Universidade Federal de Pernambuco
Centro de Informática

Bacharelado em Ciência da Computação

Análise de soluções de rastreamento *open source* no contexto de aplicações baseadas em microsserviços

Trabalho de Graduação

Matheus de Andrade Lima

Recife
2022

Universidade Federal de Pernambuco

Centro de Informática

Matheus de Andrade Lima

Análise de soluções de rastreamento open source no contexto de aplicações baseadas em microsserviços

Trabalho de Conclusão de Curso apresentado no curso de Bacharelado Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Vinicius Cardoso Garcia*

Recife
2022

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Lima, Matheus de Andrade.
Análise de soluções de rastreamento open source no contexto de aplicações baseadas em microsserviços / Matheus de Andrade Lima. - Recife, 2022.
50 : il., tab.

Orientador(a): Vinicius Cardoso Garcia
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2022.

1. sistemas distribuídos. 2. microsserviços. 3. observabilidade. 4. rastreamento distribuído. I. Garcia, Vinicius Cardoso . (Orientação). II. Título.

000 CDD (22.ed.)

Este trabalho é dedicado aos meus familiares,
amigos e professores.

AGRADECIMENTOS

Primeiramente agradeço a Deus pelos dons que me deu nesta existência que serviram na realização deste projeto.

Agradeço aos meus pais por todo o esforço investido na minha educação.

Agradeço à minha namorada que sempre esteve ao meu lado durante o meu percurso acadêmico.

Aos meus colegas de turma, por compartilharem comigo tantos momentos de descobertas e aprendizado e por todo o companheirismo ao longo deste percurso.

Sou grato pela confiança depositada na minha proposta de projeto pelo meu professor Vinícius Garcia, orientador do meu trabalho. Obrigado por me manter motivado durante todo o processo.

Por último, quero agradecer também à Universidade Federal de Pernambuco e a todo o seu corpo docente.

RESUMO

Devido a ascensão da internet e o crescimento elevado do número de usuários, a engenharia de software foi obrigada a buscar arquiteturas mais resilientes. Uma das soluções encontradas foi criar sistemas pequenos, independentes e com contexto bem definido que se comunicam através da rede, padrão arquitetural que conhecemos como microsserviços. Entretanto, este padrão arquitetural traz consigo uma dificuldade maior em relação a observabilidade do sistema. Entre as dificuldades podemos citar o rastreamento, encontrar o caminho que uma determinada chamada fez dentro do sistema, pois nesta abordagem há vários sistemas interagindo entre si. Sendo assim, o objetivo deste projeto é facilitar a escolha de ferramentas de rastreamento por meio de uma comparação dos dois principais produtos *open source* disponíveis para a plataforma JVM. Além disso, deixar disponível um repositório com um projeto exemplo de forma que sirva de referência para que desenvolvedores possam consultar a configuração e comparar os benefícios de cada ferramenta, fazendo assim a melhor escolha pro seu contexto.

Palavras-chave: sistemas distribuídos, microsserviços, observabilidade, rastreamento distribuído.

ABSTRACT

Due to the rise of the internet and the high growth in the number of users, software engineering was forced to look for more resilient architectures. One of the solutions found was to create small, independent systems with a well-defined context that communicate over the network, an architectural pattern we know as microservices. However, this architectural pattern brings with it a greater difficulty in relation to the observability of the system. Among the difficulties, we can mention the tracing, finding the path that a given request did through the system, because in this approach there are several systems interacting with each other. Therefore, the objective of this project is to facilitate the choice of tracing tools through a comparison of the two main open source products available for the JVM platform. In addition, make available a repository with an example project so that it serves as a reference for developers to consult the configuration and compare the benefits of each tool, thus making the best choice for their context.

Keywords: distributed systems, microservices, observability, distributed tracing.

LISTA DE FIGURAS

Figura 1: Exemplo de aplicação monolítica…….. 14
Figura 2: Exemplo de aplicação em microsserviços…….. 15
Figura 3: Exemplo de rastreamento…….. 20
Figura 4: Exemplo de Dockerfile com configurações pro Jaeger……. 27
Figura 5: Exemplo de Dockerfile com configurações pro Zipkin……..... 27
Figura 6: Exemplo de docker-compose com configurações pro Jaeger……. 28
Figura 7: Exemplo de docker-compose com configurações pro Zipkin…….. 29
Figura 8: Script com comandos direcionados ao Jaeger…….. 30
Figura 9: Script com comandos direcionados ao Zipkin…… 31
Figura 10: Aplicação rodando…… 32
Figura 11: Tela inicial do Jaeger…… 33
Figura 12: Tela inicial do Zipkin……. 33
Figura 13: Arquitetura do Zipkin…… 36
Figura 14: Arquitetura do Jaeger (centralizada)…… 37
Figura 15: Arquitetura do Jaeger (distribuída)…..…… 38
Figura 16: Gráfico de dependências do Jaeger…….. 39
Figura 17: Gráfico de dependências do Zipkin…… 40
Figura 18: Comparação de traces…… 41

LISTA DE SIGLAS

SOA	Arquitetura Orientada a Serviços
APM	Application Performance Monitoring
JVM	Java Virtual Machine
API	Application Programming Interface
CNCF	Cloud Native Computing Foundation

SUMÁRIO

1. INTRODUÇÃO **9**

1.1 CONTEXTO 9
1.2 OBJETIVOS 10
1.3 ORGANIZAÇÃO DO TRABALHO 11

2. CONCEITOS **12**
2.1 HISTÓRIA DA ARQUITETURA DE SOFTWARE 12
2.2 ARQUITETURA DE MICROSSERVIÇOS 14
2.3 OBSERVABILIDADE 17
2.4 SÍNTESE DO CAPÍTULO 21

3. FERRAMENTAS E CONFIGURAÇÃO DO PROJETO **22**
3.1 FERRAMENTAS DE APOIO 22
3.2 FERRAMENTAS DE RASTREAMENTO 24
3.3 IMPLEMENTAÇÃO 25
3.4 MÉTRICAS DE COMPARAÇÃO 34
3.5 SÍNTESE DO CAPÍTULO 34

4. DISCUSSÃO COMPARATIVA **36**
4.1 ARQUITETURA 36
4.2 FUNCIONALIDADES 38
4.3 COMUNIDADE E SUPORTE 41
4.4 SÍNTESE DO CAPÍTULO 42

5. CONCLUSÃO E TRABALHOS FUTUROS **44**
5.1 CONCLUSÃO 44
5.2 LIMITAÇÕES 45
5.3 TRABALHOS FUTUROS 45

6. REFERÊNCIAS **46**

1. INTRODUÇÃO

Este primeiro capítulo busca trazer um entendimento geral do problema sobre o qual está sendo realizado o trabalho, teremos uma breve passagem pela história do mesmo e tudo que será abordado.

Além disso, também se faz presente os objetivos que desejamos alcançar com a realização deste trabalho e uma visão geral do conteúdo de cada capítulo para que o leitor tenha ciência de como o trabalho está estruturado.

1.1 CONTEXTO

Devido a ascensão da internet e o crescimento elevado do número de usuários, a engenharia de software foi obrigada a buscar soluções para sistemas mais confiáveis. Ao longo do tempo diversos padrões arquiteturais foram criados com o intuito de sanar este problema, por exemplo: cliente-servidor, SOA, orientado a eventos, microsserviços, entre outros [1].

O padrão arquitetural de microsserviços, consiste em pequenos sistemas, independentes e fracamente acoplados [2]. Criar sistemas com arquitetura baseada em microsserviços é muito comum no mercado de tecnologia nos dias de hoje [1]. Dado que estamos construindo sistemas cada vez mais complexos, há uma quantidade significativa de vantagens de usar tal abordagem, tais como: conseguir escalar cada serviço de forma independente, reduzir o tempo de inatividade por meio do isolamento de falhas, fácil manutenibilidade, ganhos de produtividade pela segmentação do time, entre outros [2].

Entretanto, este padrão arquitetural traz consigo uma dificuldade maior em relação ao monitoramento e observabilidade do sistema [3]. Dentre as principais dificuldades podemos citar o rastreamento, que podemos descrever como encontrar o caminho que uma determinada chamada fez dentro do sistema, pois nesta abordagem há vários sistemas interagindo entre si [4]. O rastreamento nos diz se o sistema está funcionando conforme foi arquitetado.

Além disso, o rastreamento também nos ajuda a identificar componentes do sistema com mau funcionamento. Podemos descobrir, por exemplo, que uma lentidão na resposta está associada a uma busca muito demorada na base de dados. Dito isto, ter tal informação é de extrema importância para fins de otimização.

Com o intuito de ajudar no esclarecimento desse problema existem as ferramentas de APM e rastreamento, das quais podemos citar New Relic, Dynatrace, AppDynamics, etc [5]. As ferramentas citadas são bem avaliadas pela comunidade, conforme pode ser observado em fóruns como Reddit e Gartner [6, 7 e 8], e se bem utilizadas resolvem os problemas, contudo um obstáculo comum para empresas pequenas ou desenvolvedores é o alto valor das licenças cobradas pelos dominantes do mercado [9].

Felizmente há opções de ferramentas de rastreamento *open source* disponíveis. No entanto, escolher qual a melhor para o seu projeto é uma tarefa complicada pois há muitos aspectos a serem analisados, por exemplo: compatibilidade com a linguagem utilizada, usabilidade da UI, robustez da plataforma, modelo de implantação no ambiente de produção, entre outros. Ademais, os aspectos citados só são plenamente verificados com o uso da ferramenta, o que consome bastante tempo da equipe de desenvolvimento.

1.2 OBJETIVOS

Dito isso, o principal objetivo deste projeto é facilitar a escolha de ferramentas de rastreamento open source disponíveis para a plataforma JVM por meio de uma comparação prática. Desta forma, pretende-se conduzir um estudo com base nos seguintes pontos:

- Arquitetura, como a ferramenta pode ser implantada para uso dentro de uma organização.
- Funcionalidades, qual ferramenta apresenta o melhor conjunto de soluções.

- Suporte e comunidade, como é a aceitação na comunidade *open source.*

Além disso, este trabalho também visa analisar as ferramentas Jaeger e Zipkin, verificar a integração dessas ferramentas em uma solução completa de rastreamento e deixar disponível um repositório com um projeto exemplo de forma que sirva de referência a desenvolvedores. Sendo assim, haverá uma facilidade maior para que se possa consultar a configuração e comparar os benefícios de cada ferramenta, ajudando assim, a fazer melhor escolha pro contexto.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte forma:

O Capítulo 1 apresenta uma contextualização do problema juntamente com os objetivos almejados.

O Capítulo 2 vem com a exposição dos principais conceitos necessários para o entendimento do trabalho como um todo, conceitos aos quais não seria possível realizar o trabalho sem um bom embasamento.

Já o Capítulo 3, por sua vez, é voltado para para o entendimento das ferramentas utilizadas na realização do projeto, tanto as ferramentas que são usadas para a comparação quanto as ferramentas de apoio. Além disso, detalha como foi a configuração do projeto base e como será abordada a comparação no capítulo seguinte.

O quarto capítulo, tem como objetivo expor os detalhes de cada ferramenta, sua utilização, pontos positivos e negativos de cada critério abordado como métrica de comparação.

Por fim, o Capítulo 6 apresenta as conclusões chegadas pelo autor durante o desenvolvimento do trabalho e um levantamento de possíveis trabalhos futuros.

2. CONCEITOS

Neste capítulo serão apresentados os principais conceitos que dão fundamento a este trabalho. Primeiramente, vamos entender um pouco sobre a história da arquitetura de software para que possamos entender como chegamos no cenário atual.

Em seguida, estudaremos os conceitos de microsserviços e observabilidade, esses dois assuntos são fundamentais para a compreensão do problema do rastreamento distribuído. As ferramentas escolhidas para comparação têm como função o rastreamento distribuído dos sistemas.

2.1 HISTÓRIA DA ARQUITETURA DE SOFTWARE

Como forma de entender bem o presente, devemos olhar e estudar a história. Isso é importante não apenas em computação mas em toda ciência. O filósofo George Santayana disse, “Quem não aprende a história está condenado a repeti-la” [42], mas nós não queremos repetir os menos erros ou repetir o mesmo processo para resolver um problema já resolvido.

