



Universidade do Minho
Escola de Engenharia

Rui Pedro Chaves Silva Lousada Alves

Monitorização de uma Arquitetura de Microserviços



Universidade do Minho
Escola de Engenharia

Rui Pedro Chaves Silva Lousada Alves

Monitorização de uma Arquitetura de Microserviços

Mestrado em Engenharia Informática

Dissertation supervised by
Professor Orlando Belo

Direitos de Autor e Condições de Utilização do Trabalho por Terceiros

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho:



CC BY-NC-ND

<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Agradecimentos

Em primeiro lugar, manifesto o meu sincero agradecimento ao meu orientador, Professor Orlando Belo pela orientação inestimável, conselhos esclarecedores e dedicação inabalável ao meu desenvolvimento académico. A sua experiência e incentivo foram determinantes em todas as fases do processo de investigação e redação, não sendo este trabalho possível sem o seu contributo.

De igual modo, expresso o meu profundo reconhecimento ao DTx-Colab por me ter proporcionado a oportunidade de desenvolver esta investigação em ambiente empresarial, integrada no projeto R2UT. Esta colaboração permitiu-me obter uma perspetiva prática e aplicar o conhecimento adquirido em contexto real, aumentando significativamente o impacto e a relevância desta tese.

O meu agradecimento sentido dirige-se igualmente à minha família, cujo apoio incondicional e sacrifícios foram fundamentais para atingir estas metas académicas e pessoais. Sou especialmente grato aos meus pais, Rui e Domingas. Expresso também a minha profunda gratidão à minha namorada, Leonor, pela paciência, compreensão e encorajamento prestados ao longo desta caminhada.

Quero ainda expressar um agradecimento especial ao meu colega e amigo André, pelo tempo disponibilizado, pela partilha de conhecimento e pela ajuda prestada nos momentos de maior exigência técnica. A sua colaboração e disponibilidade foram essenciais para ultrapassar vários desafios durante o desenvolvimento deste trabalho.

Por fim, agradeço a todos os meus verdadeiros amigos pela companhia, estímulo e presença constante nos momentos de maior desafio. O vosso apoio foi, sem dúvida, uma parte fundamental desta jornada.

Declaração de Integridade

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Universidade do Minho, Braga, outubro 2025

Rui Pedro Chaves Silva Lousada Alves

Resumo

As arquiteturas de microserviços permitem construir sistemas flexíveis, escaláveis e modulares em ambientes distribuídos. Contudo, a sua natureza dinâmica aumenta a complexidade dos processos de monitorização contínua, deteção de falhas e resposta proativa a eventos críticos.

Neste trabalho de dissertação, foi implementada uma plataforma para a monitorização e gestão de alertas de infraestruturas baseadas em microserviços, com aplicação prática na indústria da construção modular. A solução desenvolvida integra ferramentas *open source* — *Prometheus*, *Grafana*, *Loki* (como alternativa à *ELK Stack*) e *Jaeger* — suportadas pelo *OpenTelemetry* para a recolha padronizada de métricas, *logs* e rastreio (*tracing*) distribuído.

Para além disso, foram definidos cenários de alerta e mecanismos de resposta automática, de forma a reforçar a resiliência e reduzir o tempo de indisponibilidade da infraestrutura em monitorização.

Os resultados obtidos demonstram ganhos significativos em visibilidade ponta-a-ponta e uma redução nos tempos médios de deteção e resolução de incidentes (*MTTD* e *MTTR*), comprovando a viabilidade de uma pilha de observabilidade aberta, escalável e alinhada com requisitos de produção.

Palavras-chave: Arquitetura de Microserviços, Contentorização, Kubernetes, Monitorização de Sistemas Distribuídos, Docker, Deteção e Resolução de Falhas.

Abstract

Microservices architectures enable the development of flexible, scalable, and modular systems in distributed environments. However, their dynamic nature increases the complexity of continuous monitoring, fault detection, and proactive response to critical events.

This dissertation presents the implementation of a monitoring and alert management platform for microservices-based infrastructures, with practical application in the modular construction industry. The proposed solution integrates *open source* tools — *Prometheus*, *Grafana*, *Loki* (as an alternative to the *ELK Stack*), and *Jaeger* — supported by *OpenTelemetry* for standardized collection of metrics, *logs*, and distributed *tracing*.

Furthermore, alert scenarios and automated response mechanisms were defined to strengthen system resilience and reduce infrastructure downtime.

The results demonstrate significant gains in end-to-end visibility and a reduction in the mean time to detect and resolve incidents (*MTTD* and *MTTR*), validating the feasibility of an open, scalable, and production-ready observability stack.

Keywords Microservices Architecture, Containerization, Kubernetes, Distributed Systems Monitoring, Docker, Fault Detection and Troubleshooting.

Conteúdo

I	Material Introdutório	1
1	Introdução	2
1.1	Contextualização	2
1.2	Motivação	3
1.3	Objetivos	4
1.4	Trabalho Realizado	5
1.5	Estrutura do Documento	6
2	Microserviços	7
2.1	Emergência e Evolução	7
2.1.1	Limitações das Arquiteturas Monolíticas	7
2.1.2	De SOA a microserviços	8
2.1.3	Fatores Tecnológicos e Organizacionais	9
2.1.4	Popularização dos Microserviços	10
2.2	O que são microserviços?	11
2.2.1	Definição Formal	11
2.2.2	Princípios e Principais Características	12
2.2.3	Componentes Típicos em Arquiteturas de microserviços	12
2.2.4	Comparação com outras Arquiteturas	13
2.3	Os Desafios das Arquiteturas de Microserviços	14
2.3.1	Organização de Sistemas de Microserviços	14
2.3.2	Principais Desafios Técnicos	15
2.3.3	Principais Desafios Organizacionais	16
2.4	Arquiteturas de microserviços em Grande Escala	16

2.4.1	Escalabilidade Horizontal	17
2.4.2	Gestão do Estado dos Serviços	17
2.4.3	Orquestração e Automação	17
2.5	microsserviços e Computação em Nuvem	18
2.5.1	Arquitetura <i>Serverless</i>	18
2.5.2	Custo e Eficiência de Escalabilidade	19
2.5.3	Desafios da Computação em Nuvem	19
3	Como Monitorizar uma Arquitetura de Microserviços	20
3.1	A Importância da Monitorização em Microserviços	20
3.1.1	Diferença entre monitorizar arquiteturas monolíticas e microserviços	20
3.1.2	Impacto da monitorização na fiabilidade e na escalabilidade	21
3.1.3	Principais Objetivos da Monitorização	21
3.2	Técnicas, Estratégias e Ferramentas de Monitorização	22
3.2.1	Logs centralizados	22
3.2.2	Monitorização de Métricas	23
3.2.3	Tracing Distribuído	23
3.2.4	Ferramentas Comuns na Monitorização	25
3.2.5	Conceito de Observabilidade	25
3.3	Desafios na Monitorização de Microserviços	26
3.3.1	Alta Cardinalidade de Dados	26
3.3.2	Correlação de Eventos Distribuídos	26
3.3.3	Problemas de latência e visibilidade	26
3.4	Estudos de Caso: Exemplos Práticos de Monitorização em Microserviços	27
3.4.1	Netflix: Monitorização em Escala Global	27
3.4.2	Amazon: Escalabilidade e Resiliência em Grande Escala	27
3.4.3	Uber: Monitorização e Observabilidade para Escalabilidade Global	28
3.4.4	Conclusão dos Estudos de Caso	28
3.5	Futuro da Monitorização de Microserviços	28
3.5.1	Inteligência Artificial e Machine Learning na Monitorização de Microserviços	28
3.5.2	Observabilidade Previsiva e Automação da Monitorização	29
3.5.3	Tecnologias Emergentes para Monitorização em Microserviços	29

4	Implementação Prática de Observabilidade com OpenTelemetry	30
4.1	Introdução e Caracterização do Sistema Observado	30
4.1.1	Visão geral da Arquitetura do Sistema R2UT	30
4.1.2	Papel do Kubernetes na Arquitetura do R2UT	31
4.1.3	Arquitetura de Microserviços no Kubernetes	31
4.1.4	Desafios da Observabilidade em Sistemas Distribuídos	32
4.2	Objetivos e Justificativa da Implementação	33
4.2.1	Objetivos Práticos	33
4.3	O Papel do OpenTelemetry na Observabilidade de Microserviços	34
4.3.1	O que é o OpenTelemetry	34
4.3.2	Principais Componentes (Visão Geral)	34
4.3.3	Sinais, Convenções Semânticas e Recursos	35
4.3.4	Vantagens do OpenTelemetry	35
4.4	Comparação entre OpenTelemetry e a Stack Tradicional de Observabilidade (Prometheus, Grafana, Jaeger e Loki)	36
4.5	Arquitetura Detalhada e Implementação da Solução	37
4.5.1	Visão Geral da Arquitetura	37
4.5.2	Fluxo de Telemetria	39
4.6	Detalhes Técnicos da Implementação	40
4.6.1	Estratégia de Instrumentação em .NET	40
4.6.2	Vantagens da Abstração da Instrumentação OpenTelemetry	41
4.6.3	Instrumentação Específica do ASP.NET Core	42
4.7	Coleta de dados com o OpenTelemetry Collector	42
4.7.1	Protocolo OTLP (gRPC/HTTP) e Configuração	43
4.7.2	OpenTelemetry Operator	43
4.7.3	Abordagem Adotada: DaemonSet (Agente por nodo)	45
4.8	Processamento e Transformação dos Dados	46
4.8.1	Processadores e as suas Aplicações	47
4.9	Persistência e Armazenamento de Dados	50
4.9.1	Armazenamento de Logs com o Loki	50
4.9.2	Armazenamento de Traces com o Jaeger	50
4.9.3	Armazenamento de Métricas com o Prometheus	50

5	Visualização e Análise de Dados	51
5.1	Visualização centralizada no Grafana	51
5.2	Organização dos Dashboards	52
5.2.1	Paineis e Dashboards	52
5.3	Correlação entre Logs e Traces	54
5.4	Alertas	55
6	Conclusões e Trabalho Futuro	56
6.1	Conclusões	56
6.2	Trabalho Futuro	56
II	Apêndices	61

