Fusão de Dados de Sinais de Observabilidade para Detectar Anomalias e Falhas em Microsserviços

Autor: Adalberto Felipe Pinheiro Chaves *
Autor: Francisco Anderson de Almada Gomes **

RESUMO

Microsserviços se tornaram uma realidade, com grandes empresas adotando essa arquitetura devido às suas vantagens, como agilidade no desenvolvimento, desacoplamento, escalabilidade, resiliência e eficiência operacional. No entanto, essa abordagem apresenta desvantagens, como a necessidade de infraestrutura de nuvem complexa e comunicação entre microsserviços por meio de protocolos. Em organizações com centenas de microsserviços, qualquer falha em um deles pode desencadear efeitos em cascata, resultando em mais falhas. Identificar falhas em microsserviços não é uma tarefa trivial. O trabalho proposto foca na fusão de dados para detectar falhas e anomalias em microsserviços, combinando dois pilares de observabilidade: rastreamento distribuído e métricas, com a ajuda do OpenTelemetry. Essa fusão visa fornecer uma detecção de falhas mais rápida e eficiente, bem como a identificação de anomalias nas métricas. Isso permite que problemas sejam detectados e resolvidos mais rapidamente, melhorando a estabilidade e a confiabilidade do sistema.

Palavras-chave: microsserviços; falhas; anomalias; observabilidade; fusão de dados.

ABSTRACT

Abstract Microservices have become a reality, with large companies adopting this architecture due to its advantages, such as development agility, decoupling, scalability, resilience, and operational efficiency. However, this approach has drawbacks, such as the need for complex cloud infrastructure and communication between microservices through protocols. In organizations with hundreds of microservices, any failure in one of them can trigger cascading effects, resulting in more failures. Identifying failures in microservices is not a trivial task. The proposed work focuses on data fusion for detecting failures and anomalies in microservices, combining two pillars of observability: distributed tracing and metrics, with the help of OpenTelemetry. This fusion aims to provide faster and more efficient failure detection, as well as the identification of anomalies in metrics. This allows problems to be detected and addressed more quickly, improving the overall stability and reliability of the system.

Keywords: microservices; failures; anomalies; observability; data fusion.

1 INTRODUÇÃO

Uma arquitetura de software bem projetada pode ajudar a garantir que um sistema atenda a requisitos-chave em áreas como desempenho, confiabilidade, portabilidade, escalabilidade e interoperabilidade (GARLAN, 2008). À medida que o tamanho e a complexidade dos sistemas de software aumentam, o problema de *design* vai além de algoritmos de computação e estruturas de dados: projetar e especificar a estrutura geral do sistema surge como um novo tipo de desafio (GARLAN; SHAW, 1993). Devido à complexidade dos sistemas modernos, a indústria de software começou a adotar um novo padrão arquitetônico, onde o sistema é modularizado em serviços (DRAGONI *et al.*, 2017a).

* felipe.pinheiro.c27@gmail.com
** almada@crateus.ufc.br

A arquitetura de microsserviços tornou-se uma realidade no meio industrial. As arquiteturas de microsserviços auxiliam gerentes de projeto e desenvolvedores ao fornecer diretrizes para o *design* e a implementação de aplicações distribuídas (DRAGONI *et al.*, 2017b). Tais arquiteturas estão presentes em vários tipos de aplicações, incluindo sistemas web, móveis e de *e-commerce*, trazendo fatores positivos como escalabilidade, flexibilidade e portabilidade. Nessa arquitetura, os serviços são pequenas aplicações que podem ser implementadas, escaladas e testadas de forma independente, cada uma com uma única responsabilidade, ou seja, uma única regra de negócio (THÖNES, 2015).

Devido à sua natureza altamente distribuída, os microsserviços são mais desafiadores de operar. Por isso, exigem tecnologias de virtualização e infraestrutura (baseadas em contêineres), como *Docker* e *Kubernetes*, que proporcionam atualizações contínuas, escalonamento automatizado e reequilíbrio em caso de falha de um nó (HEINRICH *et al.*, 2017). Dada a complexidade operacional, monitorá-los torna-se uma tarefa difícil.

A garantia de qualidade dos microsserviços é frequentemente complementada, ou até mesmo substituída, por técnicas refinadas de monitoramento em ambientes de produção expostos a cargas de trabalho reais (MENDONÇA *et al.*, 2019). Grandes sistemas podem conter centenas de serviços comunicando-se entre si. Falhas em alguns desses serviços podem levar a falhas em outros, e a rápida identificação dessas falhas pode evitar efeitos em cascata, resultando em um sistema mais estável. Além disso, a detecção de anomalias é crucial nesse contexto (RAEISZADEH *et al.*, 2023). Anomalias em métricas, são picos inesperados no uso de recursos ou tempos de resposta incomuns e podem sinalizar problemas potenciais antes que se transformem em falhas.

As soluções tradicionais de monitoramento muitas vezes carecem da capacidade de fornecer visibilidade abrangente e detectar proativamente desvios do comportamento esperado antes que impactem os serviços. Para suprir essas lacunas, é necessário um novo nível de monitoramento conhecido como observabilidade (KOSIŃSKA *et al.*, 2023). Observabilidade é definida como a capacidade de deduzir o estado de um sistema complexo a partir de suas saídas (KALMAN, 1960). Também se refere à propriedade de um sistema que permite que seu estado seja observado e comparado com condições esperadas ao longo do ciclo de desenvolvimento de software (KARUMURI *et al.*, 2021). Os três sinais de observabilidade são métricas, rastreamentos e *logs*. Devido à sua importância crítica, uma variedade de ferramentas tem sido propostas para apoiar a observabilidade. Nos últimos anos, o *OpenTelemetry* (OTel) tornou-se a ferramenta mais amplamente adotada na indústria e é considerado a ferramenta padrão para observabilidade (MAJORS *et al.*, 2022). OTel é uma estrutura de observabilidade de código aberto que oferece um conjunto de *Software Development Kit* (SDK), *Application Programming Interface* (API) e ferramentas padronizadas e independentes de fornecedor para recuperação, transformação e transmissão de dados para um *backend* de observabilidade (BOTEN; MAJORS, 2022).

A fusão de dados permite a integração de informações provenientes de diferentes sensores, sistemas e fontes (TZANETTIS *et al.*, 2022). Isso pode fornecer uma visão mais abrangente e precisa do estado do sistema, ajudando a detectar anomalias e falhas que poderiam ser difíceis de identificar usando uma única fonte de dados. Além disso, ao combinar dados de várias fontes, é possível identificar padrões anômalos que podem indicar uma falha possível.

Este estudo tem como objetivo realizar a fusão de dados dos sinais de observabilidade para aumentar a eficácia na detecção de falhas e anomalias em arquiteturas de microsserviços. A solução desenvolvida, chamada **FOCUS** (data **F**usion of **O**bservability signals to dete**C**t anomalies and fail**U**res in microservice**S**), utiliza o OTel como framework de observabilidade. Esta pesquisa aproveita técnicas de observabilidade — especificamente métricas e rastreamentos — para desenvolver um sistema mais robusto e menos suscetível a falhas. Ao integrar e analisar

esses diversos sinais de observabilidade, o estudo visa detectar e diagnosticar problemas, contribuindo para um ambiente de microsserviços mais estável e resiliente. Como resultado, através de um experimento de teste de carga, constatou-se que a solução é capaz de detectar falhas e anomalias na aplicação. No contexto apresentado, este artigo tem como objetivo responder a uma questão inicial de pesquisa (RQ):

RQ *Como a fusão de dados de diferentes sinais de observabilidade facilita a detecção de anomalias e falhas em arquiteturas de microsserviços?*

As contribuições deste artigo incluem a fusão de dados de sinais de observabilidade, combinando métricas e rastreamento distribuído, implementada por meio de instrumentação com OTel. Essa abordagem permite a associação de métricas com rastreamentos, permitindo que o rastreamento distribuído identifique serviços com falhas, detectando valores anômalos nas métricas configuradas. Uma biblioteca foi criada para facilitar essa configuração, que é integrada à aplicação de microsserviços. Além disso, é possível selecionar os serviços a serem monitorados, trazendo esses dados para um *dashboard* que classifica os serviços em níveis de risco com base nas métricas em tempo real coletadas durante a execução. Por fim, uma aplicação de referência chamada Spring Petclinic foi utilizada no experimento de teste de carga para verificar a eficácia da solução.