Uma decisão muito importante na concepção de um software é escolher um estilo arquitetural. De acordo com Herberto Graça: Os estilos arquiteturais nos dizem, em linhas muito amplas, como organizar nosso código. É o nível mais alto de granularidade e específica camadas, módulos de alto nível da aplicação e como esses módulos e camadas interagem entre si, as relações entre eles [33]. São exemplos de estilos arquiteturais:

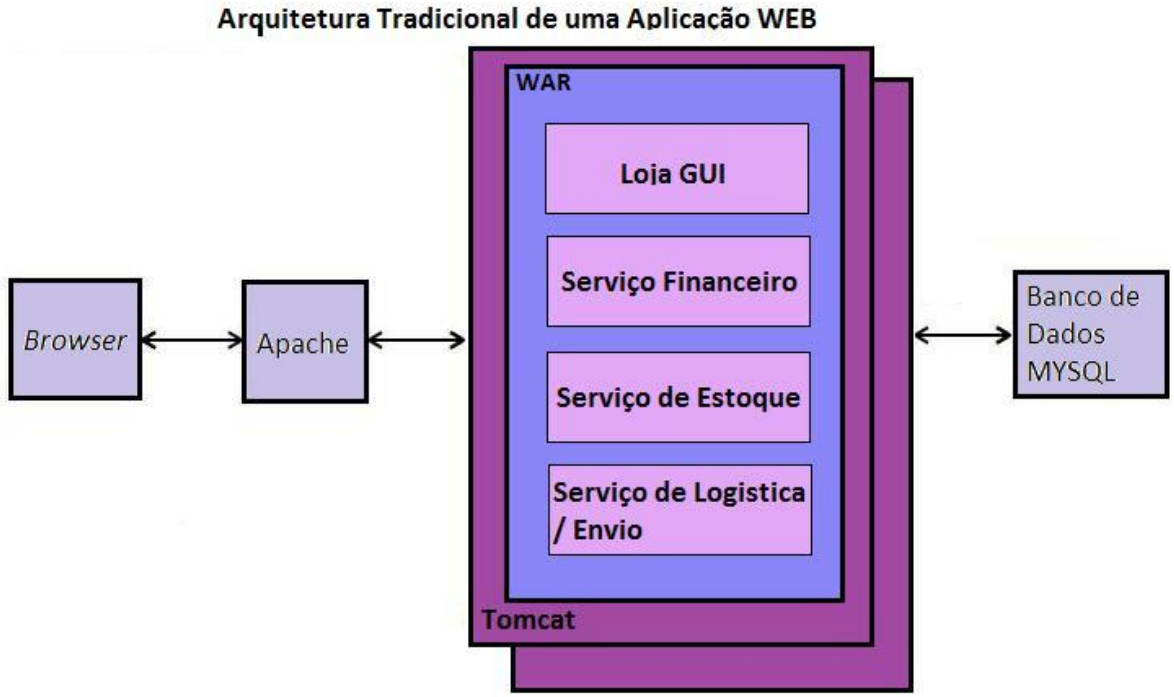
- Component-based
- Monolithic application
- Layered
- Pipes and filters
- Event-driven
- Publish-subscribe
- Plug-ins

- Client-server
- Service-oriented

Construir um monolito sempre foi o estilo arquitetural padrão e mais simples. Ter um estilo de arquitetura monolítico significa simplesmente que todo o código da aplicação está implantado e executado como um único processo em um único servidor. Por apresentar todo o código da aplicação em um único processo, começam a surgir problemas caso algumas das seguintes características são necessárias [34]:

- Escalabilidade independente de diferentes componentes do domínio;
- Diferentes componentes ou módulos a serem escritos em diferentes linguagens de programação;
- Implantação independente, talvez porque tenhamos uma taxa de lançamento maior do que a pipeline de Implantação pode suportar para uma base de código, fazendo com que a Implantação de uma versão seja lenta porque ela precisa aguardar a implementação de outras versões ou até mesmo fazendo com que a fila de Implantação cresça mais rápido do que é consumido.

Figura 1: Exemplo de aplicação monolítica



Fonte: Docplayer (2021)

Com a finalidade de resolver esses problemas começaram a surgir estilos arquiteturais com aplicações segregadas, primeiramente SOA e posteriormente microsserviços. Na próxima seção vamos entender sobre o estilo arquitetural de microsserviços, o qual é muito utilizado atualmente e de extrema importância para a continuidade deste trabalho.

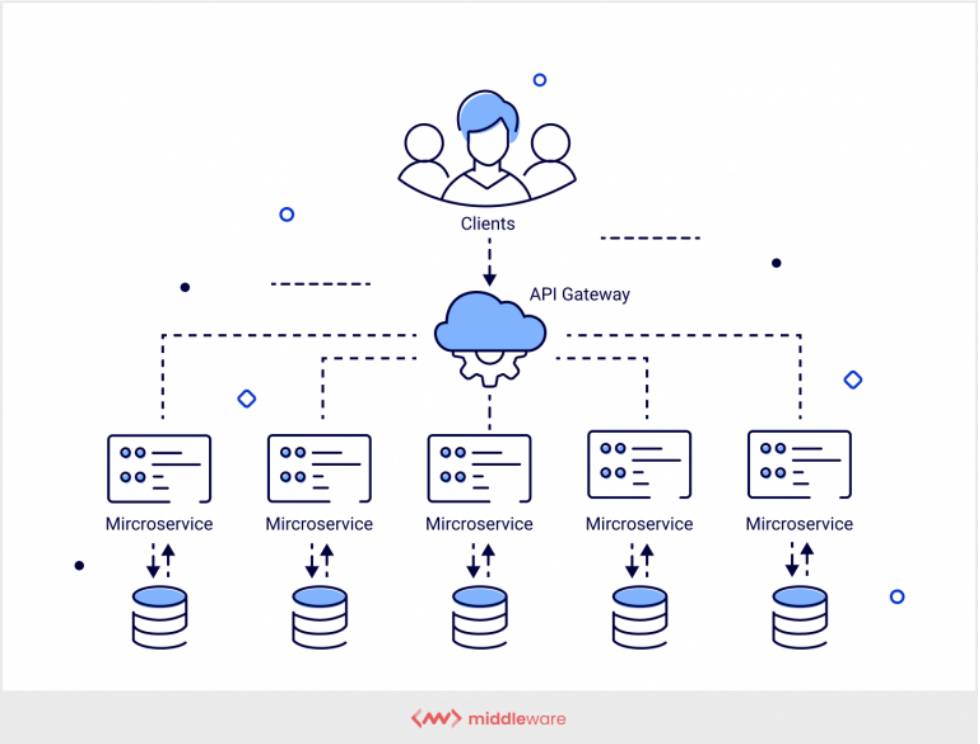
2.1 ARQUITETURA DE MICROSSERVIÇOS

De acordo com uma pesquisa direcionada a pessoas que trabalham na indústria de software, a arquitetura de microsserviços é uma das mais utilizadas nas aplicações criadas atualmente [10]. Vamos entender como chegamos neste cenário olhando os principais benefícios de aderir a este estilo arquitetural.

Criar uma aplicação com microsserviços significa dividir sua aplicação em aplicações menores e com contexto bem definido. Essas pequenas aplicações, também chamadas de micro serviço, devem ser capazes de funcionar e se comunicar de forma independente através da rede [11].

Na Figura abaixo podemos ver como uma aplicação com a arquitetura de microsserviços funciona. O uso de um API gateway, que serve como uma porta de entrada única para toda aplicação é uma característica comum, mas não obrigatória. Ademais, cada serviço tem sua própria função dentro da aplicação, e quando necessário, cada serviço também tem seu próprio meio de armazenamento de dados, evitando assim um ponto único de falha [11].

Figura 2: Exemplo de aplicação em microsserviços



Fonte: Middleware (2021)

A lista de benefícios da utilização de microsserviços é extensa, veremos aqui as principais, de acordo com [11]:

- **Escalabilidade facilitada -** Já que temos sistemas independentes, o time de desenvolvimento pode trabalhar de forma paralela nas demandas de cada micro serviço. Com isso, a equipe tem liberdade para estudar e identificar a necessidade de escalabilidade de cada um desses serviços.

- **Implantações mais rápidas e isoladas -** Um dos requisitos no contexto de independência é que cada micro serviço possa ser implantado de forma

também independente das outras aplicações. Isso faz com que tenhamos uma redução de custos e tempo na resolução de problemas.

- **Agnóstico a tecnologia -** Cada micro serviço tem uma função dentro do sistema e cada aplicação pode utilizar a tecnologia que melhor resolve o problema do seu contexto. Isso é possível devido à independência de cada aplicação.

- **Isolamento de falhas melhorado -** Exceto quando temos dependências entre micro serviços, cada aplicação está isolada de eventuais falhas que possam ocorrer em outros microsserviços do sistema.

Os benefícios citados acima revelam o motivo da gigante adoção do estilo arquitetural aqui discutido. Porém, há também diversos desafios que exigem muita maturidade do time que opta por desenvolver com o padrão arquitetural de microsserviços.

Dentre os quais podemos citar, segundo [13]:

- Consistência de dados: Primeiro, pode haver redundância nos armazenamentos de dados, com o mesmo item de dados aparecendo em vários lugares. Por exemplo, os dados podem ser armazenados como parte de uma transação e, em seguida, armazenados em outro lugar para análises, relatórios ou arquivamentos. Dados duplicados ou particionados podem levar a problemas de integridade e consistência de dados.
- Testes: São muito mais complexos em um ambiente de microsserviços devido aos diferentes serviços, integração complexa e interdependências. Outro ponto é que a equipe precisa escrever muitos mocks, mesmo para testar pequenas partes de código [36].
- Falhas de comunicação: Sistemas distribuídos devem ser construídos de forma que eventuais falhas não derrubem a aplicação. Portanto, os desenvolvedores precisam conhecer todos os modos de falha e ter backups caso ocorra alguma falha.

- Rastreamento distribuído: Encontrar os pontos de falha e gargalos é difícil, caro e demorado com microsserviços. Além disso, na maioria dos casos, os dados de falha não são propagados de maneira clara dentro dos microsserviços. E este ponto é o principal aspecto de estudo deste trabalho.

Como podemos observar, adotar uma arquitetura baseada em microsserviços tem muitos ganhos, mas há também inúmeros desafios que exigem muita maturidade da equipe de desenvolvimento. A escolha por usá-la deveria passar por uma minuciosa análise dos requisitos do produto, conhecimento do time, tempo e orçamento disponível para execução.

2.2 OBSERVABILIDADE

De acordo com [14], observabilidade é a habilidade de medir os estados internos de um sistema examinando suas saídas. Que pode ser entendido como coletar dados dos sistemas e examiná-los com o intuito de entender o seu funcionamento.

Algumas perguntas pertinentes sobre o funcionamento de um sistema são, segundo [15]:

- Por quais serviços uma requisição passou e onde estavam os gargalos de desempenho?
- Por que a requisição falhou?
- Como a execução da requisição foi diferente do comportamento esperado do sistema?
- Quantas requisições foram recebidas em um determinado intervalo de tempo?

Para responder a estas perguntas, existem três pilares levados em consideração:

- Logs: Registros imutáveis feitos para identificar o comportamento em um sistema e fornecer informações sobre o comportamento do sistema quando as coisas deram errado. É altamente recomendável ingerir logs de maneira

estruturada, como no formato JSON, para que os sistemas de visualização de log possam indexar automaticamente e tornar os logs fácil de recuperar [15].

- Métricas: São contagens ou medições que são agregadas ao longo de um período de tempo. As métricas informarão, por exemplo, a quantidade de memória usada por um método ou quantas solicitações um serviço processa por segundo e etc [15].
- Traces: Para uma transação ou solicitação individual, um único rastreamento exibe a operação conforme ela se move de um nó para outro em um sistema distribuído [15].

A observabilidade depende da telemetria que é obtida através da instrumentação para obter informações sobre o estado do sistema. Nesses ambientes modernos, todos os componentes de hardware, software, infraestrutura geram registros de todas as atividades. O objetivo da observabilidade é entender o que está acontecendo em todos esses ambientes e entre as tecnologias, para que possamos detectar e resolver problemas para manter os sistemas eficientes e confiáveis [35].

Muitas organizações também adotam uma solução de observabilidade para ajudá-las a detectar e analisar a importância de eventos para suas operações, ciclos de vida de desenvolvimento de software, segurança de aplicativos e experiências do usuário final [15].

A observabilidade tornou-se mais crítica nos últimos anos, à medida que os ambientes nativos da nuvem se tornaram mais complexos e as possíveis causas de uma falha ou anomalia tornaram-se mais difíceis de identificar. À medida que as equipes começam a coletar e trabalhar com dados de observabilidade, elas também percebem seus benefícios para os negócios, não apenas para a TI [35].

Neste trabalho, vamos focar na parte de rastreamento distribuído, o qual usa os traces obtidos por meio da observabilidade do sistema.