Lista de Figuras

1	Comparação entre arquitetura monolítica e arquitetura de microsserviços	8
2	Comparação entre arquitetura SOA e arquitetura de microsserviços	10
3	Arquitetura da Solução	38
4	Implementação de DaemonSet Collector	45
5	Painel global de Logs	52
6	Painel de Logs de Erro	53
7	<i>Dashboard de Logs</i>	53
8	Uso de CPU e Memória	53
9	Dashboard de mettricas: p95, Taxa de Erro, Pedidos por segundo	54
10	Log: Correlação Log e Trace	54
11	Link para Trace	55
12	Trace: Correlação Log e Trace	55

Lista de Tabelas

1	Comparação entre Arquitetura Monolítica, SOA e Microserviços	14
2	Comparação de Ferramentas de Monitorização	25
3	Comparação: OpenTelemetry (OTel) vs Prometheus + Grafana + Jaeger	37
4	Variáveis de ambiente para configuração OTLP Exporter (colocar as em uso)	41
5	Processadores Essenciais do OpenTelemetry Collector	47

Parte I

Material Introdutório

Capítulo 1

Introdução

1.1 Contextualização

As arquiteturas de microserviços têm emergido como uma das abordagens mais populares no desenvolvimento de software moderno, possibilitando a criação de sistemas escaláveis, modulares e fáceis de manter [Larrucea et al. \(2018\)](#). No entanto, o caráter distribuído dessa arquitetura introduz desafios significativos na monitorização, *logging* e alerta dos seus componentes, especialmente em ambientes dinâmicos e baseados em *containers*, como os geridos com *Docker* e *Kubernetes* [Liu et al. \(2020\)](#). A necessidade de uma monitorização eficaz torna-se ainda mais crítica em ambientes dinâmicos e baseados em *containers*, nos quais costumam operar aplicações distribuídas, em grande escala, que estão sujeitas a variações constantes nas cargas de trabalho com que têm de lidar. A adoção de estratégias de monitorização é essencial para garantir a estabilidade e o desempenho, permitindo a identificação proativa de anomalias e a resolução eficiente de falhas.

Ferramentas como *Prometheus*, *Grafana*, *ELK Stack* e *Jaeger* são amplamente utilizadas em aplicações de recolha e análise de *logs*, monitorização de métricas e *tracing* distribuído, proporcionando maior visibilidade sobre o comportamento dos serviços em execução. No contexto de aplicações baseadas em microserviços, em que a comunicação entre componentes é altamente distribuída, uma infraestrutura de monitorização desempenha um papel crucial na manutenção da confiabilidade e escalabilidade da plataforma. Assim, garantir a monitorização e o acompanhamento dos serviços e da aplicação desenvolvida permite não apenas detetar rapidamente problemas, mas também implementar respostas automatizadas a eventos críticos, reduzindo o tempo de inatividade e aumentando a eficiência operacional dos sistemas.

1.2 Motivação

O projeto *R2UT (Ready to Use Technology)* teve como principal objetivo impulsionar a transformação digital da indústria da construção civil em Portugal, promovendo a adoção de modelos de construção modular, industrializada e tecnologicamente avançada. Desenvolvido através da colaboração entre empresas e centros de investigação, o projeto procurou criar soluções inovadoras capazes de aumentar a produtividade, reduzir o desperdício e acelerar o processo construtivo, assegurando elevados padrões de qualidade e sustentabilidade.

No âmbito desta iniciativa, foi desenvolvida uma plataforma digital integrada destinada a suportar as diferentes fases do ciclo de vida dos edifícios pré-fabricados, desde o planeamento e conceção até à operação e manutenção. Esta plataforma combinou tecnologias de automação, *Internet of Things (IoT)* e gestão inteligente de dados, permitindo o acompanhamento em tempo real do desempenho dos sistemas e dispositivos distribuídos.

Contudo, a crescente complexidade da arquitetura da plataforma e o número elevado de serviços distribuídos introduziram novos desafios relacionados com a monitorização, deteção de falhas e gestão do desempenho. Problemas como falhas na comunicação entre serviços, anomalias de desempenho e limitações de escalabilidade podiam comprometer a fiabilidade da infraestrutura e a integridade dos dados captados pelos dispositivos conectados [Barakat \(2017\)](#).

Neste contexto, esta dissertação teve como foco o desenvolvimento de uma solução de monitorização e gestão de alertas para a plataforma *R2UT*, com o objetivo de garantir observabilidade, estabilidade e eficiência operacional. A solução proposta foi concebida de forma robusta e escalável, permitindo a deteção rápida de falhas e a implementação de respostas automáticas a eventos críticos.

Para tal, foi desenvolvida uma plataforma de monitorização baseada numa arquitetura de microserviços, responsável pela recolha, centralização e análise de *logs*, métricas e *tracing* distribuído, através da integração de ferramentas amplamente utilizadas no ecossistema de observabilidade. O sistema resultante proporciona maior visibilidade sobre o comportamento dos serviços e componentes da aplicação *R2UT*, contribuindo para um ambiente seguro, resiliente e de fácil manutenção, em alinhamento com os objetivos do projeto.

1.3 Objetivos

Este trabalho visa desenvolver uma plataforma de monitorização e alarmística para o projeto R2UT, assegurando uma gestão centralizada e em tempo real de microserviços através de componentes *open source*. Além de permitir respostas automatizadas a cenários críticos, a plataforma incluirá um *dashboard* interativo para análise e filtragem avançada dos dados de monitorização e *logs*, promovendo escalabilidade e resiliência no ambiente modular. Para este trabalho de dissertação foram estabelecidos os seguintes objetivos:

- Desenvolver uma plataforma de monitorização e alarmística para o projeto R2UT, utilizando componentes *open source* com licenças de utilização aberta (como MIT ou Apache 2.0), garantindo segurança, escalabilidade e eficiência [Mayer and Weinreich \(2017\)](#).
- Garantir a monitorização dos microserviços da infraestrutura, proporcionando uma visão unificada e em tempo real das operações.
- Estudar e implementar uma estrutura de centralização de *logs* para recolher e consolidar *logs* de todos os microserviços, facilitando a supervisão do fluxo de dados e a identificação de anomalias [Cinque et al. \(2022\)](#).
- Incluir funcionalidades de resposta automatizada para acionar ações específicas em cenários críticos, como o escalonamento automático (*autoscaling*) de serviços ou a execução de correções automáticas.
- Desenvolver um *dashboard* intuitivo e interativo para visualização, análise e aplicação de filtros avançados nos dados de monitorização e *logs*, permitindo uma análise precisa e personalizável.

1.4 Trabalho Realizado

Ao longo deste trabalho, foi concebida e implementada uma plataforma de monitorização e gestão de alertas para o projeto R2UT, com o propósito de reforçar a observabilidade e a capacidade de supervisão da sua infraestrutura de microserviços. A solução foi desenvolvida com recurso a componentes *open source* sob licenças de utilização aberta, como MIT ou Apache 2.0, garantindo elevados níveis de segurança, escalabilidade e eficiência [Mayer and Weinreich \(2017\)](#).

A plataforma proposta permitiu centralizar a monitorização dos microserviços, oferecendo uma visão consolidada e em tempo real do estado operacional do sistema. Para suportar esta monitorização, foi estudada e implementada uma estrutura de centralização de *logs*, responsável pela recolha, agregação e análise dos registos gerados pelos diferentes serviços. Esta abordagem possibilitou a supervisão contínua do fluxo de dados, bem como a deteção de anomalias e falhas na execução dos componentes distribuídos [Cinque et al. \(2022\)](#).

Complementarmente, foram desenvolvidos *dashboards* interativos e de fácil utilização, que permitem a visualização e análise detalhada das métricas, *traces* e dos *logs* recolhidos. Este painel oferece funcionalidades avançadas de filtragem e exploração de dados, facilitando a interpretação do comportamento dos serviços e suportando a tomada de decisões operacionais fundamentadas.

A solução implementada contribuiu de forma significativa para a melhoria da visibilidade e resiliência da plataforma R2UT, promovendo uma gestão mais eficiente e proativa dos serviços num ambiente modular, escalável e distribuído.

1.5 Estrutura do Documento

Além deste capítulo introdutório, a presente dissertação encontra-se organizada da seguinte forma:

- **Capítulo 2 – Arquiteturas de Microserviços**

Este capítulo apresenta a evolução e os fundamentos das arquiteturas de microserviços, explorando a sua emergência como paradigma moderno no desenvolvimento de sistemas distribuídos. São discutidos os princípios que regem este modelo arquitetônico, a sua comparação com arquiteturas monolíticas e orientadas a serviços, bem como os desafios técnicos e organizacionais associados. Por fim, aborda-se a adoção de microserviços em contextos de larga escala e em ambientes de computação em nuvem.

- **Capítulo 3 – Monitorização e Observabilidade em Microserviços**

Este capítulo analisa a importância da monitorização em sistemas distribuídos e introduz o conceito de observabilidade, sustentado nos seus três pilares fundamentais: *logs*, métricas e *tracing*. São descritas as principais ferramentas e técnicas utilizadas neste domínio, nomeadamente Prometheus, Grafana, Loki/ELK, Jaeger e OpenTelemetry. Adicionalmente, discutem-se os desafios atuais e tendências emergentes, incluindo a integração de abordagens baseadas em Inteligência Artificial para Operações (*AIOps*).

- **Capítulo 4 – Implementação da Solução de Observabilidade**

Neste capítulo é apresentada a implementação prática da solução proposta, abordando os desafios inerentes à orquestração de *containers* e à integração de ferramentas avançadas de monitorização. São explorados conceitos como *tracing* distribuído, padrões de resiliência, definição de alertas e práticas de observabilidade, fundamentais para assegurar a estabilidade e eficiência de sistemas baseados em microserviços.

- **Capítulo 5 – Conclusões e Trabalho Futuro**

Este capítulo apresenta as conclusões do trabalho desenvolvido, refletindo sobre os resultados obtidos e os desafios enfrentados. São também discutidas as contribuições do estudo para o projeto R2UT e para o avanço do conhecimento na área da observabilidade de sistemas distribuídos, bem como as perspetivas de evolução e as linhas de trabalho futuro.