O restante do artigo está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica. A Seção 3 discute trabalhos relacionados. A integração do OTel, Observabilidade para Detectar Falhas e Anomalias, e as soluções aplicadas são explicadas na Seção 4. A Seção 5 detalha os experimentos e resultados. Finalmente, a Seção 6 conclui o estudo e fornece sugestões para pesquisas futuras.

2 FUNDAMENTAÇÃO TEÓRICA

O objetivo deste Capítulo é fornecer conceitos sobre microsserviços e abordar a importância da observabilidade nessa arquitetura. Será exposto: arquitetura de microsserviços e suas características, observabilidade e sua importância na detecção de falhas e anomalias e por fim, OpenTelemetry.

2.1 Microsserviços

Microsserviços são um estilo arquitetural que enfatiza a decomposição de um sistema em serviços pequenos e leves, deliberadamente projetados para executar funções de negócios altamente coesas. Representa uma evolução tradicional da *Service-Oriented Architecture* (SOA) (PAUTASSO *et al.*, 2017). SOA é um padrão de arquitetura de software caracterizado por baixo acoplamento, o que torna os componentes reutilizáveis ao empregar serviços com uma linguagem comum em uma rede. Esses aspectos trazem vários benefícios para os microsserviços, incluindo maior agiliade, desacoplamento de funcionalidades e facilidade de implantação e manutenção (SINGLETON, 2016).

Como os microsserviços são pequenas aplicações que podem ser implantadas de forma independente, cada serviço é idealmente projetado para implementar uma regra de negócios ou funcionalidade específica do sistema. Além disso, cada serviço deve possuir sua própria lógica e evitar dependências de outros microsserviços no sistema, o que significa que sua responsabilidade deve ser única (THÖNES, 2015). Esses serviços são construídos em torno de recursos de negócios e implantados independentemente por sistemas totalmente automatizados. Há um gerenciamento centralizado mínimo desses serviços, cada serviço é um código separado que pode ser gerido por uma equipe de desenvolvimento e pode ser escrito em diferentes

linguagens de programação, além de utilizar várias tecnologias de banco de dados (WASEEM *et al.*, 2020). Consequentemente, essa arquitetura exige uma cultura *DevOps* responsável pela implementação de *pipelines* de entrega contínua e integração contínua (CI/CD) e pela orquestração dos microsserviços em um ambiente de nuvem.

Devido a essas características, essa abordagem também apresenta algumas desvantagens. Com múltiplas aplicações, todas devem ser monitoradas individualmente, ao invés de apenas uma, criando a necessidade de uma infraestrutura de monitoramento distribuída e de visualização centralizada (CHANDRAMOULI, 2019). Os testes de integração tornam-se mais desafiadores, pois é necessário um ambiente de testes no qual todos os componentes estejam operacionais e se comuniquem. Como as interações em uma aplicação baseada em microsserviços são projetadas como chamadas de API, todos os processos necessários para o gerenciamento de API devem ser implementados. Isso inclui vários componentes que precisam ser desenvolvidos juntamente com a arquitetura de microsserviços, como filas de mensagens para comunicação, um novo *pipeline* de implantação e um *gateway* de API (AL-DEBAGY; MARTINEK, 2018).

2.2 Observabilidade

A observabilidade oferece visões gerais de alto nível sobre a saúde do sistema e *insights* detalhados falhas. Um sistema observável proporciona um contexto extenso sobre suas operações internas, permitindo a descoberta de falhas sistêmicas mais profundas (USMAN *et al.*, 2022). A observabilidade capacita aqueles que desenvolvem e operam sistemas e aplicações distribuídas a entender o comportamento do código em produção (BOTEN; MAJORS, 2022). Também pode ser vista como um superconjunto do monitoramento, no sentido de que, se um sistema é observável, ele pode ser monitorado. A observabilidade fornece informações que auxiliam no monitoramento, permitindo a navegação dos efeitos para as causas em um sistema de produção, compreendendo o que, onde e por que as coisas acontecem. A observabilidade de um sistema distribuído baseia-se em três pilares principais (sinais): *logs*, métricas e rastreamentos (*traces*).

Logs são arquivos que registram eventos, avisos e erros conforme ocorrem em um ambiente de software. A maioria dos *logs* inclui informações contextuais, como o horário de ocorrência de um evento e qual usuário ou *endpoint* estava associado a ele (MAJORS *et al.*, 2022). Por exemplo, um arquivo de *log* de servidor web pode incluir quando o servidor foi iniciado, solicitações de clientes e como o servidor respondeu a essas solicitações. Ele registra informações sobre cada transação bem-sucedida, bem como erros e falhas nas conexões dos clientes.

A medição do desempenho de aplicações e sistemas através da coleta de métricas é uma prática comum no desenvolvimento de software. Esses dados são convertidos em gráficos para gerar visualizações significativas para aqueles responsáveis por monitorar a saúde do sistema. Em determinados ambientes, as métricas são usadas para automatizar fluxos de trabalho em resposta a mudanças no sistema, como aumentar o número de instâncias de aplicação ou reverter uma implantação defeituosa (MAJORS *et al.*, 2022). Métricas são dados de baixo nível que podem indicar problemas com o equipamento subjacente de computação, armazenamento ou rede. O monitoramento cuidadoso dessas métricas pode destacar a degradação de desempenho e detectar anomalias, prevenindo falhas em todo o sistema. Isso as torna cruciais para a tomada de decisões sobre melhorias no sistema.

O rastreamento de aplicações refere-se à capacidade de executar o código da aplicação e garantir que ele funcione conforme esperado, passando por pontos da aplicação e enviando parâmetros (MAJORS *et al.*, 2022). Alcançar isso, especialmente considerando o número de

serviços em projetos maiores, é uma tarefa desafiadora, exigindo o uso de mecanismos para realizar esse rastreamento. Uma abordagem nesse contexto é o rastreamento distribuído. O rastreamento distribuído é uma técnica de diagnóstico que ajuda os engenheiros a localizar falhas e problemas de desempenho em aplicações, especialmente aquelas distribuídas em vários computadores ou processos. Com o rastreamento distribuído, é possível determinar se alguma etapa de comunicação falhou, quanto tempo cada etapa levou e, potencialmente, registrar as mensagens geradas por cada etapa durante a execução. Ele observa as solicitações seguindo as etapas percorridas no sistema. Um contexto de rastreamento único (ID de rastreamento) é inserido no cabeçalho de cada solicitação, e mecanismos são implementados para garantir que esse contexto de rastreamento seja propagado ao longo do caminho da solicitação. Em outras palavras, para possibilitar a identificação, um identificador (ID) é criado na primeira solicitação e transmitido por todas as solicitações subsequentes através do cabeçalho da requisição.

