenTelemetry não apenas simplifica o

processo de rastreamento, mas também promove uma maior consistência nas métricas coletadas. Ao adotar essa abordagem padronizada, as equipes de desenvolvimento conseguem implementar facilmente o monitoramento em diferentes serviços, independentemente da linguagem de programação utilizada. Isso resulta em uma visibilidade aprimorada do comportamento do sistema, permitindo que os desenvolvedores identifiquem e resolvam problemas de forma mais ágil. Com a capacidade de visualizar as interações entre os serviços em tempo real, o Jaeger se torna uma ferramenta indispensável para garantir a eficiência e a robustez de sistemas distribuídos.

Figura 13 – Tela inicial do Jaeger



Fonte: Elaborado pelo próprio autor.

3.4 Síntese do capítulo

Neste capítulo, começamos com uma compreensão detalhada sobre o propósito e a utilização das ferramentas de observabilidade. Abordamos desde os conceitos fundamentais até a forma como essas ferramentas se integram no ambiente de microsserviços.

Em seguida, exploramos o processo completo para a criação do projeto base. Isso incluiu a escolha cuidadosa das ferramentas mais adequadas, a configuração do ambiente e a geração de dados iniciais. Ao longo do processo, discutimos como as métricas são capturadas e visualizadas, culminando na análise dos traces nas plataformas de rastreamento. Destacamos, assim, a importância dessas ferramentas no monitoramento e na detecção de problemas em sistemas distribuídos.

No próximo capítulo, daremos continuidade a essa análise prática, executando um cliente responsável por realizar requisições, gerando assim uma massa de dados mais robusta. Esse cenário nos permitirá observar o funcionamento completo do conjunto de ferramentas de observabilidade em um ambiente mais dinâmico, analisando em detalhes a coleta, monitoramento e visualização das métricas e logs gerados pelas requisições.

4 RESULTADOS

Agora que as ferramentas de observabilidade selecionadas estão devidamente configuradas no projeto, o próximo passo é demonstrar o funcionamento integrado dessas ferramentas e evidenciar o impacto positivo que sua adoção pode trazer ao sistema. A interação gerada pelo uso conjunto dessas tecnologias não apenas facilita o monitoramento, mas também aprimora a capacidade de identificar, diagnosticar e solucionar problemas em tempo real, resultando em maior eficiência e confiabilidade operacional.

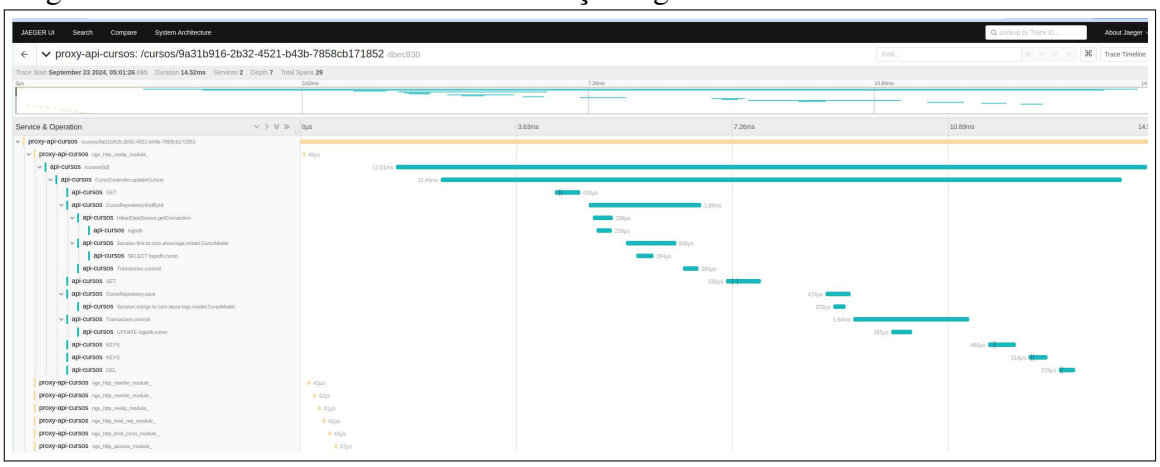
4.1 Aplicação de ferramenta de teste de carga

No intuito de demonstrar o impacto da observabilidade em um ambiente de microsserviços, foi desenvolvido um script cliente (Anexo 1) que simula diversas requisições ao microsserviço de gerenciamento de cursos, criando um cenário de teste com alta demanda de tráfego. Esse script foi projetado para simular um ambiente de produção, onde múltiplos usuários interagem simultaneamente com o serviço, enviando requisições de criação, atualização, e consulta de cursos, replicando de forma realista o comportamento esperado no dia a dia de uma aplicação em escala. A geração desse fluxo intenso de requisições foi essencial para testar a robustez da infraestrutura e identificar a capacidade de resposta do sistema sob diferentes níveis de carga, além de possibilitar uma análise detalhada de como o microsserviço lida com picos de demanda.