- **Capítulo 6 – Próximos Passos**

Capítulo 2

Microserviços

Nos últimos anos, as arquiteturas de microserviços tornaram-se uma das abordagens mais populares no desenvolvimento de sistemas de software escaláveis e resilientes. A transformação arquitetural que este tipo de abordagem provocou, impulsionou nas organizações uma crescente necessidade para inovar rapidamente, para que fossem capazes de atender a requisitos de escalabilidade global e responder com total agilidade às constantes mudanças do mercado no qual se inserem. A mudança tecnológica assentou na evolução de arquiteturas monolíticas para sistemas compostos por múltiplos serviços independentes. Uma evolução que reflete uma mudança organizacional como também cultural nas organizações. Neste capítulo abordamos o surgimento dos microserviços, a sua evolução histórica e o seu posicionamento no contexto das arquiteturas de software atuais. Em particular, discutir-se-ão alguns dos desafios inerentes à sua adoção, os seus conceitos fundamentais e o percurso que levou à sua popularização no domínio da Engenharia de Software.

2.1 Emergência e Evolução

2.1.1 Limitações das Arquiteturas Monolíticas

Antes da emergência dos microserviços, a maioria das aplicações empresariais eram desenvolvidas seguindo uma arquitetura monolítica, na qual todos os componentes do sistema - interface de utilizador, lógica de negócio e acesso a dados - estão integrados num único bloco de código. Uma única “peça” de software. Embora esta abordagem simplifique o desenvolvimento inicial, à medida que a aplicação vai crescendo e evoluindo vão surgindo vários problemas devido a essa tão grande concentração de serviços num único sistema [Villamizar et al. \(2015\)](#). A Figura 1 apresenta uma comparação estrutural entre uma arquitetura monolítica e uma arquitetura baseada em microserviços. Entre os principais problemas identificados destacam-se:

- Dificuldade de escalar equipas de desenvolvimento. Diferentes equipas precisam de trabalhar no mesmo código, o que provoca frequentemente conflitos e a necessidade de uma coordenação intensiva.
- Ciclo de *deployment* prolongados. A necessidade de testar e distribuir toda a aplicação torna os processos de atualização complexos e arriscados;
- Falta de resiliência. A ocorrência de uma falha, num único componente, pode comprometer toda a aplicação, o que pode gerar uma interrupção generalizada dos serviços do sistema.

Estas limitações tornaram-se ainda mais evidentes com o avanço da computação em nuvem e a exigência por uma disponibilização contínua de serviços.

A Figura 1 evidencia estas diferenças, mostrando como, numa arquitetura monolítica, todos os módulos se encontram num único artefacto, enquanto na arquitetura de microserviços cada componente opera de forma independente, comunicando através de um API Gateway e podendo utilizar bases de dados próprias.

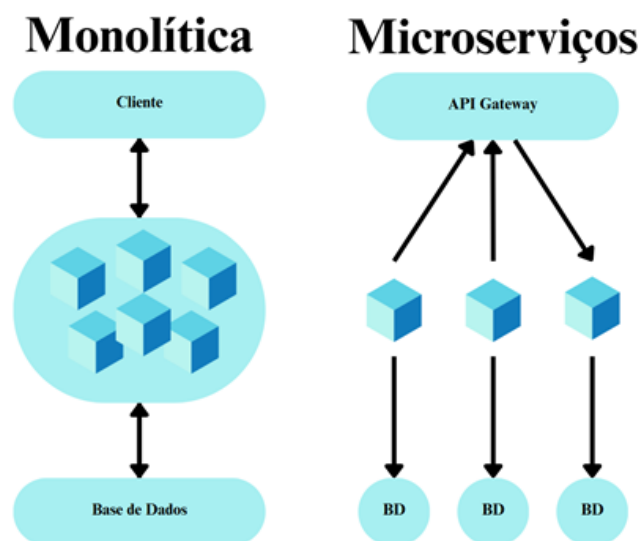


Figura 1: Comparação entre arquitetura monolítica e arquitetura de microserviços

2.1.2 De SOA a microserviços

O conceito de decompor aplicações monolíticas em serviços autónomos não é novo. As Arquiteturas Orientadas a Serviços (SOA) surgiram no final dos anos 1990 e início dos anos 2000, como uma primeira forma de abordar os problemas impostos pelas arquiteturas monolíticas. Numa arquitetura deste tipo, as

aplicações são organizadas como uma coleção de serviços que interagem por meio de um “barramento” de mensagens - *Enterprise Service Bus* (ESB). Um ESB é um sistema de *middleware* que permite a comunicação entre serviços distintos numa SOA. O ESB atua como um único intermediário central que gere toda a integração, faz o encaminhamento de mensagens, a transformação de dados e aplica as políticas de segurança definidas para os vários serviços do sistema. Embora um ESB simplifique a integração inicial, a sua centralização cria dependências e um potencial ponto de falha do sistema. Fragilidades como estas, fizeram com que se procurassem alternativas mais descentralizadas, como as arquiteturas baseadas em microsserviços [Aziz et al. \(2020\)](#).

Embora as SOA tenham introduzido avanços significativos na modularização de sistemas, também criaram alguns desafios consideráveis, nomeadamente:

- Complexidade excessiva. A utilização de ESB centralizados introduziu um ponto único de falha e complexidade operacional;
- Rigidez nos contratos de serviços. A realização de alterações nos serviços do sistema exigiam mudanças pesadas no barramento e nos consumidores.
- Foco excessivo em tecnologias pesadas. Os padrões SOAP e WS-*, por exemplo, tornaram as integrações difíceis e pouco ágeis.

A arquitetura de microsserviços pode ser vista como uma evolução pragmática das SOA, mas focada na simplicidade, na independência e na automação de operações. Numa arquitetura de microsserviços, o barramento central é eliminado. Cada serviço comunica diretamente com os outros serviços, o que permite eliminar muitas das complexidades associadas aos tradicionais sistemas orientados a serviços.

A Figura 2 ilustra estas diferenças, evidenciando a centralização do ESB nas arquiteturas SOA em contraste com a comunicação distribuída e independente entre microsserviços.

2.1.3 Fatores Tecnológicos e Organizacionais

O surgimento dos microsserviços não pode ser atribuído apenas a fatores técnicos. Os fatores organizacionais também desempenharam um papel crucial nesse processo. Três dos movimentos principais que impulsionaram esta evolução foram [Newman \(2015\)](#):

- **Computação em Nuvem.** A elasticidade da nuvem permitiu que aplicações fossem dimensionadas dinamicamente, o que incentivou arquiteturas a tirar partido dessa flexibilidade. Os microsserviços encaixam naturalmente nesse modelo, permitindo escalar apenas os componentes necessários.

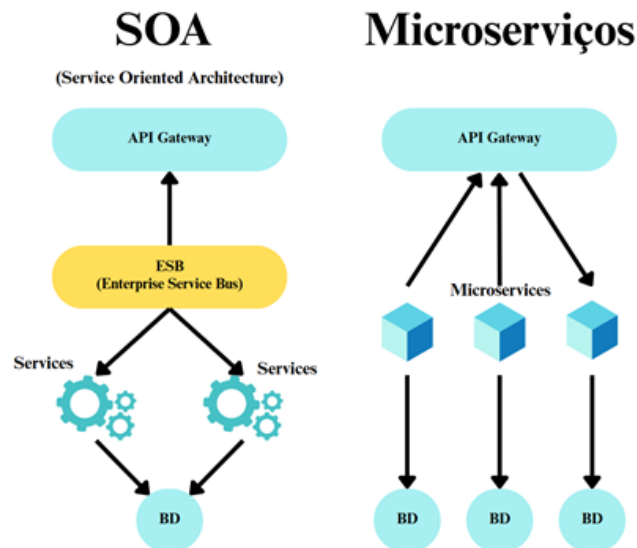


Figura 2: Comparação entre arquitetura SOA e arquitetura de microserviços

- **DevOps e Deployment Contínuo.** A cultura *DevOps* enfatizou a necessidade de integrar desenvolvimento e operações, automatizar pipelines de entrega contínua e reduzir ciclos de feedback. Os microserviços permitem ciclos de desenvolvimento independentes para cada serviço, o que os permite alinhar com esses princípios.
- **Containers e gestão de containers** Tecnologias como *Docker* ou *Kubernetes* simplificaram significativamente a criação, o *deploy* e a gestão de serviços independentes, tornando viável, em larga escala, o modelo dos microserviços.

Segundo [Lewis \(2014\)](#), a capacidade de alinhar arquitetura de software com estruturas organizacionais ágeis, inspiradas na "Lei de Conway", foi um dos principais catalisadores para a adoção dos microserviços. A "Lei de Conway", formulada por Melvin Conway em 1968, estabelece que "any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure" [Bailey et al. \(2013\)](#) ("qualquer organização que projeta um sistema (em sentido amplo) inevitavelmente produzirá um design cuja estrutura é uma cópia da estrutura de comunicação da organização"). Essa observação implica que as estruturas organizacionais moldam, de maneira direta ou indireta, a arquitetura dos sistemas que desenvolvem.

2.1.4 Popularização dos Microserviços

O termo *microservices* começou a ganhar popularidade em conferências técnicas por volta de 2011-2012, sendo posteriormente popularizado pelos trabalhos de autores como James Lewis, Martin Fowler e Sam

Newman [Lewis \(2014\)](#); [Newman \(2015\)](#). Empresas pioneiras como Netflix, Amazon, e Uber demonstraram publicamente os benefícios de arquiteturas baseadas em microsserviços, mostrando que era possível construir sistemas resilientes, escaláveis e altamente disponíveis a partir da composição de múltiplos serviços pequenos e independentes. A experiência e dimensão dessas empresas inspirou uma grande adoção no setor, suportada por uma nova geração de ferramentas de monitorização e gestão distribuída, plataformas de infraestrutura como serviço (IaaS) e metodologias ágeis de desenvolvimento. Atualmente, os microsserviços são amplamente reconhecidos como uma escolha estratégica para sistemas que exigem alta escalabilidade, independência organizacional e ciclos de entrega rápidos [Dragoni et al. \(2017\)](#). No entanto, essa popularidade não elimina a complexidade técnica e organizacional que a arquitetura de microsserviços impõe, tema que será aprofundado nas próximas secções.

2.2 O que são microsserviços?

Após as limitações evidenciadas pelas arquiteturas monolíticas e a emergência de novos paradigmas tecnológicos e organizacionais, os microsserviços consolidaram-se como uma abordagem inovadora para o desenvolvimento de sistemas distribuídos modernos.

Nesta secção caracteriza-se a arquitetura de microsserviços, destacando as suas principais propriedades no cenário atual de engenharia de software.