No monitoramento de sistemas, métricas e traces são sinais de observabilidade essenciais que fornecem *insights* diferentes, mas complementares. As métricas oferecem medições quantitativas do desempenho do sistema, como tempos de resposta ou taxas de erros. Já os rastreamentos rastreiam o fluxo e as interações das solicitações entre vários componentes do sistema. Ao fundir métricas e rastreamento, os engenheiros conseguem uma compreensão mais profunda do comportamento do sistema e melhoram a detecção de falhas. Por exemplo, se uma métrica indicar um aumento nos tempos de resposta, os rastreamentos podem ajudar a identificar qual serviço ou componente específico está causando o atraso, mostrando o caminho e o tempo das solicitações através do sistema. Essa visão combinada permite um diagnóstico mais preciso de problemas de desempenho.

2.3 OpenTelemetry

O OTel é um *framework* e kit de ferramentas para observabilidade, projetado para criar e gerenciar dados de telemetria, como rastreamentos, métricas e *logs* (BOTEN; MAJORS, 2022). O OTel é agnóstico em relação a fornecedores e ferramentas, o que significa que pode ser usado com uma ampla variedade de *backends* de observabilidade. O *OpenTracing* é um padrão aberto para rastreamento distribuído em aplicações e pacotes de software de código aberto (OSS), permitindo que desenvolvedores instrumentem seu próprio código sem depender de um fornecedor específico de rastreamento. Já o *OpenCensus* é um conjunto de bibliotecas que permite a coleta de métricas de aplicação e rastreamentos distribuídos, transferindo esses dados em tempo real para um *backend* de escolha. Essa fusão trouxe dois principais benefícios: (*i*) a unificação dos padrões para instrumentação e coleta de dados, e (*ii*) um ecossistema mais amplo e robusto.

Para ser útil, os dados de telemetria precisam ser exportados para um sistema de armazenamento, onde podem ser armazenados e analisados (BOTEN; MAJORS, 2022). Para isso, cada implementação oferece uma gama de mecanismos para gerar, processar e transmitir telemetria, conhecidos como *pipelines*. Esses *pipelines* são executados no início da implementação para garantir que nenhum dado seja perdido e, após essa etapa, o papel do *pipeline* é coletar e enviar dados, sendo caracterizado por três processos principais: *receivers*, *processors* e *exporters*.

Os *receivers* são usados para fornecer acesso ao código da aplicação ou a uma entidade responsável por gerar os dados de telemetria e podem ser configurados desde o início, mesmo que ainda não haja dados gerados. Os *processors* realizam a agregação de dados, filtragem, amostragem e outras lógicas de processamento nos dados coletados, considerando possíveis alterações nos dados previamente processados. É possível encadear vários *processors*

para criar uma lógica de processamento mais complexa (THAKUR; CHANDAK, 2022). Os *exporters*, por fim, lidam com a emissão de dados de telemetria para um ou mais destinos em diferentes formatos e protocolos, com exemplos como *Prometheus*, *Jaeger* e *Zipkin* (THAKUR; CHANDAK, 2022).

3 TRABALHOS RELACIONADOS

Nesta seção, será apresenta os trabalhos relacionados à presente proposta.

3.1 *Automated Analysis of Distributed Tracing: Challenges and Research Direction*

Bento et al. (BENTO *et al.*, 2021) conduziram um estudo focado em encontrar regiões anômalas por meio de rastreamento distribuído. Eles afirmam que consultas manuais à ferramenta de rastreamento distribuído são necessárias, considerando o tempo e as anotações envolvidas. Portanto, é enfatizada a abordagem automatizada. Os autores desenvolveram um *Open Trace Processor* (OTP) para extrair métricas de rastreamentos e alimentá-las em um analisador de dados, que identifica anomalias em séries temporais, como o número de chamadas recebidas, chamadas feitas e tempos de resposta. Com base nos rastreamentos existentes e nas métricas derivadas deles—como o número de solicitações recebidas e enviadas, tempos de resposta ou códigos de erro de serviço—o objetivo era identificar ameaças à resiliência do *software*. Uma desvantagem desse trabalho é sua dependência do *OpenTracing*, um padrão mais antigo para observabilidade.

3.2 *Using Distributed Tracing to Identify Inefficient Resources Composition in Cloud Applications*

Casse et al. (CASSÉ *et al.*, 2021) focaram em explorar dados de rastreamento do OTel para detectar comunicações ineficientes em uma aplicação em nuvem. O objetivo era expor alocações ineficientes de recursos feitas pelo *Kubernetes* em um *cluster* geograficamente distribuído. Para obter os dados, o *Kubernetes* foi estendido com um *service mesh* em cada microsserviço, com o objetivo de observar as chamadas de rede entre os serviços. O *service mesh Linkerd* foi escolhido devido à sua compatibilidade com o formato *OpenCensus*, garantindo compatibilidade com os binários do OTel. Em um ambiente *Kubernetes*, o *Linkerd* injeta e configura automaticamente *proxies Hypertext Transfer Protocol* (HTTP) entre cada instância de microsserviço para habilitar a observação das comunicações. O próximo passo foi criar um grafo de propriedades, onde todos os rastreamentos foram considerados como um único grafo, decompondo os dados de rastreamento em múltiplos vértices e arestas, que foram mesclados com dados previamente observados, estabelecendo correlações entre múltiplos rastreamentos. Uma desvantagem desse trabalho é sua dependência exclusiva dos rastreamentos para detectar ineficiências de recursos, enquanto a fusão de dados, incorporando métricas, poderia contribuir para uma detecção mais eficaz.

3.3 *Demonstration of an Observability Framework for Cloud Native Microservices*

Marie et al. (MARIE-MAGDELAINE *et al.*, 2019) realizaram um estudo demonstrando a viabilidade de um monitoramento orientado para observabilidade usando um conjunto de ferramentas personalizadas. A arquitetura proposta inclui uma camada de observabilidade que extrai dados de aplicações nativas em nuvem, utilizando esses dados para informar as equipes

de *DevOps*. O *framework* de observabilidade foi implementado no ambiente da empresa *Lectra SA*, com a restrição de que todos os microsserviços dependem da infraestrutura da *Microsoft Azure*. Esse framework oferece funcionalidades como coleta, armazenamento e gerenciamento de métricas, utilizando *Prometheus* e *Grafana*. Os microsserviços são implantados por um orquestrador baseado em um arquivo manifesto que contém todos os parâmetros necessários para a implantação e execução das instâncias. A pesquisa fornece *insights* sobre conceitos, funcionalidades e protótipos relacionados à observabilidade em aplicações nativas de nuvem. Uma limitação deste trabalho é sua dependência exclusiva de métricas para detectar a necessidade de auto-escalonamento de recursos.

3.4 *Data fusion of observability signals for assisting orchestration of distributed applications*

Tzanettis et al. (TZANETTIS *et al.*, 2022) abordam o desafio de integrar dados de plataformas de orquestração, rastreamento distribuído e ferramentas de registro de *logs* para simplificar a análise de desempenho de aplicações distribuídas. Sua principal contribuição é a especificação de um esquema de vinculação de dados que suporta a coleta e integração de dados de recursos de contêiner, métricas de desempenho, rastreamento distribuído e *logs*. O esquema proposto é capaz de mesclar e correlacionar dados de diferentes tipos de sinais, implementado por meio de ferramentas de código aberto. Essas ferramentas utilizam o mecanismo de monitoramento *Prometheus* (suportado pelo *Kubernetes*), a ferramenta de rastreamento distribuído *Zipkin*, o software de registro de *logs Fluentd* e a biblioteca de instrumentação *Prometheus Python* para definir instâncias no código-fonte. Uma limitação deste trabalho é que, embora realize a fusão de dados de todos os sinais, ele apenas valida o esquema de dados e não realiza um experimento para demonstrar que a escalabilidade realmente melhora com a solução proposta.