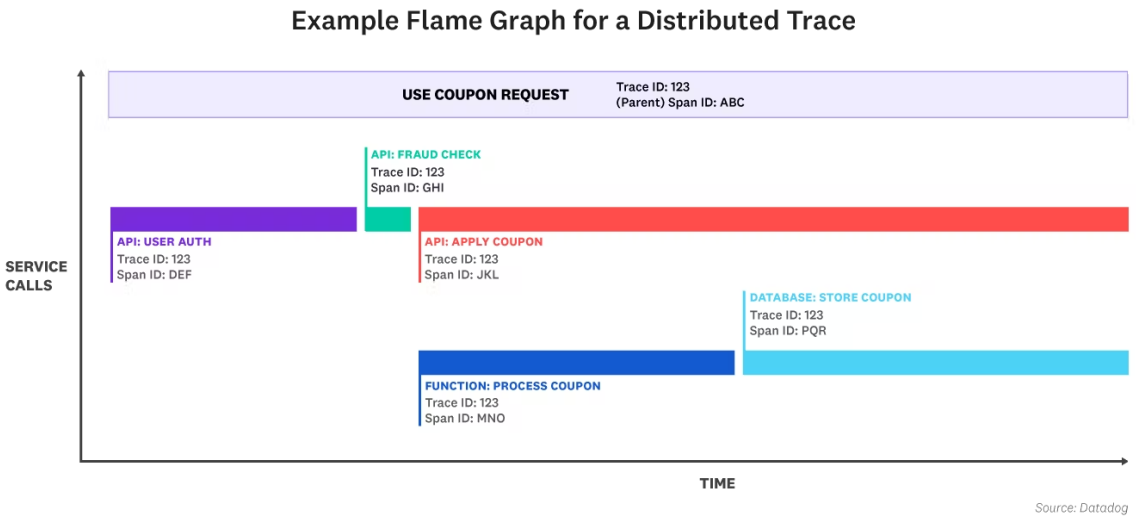
RASTREAMENTO DISTRIBUÍDO

Imagine que você é um desenvolvedor backend e ao chegar para trabalhar na segunda-feira se depara com a notícia de que algumas funcionalidades dos sistema não estão funcionando. Neste momento você começa a fazer troubleshoot nos logs de todos os serviços do sistema na tentativa de achar o ponto em que o fluxo está travando. Isso é uma tarefa que demanda bastante tempo, em sistemas que rodam dezenas de serviços é totalmente inviável fazer troubleshoot dessa forma. Para nos ajudar nessa situação crítica podemos nos apoiar no rastreamento distribuído de requisições.

O rastreamento de requisições não é algo novo na indústria de software. A ideia é saber o comportamento de uma requisição dentro do sistema, desde a interface de entrada até a camada de saída [16]. O rastreamento permite que você entre em detalhes de requisições específicas para determinar quais componentes causam erros no sistema, monitorar o fluxo pelos módulos e encontrar gargalos de desempenho [15].

Com a adoção de microsserviços, o rastreamento de requisições não é mais uma tarefa simples, agora se faz necessário fazer um rastreamento distribuído da requisição em questão [17]. Isso é possível por meio da etiquetagem de requisições com um identificador único que é passado em todo o percurso que a requisição faz dentro do sistema. A imagem abaixo mostra um exemplo de rastreamento.

Figura 3: Exemplo de rastreamento



Fonte: Datadoghq (2022)

Cada passo do percurso gera *traces,* que são nada mais do que logs automatizados com a informações do passo atual. Esses traces, por sua vez, são agregados por alguma ferramenta de rastreamento e disponibilizados através de uma interface gráfica para visualização e análise [17].

Dado isso, o rastreamento distribuído traz diversos benefícios, segundo [39]:

- Melhor entendimento das relações entre os microsserviços - Ao visualizar rastreamentos distribuídos, os desenvolvedores podem entender as relações de causa e efeito entre os serviços e otimizar seu desempenho. Por exemplo, visualizar um intervalo gerado por uma chamada de banco de dados pode revelar que adicionar uma nova entrada de banco de dados causa latência em um serviço.

- Avaliar as ações específicas do usuário - O rastreamento distribuído ajuda a medir o tempo necessário para concluir as principais ações do usuário, como a compra de um item. Os rastreamentos podem ajudar a identificar gargalos de back-end e erros que estão prejudicando a experiência do usuário.

- Melhora a colaboração e a produtividade - Em arquiteturas de microsserviços, equipes diferentes podem possuir os serviços envolvidos na conclusão de uma solicitação. O rastreamento distribuído deixa claro onde ocorreu um erro e qual equipe é responsável por corrigi-lo.

2.2 SÍNTESE DO CAPÍTULO

Neste capítulo foram apresentados os conceitos e fundamentos teóricos referentes à arquitetura de microsserviços, observabilidade e rastreamento distribuído. Esses conceitos são de extrema importância para o entendimento da continuidade do trabalho e projeto desenvolvido.

No próximo capítulo vamos conhecer quais ferramentas foram escolhidas para ajudar na fabricação e configuração do projeto. Com isso, falaremos de ferramentas muito utilizadas no mercado: spring boot, docker e opentelemetry. Além disso, deixaremos detalhado quais serão as métricas de comparação das ferramentas.

3. FERRAMENTAS E CONFIGURAÇÃO DO PROJETO

Este capítulo mostra, primeiramente, quais ferramentas de apoio foram utilizadas e qual é a sua utilidade dentro do projeto. Imediatamente, veremos a escolha das ferramentas de rastreamento usadas para comparação no projeto.

Posteriormente, está escrito como foi a configuração e implementação do projeto com todas essas ferramentas juntas, aqui não está presente a comparação em si, que fica no próximo capítulo. Por último, olharemos para as métricas de comparação das ferramentas que serão debatidas no capítulo seguinte.

3.1 FERRAMENTAS DE APOIO

Com o intuito de facilitar a execução do projeto, algumas ferramentas foram utilizadas, vamos descrever brevemente qual o objetivo de uso de cada uma delas.

DOCKER

Antes de pensarmos em saber o que é como funciona o docker, precisamos definir seu principal componente, o container. Segundo a Docker, um container é um software que empacota o código de uma aplicação e todas as suas dependências para que seja executada de forma rápida e confiável de diversos ambientes de computação [18].

O docker, por sua vez, é uma plataforma que facilita a criação e administração de containers docker [19]. Esses containers, que nada mais são do que o código da aplicação empacotado em execução, são criados a partir de imagens docker [18]. As imagens são criadas a partir de arquivos *Dockerfile*, exemplificados nas figuras 2 e 3.

No projeto exemplo utilizaremos o docker para criar os containers contendo cada micro serviço da aplicação e outro container contendo a ferramenta de

rastreamento: Jaeger e Zipkin. Assim, conseguimos simular um ambiente de execução local completo para testes.

SPRING BOOT

O Spring boot é um framework desenvolvido para a plataforma java que tem por objetivo facilitar o desenvolvimento de aplicações para este ecossistema por meio da abstração da complexidade de configuração e disponibilização de componentes de uso geral [20].

Dentre esses componentes úteis podemos citar: servidor web, bibliotecas de integração com banco de dados, integração com serviços de computação em nuvem, funcionalidades de segurança, entre outros. Spring boot é um framework completo e bastante utilizado pela indústria de desenvolvimento de software [21].

No projeto utilizaremos o spring boot auxiliar na criação e configuração de cada micro serviço java que compõe a aplicação de exemplo. Serão três aplicações cada uma com sua função dentro do sistema.

OPENTELEMETRY

OpenTelemetry é um framework de observabilidade *open source* que nasceu da necessidade de existir uma ferramenta única com todas as funcionalidades necessárias à observabilidade: logs, métricas e rastreamento de requisições [22].

Além disso, outra importante característica é não ser restritivo a uma linguagem específica, a ferramenta pode ser utilizada para aplicações escritas em Java, Python, JavaScript e etc. Existem aplicações, chamadas de agentes, específico para cada linguagem, que fazem a coleta de métricas, traces e logs de forma automatizada [23].

O OpenTelemetry oferece ainda uma forma de exportação dessas métricas de forma agnóstica ao provedor de armazenamento e análise visual dos dados. No

caso de rastreamento esse provedor pode ser: Jaeger ou Zipkin [23]. No projeto utilizaremos esta ferramenta com o agente injetado na JVM para que os traces coletados sejam enviados para o Jaeger ou Zipkin.

3.2 FERRAMENTAS DE RASTREAMENTO

Ao analisar as ferramentas *open source* para rastreamento distribuído disponíveis no mercado é possível notar a dominância de duas, de acordo com número de interações com o repositório oficial de cada uma delas: Jaeger e Zipkin. As duas ferramentas apresentam propostas semelhantes, mas foram desenvolvidas em épocas diferentes.

Zipkin é uma versão *open source* do Dapper do Google que foi desenvolvida pelo Twitter. Em sua essência, o Zipkin é uma aplicação escrita em Java que fornece o serviço de armazenamento e visualização de traces. Como opções de armazenamento para manter os dados registrados, há integração com banco de dados em memória, MySQL, Cassandra e Elasticsearch [24].

Já o Jaeger foi criado pela Uber e foi escrito em Go. Além do conjunto de recursos do Zipkin, o Jaeger também fornece alguns recursos adicionais como: amostragem dinâmica, que é uma forma mais inteligente de colher os traces; uma API REST, que serve para fazer consulta aos dados dos seus traces de forma programática; e suporte para armazenamento em memória Cassandra e Elasticsearch [24].

A motivação para escolha de Jaeger e Zipkin se deu por serem as ferramentas *open source* mais relevantes para rastreamento de acordo com as interações no github, quantidade de posts escritos em blogs na comunidade sobre cada ferramenta e recomendações de uso da comunidade. Entre as ferramentas pesquisadas estão também: Uptrace, Grafana Tempo e Signos, porém nenhuma delas se mostrou tão relevante dentro dos aspectos já citados, quanto as escolhidas [29, 30, 38, 39, 40, 41, 42].

3.3 IMPLEMENTAÇÃO

Nesta seção vamos ver o passo a passo para a criação do projeto exemplo utilizando todas as ferramentas e conceitos discutidos até o presente momento. O projeto a ser desenvolvido aqui é um dos objetivos deste trabalho e permitirá que desenvolvedores tenham uma fonte de conhecimento para comparação e consulta durante a escolha da ferramenta de rastreamento mais adequada ao seu projeto.

ESCOLHA DO PROJETO BASE

Como mencionado anteriormente, o projeto base consiste de uma aplicação que faz uso da arquitetura de microsserviços. Com o intuito de agilizar o desenvolvimento, usaremos uma aplicação criada por terceiros que se encontra disponível no github com uma licença que permite o uso. A aplicação base contém três micro serviços que se chamam: *name-generator-service*, *animal-generator-service* e *scientist-generator-service*.

A missão é gerar nomes aleatórios concatenando nomes de cientistas famosos a nomes de animais. Cada micro serviço tem sua função dentro de um sistema e os microsserviços *scientist-generator-service* e *animal-generator-service* apenas expõem uma interface que retorna um nome aleatório de um cientista e animal, respectivamente.

Já o *name-generator-service,* expõe uma interface que ao ser invocada, acessa os micro serviços *scientist-generator-service* e *animal-generator-service,* obtendo assim o nome de um cientista e de um animal. Feito isso, os nomes são concatenados e um temos assim um nome aleatório gerado, como por exemplo: *alan-turing-canaan-dog*.

Apesar de simples, a aplicação explora um cenário de chamadas síncronas a outros serviços do sistema, aspecto muito comum em uma aplicação distribuída. Agora teremos que fazer a configuração do docker para fazer a geração de imagens

docker a partir do nosso código. O código base usado está disponível neste repositório: https://bit.ly/3BmtmBC.

CONFIGURAÇÃO DO DOCKER E OPENTELEMETRY

Antes de começarmos esta fase, devemos nos certificar de que nossa aplicação Java está compilando e executando corretamente. Dito isto, vamos criar um *Dockerfile*, que é um arquivo usado para definir como o docker irá criar a nossa imagem, que irá conter o código desenvolvido na nossa aplicação, e futuramente será estará rodando no container.

Na criação do *Dockerfile* devemos indicar a partir de qual imagem queremos criar a nossa, já que estamos usando java, devemos usar uma imagem que tenha a JVM disponível para rodarmos nossa aplicação. Depois, devemos copiar o arquivo *.jar*, o qual contém as classes java compactadas geradas na compilação do programa, para dentro da imagem.