2.2.1 Definição Formal

Um microsserviço é uma unidade modular de software criada com o intuito de executar uma funcionalidade específica integrada num sistema maior. Os microsserviços são independentes e autónomos, podendo assim serem desenvolvidos, testados e escalados separadamente de qualquer outro componente do sistema [Jamshidi et al. \(2018\)](#). A principal característica dos microsserviços é a separação de responsabilidades. Cada serviço tem o seu propósito no contexto do sistema geral e, por isso, deve executar apenas o seu código, focando-se num único problema, aquilo que realmente lhe compete [Newman \(2015\)](#). Estes serviços são flexíveis e altamente escaláveis. Além disso permitem a utilização de tecnologias distintas na sua implementação; bem como o desenvolvimento paralelo de serviços e dimensionamento granular por serviço (isto é, a capacidade de escalar apenas os serviços que efetivamente necessitam de mais recursos) [Lewis \(2014\)](#). Mais do que o tamanho do código, o termo "micro" enfatiza a responsabilidade limitada de cada serviço e a independência operacional, permitindo que estes sejam desenvolvidos, implementados e escalados de maneira isolada. Segundo [Dragoni et al. \(2017\)](#) o conceito

de microsserviços surgiu como um refinamento de princípios preexistentes, como modularidade, separação de responsabilidades e princípios do desenvolvimento das SOA, mas com ênfase na autonomia e no alinhamento com domínios de negócio.

2.2.2 Princípios e Principais Características

Podemos caracterizar uma arquitetura de microsserviços como tendo as seguintes características:

- **Autonomia de desenvolvimento e deployment.** Cada microsserviço pode ser desenvolvido, testado, implementado e mantido de forma independente [Newman \(2015\)](#).
- **Especialização funcional.** Os serviços são organizados em torno de capacidades de negócio específicas, refletindo o princípio de responsabilidade única [Newman \(2015\)](#).
- **Comunicação leve.** Os microsserviços utilizam protocolos de comunicação simples, como REST sobre HTTP, gRPC ou filas de mensagens assíncronas [Dragoni et al. \(2017\)](#).
- **Independência tecnológica.** Os serviços podem ser implementados em diferentes linguagens de programação ou *frameworks*, promovendo poliglotismo arquitetural [Richardson \(2018\)](#).
- **Escalabilidade granular.** Serviços com maior procura podem ser escalados individualmente, permitindo alocação eficiente de recursos [Lewis \(2014\)](#).
- **Observabilidade.** Cada serviço é projetado para expor métricas, logs e *tracing* distribuído, possibilitando monitorização e diagnóstico isolados [Soldani et al. \(2018\)](#).

Este conjunto de princípios alinha a arquitetura de microsserviços a práticas modernas de desenvolvimento ágil, DevOps e computação em nuvem.

2.2.3 Componentes Típicos em Arquiteturas de microsserviços

Geralmente, a implementação prática de microsserviços envolve diversos componentes arquiteturais adicionais, nomeadamente:

- **APIs Públicas.** Cada serviço expõe a sua funcionalidade por meio de uma interface bem definida, normalmente baseada em padrões como RESTful APIs ou gRPC.
- **As Base de Dados são Privadas.** Cada microsserviço é responsável pela sua própria persistência de dados, evitando assim dependências diretas entre serviços [Dragoni et al. \(2017\)](#).

- **Mensagens Assíncronas.** A comunicação baseada em eventos, utilizando tecnologias como Kafka, RabbitMQ ou SQS, reduz o acoplamento entre serviços e facilita a escalabilidade horizontal.
- **Service Discovery e Load Balancing.** Estes mecanismos automáticos são utilizados para fazerem a localização e o balanceamento de serviços dinâmicos em ambientes distribuídos [Newman \(2015\)](#).
- **API Gateway.** Uma API que serve para unificar o acesso externo aos serviços e gerir autenticação, encaminhamento, caching e controlo de versões [Richardson \(2018\)](#).

Estes componentes são fundamentais para garantir que uma arquitetura baseada em microsserviços seja robusta, escalável e de fácil manutenção.

2.2.4 Comparação com outras Arquiteturas

Após a apresentação dos princípios e componentes fundamentais da arquitetura de microsserviços, torna-se pertinente posicioná-la relativamente a outras abordagens arquiteturais, como o modelo monolítico e a SOA. Esta comparação é essencial para compreender as diferenças estruturais e organizacionais que motivam a adoção dos microsserviços em determinados contextos. Embora as três abordagens procurem suportar a construção de sistemas robustos e escaláveis, estas diferem significativamente quanto ao seu âmbito funcional, à forma de comunicação entre componentes e à autonomia de desenvolvimento e operação. Enquanto o modelo monolítico centraliza toda a aplicação num único bloco, com forte acoplamento interno, e a SOA tradicional procura modularizar sistemas através de serviços de grande escala coordenados por infraestruturas centrais (como Enterprise Service Buses — ESBs), a arquitetura de microsserviços distingue-se pela sua granularidade fina, descentralização operacional e leveza na comunicação entre componentes.

Segundo [Newman \(2015\)](#) e [Dragoni et al. \(2017\)](#), os microsserviços são concebidos para que cada serviço corresponda a uma capacidade de negócio específica, podendo ser desenvolvido, implementado e escalado de forma totalmente autónoma, sem dependências centralizadas.

A Tabela 1 sintetiza as principais diferenças entre as três abordagens arquiteturais, destacando aspetos como o âmbito funcional, modelo de comunicação, estratégia de *deployment*, gestão de dados e autonomia de desenvolvimento. Esta comparação permite observar de forma clara a evolução de um modelo fortemente centralizado para arquiteturas cada vez mais distribuídas e independentes, evidenciando o papel dos microsserviços enquanto paradigma orientado à escalabilidade organizacional e técnica.

Tabela 1: Comparação entre Arquitetura Monolítica, SOA e Microserviços

Característica	Arquitetura Mono- lítica	Arquitetura SOA	Arquitetura de Mi- croserviços
Âmbito funcional	Abrangente e inte- grado	Serviços de grande escala	Serviços pequenos e focados
Comunicação	Interna (memória lo- cal)	Middleware corpora- tivo (ESB)	APIs leves (HTTP/- gRPC)
<i>Deployment</i>	Único e centralizado	Parcial, frequente- mente acoplado ao ESB	Independente por ser- viço
Dados	Centralizados	Parcialmente descen- tralizados	Totalmente descen- tralizados
Autonomia no desen- volvimento	Reduzida	Moderada	Elevada

2.3 Os Desafios das Arquiteturas de Microserviços

A arquitetura de microserviços representa uma mudança paradigmática no desenvolvimento de sistemas de software, uma promessa de maior escalabilidade, flexibilidade e capacidade de inovação. Porém, na prática, a construção e a operação de sistemas baseados em microserviços trazem consigo um conjunto de desafios técnicos e organizacionais que não podem ser ignorados. De seguida, serão analisados os principais aspetos relacionados com a organização de sistemas de microserviços, bem como os desafios emergentes da sua adoção em larga escala.

2.3.1 Organização de Sistemas de Microserviços

Os sistemas baseados em microserviços são compostos por um conjunto de serviços pequenos, especializados e autonomamente desenvolvidos. A sua organização não se limita à existência de vários serviços independentes, mas exige uma conceção cuidadosa das relações entre serviços, das suas formas de comunicação e da delimitação das suas fronteiras [Railic and Savic \(2021\)](#); [Lewis \(2014\)](#).

A comunicação entre microserviços pode ser síncrona, utilizando APIs RESTful ou gRPC, ou assíncrona, através de sistemas de filas, como Apache Kafka ou RabbitMQ. A comunicação síncrona facilita o

desenvolvimento inicial, mas introduz dependências temporais entre serviços, enquanto a comunicação assíncrona promove maior tolerância a falhas, embora acrescente alguma complexidade na gestão da consistência dos dados e na monitorização de transações distribuídas.

A gestão das fronteiras de serviços é um aspeto crucial. Um serviço deve encapsular uma capacidade de negócio bem definida, evitando tanto o excesso de granularidade, que aumenta a complexidade operacional, como a agregação de múltiplas funcionalidades distintas, que reintroduz problemas típicos de arquiteturas monolíticas.

Técnicas como *Domain-Driven Design* (DDD) [Rogers \(2022\)](#) são frequentemente utilizadas para identificar limites de contexto adequados e promover o bom funcionamento interno de cada serviço. A dependência de componentes como API Gateways, mecanismos de *service discovery* e comunicação assíncrona, referidos anteriormente, não deve ser vista apenas como um requisito tecnológico, mas como uma estratégia fundamental para garantir a escalabilidade, resiliência e segurança dos serviços.

2.3.2 Principais Desafios Técnicos

Apesar das vantagens teóricas, a implementação prática de sistemas baseados em microserviços traz um conjunto significativo de desafios técnicos que se intensificam com o aumento da complexidade e da escala do sistema. A comunicação distribuída é um dos principais pontos de fragilidade: a utilização de redes para interligar serviços introduz atrasos variáveis, falhas de ligação e a necessidade de implementar mecanismos de *retry*, *circuit breaker* e *timeouts* devidamente controlados [NYGARD \(2018\)](#); [Newman \(2015\)](#). Além disso, a gestão de APIs torna-se crítica, exigindo práticas rigorosas de versionamento para evitar incompatibilidades em tempo de execução [Richardson \(2018\)](#).

A gestão de dados distribuídos constitui outro desafio importante. Ao promover a descentralização dos repositórios de dados, a arquitetura de microserviços inviabiliza a utilização de transações ACID tradicionais entre serviços. Como alternativa, padrões como *Sagas* ? e a adoção de consistência eventual tornam-se necessários, aumentando a complexidade do desenvolvimento e das operações. A monitorização de sistemas distribuídos exige uma abordagem abrangente de observabilidade: métricas detalhadas por serviço, logs estruturados e *tracing* distribuído são essenciais para a deteção precoce de problemas e para a análise eficaz de incidentes [Burns \(2015\)](#). Ferramentas como Prometheus, Grafana e Jaeger têm sido amplamente utilizadas para este fim, mas exigem configuração e manutenção especializadas.

A gestão de *deployments* e versões em ambientes de microserviços também se torna mais complexa. A coordenação de atualizações entre serviços dependentes, a manutenção da compatibilidade de APIs e a implementação de estratégias de *deployment* seguras, como *blue-green deployments* e *canary releases*,

são práticas indispensáveis para reduzir o risco de interrupções de serviço [Humble and Farley \(2010\)](#).