Nenhum dos estudos avaliados utiliza fusão de dados por meio de sinais de observabilidade para detectar anomalias e falhas em microsserviços usando o padrão moderno de observabilidade do OTel.

4 METODOLOGIA

Esta seção descreve a metodologia utilizada em nosso estudo. A Seção 4.1 discute a implementação da biblioteca FOCUS para fusão de dados, enquanto a Seção 4.2 apresenta a aplicação Petclinic. A Seção 4.3 descreve como o projeto *Petclinic* foi integrado à biblioteca, e a Seção 4.4 abrange a criação de um serviço que gerencia as informações da FOCUS e detecta anomalias e falhas. Por fim, a Seção 4.5 detalha a criação de um serviço de dashboard para visualização dos serviços configurados.

4.1 FOCUS Biblioteca para Fusão de Dados

O primeiro passo do estudo foi desenvolver a biblioteca FOCUS, baseada em *Java* e OTel, para coleta e fusão de dados, com foco em métricas e traces dos sinais de observabilidade. Na fusão de dados, cada *span* de um *trace* inclui, além dos valores padrão como *traceId* e *spanId*, as métricas associadas ao *trace*. A biblioteca é acessada por meio de uma anotação *Java* chamada *@ObservabilityParam*, onde são passados parâmetros para monitorar métricas com valores máximos (mais detalhes na Seção 4.3) para identificar métricas anômalas, que são aquelas que excedem os valores máximos configurados. Os parâmetros configurados pelo desenvolvedor incluem o uso de CPU, memória, tempo de resposta e *throughput* que é a quantidade de entradas e saídas de dados que ocorreram no serviço. Assim, tanto métricas de infraestrutura quanto de

aplicação são avaliadas.

Para coletar os sinais de observabilidade, como descrito anteriormente, o OTel opera com o *Collector*, que possui componentes específicos para recepção e exportação de dados. O ***OTLP Receiver*** foi utilizado para receber os sinais de observabilidade da aplicação no formato OTLP (protocolo de dados do OTel). As métricas são enviadas para o ***Prometheus Exporter***, que exporta as métricas para um servidor *Prometheus* via *scraping*. Para obter métricas de infraestrutura, foi utilizado o ***HostMetrics Receiver***, que gera uma ampla variedade de métricas do sistema *host*, como uso de CPU, memória, I/O de disco, entre outras. Adicionalmente, o ***OTLP Exporte*****r** foi utilizado para exportar dados no formato *OTLP* via *gRPC*, o que permite a recuperação dos tempos de resposta de cada requisição. O cálculo de *throughput* foi feito somando os *bytes* recebidos e transmitidos pela interface de rede, fornecidos pelo *HostMetrics*, divididos pelo tempo de resposta da requisição. Por fim, um componente chamado ***Logging Exporter*** foi utilizado para enviar dados ao console via *zap.Logger*, suportando *traces*, métricas e *logs* no *pipeline*. Assim, embora não faça parte da fusão de dados neste estudo, os *logs* podem ser recuperados e analisados.

Para recuperar os valores das métricas coletadas e compará-los com os valores configurados pelo desenvolvedor, foi necessária uma implementação de Programação Orientada a Aspectos (AOP) em *Java* usando *AspectJ*. Nessa abordagem, um *advice* pode ser definido, que são trechos de código executados em pontos específicos (*join points*) do programa. Dessa forma, o código pode ser incluído antes, depois ou ao redor da execução de métodos. Isso possibilitou a recuperação das métricas de uso de CPU e memória durante a execução de qualquer método anotado com *@ObservabilityParam*. No passo seguinte, essas métricas precisaram ser incorporadas ao *span* do OTel para realizar a fusão dos dados. Para isso, foi utilizado o método *setAttribute* do OTel no *span* atual da requisição para atribuir as métricas coletadas. Assim, cada span contém as métricas de CPU, memória, tempo de resposta e *throughput*. Como resultado, a fusão dos dados de métricas e *traces* foi concluída, permitindo que ambos sejam visualizados em um único *span* nos sistemas de rastreamento distribuído. O *Jaeger*, um sistema de rastreamento distribuído criado pela *Uber*, que oferece *sampling* adaptativo e possui suporte nativo para *OTLP*, foi utilizado na solução proposta.

4.2 Sobre a aplicação

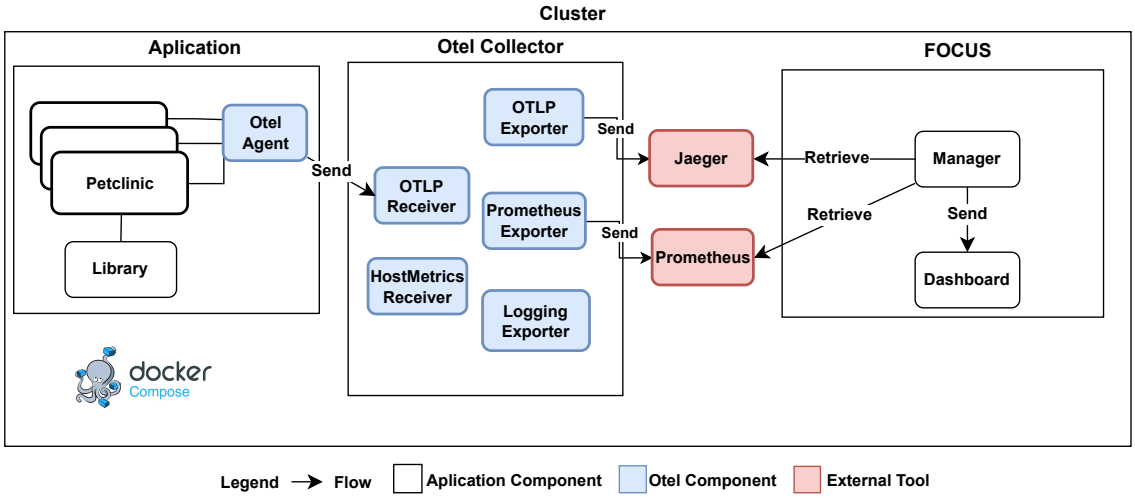
O *Petclinic* é uma aplicação de referência criada para demonstrar práticas do *Spring Framework* em microserviços, simulando o sistema de gestão de uma clínica veterinária. Suas principais funcionalidades incluem a gestão de clientes, onde é possível registrar e gerenciar as informações dos proprietários de pets; a gestão de pets, que permite o registro de pets associados a um proprietário e a visualização do histórico médico desses pets; e a gestão de veterinários, com a capacidade de registrar e listar veterinários e suas especializações. Desenvolvido em 2021 e com modificações contínuas, atualmente consiste em 4 microserviços relacionados à lógica de negócios e utiliza *Java*. Esta aplicação foi utilizada para validar a solução proposta e conduzir os experimentos para este trabalho.

4.3 Integração da Aplicação com a Biblioteca

Para integrar a aplicação *Petclinic* com a biblioteca FOCUS, foi necessário adicionar o agente OTel à imagem *Docker* e incluir a dependência da biblioteca em cada projeto de serviço. A Figura 1 apresenta uma visão geral da arquitetura da solução desenvolvida, que opera por meio do *docker-compose*, uma ferramenta para definir e executar múltiplos containeres *Docker*. Além

dos componentes do OTel *Collector* explicados anteriormente, os dados foram exportados para *backends* de observabilidade, que, conforme ilustrado na Figura 1, são o *Prometheus* e o *Jaeger*.