Agora só nos resta configurar o OpenTelemetry e indicar como a imagem deverá rodar a nossa aplicação. A documentação oficial do OpenTelemetry recomenda usar o agente específico da linguagem, por isso estamos copiando também o agente para dentro da nossa imagem. Por último, devemos indicar como a imagem irá rodar nossa aplicação quando o container for criado e passar as as configurações que o agente deve usar para que mande o traces para o rastreador que quisermos.

As imagens abaixo mostram como ficaram os arquivos com configuração voltada a cada rastreador, Jaeger e Zipkin.

Figura 4: Exemplo de Dockerfile com configurações pro Jaeger

```
FROM openjdk:8-jdk-alpine

ARG JAR_FILE=target/demo-*.jar
COPY ${JAR_FILE} /app/bin/app.jar

COPY opentelemetry-javaagent.jar /app/bin

CMD java -Dotel.traces.exporter=jaeger \
         -Dotel.exporter.jaeger.endpoint=http://jaeger:14250/ \
         -Dotel.service.name=name-generator-service \
         -Dapplication.home=/app/bin/ \
         -Dapplication.name=name-generator-service \
         -javaagent:/app/bin/opentelemetry-javaagent.jar \
         -jar \
         /app/bin/app.jar
```

Fonte: Autor (2022)

Figura 5: Exemplo de Dockerfile com configurações pro Zipkin

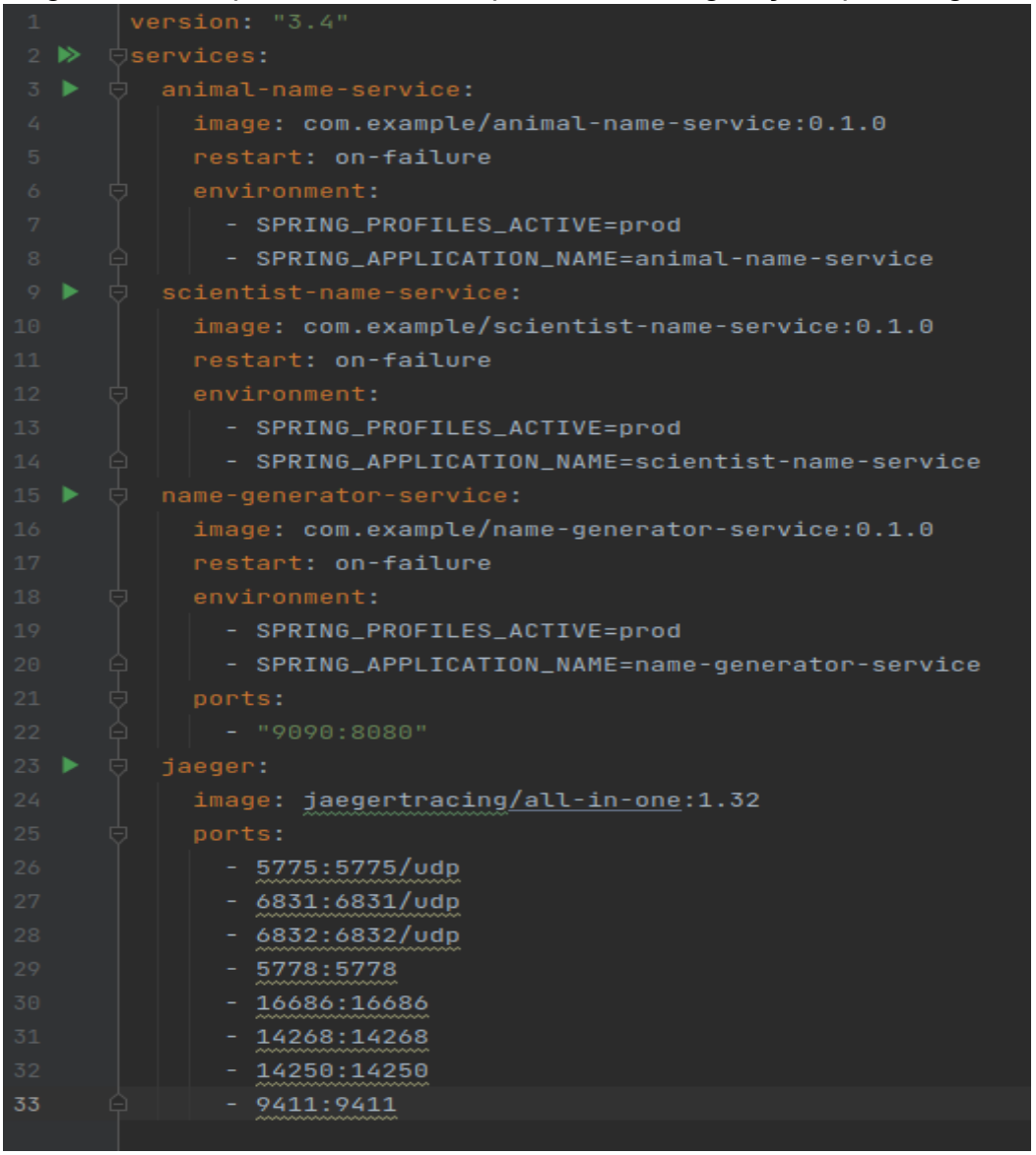
```
FROM openjdk:8-jdk-alpine

ARG JAR_FILE=target/demo-*.jar
COPY ${JAR_FILE} /app/bin/app.jar

COPY opentelemetry-javaagent.jar /app/bin

CMD java -Dotel.traces.exporter=zipkin \
         -Dotel.exporter.zipkin.endpoint=http://zipkin:9411/api/v2/spans \
         -Dotel.service.name=name-generator-service \
         -Dapplication.home=/app/bin/ \
         -Dapplication.name=name-generator-service \
         -javaagent:/app/bin/opentelemetry-javaagent.jar \
         -jar \
         /app/bin/app.jar
```

Fonte: Autor (2022)

Agora temos os arquivos de geração de imagem da nossa aplicação direcionado a cada rastreador, agora precisamos criar um arquivo que nos ajude a rodar todas imagens para criar os containers ao mesmo tempo para que possamos deixar nossa aplicação disponível de maneira mais rápida.

Para fazer isso, vamos criar um arquivo de configuração do docker-compose. Dito isso, precisamos definir os três micro serviços e o rastreador desejado. Devemos definir cada um deles para que sejam criados isoladamente, na definição de cada serviço podemos passar alguns parâmetros: o nome da imagem, política de reinicialização do container, variáveis de ambiente e expor portas quando necessário. Há muitos outros parâmetros disponíveis mas iremos utilizar apenas os citados.

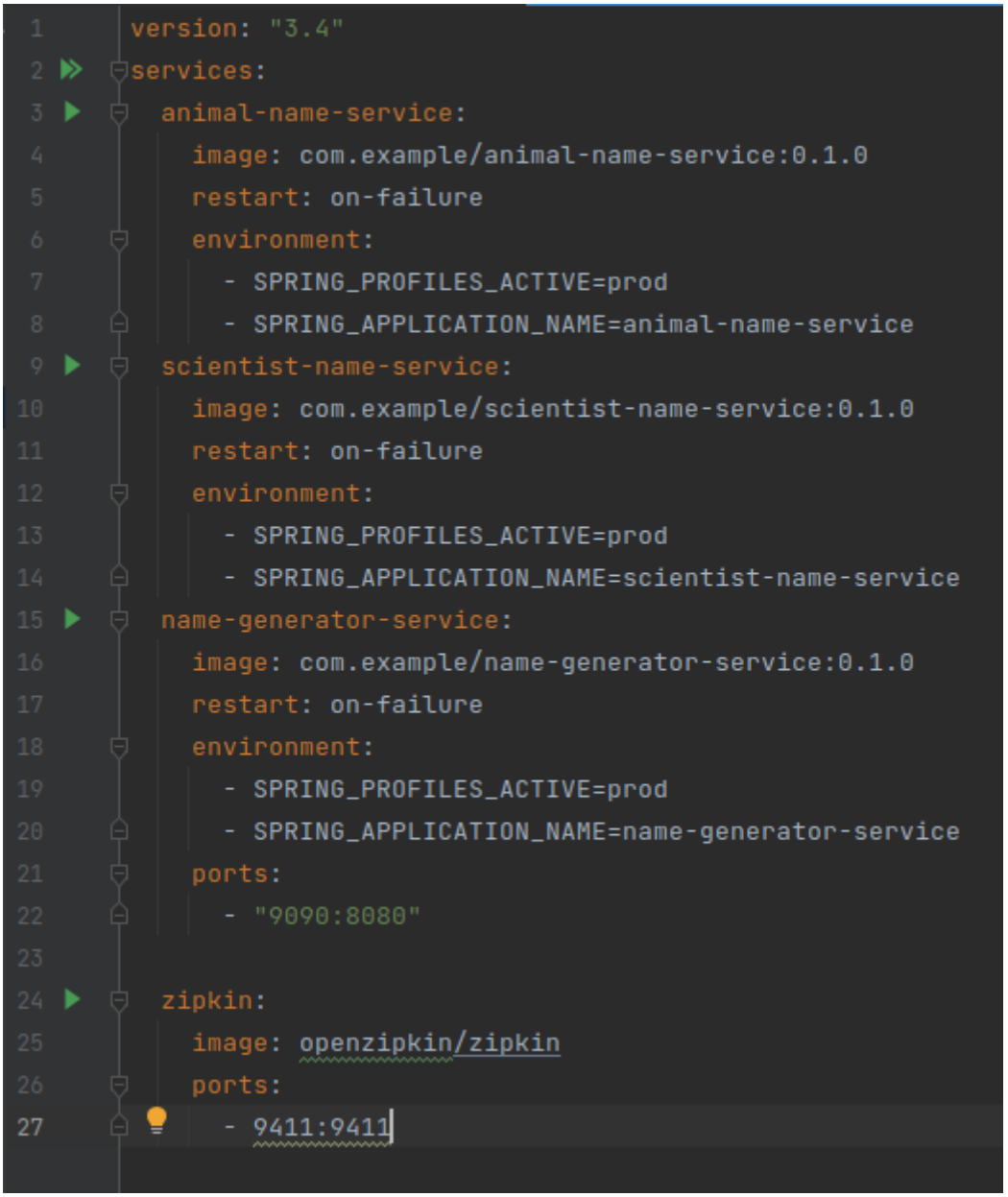
Figura 6: Exemplo de docker-compose com configurações pro Jaeger



```
version: "3.4"
services:
  animal-name-service:
    image: com.example/animal-name-service:0.1.0
    restart: on-failure
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - SPRING_APPLICATION_NAME=animal-name-service
  scientist-name-service:
    image: com.example/scientist-name-service:0.1.0
    restart: on-failure
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - SPRING_APPLICATION_NAME=scientist-name-service
  name-generator-service:
    image: com.example/name-generator-service:0.1.0
    restart: on-failure
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - SPRING_APPLICATION_NAME=name-generator-service
    ports:
      - "9090:8080"
  jaeger:
    image: jaegertracing/all-in-one:1.32
    ports:
      - 5775:5775/udp
      - 6831:6831/udp
      - 6832:6832/udp
      - 5778:5778
      - 16686:16686
      - 14268:14268
      - 14250:14250
      - 9411:9411
```

Fonte: Autor (2022)

Figura 7: Exemplo de docker-compose com configurações pro Zipkin



```yaml
version: "3.4"
services:
  animal-name-service:
    image: com.example/animal-name-service:0.1.0
    restart: on-failure
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - SPRING_APPLICATION_NAME=animal-name-service
  scientist-name-service:
    image: com.example/scientist-name-service:0.1.0
    restart: on-failure
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - SPRING_APPLICATION_NAME=scientist-name-service
  name-generator-service:
    image: com.example/name-generator-service:0.1.0
    restart: on-failure
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - SPRING_APPLICATION_NAME=name-generator-service
    ports:
      - "9090:8080"

  zipkin:
    image: openzipkin/zipkin
    ports:
      - 9411:9411
```

Fonte: Autor (2022)

Depois de termos configurado o arquivo docker-compose já temos tudo pronto para rodar nossa aplicação e gerar os nomes aleatórios. Porém como temos três serviços, construir a imagem de cada um de cada vez pode se tornar uma tarefa tediosa e repetitiva. Para resolver isso vamos escrever um script que contém a sequência de comandos para subir tudo que precisamos direcionado a cada rastreador.

Primeiramente, o arquivo de script mostrado nas imagens abaixo entra na pasta de cada micro serviço, cria o arquivo *.jar* da aplicação e cria a imagem a partir do *Dockerfile* passando como parâmetro.