2.3.3 Principais Desafios Organizacionais

Para além das dificuldades técnicas, a adoção de microsserviços implica grandes transformações na organização das equipas de desenvolvimento e na cultura empresarial. A autonomia das equipas é um dos princípios fundamentais desta abordagem: cada equipa deve ser responsável pelo ciclo de vida completo dos serviços que desenvolve, desde a conceção até à operação em produção. Esta autonomia reduz a necessidade de coordenação centralizada, mas exige uma forte disciplina na gestão de interfaces e na comunicação entre equipas. A Lei de Conway ensina que a arquitetura dos sistemas tende a refletir a estrutura de comunicação da organização [Bailey et al. \(2013\)](#). Assim, para beneficiar das vantagens dos microsserviços, é necessário que as fronteiras organizacionais estejam alinhadas com as dos serviços, promovendo equipas pequenas, multifuncionais e responsáveis por domínios de negócio bem delimitados.

A maturidade em práticas de DevOps é outro requisito essencial. A automação de pipelines de integração e entrega contínuas, a gestão centralizada de configurações e a monitorização proativa são práticas indispensáveis para garantir a eficácia operacional em ambientes de microsserviços. Organizações que não possuam essa maturidade tendem a enfrentar dificuldades na gestão da complexidade e na manutenção da fiabilidade dos sistemas [Lewis \(2014\)](#).

Finalmente, a cultura de responsabilização deve ser reforçada: cada equipa não deve apenas entregar código funcional, mas assumir a responsabilidade contínua pela qualidade, desempenho e estabilidade dos seus serviços em produção. Este paradigma, frequentemente resumido na expressão *"you build it, you run it"*, requer mudanças culturais significativas e um compromisso claro com a excelência operacional [Khan et al. \(2022\)](#).

2.4 Arquiteturas de microsserviços em Grande Escala

A implementação de microsserviços em grande escala apresenta desafios únicos em termos de escalabilidade, resiliência e orquestração. Com o aumento da complexidade das aplicações, as soluções tradicionais baseadas em arquiteturas monolíticas não são suficientes para suportar exigências de alta disponibilidade e escalabilidade. Para lidar com sistemas complexos compostos por centenas ou milhares de microsserviços, é fundamental adotar práticas e tecnologias específicas que garantam o bom funcionamento e a escalabilidade das plataformas.

2.4.1 Escalabilidade Horizontal

A escalabilidade horizontal é um dos principais benefícios oferecidos pelos microsserviços, permitindo que os serviços sejam escalados individualmente conforme a demanda. Esta abordagem contrasta com a escalabilidade vertical, comum em sistemas monolíticos, onde a capacidade do sistema é aumentada através do reforço de um único componente [Blinowski et al. \(2022\)](#). Nos microsserviços, cada serviço pode ser replicado de forma independente para lidar com picos de carga sem impactar outros serviços.

A gestão da escalabilidade horizontal em ambientes distribuídos exige ferramentas de orquestração como o Kubernetes, que permitem o dimensionamento automático dos serviços com base em métricas de desempenho em tempo real. O Kubernetes facilita a criação, gestão e monitorização de *containers*, permitindo que os microsserviços sejam escalados automaticamente de acordo com a carga de trabalho [Rocha et al. \(2023\)](#). Este tipo de escalabilidade garante que os sistemas sejam capazes de lidar com grandes volumes de tráfego sem sobrecarregar recursos ou comprometer a disponibilidade.

2.4.2 Gestão do Estado dos Serviços

Em sistemas distribuídos, a gestão do estado dos serviços é um desafio crítico. No modelo de microsserviços, é comum que cada serviço tenha a sua própria base de dados, promovendo a descentralização do armazenamento. Embora esta abordagem permita maior flexibilidade e agilidade no dimensionamento dos serviços, ela também traz desafios no que diz respeito à consistência dos dados.

A descentralização dos dados pode exigir o uso de técnicas como *event sourcing* e *CQRS* (Command Query Responsibility Segregation), que ajudam a garantir a integridade dos dados entre os serviços [Richardson \(2018\)](#). Além disso, a sincronização entre serviços independentes pode ser complexa, especialmente quando se lida com falhas de rede e inconsistências temporárias. O uso de comunicação assíncrona e sistemas de filas de mensagens, como Kafka ou RabbitMQ, permite que os microsserviços comuniquem de forma eficiente mesmo em cenários com alta latência ou falhas temporárias [Dragoni et al. \(2017\)](#).

2.4.3 Orquestração e Automação

A orquestração e automação são fundamentais para a gestão de microsserviços em grande escala. Ferramentas como o Kubernetes não só gerem a criação e escalabilidade dos serviços, mas também garantem mecanismos de *self-healing*, reiniciando ou substituindo automaticamente serviços que falham, minimizando o impacto nos utilizadores finais [Burns et al. \(2016\)](#).

Além disso, a automação do processo de *deployment* é um fator crítico. Tecnologias de *CI/CD* (Integração Contínua/Entrega Contínua), combinadas com Kubernetes e Docker, possibilitam o *deployment* contínuo e a validação automática de novas versões dos serviços, garantindo atualizações rápidas e seguras, sem interrupção do funcionamento do sistema [Taherizadeh and Grobelnik \(2020\)](#).

Em suma, a implementação de microsserviços em grande escala exige uma abordagem estratégica que combine escalabilidade horizontal eficiente, gestão robusta de estado e orquestração automatizada. Ferramentas como Kubernetes, Docker e sistemas de mensagens assíncronas desempenham um papel fundamental na gestão e operação destas arquiteturas complexas, garantindo que plataformas baseadas em microsserviços possam atender aos requisitos modernos de alta disponibilidade e resiliência.

2.5 microsserviços e Computação em Nuvem

A integração de microsserviços com plataformas de computação em nuvem transformou a forma como as empresas desenham e operam os seus sistemas. A nuvem oferece uma infraestrutura elástica que facilita a escalabilidade dinâmica, a gestão simplificada e a alta disponibilidade, características essenciais para ambientes de microsserviços. Esta secção explora a relação entre microsserviços e computação em nuvem, destacando as vantagens, desafios e oportunidades que surgem com o uso de tecnologias de nuvem.

2.5.1 Arquitetura Serverless

O paradigma *serverless* representa uma evolução dos microsserviços, na qual a gestão da infraestrutura é totalmente abstraída pelo provedor de nuvem. Numa arquitetura *serverless*, como as oferecidas pelo AWS Lambda, Azure Functions e Google Cloud Functions, as equipas de desenvolvimento podem focar-se apenas na lógica de negócio, sem se preocupar com a gestão de servidores, escalabilidade ou manutenção da infraestrutura subjacente [Dragoni et al. \(2017\)](#). Embora o *serverless* ofereça grande flexibilidade e escalabilidade automática, também apresenta desafios, especialmente quando se lida com tempos de execução curtos e recursos limitados, que podem afetar a performance em sistemas complexos. Ainda assim, a adoção de microsserviços *serverless* permite reduzir significativamente o custo de operação, dado que os utilizadores pagam apenas pelo tempo de execução dos serviços, tornando-se uma escolha vantajosa para muitas aplicações [Richardson \(2018\)](#).

2.5.2 Custo e Eficiência de Escalabilidade

A escalabilidade de microsserviços na nuvem também traz implicações em termos de eficiência de custos. A nuvem permite que as empresas escalem os seus serviços de acordo com a procura, evitando a necessidade de provisionar recursos fixos. Com ferramentas como o *Auto Scaling* da AWS e o Google Kubernetes Engine (GKE), as empresas podem ajustar dinamicamente a capacidade dos seus serviços para se adaptarem a picos de tráfego e garantir uma utilização de recursos otimizada [Dragoni et al. \(2017\)](#). Esta escalabilidade sob demanda permite uma gestão mais eficiente dos custos operacionais, pois as organizações pagam apenas pelos recursos efetivamente consumidos. Além disso, ao combinar computação em nuvem com a orquestração de microsserviços, as empresas conseguem dimensionar os seus sistemas de maneira eficiente, sem comprometer a performance ou a disponibilidade [Blinowski et al. \(2022\)](#).

2.5.3 Desafios da Computação em Nuvem

Apesar das inúmeras vantagens, a computação em nuvem apresenta desafios que as organizações precisam de abordar ao implementar microsserviços. A dependência de fornecedor (*vendor lock-in*) é um dos principais desafios, pois as organizações podem ficar fortemente dependentes das ferramentas e serviços específicos de um provedor de nuvem. Por exemplo, a portabilidade entre plataformas pode ser limitada, dificultando uma eventual migração para outro fornecedor caso as necessidades da empresa evoluam [Richardson \(2018\)](#).

Outro desafio importante está relacionado com a segurança e a privacidade dos dados, especialmente quando se lida com informação sensível. Embora os fornecedores de nuvem ofereçam mecanismos robustos de segurança, a responsabilidade de garantir configurações adequadas recai sobre a organização, que deve assegurar a proteção de dados em trânsito e em repouso.

Capítulo 3

Como Monitorizar uma Arquitetura de Microserviços

todo: (texto copiado do documento word), meter cites, italicos e acronimos depois da revisao por parte do orientador

A monitorização de sistemas de software é um pilar fundamental para garantir disponibilidade, boa performance e evolução contínua. Com a emergência de arquiteturas de microserviços, a importância da monitorização elevou-se substancialmente, refletindo a complexidade e a natureza altamente distribuída destes sistemas. Ao contrário das arquiteturas monolíticas, onde a observabilidade podia ser alcançada através da análise de poucos componentes centralizados, os microserviços exigem uma abordagem descentralizada, integrando métricas, logs e tracing distribuído para alcançar uma visibilidade de todo o sistema. Este capítulo analisa a importância estratégica da monitorização em microserviços, discute as principais estratégias e ferramentas utilizadas e identifica os desafios inerentes a este novo paradigma arquitetural, fornecendo uma perspetiva crítica e fundamentada sobre o tema.

3.1 A Importância da Monitorização em Microserviços

3.1.1 Diferença entre monitorizar arquiteturas monolíticas e microserviços

Em sistemas monolíticos tradicionais, toda a aplicação é geralmente executada num único processo ou num pequeno conjunto de processos homogêneos (Villa-mizar et al., 2015), a monitorização desses sistemas pode centrar-se em métricas simples como utilização de CPU, tempo de resposta global e disponibilidade de uma base de dados centralizada. Em contraste, em arquiteturas de microserviços, o sistema é composto por dezenas ou centenas de serviços autónomos, cada um com o seu próprio ciclo de vida,

ambiente de execução e armazenamento de dados (Newman, 2015), cada interação entre serviços é uma potencial fonte de falha, e as comunicações distribuídas aumentam significativamente a complexidade de rastrear e diagnosticar problemas. Monitorizar um sistema baseado em microserviços requer, por isso, uma abordagem holística e descentralizada, onde cada serviço deve ser instrumentado individualmente e os dados devem ser agregados de forma coerente para análise e diagnóstico.