Antes de realizar os testes de carga, foi implementado um proxy reverso com Nginx. Essa etapa foi essencial para garantir uma melhor distribuição das requisições e facilitar o balanceamento de carga entre as instâncias do microsserviço, com ele em funcionamento foi possível simular um ambiente mais próximo ao de produção, onde o tráfego passa por um ponto de controle antes de atingir o microsserviço.

Ao executar o script em conjunto com a aplicação, é possível observar a rastreabilidade do sistema, conforme ilustrado na Figura 14. O diagrama mostra todos os pontos pelos quais o sistema transita antes de retornar a resposta ao cliente, começando pelo proxy, passando pela aplicação e incluindo as consultas realizadas no banco de dados. Essa visualização permite uma compreensão clara do fluxo de dados e das interações envolvidas no processo.

Figura 14 – Rastreabilidade do microsserviço de gerenciamento de cursos



Fonte: Elaborado pelo próprio autor.

Com a massa de teste de requisições podemos observar isso também no dashboard do Grafana, conforme ilustrado na Figura 15. onde temos o gráfico o numero de requisições por endpoint e a duranção média de requisições por minuto.

Figura 15 – Dashboard quantidade e duração de requisições por minuto

Fonte: Elaborado pelo próprio autor.

4.2 Analise do comportamento da ferramenta em cenário de falha

A observabilidade é crucial para a manutenção e operação eficaz de sistemas complexos, especialmente em arquiteturas de microsserviços. Neste capítulo, analisamos o comportamento de ferramentas de observabilidade quando expostas a falhas simuladas, destacando a importância de identificar e resolver problemas rapidamente em ambientes de produção.

Para avaliar a eficácia das ferramentas de observabilidade, realizamos uma simulação de falha derrubando o banco de dados. Este cenário foi escolhido devido à sua frequência em sistemas de produção e ao impacto potencial em diversas operações do microsserviço.

Durante a execução da simulação, o contêiner do banco de dados foi interrompido

intencionalmente. O monitoramento em tempo real permitiu observar as reações das ferramentas de observabilidade.

Inicialmente, foi observado um aumento significativo na taxa de erros logo após o banco de dados ser interrompido, permitindo identificar rapidamente a ocorrência da falha no dashboard do Grafana, conforme ilustrado na figura 16. Juntamente com o aumento da taxa de erros, as latências nas requisições ao microsserviço dispararam, refletindo o impacto da falta de resposta do banco de dados. Os gráficos de latência ajudaram a quantificar o tempo médio de resposta e a identificar picos durante a falha.

Figura 16 – Dashboard quantidade de erro na conexão com o banco de dados.

Fonte: Elaborado pelo próprio autor.

Além disso, as métricas geradas pelo Prometheus e exportadas para o Grafana mostraram que o uso de CPU e memória do microsserviço foi monitorado, revelando como o sistema lidou com a sobrecarga de requisições sem o suporte do banco de dados. Essa análise permitiu entender melhor o comportamento do sistema sob condições adversas e destacou a necessidade de otimizações para aumentar a resiliência do microsserviço em situações semelhantes.

Durante a análise dos traces de requisições no Jaeger, foi possível identificar as etapas que levaram às falhas e a propagação dos erros por diferentes serviços. Essa análise forneceu um panorama detalhado das interações no sistema, permitindo entender como cada componente contribui para o funcionamento geral. A visualização dos traces revelou não apenas onde as falhas ocorreram, mas também como as requisições se comportaram sob estresse, o que é essencial para diagnosticar problemas e implementar soluções eficazes.

Os dados de tempo de execução, por sua vez, revelaram onde as latências estavam concentradas, permitindo identificar gargalos críticos na comunicação entre os microsserviços durante a falha. O gráfico de dependências destacou quais microsserviços foram impactados pela indisponibilidade do banco de dados, evidenciando a interconexão entre eles. Essa visão

reforça a importância de uma abordagem holística para a observabilidade, que considere não apenas o desempenho individual de cada serviço, mas também as interações complexas entre os componentes do sistema. Essa compreensão é vital para a otimização do desempenho e para garantir a resiliência do sistema em face de falhas.

4.3 Análise dos resultados e impacto na eficiência do sistema

O impacto na eficiência do sistema foi notório, pois a falta de resposta do banco de dados não apenas causou falhas nas requisições, mas também resultou em uma sobrecarga significativa dos recursos computacionais. O monitoramento das métricas de CPU e memória demonstrou que, durante a falha, o microsserviço consumiu recursos de maneira ineficiente, o que poderia levar a um aumento nos custos operacionais e à degradação do desempenho em outros serviços interconectados.

Os dados coletados permitiram identificar áreas específicas que necessitam de melhorias. A análise das latências revelou gargalos críticos na comunicação entre serviços, sugerindo a necessidade de otimizações no fluxo de dados. Além disso, a implementação de circuit breakers e técnicas de caching poderia melhorar a resiliência do sistema, reduzindo o impacto de falhas futuras e aumentando a eficiência geral.