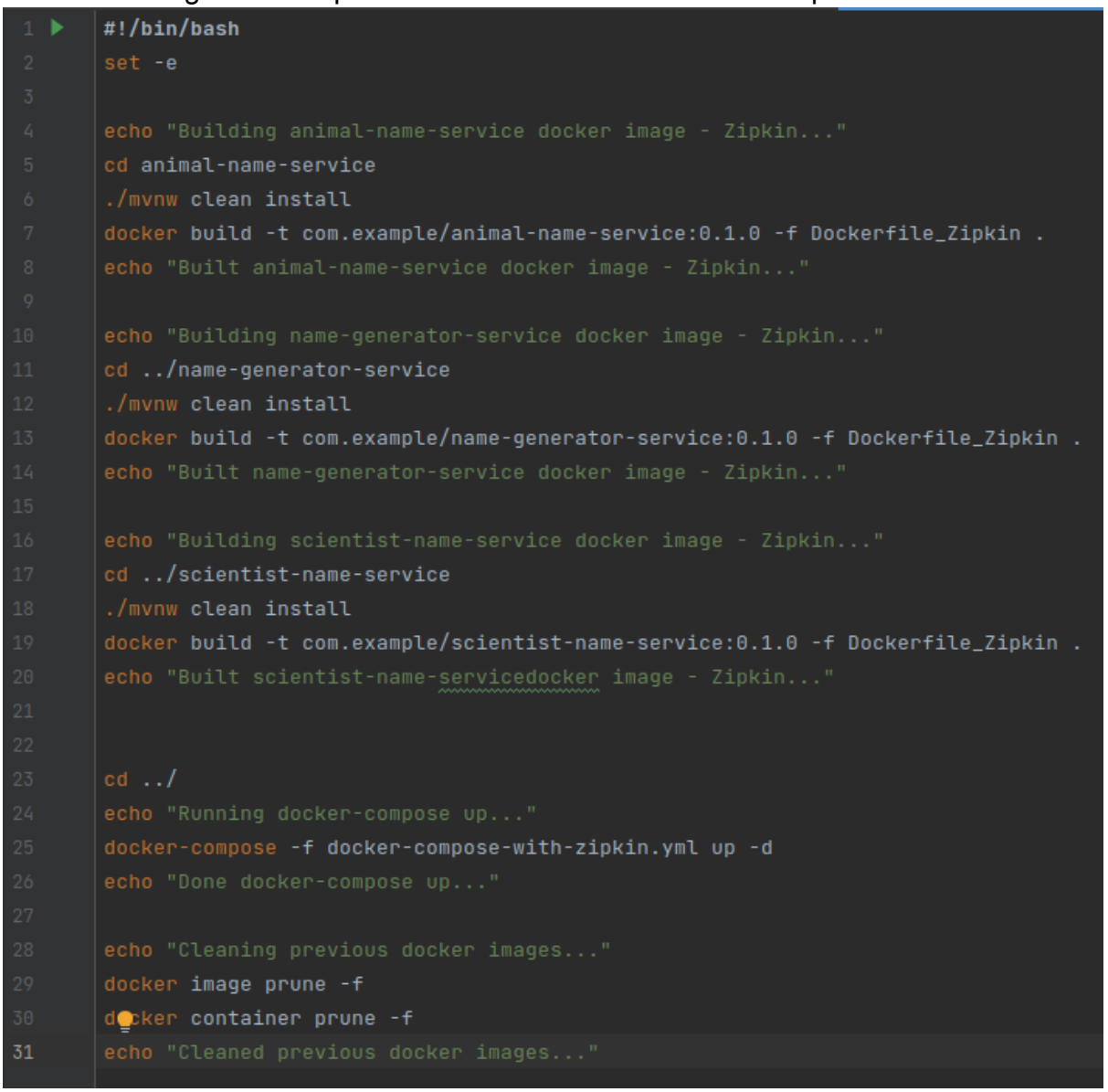
Feito isso, executamos o comando docker-compose para subir toda a nossa infraestrutura e deixar disponível nossa aplicação e o serviço de rastreamento escolhido. Por último, como boa prática, limpamos as imagens e containers sem uso.

Figura 8: Script com comandos direcionados ao Jaeger

```
#!/bin/bash
set -e

echo "Building animal-name-service docker image - jaeger..."
cd animal-name-service
./mvnw clean install
docker build -t com.example/animal-name-service:0.1.0 -f Dockerfile_Jaeger .
echo "Built animal-name-service docker image - jaeger..."

echo "Building name-generator-service docker image - jaeger..."
cd ../name-generator-service
./mvnw clean install
docker build -t com.example/name-generator-service:0.1.0 -f Dockerfile_Jaeger .
echo "Built name-generator-service docker image - jaeger..."

echo "Building scientist-name-service docker image - jaeger..."
cd ../scientist-name-service
./mvnw clean install
docker build -t com.example/scientist-name-service:0.1.0 -f Dockerfile_Jaeger .
echo "Built scientist-name-servicedocker image - jaeger..."


cd ../
echo "Running docker-compose up..."
docker-compose -f docker-compose-with-jaeger.yml up -d
echo "Done docker-compose up..."

echo "Cleaning previous docker images..."
docker image prune -f
docker container prune -f
echo "Cleaned previous docker images..."
```

Fonte: Autor (2022)

Figura 9: Script com comandos direcionados ao Zipkin



```
#!/bin/bash
set -e

echo "Building animal-name-service docker image - Zipkin..."
cd animal-name-service
./mvnw clean install
docker build -t com.example/animal-name-service:0.1.0 -f Dockerfile_Zipkin .
echo "Built animal-name-service docker image - Zipkin..."

echo "Building name-generator-service docker image - Zipkin..."
cd ../name-generator-service
./mvnw clean install
docker build -t com.example/name-generator-service:0.1.0 -f Dockerfile_Zipkin .
echo "Built name-generator-service docker image - Zipkin..."

echo "Building scientist-name-service docker image - Zipkin..."
cd ../scientist-name-service
./mvnw clean install
docker build -t com.example/scientist-name-service:0.1.0 -f Dockerfile_Zipkin .
echo "Built scientist-name-servicedocker image - Zipkin..."


cd ../
echo "Running docker-compose up..."
docker-compose -f docker-compose-with-zipkin.yml up -d
echo "Done docker-compose up..."

echo "Cleaning previous docker images..."
docker image prune -f
docker container prune -f
echo "Cleaned previous docker images..."
```

Fonte: Autor (2022)

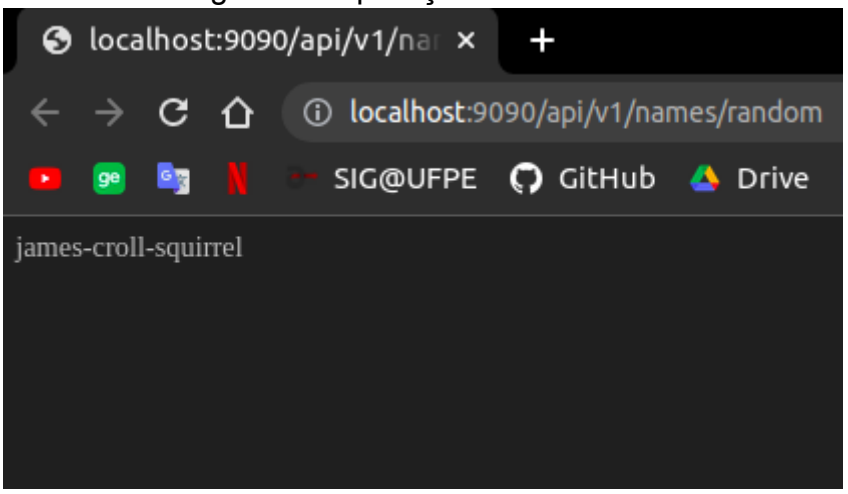
RODANDO O PROJETO E OLHANDO OS TRACES

Neste momento, finalmente, temos tudo pronto para rodar a aplicação com o rastreador que quisermos. Podemos, inclusive, subir as duas para comparar ao mesmo tempo, entretanto devemos colocar configurações de portas diferentes para o *name-generator-service.*

Para rodar compilar e rodar a aplicação com o Zipkin basta digitar o seguinte comando: ***sudo bash docker-setup-with-zipkin.sh***, já para rodar com o Jaeger, devemos escrever: ***sudo bash docker-setup-with-jaeger.sh.***

Dentro de alguns instantes a aplicação estará disponível para uso, podemos então acessar o link: *http://localhost:9090/api/v1/names/random*, e deveremos obter um nome aleatório tal como na imagem abaixo. Sempre que atualizarmos a página um novo nome será gerado.

Figura 10: Aplicação rodando

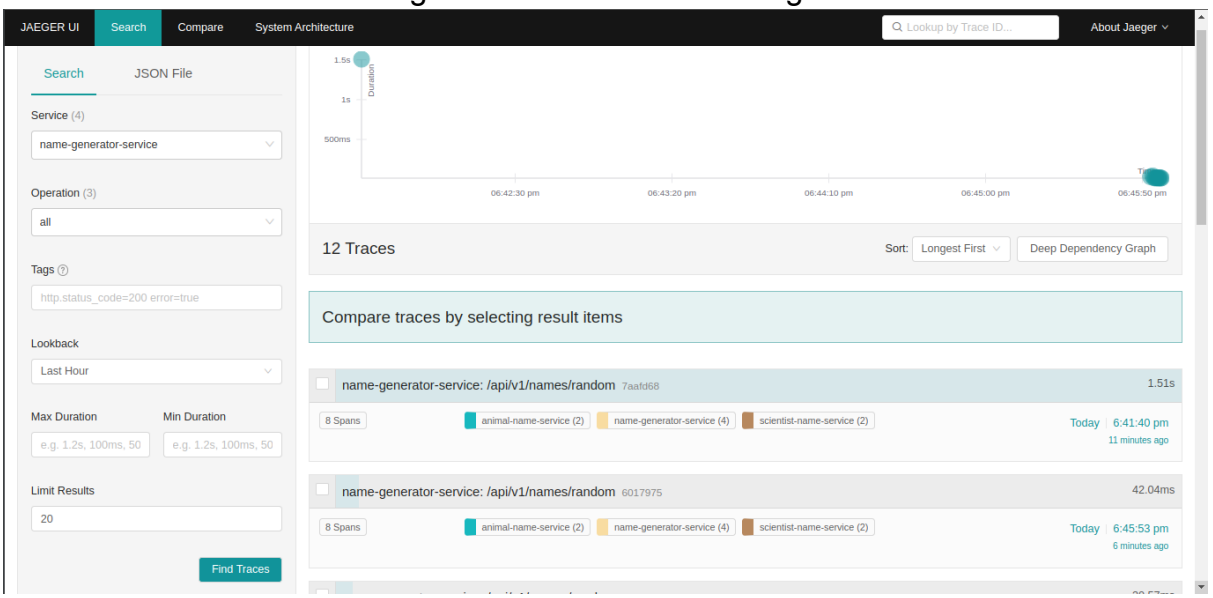


Fonte: Autor (2022)

Após termos feito alguns testes na aplicação, podemos ver como está o rastreamento das requisições. Primeiramente, vamos ver como o Jaeger nos mostra as informações de rastreamento, e para isso vamos acessar: *http://localhost:16686/search*.

A imagem abaixo mostra a tela inicial do Jaeger, na qual podemos ver a possibilidade de buscar os traces. Há filtros pelo nome do serviço, operação executada, tags e horário. No lado direito vemos um gráfico que mostra os traces coletados de acordo com o tempo. E por fim, a lista de traces para serem analisados individualmente.