3.1.2 Impacto da monitorização na fiabilidade e na escalabilidade

A monitorização eficaz é essencial para:

- Detetar falhas de forma precoce: Pequenas anomalias podem ser indícios de problemas maiores em formação (Burns, 2015);
- Manter a fiabilidade operacional: Ao identificar e isolar serviços degradados rapidamente, evita-se o efeito de cascata de falhas;
- Facilitar a escalabilidade dinâmica: Dados da utilização em tempo real permitem ajustar a capacidade dos serviços conforme necessário, beneficiando plenamente das características da computação em nuvem (Dragoni et al., 2017).

Sem uma infraestrutura de monitorização robusta, as operações em sistemas de microserviços tornam-se arriscadas e insustentáveis.

3.1.3 Principais Objetivos da Monitorização

Os principais objetivos da monitorização em microserviços podem ser resumidos em três grandes áreas (Richardson, 2018):

- Deteção de falhas: Identificar problemas técnicos antes que impactem os utilizadores;
- Medição de desempenho: Avaliar a performance de serviços individuais e do sistema como um todo;
- Tracing de erros: Seguir o percurso de requisições entre serviços para identificar rapidamente o ponto de falha.

3.2 Técnicas, Estratégias e Ferramentas de Monitorização

Existem três pilares clássicos da observabilidade em microserviços (Soldani et al., 2018):

- Logs Centralizados: Registo detalhado de eventos e exceções, estruturados e centralizados num sistema de busca e análise;
- Métricas: Indicadores quantitativos agregados, como taxas de erro, latência e throughput;
- Tracing Distribuído: Rastreo de transações distribuídas que cruzam múltiplos serviços.

Cada pilar fornece uma perspetiva complementar sobre o estado do sistema e, combinados, permitem um diagnóstico completo.

3.2.1 Logs centralizados

Os “logs” são registos escritos de eventos específicos que ocorrem numa aplicação, detalhando o que aconteceu e quando, estes são fundamentais para uma resolução de problemas eficaz e para a compreensão do comportamento do sistema. Numa arquitetura de microserviços, cada serviço independente gera os seus próprios logs. Sem um sistema centralizado, gerir e correlacionar estes registos dispersos torna-se extremamente difícil, dificultando o rastreamento e a depuração de problemas [Soldani and Brogi \(2022\)](#). Um sistema de registo centralizado agrega os registos de várias fontes num único local, permitindo a monitorização em tempo real, a pesquisa avançada e análise visual. A estruturação e a utilização de IDs de correlação são de importância crítica, a formatação de registos (por exemplo, em JSON) com campos consistentes (data/hora, nomes de serviço, códigos de erro) torna-os mais fáceis de analisar automaticamente e de consultar eficientemente. A atribuição de IDs de correlação únicos aos pedidos e a sua propagação por todos os serviços envolvidos numa transação é crucial, estes IDs ligam eventos relacionados entre diferentes microserviços, simplificando a resolução de problemas ao permitir que os engenheiros rastreiem todo o fluxo de um pedido através do sistema distribuído (Fu et al., 2012). Ferramentas como o Elastic Stack (ELK), Grafana Loki, permitem armazenar, processar e visualizar logs de forma eficiente (Bajer, 2017). Para maximizar a utilidade dos logs:

- Utilizar logs estruturados (ex.: formato JSON);
- Incluir correlações como IDs de transação;
- Manter políticas de retenção e rotação de logs bem definidas.

3.2.2 Monitorização de Métricas

As métricas são pontos de dados numéricos recolhidos ao longo do tempo, fornecendo uma visão geral quantificável e de alto nível da saúde e das tendências de desempenho de um sistema. Funcionam como os "indicadores do painel de controlo" que alertam as equipas para potenciais problemas antes que estes se agravem (Burns, 2015). As métricas essenciais para microserviços incluem:

- Latência: O tempo que um microserviço demora a responder a um pedido. Uma latência elevada indica respostas lentas, o que afeta diretamente a experiência do utilizador e os resultados do negócio;
- Débito (Throughput): O número de pedidos que um microserviço consegue processar com sucesso por segundo. Um débito elevado indica a capacidade do serviço para lidar com cargas pesadas, o que é crucial durante os períodos de pico;
- Latência: O tempo que um microserviço demora a responder a um pedido. Uma latência elevada indica respostas lentas, o que afeta diretamente a experiência do utilizador e os resultados do negócio;
- Taxa de Erros: A percentagem de pedidos que falham ou apresentam erros. Este é um indicador direto da saúde de um microserviço, uma vez que os pedidos falhados podem bloquear os utilizadores.

As métricas podem ser categorizadas pelo seu âmbito: de nível de infraestrutura (por exemplo, utilização de CPU, memória, disco), de nível de aplicação (por exemplo, latência de pedidos de serviço, número de pedidos) e métricas de utilizador final (por exemplo, tempos de carregamento da aplicação). A recolha de métricas pode ocorrer através de mecanismos de "push" (os serviços enviam métricas para um servidor central) ou "pull" (os sistemas de monitorização recolhem métricas dos serviços). A captura e análise de métricas são normalmente realizadas através de sistemas como Prometheus. Este sistema recolhe métricas de série temporal de serviços instrumentados, permitindo análises de tendências e geração de alertas com base em limites configuráveis (Burns, 2015).

3.2.3 Tracing Distribuído

O tracing distribuído é um componente essencial e único da observabilidade que rastreia pedidos individuais à medida que fluem através de um sistema complexo e distribuído (Sambasivan et al., 2014).

Proporciona visibilidade de ponta a ponta, e revela o percurso de um pedido através de múltiplos serviços, bases de dados e saltos de rede.(Zhang et al., 2023) Os conceitos fundamentais do rastreamento distribuído incluem:

- Spans: Representam operações individuais dentro de um rastreamento (por exemplo, uma consulta a uma base de dados, uma chamada de API), cada span inclui um nome, tempos de início e fim, e pode ter relações pai-filho com outros spans para mostrar causalidade. Podem também conter etiquetas e registos para contexto adicional (Sambasivan et al., 2014);
- Traces: Uma coleção de spans logicamente conectados que representam o caminho de execução completo de ponta a ponta de um único pedido ou transação através do sistema distribuído (Sambasivan et al., 2014);

Os benefícios chave do tracing distribuído são múltiplos:

- Identificação de Gargalos: Permite identificar exatamente qual serviço ou operação está a causar atrasos ou gargalos de desempenho;
- Depuração de Problemas em Produção: Fornece o contexto necessário para depurar problemas complexos em produção, visualizando todo o fluxo do pedido;
- Otimização de Desempenho: Ajuda a identificar chamadas desnecessárias ou problemas de alta latência na cadeia de serviços;
- Compreensão de Dependências: Mapeia como os serviços se conectam e interagem, oferecendo insights sobre relações de serviço que podem não ser óbvias a partir do código ou de diagramas arquitetónicos.

Para que o tracing funcione eficazmente através dos limites dos serviços, a informação contextual (como o ID do rastreamento) deve ser propagada de um serviço para o seguinte. O OpenTelemetry, mencionado em secções seguintes, surgiu como o novo padrão de código aberto para instrumentação, fornecendo uma forma unificada de recolher dados de telemetria (métricas, registos e rastreamentos) [Thakur and Chandak \(2022\)](#). A adoção do OpenTelemetry garante que os dados recolhidos não estão vinculados a uma plataforma de observabilidade de backend específica oferecendo flexibilidade e preparação para o futuro. Cada serviço propaga informações de tracing em cabeçalhos HTTP ou de RPC, o que permite re-construir o fluxo completo (Sigelman et al., 2010).

Ferramentas populares incluem:

- Jaeger: Uma plataforma open-source para tracing;
- OpenTelemetry: Iniciativa para padronizar a coleta de logs, métricas e traces.

3.2.4 Ferramentas Comuns na Monitorização

Tabela 2: Comparação de Ferramentas de Monitorização

Ferramenta	Tipo de Monitorização	Funcionalidades Principais	Licença
Prometheus	Monitorização de métricas	Coleta de métricas, alertas	Apache 2.0
Grafana	Visualização de métricas	Dashboards interativos	AGPL
ELK Stack	Gestão de logs	Recolha, análise de logs	Apache 2.0
Jaeger	Tracing distribuído	Rastreio de chamadas	Apache 2.0

3.2.5 Conceito de Observabilidade

A observabilidade de um sistema é a capacidade de inferir o seu estado interno apenas a partir de saídas externas (Kalman, 1960).

Num contexto de microserviços, isso implica:

- Apresentar de logs detalhados e coerentes;
- Ter métricas ricas e acionáveis;
- Conseguir rastrear a jornada de uma requisição ponta-a-ponta.

A observabilidade eficaz permite que equipas de operações identifiquem rapidamente a causa e origem de problemas complexos.

3.3 Desafios na Monitorização de Microserviços

3.3.1 Alta Cardinalidade de Dados

Microserviços geram imensas métricas e logs, muitas vezes com altos níveis de cardinalidade (ex.: IDs únicos de utilizadores, IPs, etc.), este grande número de dados pode:

- Saturar bases de dados de séries temporais;
- Dificultar a construção de queries de análise úteis.

Ferramentas como Prometheus lidam mal com métricas com cardinalidade extremamente alta, sendo necessária uma gestão criteriosa da instrumentação.

3.3.2 Correlação de Eventos Distribuídos

A identificação da causa raiz de uma falha muitas vezes exige a correlação de múltiplos Eventos dispersos em logs, métricas e traces. Sem tracing distribuído ou logs estruturados com IDs de correlação, esta tarefa torna-se extremamente difícil (Sigelman et al., 2010).

Estratégias recomendadas:

- Propagação consistente de IDs de tracing;
- Injeção automática de metadados relevantes em logs e métricas.

3.3.3 Problemas de latência e visibilidade

Em sistemas distribuídos, a latência é inevitável e pode surgir em múltiplos pontos, chamadas de serviço, acesso a bases de dados, comunicação de rede. Sem visibilidade detalhada, é impossível identificar rapidamente o local de origem da latência [Railic and Savic \(2021\)](#). Além disso, a monitorização deve ser projetada para ser não intrusiva, ou seja, não pode adicionar overhead significativo que degrade ainda mais o desempenho.