Figura 1 – Arquitetura da Solução



Após configurar o OTel e a aplicação, foi necessário configurar o uso da Anotação *@ObservabilityParam* nos métodos dos serviços que possuem os *endpoints (controllers)* e definir os valores de métricas que não devem ser excedidos. O uso da CPU é dado em porcentagem, a memória em MB, o tempo de resposta em milissegundos (ms) e a taxa de transferência (*throughput*) em bytes por segundo. Por exemplo, a anotação tem os seguintes parâmetros:

Código-fonte 1 – Exemplo de Anotação

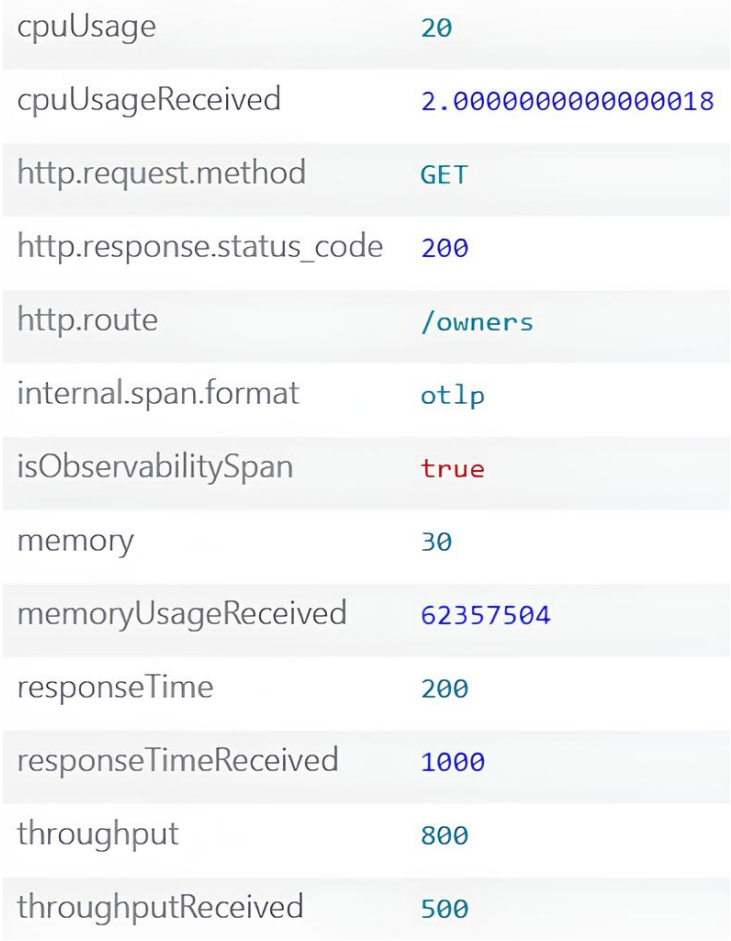
```
@ObservabilityParam(params = {
        @Param(key = "cpuUsage", value = "20"),
        @Param(key = "memory", value = "30"),
        @Param(key = "responseTime", value = "200"),
        @Param(key = "throughput", value = "800")
})
```

A partir do Listagem 1, pode-se observar que o uso da CPU é configurado para não exceder 20%, o uso de memória é limitado a 30 MB, o tempo de resposta não deve ultrapassar 200 ms, e a taxa máxima de transferência (*throughput*) é definida em 800 Bytes/ms. As configurações de métricas são salvas pela biblioteca no *span* e enviadas para o *Jaeger*. A Figura 2 mostra como esses dados são armazenados. Esses dados são posteriormente recuperados pelo serviço *Manager* do FOCUS, que é responsável por analisar as métricas para determinar se são anômalas, bem como identificar falhas nas requisições e realizar a exportação dos dados. Mais detalhes são apresentados na próxima seção.

4.4 FOCUS *Manager*

Após a integração com o projeto *Petclinic*, a configuração da biblioteca e o envio de dados para o *Jaeger*, o próximo passo foi criar o serviço FOCUS *Manager* para exportar dados, realizar a análise de métricas para detectar anomalias e falhas nas requisições, além

Figura 2 – Dados Salvos no Jaeger



de fornecer informações para o serviço de *dashboard*, que será apresentado posteriormente. O serviço *Manager* possui quatro *endpoints* principais: */api/metrics/system-info*, que retorna informações gerais sobre a aplicação, neste caso o *Petclinic*, incluindo o número de requisições, número de erros e requisições por segundo; */api/metrics/all*, que retorna os valores recebidos em tempo de execução dos métodos onde a Anotação foi aplicada; */api/metrics/system-metrics*, que retorna métricas da API do *Prometheus* sobre a saúde do sistema naquele momento, incluindo informações sobre uso de CPU, uso de memória e *throughput*. Por fim, */api/analysis* retorna os resultados da detecção de anomalias nas métricas, bem como as requisições com falhas.

Para a detecção de anomalias nas métricas, é realizada uma classificação de risco da integridade da aplicação. O algoritmo determina se um método está em baixo risco se estiver abaixo de 50% do valor definido como parâmetro na Anotação. Por exemplo, se foi especificado que o método *getAllUsers* não deve exceder 20% de uso de CPU, resultados abaixo de 10% são classificados como de baixo risco. O risco médio é atribuído a métodos com valores entre 80% e 99% do valor definido na Anotação. Por fim, risco alto é atribuído a métodos que excedem os valores definidos. Essa classificação foi determinada pelos autores do estudo. Assim, todas as métricas classificadas como de alto risco são consideradas anômalas, e o restante é considerado normal. Essa classificação de risco é usada no serviço de *Dashboard* para indicar aos operadores do sistema o *status* das métricas dos métodos anotados. No FOCUS, os *traces* das requisições têm um dos dois estados: normal ou falha. Qualquer método anotado com um código de *status* 4xx ou 5xx é classificado como estado de falha, seguindo o padrão *HTTP*. Portanto, o FOCUS também pode detectar e reportar requisições com falhas.

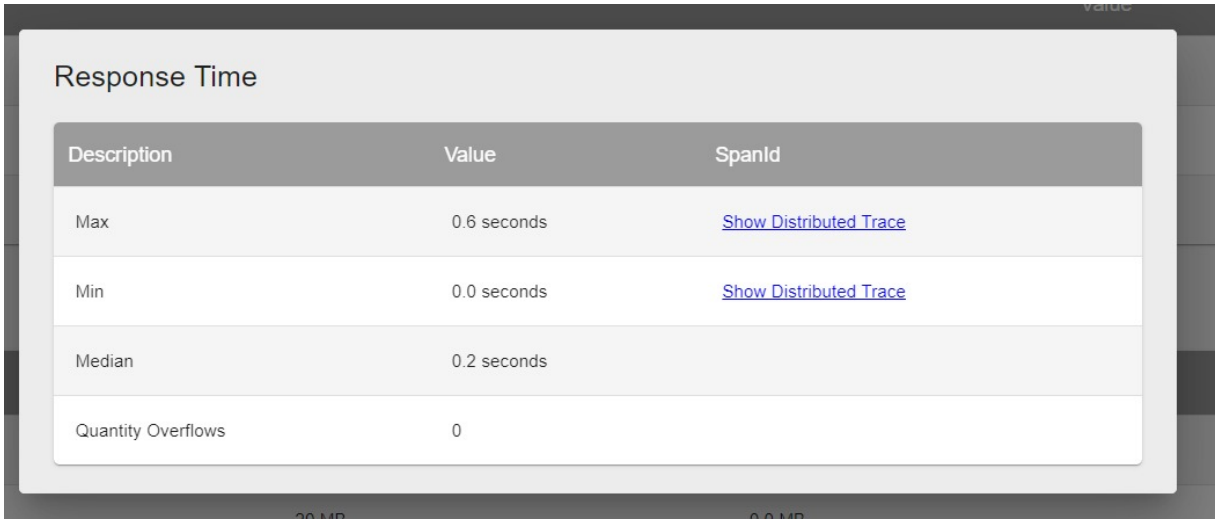
4.5 Focus *Dashboard*

Após a criação do FOCUS *Manager* para expor a fusão de dados e analisar métricas e *traces*, foi desenvolvido um serviço de *dashboard* para facilitar a visualização. O *Dashboard* possui dois fluxos principais: na tela inicial, todos os métodos com a Anotação são listados, juntamente com informações sobre se as métricas são anômalas e se os *traces* possuem falhas.