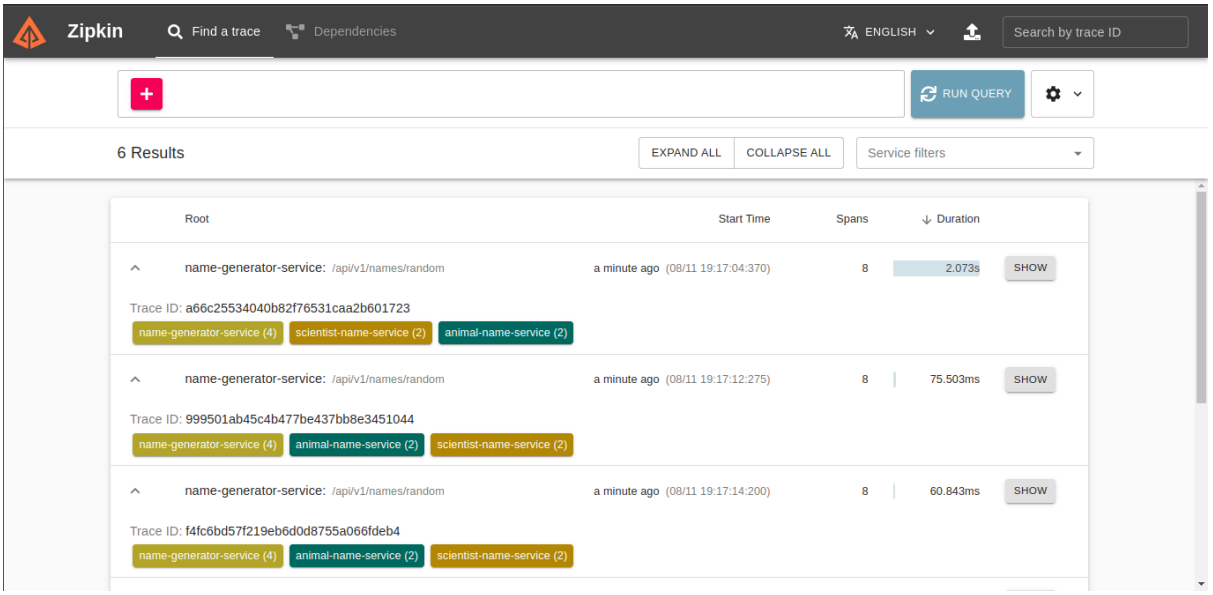
Figura 11: Tela inicial do Jaeger



Fonte: Autor (2022)

Agora vamos analisar os rastreamentos usando o Zipkin, para isso devemos acessar: http://localhost:9411. A tela inicial também nos mostra a lista de traces e tem todos os filtros presentes no Jaeger, a maior diferença é a ausência do gráfico que mostra as requisições com base no tempo.

Figura 12: Tela inicial do Zipkin



Fonte: Autor (2022)

O código fonte do projeto base está disponível no repositório: https://bit.ly/3qpPlf2. O repositório é público e pode ser utilizado para quaisquer fins.

3.4 MÉTRICAS DE COMPARAÇÃO

Agora que já sabemos configurar o projeto com as ferramentas escolhidas, nos resta fazer a discussão comparativa, que é um dos objetivos deste trabalho. Entretanto, isso será realizado no capítulo seguinte, nesta seção trataremos apenas da escolha das métricas a serem utilizadas na comparação.

Como o intuito deste trabalho é facilitar a escolha para que a ferramenta possa ser implementada com mais agilidade dentro de uma organização, não podemos deixar de falar da **arquitetura e forma de implantação utilizada**. Uma empresa maior pode necessitar de uma ferramenta que tenha uma arquitetura mais robusta, já uma empresa pequena deve escolher uma arquitetura mais simples. Com isso, saber quais modelos de arquitetura estão disponíveis em cada ferramenta é de suma importância.

Ademais, também é importante saber se o **conjunto de funcionalidades** de cada ferramenta se adequa às necessidades da organização. Neste quesito, faremos uma comparação entre as funcionalidades semelhantes e será ressaltado também as funcionalidades adicionais que uma ferramenta pode ter de diferente da outra [31].

Por último, um fator essencial na escolha de ferramentas open source é saber como é a **adoção pela comunidade**. Perguntas que podem nos ajudar: A comunidade continua fazendo manutenções na ferramenta? Existe alguma organização que dá suporte? A comunidade fala sobre essa ferramenta em blogs e fóruns? Essas perguntas vão nos ajudar a saber se a ferramenta tem um futuro longo ou se está caindo em desuso, a intenção é escolher ferramentas que tenham um longo futuro para não ter retrabalho [31, 32].

3.5 SÍNTESE DO CAPÍTULO

Iniciamos este capítulo entendendo a para que serve e como usamos as ferramentas de apoio, e posteriormente, vimos todo o processo para criação do

projeto base, desde a escolha das ferramentas até a visualização dos traces nas ferramentas de rastreamento. Por último, também foi importante analisarmos as métricas de comparação a serem utilizadas na discussão.

No próximo capítulo vamos nos aprofundar na comparação entre os rastreadores, falar dos pontos positivos e negativos e negativos de cada ferramenta e discutir alguns casos de uso. Utilizaremos como base as métricas abordadas na seção anterior deste capítulo.

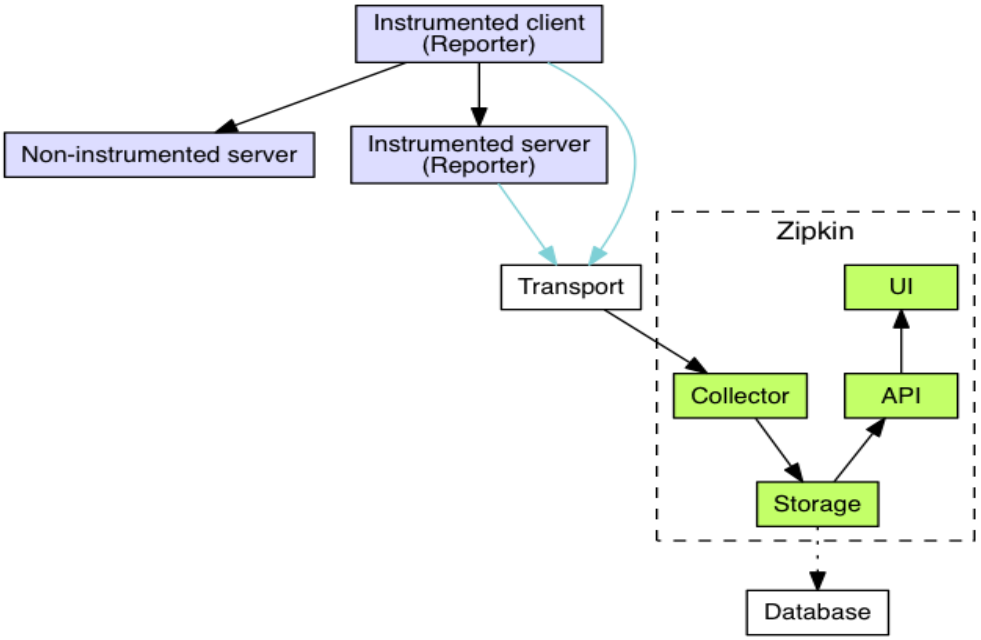
4. DISCUSSÃO COMPARATIVA

Chegou o momento de refletirmos sobre as duas ferramentas apresentadas neste trabalho e quais as vantagens e desvantagens de ambas. As ferramentas apresentadas existem para resolver o mesmo problema, o rastreamento de requisições, porém foram construídas em momentos distintos e de forma diferente. Utilizaremos as métricas descritas no capítulo 3, seção 4 como um guia para a discussão deste capítulo. Como um lembrete, a motivação de escolha das ferramentas comparadas aqui se encontram no capítulo 3, seção 2.

4.1 ARQUITETURA

A arquitetura do Zipkin consiste de quatro serviços: Coletor, que tem como função receber os traces das aplicações e repassar para serem armazenados; Armazenamento deve fazer a ponte entre a API e o banco de dados escolhido; API serve como uma camada para expor os dados; E a interface gráfica consome a API para exibir os traces. A arquitetura montada fica de acordo com a imagem abaixo.

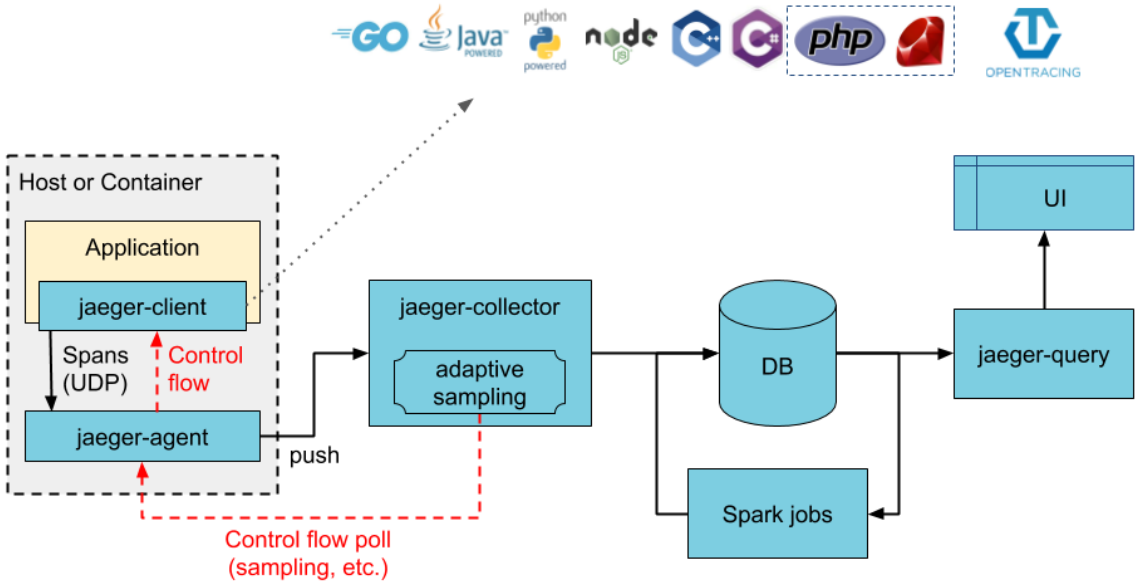
Figura 13: Arquitetura do Zipkin



Fonte: Zipkin (2022)

O Jaeger, contudo, possui uma arquitetura um pouco diferente e mais flexível. Esta ferramenta foi pensada para ser disponibilizada de duas formas, com todos os componentes em um único processo, ou de forma distribuída. Vamos estudar primeiro a forma centralizada.

Figura 14: Arquitetura do Jaeger (centralizada)



Fonte: Jaeger (2022)

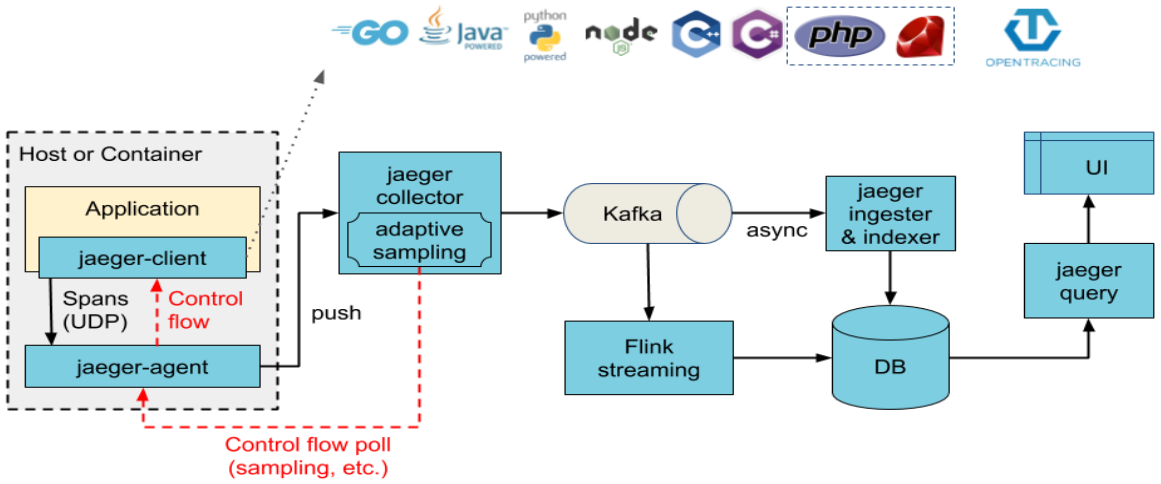
A arquitetura acima é composta por elementos muito semelhantes à arquitetura do Zipkin, com exceção de que não existe um serviço para gravação de dados e acesso à base de dados. O coletor grava diretamente no banco e a busca faz as consultas direto também.

Importante ressaltar também que com o Jaeger é possível ter uma abordagem mais distribuída e resiliente. Essa abordagem consiste em colocar o Kafka como intermediário entre o coletor e o banco de dados, isso faz com que o sistema de ingestão de dados na base seja mais confiável.

A utilização do Kafka, que é uma plataforma distribuída de envio de mensagens, faz com que a arquitetura do Jaeger seja mais confiável pois provê

alguns mecanismos como: escalabilidade, armazenamento permanente, alta disponibilidade, entre outros [25].