3.4 Estudos de Caso: Exemplos Práticos de Monitorização em Microserviços

A monitorização de microserviços é um desafio significativo para as organizações que implementam esta arquitetura, dada a natureza distribuída e a complexidade que a caracteriza. Empresas como a Netflix, Amazon e Uber são exemplos de sucesso na implementação de sistemas de monitorização em grande escala. Estes casos demonstram como as empresas podem manter a fiabilidade e a escalabilidade dos seus serviços enquanto lidam com a complexidade inerente aos microserviços.

3.4.1 Netflix: Monitorização em Escala Global

A Netflix, pioneira na adoção de microserviços, gere um dos maiores sistemas distribuídos do mundo. Para garantir a disponibilidade e o desempenho para mais de 200 milhões de utilizadores, a empresa desenvolveu ferramentas internas de monitorização. O Atlas, uma plataforma de métricas em tempo real, e o Eureka, um serviço de descoberta, são fundamentais para que os microserviços se localizem e se registem automaticamente, suportando a escalabilidade dinâmica da arquitetura (Newman, 2015). A Netflix elevou a monitorização a um novo patamar com o Chaos Engineering, uma prática que envolve a injeção intencional de falhas no sistema. Testes como a suspensão de serviços ou o aumento de latência permitem à equipa identificar pontos de fragilidade e garantir que a plataforma é capaz de se recuperar rapidamente de falhas imprevistas (Lewis, 2014). A utilização de ferramentas de monitorização como Atlas e Eureka, em conjunto com a prática de Chaos Engineering, contribui para a fiabilidade do sistema, permitindo à Netflix operar em uma escala global com alta disponibilidade.

3.4.2 Amazon: Escalabilidade e Resiliência em Grande Escala

A Amazon, através da sua vasta plataforma de e-commerce e dos serviços da Amazon Web Services (AWS), lida com uma quantidade massiva de transações e dados. A monitorização da sua arquitetura de microserviços é centralizada em ferramentas nativas da nuvem para garantir a eficiência operacional e a resiliência (Dragoni et al., 2017). O Amazon CloudWatch é a ferramenta primária para monitorizar métricas e logs em tempo real, permitindo a criação de alarmes e a optimização automática da utilização de recursos. Complementarmente, o AWS X-Ray fornece uma solução de tracing distribuído que permite seguir a trajetória das requisições entre os múltiplos microserviços. Essa visibilidade de ponta a ponta é crucial para identificar gargalos e falhas, garantindo que a infraestrutura se mantenha robusta e escalável.

(Dragoni et al., 2017).

3.4.3 Uber: Monitorização e Observabilidade para Escalabilidade Global

A Uber, com a sua operação em escala global, precisa de uma plataforma de monitorização eficaz para gerir milhões de transações por minuto. A empresa utiliza uma combinação de ferramentas open-source para alcançar uma observabilidade completa (Newman, 2015). O Jaeger, uma plataforma de tracing distribuído, é a peça central para rastrear a jornada de cada requisição através dos seus microserviços, o que permite à Uber identificar rapidamente a origem de falhas e gargalos de desempenho. O Prometheus, por sua vez, é utilizado para a recolha de métricas de cada serviço, como latência e taxa de erros, permitindo a análise contínua do comportamento do sistema e a criação de alertas proativos. A orquestração desses serviços é gerida por Kubernetes, garantindo que a infraestrutura possa ser dimensionada de forma automática e segura, suportando a demanda crescente.

3.4.4 Conclusão dos Estudos de Caso

Os casos de estudo da Netflix, Amazon e Uber ilustram que, embora os desafios de monitorização em microserviços sejam significativos, podem ser superados com a adoção de uma abordagem multifacetada. A combinação dos três pilares da observabilidade, logs, métricas e tracing e o uso de ferramentas específicas, sejam elas proprietárias ou open-source, são essenciais para garantir que os sistemas distribuídos se mantenham robustos, escaláveis e resilientes em ambientes de alta exigência (Dragoni et al., 2017).

3.5 Futuro da Monitorização de Microserviços

À medida que as arquiteturas de microserviços continuam a evoluir, as ferramentas e práticas de monitorização também acompanham esta evolução. O futuro da monitorização em microserviços está ligado à integração de novas tecnologias e à melhoria da observabilidade, visando garantir um funcionamento ainda mais eficiente e autónomo dos sistemas distribuídos.

3.5.1 Inteligência Artificial e Machine Learning na Monitorização de Microserviços

Uma das áreas mais promissoras na monitorização de microserviços é o uso de inteligência artificial (IA) e machine learning (ML). Estas tecnologias podem ser aplicadas para detectar anomalias, prever falhas

antes que elas ocorram e otimizar o desempenho de sistemas distribuídos. A monitorização preditiva permite que os sistemas identifiquem padrões de comportamento e ajustem automaticamente os recursos para prevenir falhas (Khan et al., 2022). A integração de AIOps (Artificial Intelligence for IT Operations) nas plataformas de monitorização também está a transformar a forma como os problemas são identificados e resolvidos em tempo real. Ao usar algoritmos de ML para analisar grandes volumes de dados de telemetria, as empresas podem automatizar o processo de diagnóstico e correção de falhas, tornando a operação de microserviços ainda mais eficiente (Khan et al., 2022).

3.5.2 Observabilidade Previsiva e Automação da Monitorização

A observabilidade preditiva está a tornar-se uma tendência emergente na monitorização de microserviços. Ao analisar dados históricos e comportamentais, os sistemas poderão prever falhas e otimizar a alocação de recursos antes mesmo que um problema ocorra. Esta abordagem reduzirá o tempo de inatividade e melhorará a resposta a falhas [Kusuma and Oktawati \(2022\)](#). A automação na instrumentação de microserviços também será um ponto-chave no futuro da monitorização. Ferramentas como OpenTelemetry, que permitem a coleta unificada de métricas, logs e traces, irão se tornar ainda mais populares, garantindo que todos os dados de telemetria possam ser acessados e analisados de forma coesa e eficiente [Kusuma and Oktawati \(2022\)](#).

3.5.3 Tecnologias Emergentes para Monitorização em Microserviços

Além da IA e do ML, outras tecnologias emergentes estão a moldar o futuro da monitorização, como 5G e edge computing. O 5G vai permitir uma comunicação ainda mais rápida e eficiente entre microserviços, enquanto o edge computing ajudará a descentralizar o processamento de dados, reduzindo a latência e aumentando a performance dos sistemas distribuídos (Dragoni et al., 2017). As tecnologias de serverless computing e containers também continuam a evoluir, exigindo novas abordagens para a monitorização. A integração de ferramentas de monitorização com plataformas como Kubernetes e Docker será fundamental para garantir que as aplicações possam escalar e funcionar sem problemas, independentemente de como os microserviços sejam orquestrados ou implementados (Dragoni et al., 2017).

Capítulo 4

Implementação Prática de Observabilidade com Open-Telemetry

todo: (texto copiado do documento word), meter cites, italicos e acronimos depois da revisao por parte do orientador

4.1 Introdução e Caracterização do Sistema Observado

O cenário atual do desenvolvimento de software é marcado pela crescente adoção de arquiteturas de microsserviços e ambientes cloud-native, com plataformas de gestão como o Kubernetes. Embora essa abordagem promova agilidade, escalabilidade e resiliência, ela introduz uma complexidade significativa, especialmente no monitoramento e depuração. A proliferação de serviços distribuídos, cada um com sua própria lógica, e a comunicação assíncrona entre eles, tornam o seguimento de uma única requisição de ponta a ponta uma tarefa desafiadora. Neste contexto, a observabilidade emerge como uma disciplina fundamental para garantir a fiabilidade, o desempenho e a resiliência desses sistemas (Salah et al., 2017)

4.1.1 Visão geral da Arquitetura do Sistema R2UT

A arquitetura do R2UT segue os princípios das arquiteturas de microsserviços para garantir flexibilidade, escalabilidade e resiliência no gerenciamento do ciclo de vida de construção modular. Como apresentado no documento, a plataforma é composta por diversos componentes (ou módulos), cada um responsável por uma funcionalidade específica. A estrutura modular facilita a manutenção, o desenvolvimento e a implementação de novas funcionalidades sem impactar a operação de outros componentes. A arquitetura é organizada em camadas, e cada serviço é isolado em um container Docker, gerido e orquestrado pelo Kubernetes. Isso permite que os serviços sejam executados de maneira autônoma, escalando conforme necessário, e se comunicando por meio de REST APIs, gRPC e mensagens via Kafka. A comunica-

ção entre os componentes é feita através de interfaces bem definidas, o que garante um alto nível de desacoplamento entre os microserviços.

4.1.2 Papel do Kubernetes na Arquitetura do R2UT

O Kubernetes desempenha um papel central na arquitetura do R2UT, fornecendo a infraestrutura necessária para orquestrar e gerenciar os containers dos microserviços, garantindo:

- Escalabilidade automática (Auto-scaling): O Kubernetes permite que os pods (unidades de execução de containers) sejam escalados automaticamente com base na carga de trabalho e na demanda de recursos. Isso é essencial para garantir que o sistema possa lidar com picos de uso sem comprometimento de desempenho, como no caso de um aumento de utilizadores que acessam simultaneamente os serviços;
- Alta disponibilidade (High Availability - HA): O Kubernetes garante que, caso um pod falhe, outro seja automaticamente reiniciado em um nó diferente do cluster, minimizando o impacto de falhas no sistema. Isso assegura que os serviços essenciais, como autenticação e autorização, ou até mesmo o gerenciamento de IoT devices, continuem operando sem interrupção;
- Gerenciamento de containers: O Kubernetes gerencia a execução de containers, garantindo que todos os microserviços estejam devidamente implantados e funcionando corretamente. Ele também facilita a atualização e a implementação de novas versões dos serviços, sem tempo de inatividade, por meio do rolling update;
- Orquestração e balanceamento de carga: O Kubernetes pode ser configurado para balancear automaticamente a carga de trabalho entre diferentes réplicas de um serviço, garantindo que o tráfego seja distribuído de maneira eficiente, sem sobrecarregar nenhum servidor individualmente. Essa orquestração é crucial para garantir que as interações entre os microserviços sejam rápidas e confiáveis.