O primeiro fluxo exibe a classificação de acordo com o risco à integridade da aplicação. As classificações, como mostrado anteriormente, vêm do FOCUS *Manager*. O segundo fluxo corresponde à tela de detalhes, que fornece *insights* sobre as métricas do método anotado, incluindo informações gerais como o número de requisições, erros e a média de *throughput* de dados.

A Figura 3 fornece informações sobre os valores máximos, mínimos, médias e excedentes das métricas, incluindo os *traces* das requisições. Ao clicar no trace, o usuário é direcionado ao *Jaeger* para analisar o rastreamento distribuído, o que facilita a identificação do motivo pelo qual as métricas receberam esses valores naquele momento. Isso destaca a importância da fusão de dados. As métricas nos permitem visualizar a saúde do sistema, e o rastreamento distribuído nos ajuda a encontrar o que causou as métricas excederem os valores definidos.

Figura 3 – Valores de Métricas Específicos



5 RESULTADOS

Foi conduzido experimentos para avaliar nossa solução na identificação de métodos que excedem as métricas pré-definidas na aplicação *Petclinic*, bem como na detecção de falhas.

5.1 Configuração do Experimento

Implantamos a aplicação em um *cluster* usando *Docker Compose* em uma infraestrutura local composta por um processador Intel® Core™ i7-11390H de 11ª geração com 4 núcleos, 2918 MHz, 16 GB de RAM e Windows. Foi executado dois cenários no *Petclinic*, onde modificamos um dos serviços para incluir um método específico que, quando acionado, aumenta significativamente o uso de memória e o tempo de resposta. A execução desse método ocorre

aleatoriamente, gerada por um valor randômico de 1 a 100 no início do serviço. Se o valor estiver entre 1 e 8, o método anômalo é acionado, resultando em aproximadamente 8% de chance de ocorrência.

Essa estratégia foi projetada para simular condições em que falhas inesperadas podem ocorrer, oferecendo uma maneira prática de testar a resiliência do sistema e sua capacidade de detectar anomalias. Os cenários do projeto foram: (*i*) a versão original do *Petclinic*, com as modificações mencionadas, e (*ii*) a versão modificada, incluindo a dependência da biblioteca e o uso da anotação. O objetivo era avaliar se o acionamento do método que sobrecarrega as métricas permitiria que a solução proposta facilitasse a identificação de problemas. Para esse teste, foi utilizado o *JMeter* com 50 usuários e um total de 4 execuções.

5.2 Resultados do Experimento

Os resultados obtidos nos experimentos são apresentados a seguir. No teste de carga, foi observado que, no cenário (*i*), sem a nossa solução, foi muito difícil identificar o serviço anômalo usando apenas o *Jaeger* com a listagem de chamadas, e o método acabou se misturando com as diversas chamadas que ocorreram. No cenário (*ii*), com a nossa solução, o *endpoint* foi configurado com a anotação especificando os valores que não deveriam ser excedidos, conforme mostrado no código fonte 5.2.

```
1 @GetMapping
2 @ObservabilityParam(params = {
3         @Param(key = "cpuUsage", value = "15"),
4         @Param(key = "memory", value = "25"),
5         @Param(key = "responseTime", value = "550"),
6         @Param(key = "throughput", value = "940")
7 })
8 public List<Owner> findAll(){ ... }
```

Além de ser configurada nesse *endpoint*, a anotação foi aplicada a outros serviços com diferentes valores máximos, já que esses valores são escolhidos pelo desenvolvedor. Ao realizar o teste de carga e verificar o painel, foi observado que foram chamados 5 serviços com a anotação, conforme mostrado na Figura 4. Foi decidido examinar os detalhes do *endpoint* destacado, pois ele foi sinalizado com um risco crítico e sabíamos que poderia invocar o método que aumenta o uso de memória.

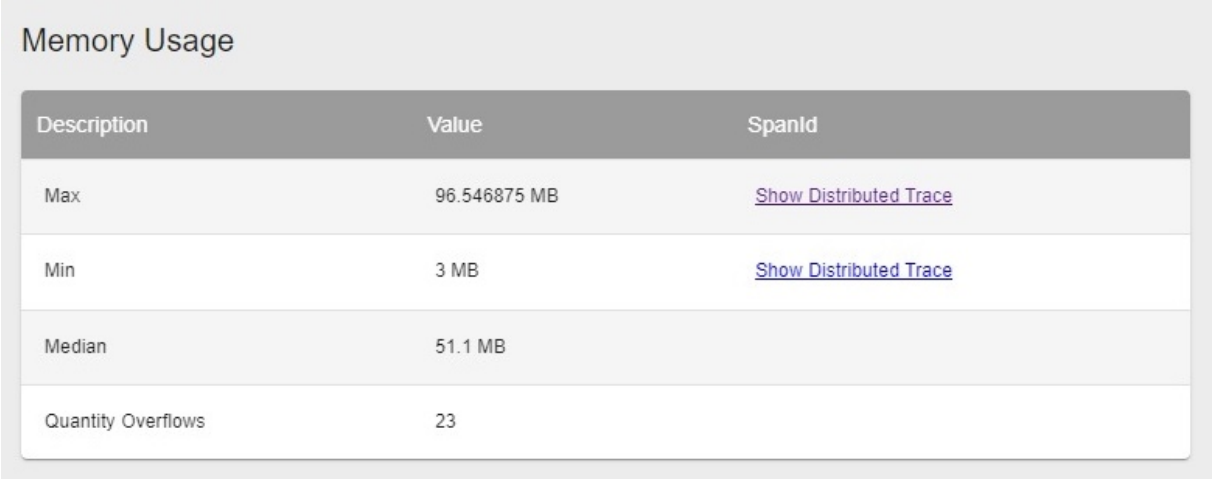
Ao examinar as métricas do serviço, constatou-se que o consumo de memória estava em um nível de risco crítico. A Figura 5 exibe as métricas de uso de memória coletadas desse serviço, incluindo os valores máximos, mínimos, médios e os excessos de quantidade, juntamente com o *trace* correspondente.

Ao examinar ambos os *traces*, é possível compará-los por meio de rastreamento distribuído. Conforme mostrado na Figura 6, um dos *traces* incluiu uma chamada ao método *causeMemoryLeak* (responsável pela anomalia), enquanto o outro, conforme evidenciado na Figura 7, não incluiu essa chamada. Assim, com a fusão de métricas e rastreamento distribuído, a detecção de falhas ou anomalias em microsserviços se torna mais simples.