Figura 15: Arquitetura do Jaeger (distribuída)



Fonte: Jaeger (2022)

A escolha da arquitetura usada para um sistema de rastreamento distribuído depende muito das necessidades que os sistemas a serem monitorados apresentam. No contexto de uma grande empresa, é provável que usar o Jaeger com uma arquitetura distribuída faça mais sentido, pois a quantidade de traces gerados será maior.

Quando se trata de empresas não tão grandes, com sistemas menos críticos, tanto Zipkin quanto Jaeger podem ser utilizados com a arquitetura centralizada, isto é, o deploy pode ser feito usando a imagem docker que contém todos os serviços em um só. Cabe ao time analisar o seu cenário e escolher a melhor solução.

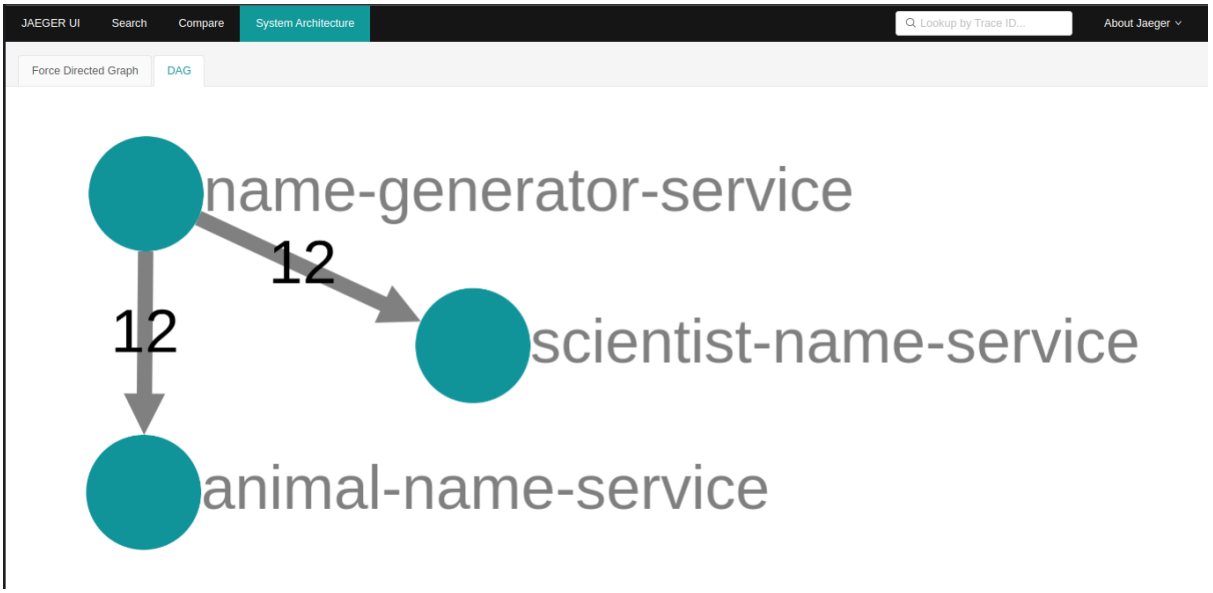
4.2 FUNCIONALIDADES

Um fator que contribui muito para a escolha de uma ferramenta são as suas funcionalidades e os detalhes podem fazer a diferença a favor de uma ferramenta em comparação a outra.

As duas ferramentas apresentadas têm a funcionalidade de busca de traces nas suas telas iniciais (figuras 9 e 10). Ambas apresentam filtros por serviço, duração, tags, horário e etc. Podemos notar que as funcionalidades são muito parecidas, porém o Jaeger apresenta um gráfico com a duração das requisições ao longo do tempo, o que ajuda a identificar de forma rápida e visual um possível gargalo no sistema.

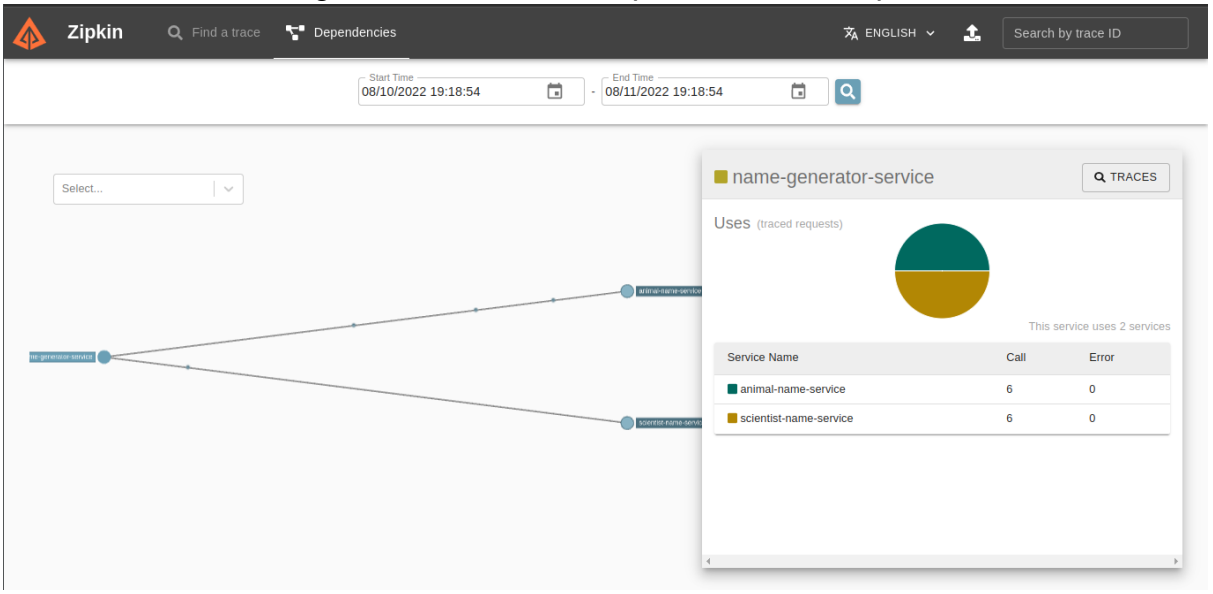
Outra funcionalidade muito interessante presente nos dois rastreadores é mostrar como estão dispostas as dependências entre os serviços. Esse tipo de abordagem é útil para a identificação de dependências cíclicas entre os serviços que compõem o sistema, tal problema pode causar fluxos que não têm fim. No caso de grandes sistemas, que rodam centenas ou até milhares de serviços, para a correta identificação desse tipo de problema é necessário uma ferramenta que faça esse trabalho. As imagens abaixo mostram como é montado esse gráfico em cada ferramenta.

Figura 16: Gráfico de dependências do Jaeger



Fonte: Autor (2022)

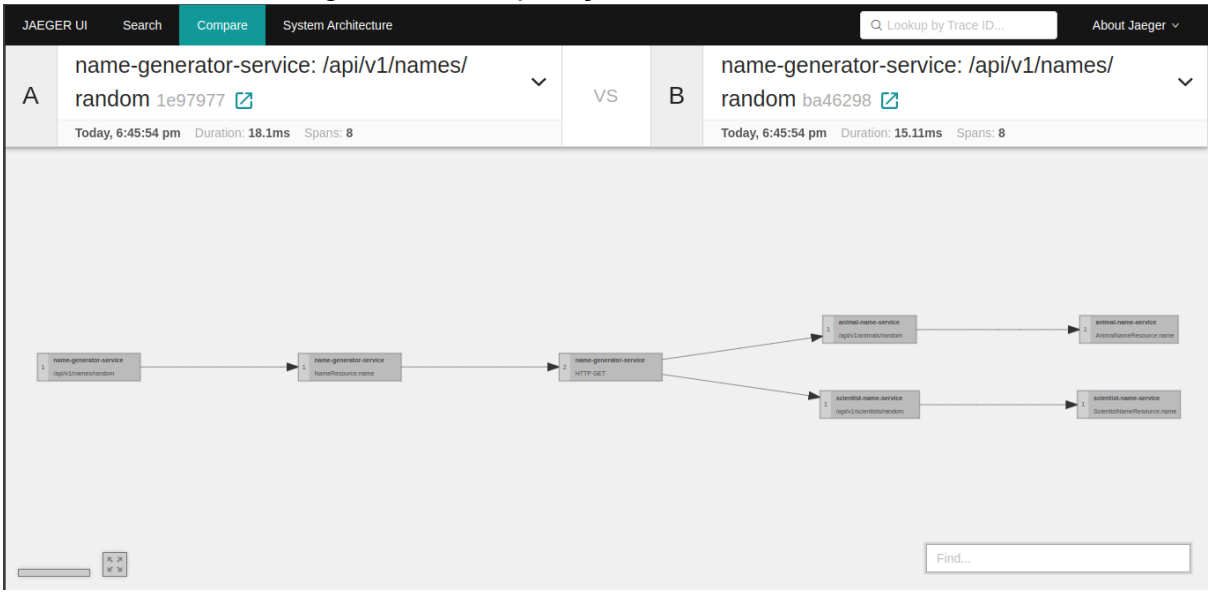
Figura 17: Gráfico de dependências do Zipkin



Fonte: Autor (2022)

Há funcionalidades, porém, que só estão disponíveis no Jaeger. Uma delas é a comparação de traces, que fornece uma visualização de dois traces na mesma tela para que o desenvolvedor possa analisar as similaridades e diferenças entre eles. Essa funcionalidade é útil para depurar o fluxo de uma requisição com problema, que pode ser comparada com uma requisição de sucesso. A imagem abaixo mostra como é a interface de comparação.

Figura 17: Comparação de traces



Fonte: Autor (2022)

4.3 COMUNIDADE E SUPORTE

As ferramentas aqui utilizadas, por se tratarem de softwares *open source,* um fator muito importante que deve ser levado em consideração é como é o uso e aceitação da ferramenta na comunidade. Ferramentas com comunidades ativas têm melhor resolução de problemas e ajuda para configuração [26].

Um bom norte para olhar como é o engajamento da comunidade com a ferramenta é olhar o repositório oficial no github. Falando em números, Jaeger tem 16.4 mil stars, mais de 2 mil forks, 311 issues e cerca de 250 contribuidores. Já o Zipkin apresenta 15.6 mil stars, mais de 3 mil forks, 180 issues e aproximadamente 152 contribuidores. Os números apresentados mostram uma semelhança de engajamento entre as duas ferramentas, Jaeger apresenta mais stars, porém o Zipkin tem mais forks, e etc [27, 28].

Além do github, podemos olhar também para a quantidade de posts em blogs da comunidade. No site da dev.to, que é um site para engajamento da comunidade de desenvolvedores, existe conteúdo sobre quase todas as ferramentas do mundo de desenvolvimento de software. Comparando a quantidade de posts sobre cada

ferramenta, podemos ver que o Jaeger leva uma certa vantagem contra o Zipkin, 18 a 3 [29, 30].

O Zipkin tem um menor número de issues abertas no repositório em relação ao Jaeger, isso pode ser um indício de que tem uma plataforma mais estável. Além disso, é amplamente utilizado na indústria e tem uma comunidade bastante ativa. Isso pode ser importante para empresas mais conservadoras e que prezam por confiabilidade e segurança [24].

Já o Jaeger, apesar de ser uma ferramenta mais nova, também tem uma comunidade ativa. Outro ponto importante é que o Jaeger tem suporte da Cloud Native Computing Foundation (CNCF), que é uma organização voltada a ajudar projetos *open source* que contribuam para aplicações implantadas na cloud [24].

4.4 SÍNTESE DO CAPÍTULO

Neste capítulo olhamos detalhadamente para a comparação entre Jaeger e Zipkin, vimos os pontos positivos e negativos de cada ferramenta para que possamos fazer a escolha que melhor se adequa ao nosso contexto. A fim de termos uma comparação mais visual entre as ferramentas, podemos analisar as tabelas abaixo.

	Jaeger	Zipkin
Busca de traces	X	X
Gráfico de dependencias	X	X
Comparação de traces	X	
Filtros de traces	X	X

Tabela 4.1. Funcionalidades de cada ferramenta

	Jaeger	Zipkin
Forks	2 k	3 k
Stars	16.4 k	15.6 k
Contribuidores	250	152
Issues	311	180

Tabela 4.2. Números dos repositórios no github

5. CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo apresenta a conclusão deste trabalho, algumas limitações encontradas no projeto e perspectivas de trabalhos futuros. Primeiro veremos as considerações finais sobre o projeto aqui apresentado, depois abordaremos as limitações presentes e posteriormente os possíveis trabalhos futuros que podem vir a ser realizados.