Em resumo, a análise dos resultados da simulação de falha não apenas destacou as vulnerabilidades do sistema, mas também proporcionou uma oportunidade valiosa para implementar melhorias. O impacto na eficiência do microsserviço, exacerbado pela falta de resposta do banco de dados, reforça a importância de uma arquitetura de sistemas que priorize a observabilidade e a resiliência. Ao adotar as lições aprendidas, é possível criar um ambiente mais robusto e eficiente, capaz de suportar a pressão de um cenário real de produção.

5 CONCLUSÃO

A implementação de práticas de observabilidade específicas para microsserviços representa um marco significativo na busca por sistemas mais resilientes e eficientes. Ao longo desta análise, foi possível constatar que a observabilidade desempenha um papel crucial na identificação e resolução proativa de problemas em ambientes distribuídos. A capacidade de monitorar, analisar e entender o comportamento dos microsserviços em tempo real permite uma resposta ágil a incidentes, contribuindo para a manutenção da estabilidade e desempenho do sistema.

A adoção dessas práticas não apenas proporciona uma visibilidade aprimorada, mas também promove uma cultura de melhorias contínuas. Ao integrar métricas, logs e traces de forma unificada, as equipes de desenvolvimento e operações ganham dados valiosos sobre o desempenho do sistema como um todo. Essa abordagem detalhada não apenas simplifica a detecção de anomalias no sistema, mas também facilita a otimização de recursos e a identificação de oportunidades para aprimoramento da arquitetura de microsserviços. Portanto, a implementação de práticas de observabilidade específicas para microsserviços representa um investimento estratégico na construção e manutenção de sistemas robustos e eficientes.

Além disso, a observabilidade não se limita apenas à detecção de problemas; ela também desempenha um papel fundamental na colaboração entre equipes. Através de uma plataforma de observabilidade centralizada, os desenvolvedores e operadores podem compartilhar informações em tempo real, promovendo uma comunicação mais eficaz e reduzindo o tempo de resposta a incidentes. Essa transparência no fluxo de dados e na análise de desempenho cria um ambiente de trabalho mais colaborativo, onde as equipes podem aprender umas com as outras e alinhar seus objetivos em torno da resiliência e eficiência do sistema. Assim, a implementação de práticas de observabilidade não só fortalece a integridade dos microsserviços, mas também cultiva uma cultura organizacional focada na inovação e na excelência contínua.

REFERÊNCIAS

ALSUBHI K. M., . S. A. **A Review on Microservices Architecture and Its Adoption Challenges. In Proceedings of the 2018 IEEE/ACM 1st International Workshop on Rapid Continuous Software Engineering**. [S.l.: s.n.], 2018.

ASPECTO. **what-is-opentelemetry-the-infinitive-guide**. 2023. Opentelemetry. Disponível em: <https://www.aspecto.io/blog/what-is-opentelemetry-the-infinitive-guide/>. Acesso em: 06 Dez. 2023.

BLOG, U. **Introducing Domain-Oriented Microservice Architecture**. 2020. Uber. Disponível em: <https://www.uber.com>. Acesso em: 06 Fev. 2024.

DIFRANCESCO, P. **Architecting microservices . In 1st International Conferenceon Software Architecture Workshops (ICSAW)**. [S.l.: s.n.], 2017.

DYNATRACE. **what is opentelemetry**. 2023. Opentelemetry. Disponível em: <https://www.dynatrace.com>. Acesso em: 06 Dez. 2023.

FOWLER, M.; LEWIS., J. **Microservices, a definition of this new architectural term,**. 2014. FOWLER. Disponível em: <https://martinfowler.com>. Acesso em: 06 Dez. 2023.

LABS, G. **GRAFANA**. 2023. GRAFANA LABS. Disponível em: <https://grafana.com>. Acesso em: 05 Nov. 2023.

LIMA, M. d. A. **Análise de soluções de rastreamento open source no contexto de aplicações baseadas em microsserviços.** [S.l.: s.n.], 2022.

OVERVIEW. **What is Prometheus?** 2023. OVERVIEW. Disponível em: <https://prometheus.io/docs/introduction/overview>. Acesso em: 03 Set. 2023.

SOUNDARARAJAN V., . K. S. **Implementing Microservices: An Experience Report on Challenges and Lessons Learned. In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2019.

V., N. **Processing Time Series Data in Real-Time with InfluxDB and Structured Streaming**. 2018. Medium. Disponível em: <https://medium.com>. Acesso em: 06 Jan. 2024.

YAGOUB M., S. E. . J. Z. M. **Understanding Microservices Challenges: A Multi-tenant Application Case Study. In Proceedings of the 16th International Conference on Mining Software Repositories (MSR)**. [S.l.: s.n.], 2019.