Além dos testes de anomalia, foram realizados testes para determinar a porcentagem de falhas que o sistema consegue identificar. Para isso, um *endpoint* com a Anotação foi configurado para chamar um serviço inexistente, resultando em uma exceção. O resultado deste teste, ilustrado na Figura 8, mostra que a solução proposta conseguiu identificar 100% das falhas.

Figura 4 – Lista de Serviços Chamados

Figura 5 – Detalhes do Serviço



Como os dados são extraídos da API do *Jaeger*, a solução recupera o status dos *traces*, e todas as solicitações com falhas foram corretamente exibidas no *dashboard*.

5.3 Lições Aprendidas

Profissionais de tecnologia podem usar esta solução para otimizar a manutenção de suas aplicações baseadas em microsserviços, reduzindo a ocorrência de falhas e melhorando a eficiência da detecção de anomalias. Como demonstrado pelos resultados, a integração de técnicas de observabilidade, combinando métricas e rastreamento distribuído, prova ser altamente eficaz. Esta fusão permite que ambas as técnicas trabalhem juntas, fornecendo *insights* valiosos que ajudam a identificar problemas que, devido ao tamanho dessa arquitetura, poderiam de outra forma permanecer ocultos. Ela ajuda a identificar serviços anômalos e a melhorar a estabilidade e o desempenho geral da aplicação. Dessa forma, a questão de pesquisa é respondida.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este documento apresentou a solução FOCUS, que foi desenvolvida usando uma ferramenta de observabilidade OTel para realizar a fusão de rastreamento distribuído com

Figura 6 – Trace com o Método Anômalo

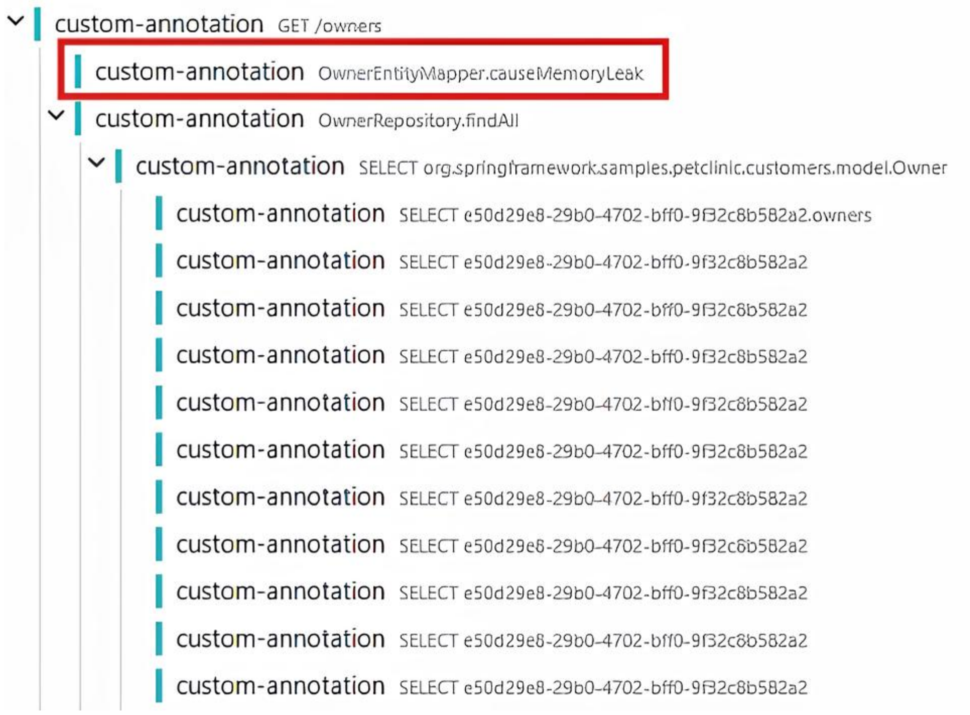
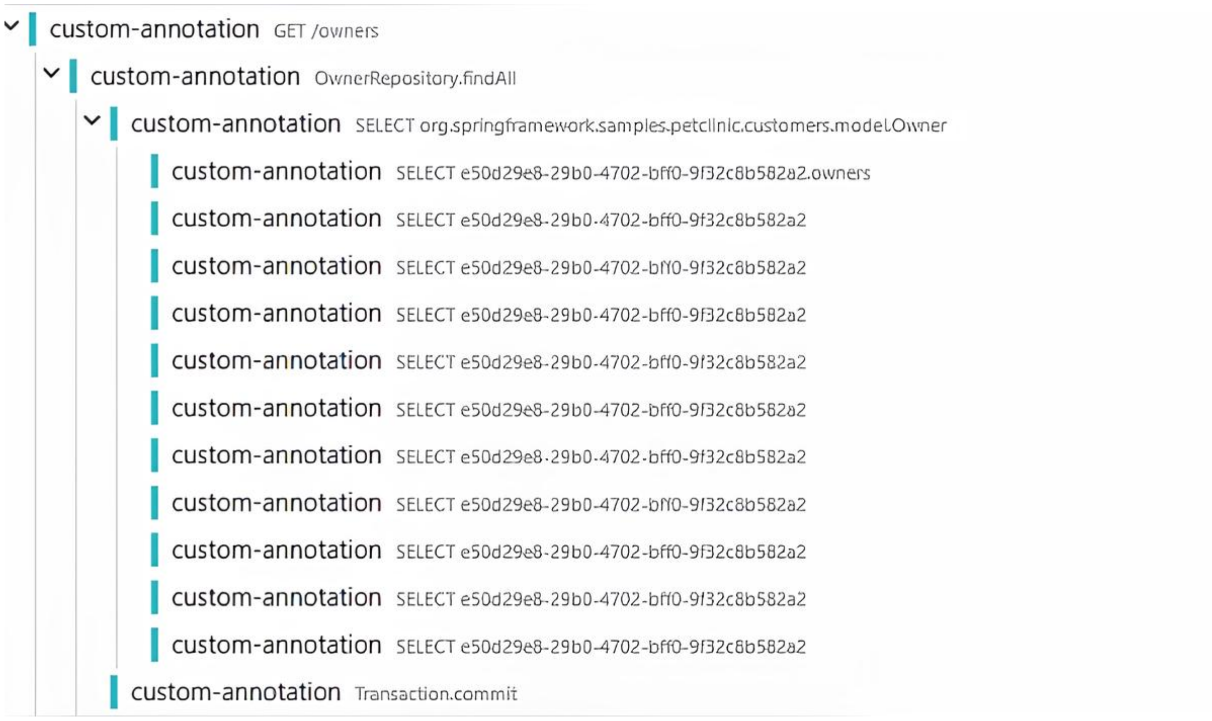
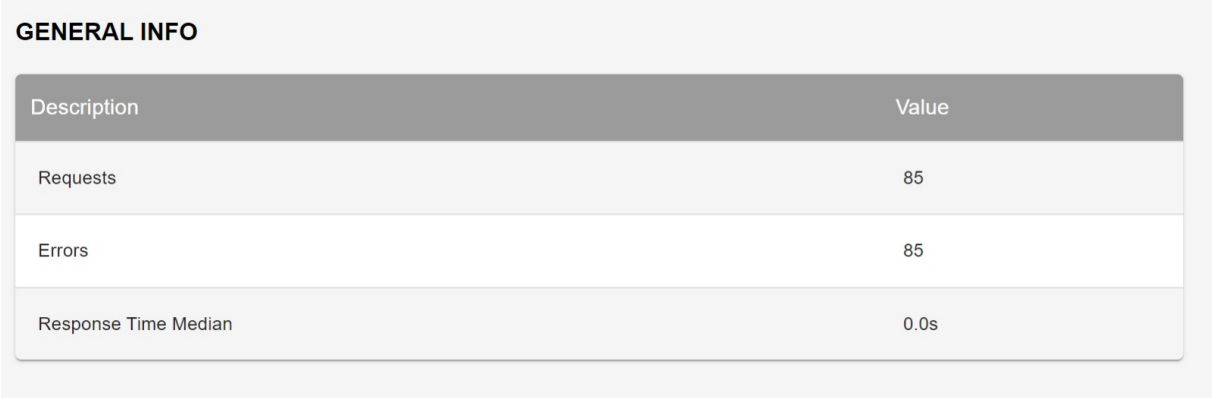