5.1 CONCLUSÃO

A arquitetura de microsserviços é uma arquitetura muito robusta e a sua utilização deve sempre ser analisada para projetos de larga escala. Principalmente em negócios onde alta disponibilidade é um requisito primordial.

Como já sabemos, com o uso da arquitetura de microsserviços, há inúmeros benefícios, mas também vários desafios. Um dos principais desafios é que a maior complexidade causa dificuldades no quesito da observabilidade do sistema como um todo. Mais especificamente, tratamos neste trabalho sobre o problema de rastreamento de requisições em um sistema distribuído.

Para auxiliar nesta tarefa, existem ferramentas *open source* no mercado tão efetivas quanto outros softwares pagos, que são inacessíveis para pequenas empresas e desenvolvedores independentes. Depois de um estudo sobre quais ferramentas são mais relevantes para serem comparadas, as escolhidas foram Jaeger e Zipkin devido a grande aceitação de ambas pela comunidade Java.

O Zipkin é uma ferramenta mais antiga, mas ainda assim muito usada, estável e com um bom leque de funcionalidades. O Jaeger é uma ferramenta mais recente e traz consigo uma personalização maior e funcionalidades inovadoras. As duas ferramentas escolhidas entregam bem o prometido.

Este trabalho conseguiu alcançar o resultado esperado: ajudar na escolha de ferramentas de rastreamento por meio de uma comparação prática das duas ferramentas *open source* mais relevantes e deixar disponível um repositório de

exemplo para consulta. O código criado e utilizado neste trabalho está disponível em https://bit.ly/3qpPlf2.

5.2 LIMITAÇÕES

Na análise realizada durante o desenvolvimento deste trabalho foi utilizada uma aplicação com três serviços. Seria interessante uma aplicação com mais serviços executando ao mesmo tempo. Porém, devido ao esforço de configuração ser muito grande e exigir uma capacidade de processamento maior do que está disponível para o autor, este trabalho não contemplou uma quantidade maior de serviços.

5.3 TRABALHOS FUTUROS

Como trabalhos futuros a serem desenvolvidos a partir deste, podem-se sugerir:

1. **Estudo da economia obtida pela adoção de uma ferramenta de rastreamento *open source*** em detrimento às ferramentas pagas. Como discutido anteriormente, ferramentas de APM e rastreamento pagas são inacessíveis para muitos desenvolvedores independentes e empresas de pequeno porte, por isso a existência de ferramentas *open source* é tão importante. O estudo proposto faria uma análise da economia mensal com a adoção de um rastreador *open source* em comparação com um pago, mostrando a diferença de valores à medida em que o projeto cresce.

2. Outra opção muito interessante seria uma **Análise comparativa das variações de implantação do Jaeger** e qual se encaixa melhor para cada contexto dentro de uma empresa. Na nossa análise de arquitetura está presente a discussão sobre os modelos de implantação do Jaeger dentro de uma organização, que podem ser vários, visto que quanto maior for a empresa, maior é a necessidade de se utilizar uma arquitetura mais robusta, a fim de minimizar a perda de traces por indisponibilidade do rastreador.

REFERÊNCIAS

[1] WALPITA, Priyal. **Evolution in Software Architecture**. medium.com, 2022. Disponível em: <https://priyalwalpita.medium.com/evolution-in-software-architecture-a607db649586>. Acesso em: 17. ago. 2022.

[2] DIGUER, Sarah. **Microservices Advantages and Disadvantages: Everything You Need to Know**. Solace, 2020. Disponível em: <https://solace.com/blog/microservices-advantages-and-disadvantages/>. Acesso em: 16. fev. 2022.

[3] KUMAR JHA, Abhishek. **4 Difficult Microservices Observability Challenges and How to Address Them**. TechWorm, 2022. Disponível em: <https://www.techworm.net/2021/07/microservices-observability-challenges.html>. Acesso em: 08. fev. 2022.

[4] KOWALL, Jonah. **How microservices broke monitoring (and how to fix it)**. Tech Beacon, 2022 . Disponível em: <https://techbeacon.com/app-dev-testing/how-microservices-broke-monitoring-how-fix-it>. Acesso em: 18. fev. 2022.

[5] sem autor. **Application Monitoring Tools List – Top 16 Compared?**. Dotcom tools, 2022. Disponível em: <https://www.dotcom-tools.com/web-performance/list-of-application-monitoring-tools-apm/>. Acesso em: 18. fev. 2022.

[6] u/mrjenkins2017. **“What are people's opinion on Datadog for monitoring?”**. Reddit, 2022. Disponível em: <https://www.reddit.com/r/devops/comments/7bb2ao/what_are_peoples_opinion_on_datadog_for_monitoring/>. Acesso em: 21 Fev. 2021.

[7] u/deleted. **“What do you guys think of AppDynamics?”**. Reddit, 2022. Disponível em: <https://www.reddit.com/r/sysadmin/comments/6ef90e/what_do_you_guys_think_of_appdynamics/>. Acesso em: 21 Fev. 2021.

[8] sem autor. **Application Performance Monitoring Reviews and Ratings**. Gartner 2022. 21 Fev. 2022. Disponível em: <https://www.gartner.com/reviews/market/application-performance-monitoring/>. Acesso em: 18. fev.

[9] HUBBARD, Patrick. **The Expensive History of APM**. pingdom, 2022. Disponível em: https://www.pingdom.com/blog/the-expensive-history-of-apm/. Acesso em: 18. fev. 2022.

[10] LOUKIDES, M; SWOYER, S. **Microservices Adoption in 2020**. oreilly, 2020. Disponível em:

https://www.oreilly.com/radar/microservices-adoption-in-2020. Acesso em: 23. ago. 2022.

[11] MW Team. **What Are Microservices? How Microservices Architecture Works**. middleware, 2021. Disponível em: https://middleware.io/blog/microservices-architecture/ . Acesso em: 23. ago. 2022.

[12] RICHARDSON, Mary Ann. **Top 10 Challenges of Using Microservices for Managing Distributed Systems**. spiceworks, 2021. Disponível em: https://www.spiceworks.com/tech/data-management/articles/top-10-challenges-of-using-microservices-for-managing-distributed-systems/. Acesso em: 23. ago. 2022.

[13] KURMI, Anil. **What are the challenges in Microservices Architecture?**. medium, 2020. Disponível em: https://medium.com/microservices-architecture/what-are-the-challenges-in-microservices-architecture-2ee9149cfc4e. Acesso em: 23. ago. 2022.

[14] sem autor. **What Is Observability?**. splunk, 2022. Disponível em: https://www.splunk.com/en_us/data-insider/what-is-observability.html. Acesso em: 23. ago. 2022.

[15] EGILMEZ, Ismail. **Monitoring vs. Observability: What's the Difference?**. thenewstack, 2020. Disponível em: https://thenewstack.io/monitoring-vs-observability-whats-the-difference/. Acesso em: 23. ago. 2022.

[16] BIGELOW, Stephen J. **Distributed tracing**. techtarguet, 2022. Disponível em: https://www.techtarget.com/searchitoperations/definition/distributed-tracing. Acesso em: 23. ago. 2022.

[17] MOREHOUSE, John. **What Is Distributed Tracing? Key Concepts and Definition**. orangematter, 2022. Disponível em: https://orangematter.solarwinds.com/2022/01/28/what-is-distributed-tracing. Acesso em: 23. ago. 2022.

[18] sem autor. **Use containers to Build, Share and Run your applications**. docker, 2022. Disponível em: https://www.docker.com/resources/what-container. Acesso em: 23. ago. 2022.

[19] sem autor. **O que é o Docker?**. oracle, 2022. Disponível em: https://www.oracle.com/br/cloud/cloud-native/container-registry/what-is-docker/#docker-basics. Acesso em: 4. set. 2022.

[20] sem autor. **Spring Boot: Tudo que você precisa saber!**. geekhunter, 2022. Disponível em: https://blog.geekhunter.com.br/tudo-o-que-voce-precisa-saber-sobre-o-spring-boot/. Acesso em: 4. set. 2022.

[21] sem autor. **Spring Boot**. spring, 2022. Disponível em: https://spring.io/projects/spring-boot. Acesso em: 4. set. 2022.

[22] sem autor. **What is OpenTelemetry?**. opentelemetry, 2022. Disponível em: https://opentelemetry.io/docs/concepts/what-is-opentelemetry/. Acesso em: 4. set. 2022.

[23] sem autor. **What is OpenTelemetry?**. splunk, 2022. Disponível em: https://www.splunk.com/en_us/data-insider/what-is-opentelemetry.html. Acesso em: 4. set. 2022.

[24] ÖZAL, Serkan. **Jaeger vs. Zipkin: Battle of the Open Source Tracing Tools**. splunk, 2022. Disponível em: https://thenewstack.io/jaeger-vs-zipkin-battle-of-the-open-source-tracing-tools/. Acesso em: 4. set. 2022.

[25] sem autor. **APACHE KAFKA**. kafka, 2022. Disponível em: https://kafka.apache.org/. Acesso em: 9. set. 2022.

[26] Avyaa. **Role of Community in Scaling Open Source Projects**. dev.to, 2021. Disponível em: https://dev.to/aviyel/role-of-community-in-scaling-open-source-projects-19g6. Acesso em: 13. set. 2022.

[27] sem autor. [Lista de artigos sobre o Jaeger]. dev.to, 2022. Disponível em: https://dev.to/t/jaeger. Acesso em: 13. set. 2022.

[28] sem autor. [Lista de artigos sobre o Zipkin]. dev.to, 2022. Disponível em: https://dev.to/t/zipkin. Acesso em: 13. set. 2022.

[29] sem autor. [Repositório oficial do Jaeger]. github, 2022. Disponível em: https://github.com/jaegertracing/jaeger. Acesso em: 13. set. 2022.

[30] sem autor. [Repositório oficial do Zipkin]. github, 2022. Disponível em: https://github.com/openzipkin/zipkin. Acesso em: 13. set. 2022.

[31] CROUCH, Steve. **Choosing the right open-source software for your project**. software.ac, 2022. Disponível em: https://www.software.ac.uk/choosing-right-open-source-software-your-project. Acesso em: 28. set. 2022.

[32] sem autor. **Tips To Choose the right open source tool**. eclature, 2022. Disponível em: https://eclature.com/open-source-tool/. Acesso em: 28. set. 2022.

[33] GRAÇA, Herberto. **Architectural Styles vs. Architectural Patterns vs. Design Patterns**. herbertograca.com, 2022. Disponível em: https://herbertograca.com/2017/07/28/architectural-styles-vs-architectural-patterns-vs-design-patterns/. Acesso em: 30. set. 2022.

[34] GRAÇA, Herberto. **Monolithic Architecture**. herbertograca.com, 2022. Disponível em: https://herbertograca.com/2017/07/31/monolithic-architecture/. Acesso em: 30. set. 2022.

[35] LIVENS, Jay. **What is observability? Not just logs, metrics and traces**. dynatrace.com, 2021. Disponível em: https://www.dynatrace.com/news/blog/what-is-observability-2/. Acesso em: 3. out. 2022.

[36] FOWLER, Martin. **Testing Strategies in a Microservice Architecture**. martinfowler.com, 2022. Disponível em: https://martinfowler.com/articles/microservice-testing/. Acesso em: 3. out. 2022.

[37] DINUWAN, Chanuka. **Getting started with Microservices Architecture**. blog.devgenius.io, 2021. Disponível em: https://blog.devgenius.io/getting-started-with-microservices-architecture-203172390928. Acesso em: 3. out. 2022.

[38] sem autor. [Repositório oficial do Uptrace]. github, 2022. Disponível em: https://github.com/uptrace/uptrace. Acesso em: 6. out. 2022.

[39] sem autor. [Repositório oficial do Grafana Tempo]. github, 2022. Disponível em: https://github.com/grafana/tempo. Acesso em: 13. set. 2022.

[40] sem autor. [Repositório oficial do Signoz]. github, 2022. Disponível em: https://github.com/SigNoz/signoz. Acesso em: 13. set. 2022.

[41] BARKER, Dan. **3 open source distributed tracing tools**. opensource.com, 2021. Disponível em: https://opensource.com/article/18/9/distributed-tracing-tools. Acesso em: 6. out. 2022.

[41] sem autor. **Free and Open Source Distributed Tracing Tools**. uptrace.dev, 2021. Disponível em: https://uptrace.dev/get/compare/distributed-tracing-tools.html#uptrace. Acesso em: 6. out. 2022.

[42] sem autor. **George Santayana**. wikiquote.org, 2021. Disponível em: https://en.wikiquote.org/wiki/George_Santayana#Quotes_about_Santayana. Acesso em: 18. out. 2022.