Figura 7 – Trace sem o Método Anômalo



métricas para a detecção de anomalias e falhas. Essa fusão proporcionou uma visão mais ampla da aplicação de microsserviços, ao combinar métricas com rastreamento distribuído para localizar facilmente os rastreamentos em caso de excedência de métricas. Isso resulta em uma análise mais rápida e reduz as chances de falhas, já que valores máximos são definidos para os métodos estudados, e quaisquer valores que ultrapassem esses limites são exibidos no *dashboard* pelo FOCUS. Uma limitação dessa solução é que ela é específica para projetos de microsserviços baseados em *Java*, pois depende de *Annotations*, um recurso exclusivo dessa linguagem. Além

Figura 8 – Porcentagem de Falhas



disso, outra limitação é a necessidade de sincronização da equipe do projeto para selecionar os valores de parâmetros mais apropriados, a fim de evitar falsos positivos na detecção de anomalias.

Para trabalhos futuros, foi planejado expandir a biblioteca FOCUS para suportar outras linguagens de programação além de *Java* e buscar métodos mais eficientes para a detecção de anomalias, bem como aplicar algoritmos de aprendizado de máquina usando os dados coletados pelo OTel para recomendar melhores configurações de métricas.

REFERÊNCIAS

AL-DEBAGY, O.; MARTINEK, P. A comparative review of microservices and monolithic architectures. In: IEEE. **2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)**. [*S. l.*], 2018. p. 000149–000154.

BENTO, A.; CORREIA, J.; FILIPE, R.; ARAUJO, F.; CARDOSO, J. Automated analysis of distributed tracing: Challenges and research directions. **Journal of Grid Computing**, Springer, v. 19, n. 1, p. 9, 2021.

BOTEN, A.; MAJORS, C. **Cloud-Native Observability with OpenTelemetry: Learn to gain visibility into systems by combining tracing, metrics, and logging with OpenTelemetry**. [*S. l.*]: Packt Publishing, 2022. ISBN 9781801071901.

CASSÉ, C.; BERTHOU, P.; OWEZARSKI, P.; JOSSET, S. Using distributed tracing to identify inefficient resources composition in cloud applications. In: IEEE. **2021 IEEE 10th International Conference on Cloud Networking (CloudNet)**. [*S. l.*], 2021. p. 40–47.

CHANDRAMOULI, R. Microservices-based application systems. **NIST Special Publication**, v. 800, n. 204, p. 800–204, 2019.

DRAGONI, N.; GIALLORENZO, S.; LAFUENTE, A. L.; MAZZARA, M.; MONTESI, F.; MUSTAFIN, R.; SAFINA, L. Microservices: yesterday, today, and tomorrow. **Present and ulterior software engineering**, Springer, p. 195–216, 2017.

DRAGONI, N.; LANESE, I.; LARSEN, S. T.; MAZZARA, M.; MUSTAFIN, R.; SAFINA, L. Microservices: How to make your application scale. In: SPRINGER. **International Andrei Ershov Memorial Conference on Perspectives of System Informatics**. [*S. l.*], 2017. p. 95–104.

GARLAN, D. Software architecture. Carnegie Mellon University, 2008.

GARLAN, D.; SHAW, M. An introduction to software architecture. In: **Advances in software engineering and knowledge engineering**. [*S. l.*]: World Scientific, 1993. p. 1–39.

HEINRICH, R.; HOORN, A. V.; KNOCHE, H.; LI, F.; LWAKATARE, L. E.; PAHL, C.; SCHULTE, S.; WETTINGER, J. Performance engineering for microservices: research challenges and directions. In: **Proceedings of the 8th ACM/SPEC on international conference on performance engineering companion**. [*S. l.*: *s. n.*], 2017. p. 223–226.

KALMAN, R. E. On the general theory of control systems. In: **Proceedings First International Conference on Automatic Control, Moscow, USSR**. [*S. l.*: *s. n.*], 1960. p. 481–492.

KARUMURI, S.; SOLLEZA, F.; ZDONIK, S.; TATBUL, N. Towards observability data management at scale. **ACM SIGMOD Record**, ACM New York, NY, USA, v. 49, n. 4, p. 18–23, 2021.

KOSIŃSKA, J.; BALIŚ, B.; KONIECZNY, M.; MALAWSKI, M.; ZIELIŃSKI, S. Towards the observability of cloud-native applications: The overview of the state-of-the-art. **IEEE Access**, IEEE, 2023.

MAJORS, C.; FONG-JONES, L.; MIRANDA, G. **Observability Engineering: Achieving Production Excellence**. [*S. l.*]: O'Reilly Media, Incorporated, 2022. ISBN 9781492076445.

MARIE-MAGDELAINE, N.; AHMED, T.; ASTRUC-AMATO, G. Demonstration of an observability framework for cloud native microservices. In: IEEE. **2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. [*S. l.*], 2019. p. 722–724.

MENDONÇA, N. C.; JAMSHIDI, P.; GARLAN, D.; PAHL, C. Developing self-adaptive microservice systems: Challenges and directions. **IEEE Software**, IEEE, v. 38, n. 2, p. 70–79, 2019.

PAUTASSO, C.; ZIMMERMANN, O.; AMUNDSEN, M.; LEWIS, J.; JOSUTTIS, N. Microservices in practice, part 1: Reality check and service design. **IEEE software**, IEEE Computer Society, v. 34, n. 01, p. 91–98, 2017.

RAEISZADEH, M.; EBRAHIMZADEH, A.; SALEEM, A.; GLITHO, R. H.; EKER, J.; MINI, R. A. Real-time anomaly detection using distributed tracing in microservice cloud applications. In: IEEE. **2023 IEEE 12th International Conference on Cloud Networking (CloudNet)**. [*S. l.*], 2023. p. 36–44.

SINGLETON, A. The economics of microservices. **IEEE Cloud Computing**, IEEE, v. 3, n. 5, p. 16–20, 2016.

THAKUR, A.; CHANDAK, M. A review on opentelemetry and http implementation. **International journal of health sciences**, v. 6, p. 15013–15023, 2022.

THÖNES, J. Microservices. **IEEE software**, IEEE, v. 32, n. 1, p. 116–116, 2015.

TZANETTIS, I.; ANDRONA, C.-M.; ZAFEIROPOULOS, A.; FOTOPOULOU, E.; PAPAVASSILIOU, S. Data fusion of observability signals for assisting orchestration of distributed applications. **Sensors**, MDPI, v. 22, n. 5, p. 2061, 2022.

USMAN, M.; FERLIN, S.; BRUNSTROM, A.; TAHERI, J. A survey on observability of distributed edge & container-based microservices. **IEEE Access**, IEEE, 2022.

WASEEM, M.; LIANG, P.; SHAHIN, M. A systematic mapping study on microservices architecture in devops. **Journal of Systems and Software**, Elsevier, v. 170, p. 110798, 2020.