4.1.3 Arquitetura de Microserviços no Kubernetes

A plataforma R2UT é composta por diversos serviços que interagem entre si, cada um implementado como um microserviço em containers. Esses serviços são organizados em um cluster Kubernetes, e a comunicação entre os componentes é feita através de APIs e mensagens assíncronas via Kafka, permitindo

que o sistema seja altamente escalável e eficiente. A seguir, destacam-se alguns componentes chave da arquitetura:

- **Global Database:** Uma base de dados compartilhada por todos os serviços que necessita de escalabilidade horizontal. O Kubernetes facilita o gerenciamento de bancos de dados distribuídos, como o PostgreSQL com Citus, para garantir alta performance nas consultas e escalabilidade;
- **Middleware e Cloud Platform:** Os serviços responsáveis pela interoperabilidade entre os sistemas internos e o deploy em nuvem. O Kubernetes gerencia a infraestrutura necessária para escalar esses serviços conforme a demanda de tráfego;
- **Módulos Específicos:** Tenant Management, Ticket Management, PDFBuilder, IoT Manager, Rules Engine, DAE Authentication. Cada um desses módulos opera de forma independente em containers, com comunicação entre os serviços mediada através de APIs e filas de mensagens (Kafka);
- **Kafka Cluster:** O Kafka é utilizado como broker de mensagens para garantir a comunicação assíncrona entre os microserviços, especialmente para o gerenciamento de grandes volumes de dados e eventos gerados por dispositivos IoT ou interações de usuários. O Kubernetes permite a escalabilidade do Kafka através de clusters e replica os brokers para garantir alta disponibilidade e tolerância a falhas.

4.1.4 Desafios da Observabilidade em Sistemas Distribuídos

A implementação de sistemas distribuídos, como o ambiente Kubernetes com microserviços, cria desafios significativos em relação à observabilidade. A complexidade do sistema é aumentada pela comunicação assíncrona entre os serviços, pela escalabilidade dinâmica dos pods e pela necessidade de correlacionar eventos gerados por diferentes componentes. A detecção de falhas ponta a ponta e a análise de métricas, logs e traces de forma eficiente são cruciais para garantir a saúde e o desempenho do sistema. A observabilidade adequada exige que o sistema seja capaz de capturar, correlacionar e analisar as interações entre os microserviços, além de monitorar de forma eficaz as métricas de desempenho, os logs estruturados e os traces distribuídos.

4.2 Objetivos e Justificativa da Implementação

4.2.1 Objetivos Práticos

A principal motivação para a implementação desta solução foi a necessidade de garantir visibilidade completa sobre o comportamento do sistema, oferecendo informações em tempo real sobre a saúde e o desempenho da aplicação. O objetivo foi criar uma solução de observabilidade integrada que permitisse à equipa de desenvolvimento atuar de forma proativa. Os objetivos práticos da implementação incluem:

1. Visibilidade em Tempo Real: Fornecer uma visão consolidada e interativa do comportamento da aplicação, com a capacidade de monitorar métricas, logs e traces de forma centralizada. Isso permite a deteção imediata de anomalias, minimizando o impacto no desempenho e na experiência do usuário;
2. Redução do MTTR (Mean Time to Resolution): Diminuir o tempo necessário para identificar e resolver problemas. A correlação de dados de diferentes fontes (métricas, logs e traces) é fundamental para diagnosticar falhas de forma eficiente, permitindo assim identificar rapidamente a causa raiz;
3. Otimização de Desempenho: Capacitar a equipa de desenvolvimento a identificar gargalos de desempenho e áreas de melhoria antes que eles afetem os utilizadores finais. O uso de alertas e dashboards de desempenho permite a otimização contínua do sistema, ajustando componentes e recursos de acordo com a carga e as necessidades operacionais

Importa referir que o escopo da solução de observabilidade foi deliberadamente delimitado aos microserviços desenvolvidos internamente pela equipa do projeto. Assim, a monitorização abrange apenas os componentes proprietários da plataforma R2UT, excluindo serviços externos ou de terceiros (como bases de dados geridas por fornecedores, APIs externas ou serviços cloud nativos). Esta decisão foi motivada pela necessidade de garantir visibilidade e controlo direto sobre os módulos sob responsabilidade da equipa, assegurando que o esforço de instrumentação se concentra nas partes do sistema onde é possível atuar de forma proativa.

O escopo da observabilidade contempla os microserviços internos da equipa, excluindo serviços externos e terceiros, garantindo controle direto e foco nas partes modificáveis.

4.3 O Papel do OpenTelemetry na Observabilidade de Microserviços

Em arquiteturas de microserviços, onde múltiplos serviços isolados cooperam para servir uma única requisição, rastrear o comportamento global do sistema torna-se um desafio. Problemas como latências interserviço, falhas silenciosas ou dependências ocultas exigem que a observabilidade vá além do monitoramento tradicional. Nesse contexto, o OpenTelemetry surge como uma camada padronizada de telemetria, capaz de unificar métricas, logs e traces e de oferecer correlação ponta a ponta, com menor acoplamento ao backend.

4.3.1 O que é o OpenTelemetry

O OpenTelemetry (OTel) é um projeto open-source da CNCF que fornece APIs, SDKs e o OpenTelemetry Collector para instrumentar e transportar os três sinais de observabilidade, traces, métricas e logs de forma agnóstica de fornecedor e independente do backend. O objetivo é padronizar a geração e a exportação de telemetria, permitindo encaminhar os dados para sistemas como Prometheus, Jaeger, Loki e outros, sem alterações no código da aplicação. O Collector atua como um binário vendor-agnostic que recebe, processa e exporta telemetria para um ou múltiplos destinos, removendo a necessidade de operar coletores específicos por ferramenta e suportando protocolos abertos (OTLP, Prometheus, Jaeger, etc.).

4.3.2 Principais Componentes (Visão Geral)

Embora os detalhes sejam explicados nas seções seguintes, vale apresentar aqui os blocos conceituais do OTel:

- APIs / SDKs / Instrumentação: as APIs definem o contrato genérico para traces, métricas e logs; os SDKs concretizam esse contrato em cada linguagem e permitem a instrumentação manual ou automática;
- Exporters / Receivers: componentes que enviam ou recebem telemetria em formatos como OTLP, Jaeger, Prometheus;
- Collector: componente neutro que organiza pipelines receivers - processors - exporters, permitindo filtragem, enriquecimento, amostragem e fan-out.

Esses componentes trabalham juntos para garantir que a telemetria gerada pelo sistema seja relevante, consistente e útil para análise.

4.3.3 Sinais, Convenções Semânticas e Recursos

O OpenTelemetry organiza a observabilidade em três sinais principais:

- Traces: descrevem operações distribuídas através de spans correlacionados;
- Métricas: valores quantitativos observados ao longo do tempo;
- Logs estruturados: eventos com contexto adicional.

Para garantir comparabilidade entre serviços e linguagens, o OTel define Convenções Semânticas (Semantic Conventions, SemConv), um vocabulário padronizado para atributos de HTTP, bases de dados, mensageria e outros domínios. Além disso, atributos de Resource (como `service.name`, `service.version`, `service.namespace`) identificam consistentemente a origem da telemetria. A aplicação sistemática das SemConv melhora a filtragem, correlação e exploração nos dashboards, sendo `service.name` um dos atributos obrigatórios.

4.3.4 Vantagens do OpenTelemetry

- Padronização e portabilidade: Uma só camada de instrumentação para todos os sinais e múltiplos backends, reduzindo lock-in e retrabalho em migrações;
- Flexibilidade via Collector: Multi-export, buffering/retry, amostragem inteligente, enriquecimento e saneamento centralizados, antes do armazenamento;
- Alinhamento cloud-native: Suporte maduro para Kubernetes e frameworks populares, além de auto-instrumentation em linguagens como .NET.

Limitações e Desafios

- Curva de configuração/operacional: Pipelines mal desenhadas podem causar perda de dados ou latência; o Collector também consome recursos e precisa de monitorização;
- Maturidade desigual por sinal: Tracing e métricas estão muito estáveis; logs continuam a evoluir e dependem mais de integrações (ex.: Loki/ELK);

- OTel não é backend: Continua a ser necessário armazenamento/visualização (Prometheus, Jaeger, Loki, Grafana).

4.4 Comparação entre OpenTelemetry e a Stack Tradicional de Observabilidade (Prometheus, Grafana, Jaeger e Loki)

Importa clarificar que a comparação estabelecida entre OpenTelemetry e a stack composta por Prometheus, Grafana, Jaeger e Loki não deve ser interpretada como uma análise entre ferramentas equivalentes ou mutuamente exclusivas. O OpenTelemetry não se configura como um backend de armazenamento ou visualização de dados. O seu papel principal situa-se na camada de instrumentação, coleta, normalização e encaminhamento de dados de observabilidade (métricas, rastreamentos e registos), ou seja, o OpenTelemetry atua como um padrão unifica-dor para a geração e transporte da telemetria, permitindo que diferentes linguagens, bibliotecas e serviços emitam dados num formato consistente (OTLP) e que estes possam ser processados e exportados através do Collector.

Por contraste, a stack tradicional composta por Prometheus, Grafana, Jaeger e Loki cobre funções predominantemente associadas ao armazenamento persistente, consulta e visualização da telemetria:

- Prometheus provê a recolha e o armazenamento de séries temporais de métricas, com motor de consulta baseado em PromQL;
- Grafana oferece a camada de visualização e alertas sobre métricas, logs e outros dados observacionais;
- Jaeger é especializado no armazenamento, consulta e visualização de rastreamentos distribuídos;
- Loki realiza a ingestão e indexação de registos (logs) para fins de pesquisa e correlação.

Assim, mesmo num cenário em que o OpenTelemetry é utilizado como base para a instrumentação e coleta, permanece necessária a adoção de sistemas de backend que assegurem a persistência e exploração dos dados. No caso em estudo, esses papéis continuam a ser desempenhados pelo Prometheus, Grafana, Jaeger e Loki. A comparação, portanto, deve ser entendida no sentido de abordagens distintas para observabilidade:

- No modelo tradicional, cada ferramenta impõe o seu próprio agente ou exporter (e.g., Node Exporter para Prometheus, Jaeger Agent, Promtail para Loki), originando uma coleta fragmentada e heterogénea;

- No modelo baseado em OpenTelemetry, a coleta é unificada (através do Collector em modo DaemonSet e dos SDKs ou mecanismos de auto-instrumentação), e os mesmos dados podem ser distribuídos de forma flexível para múltiplos backends.

Deste modo, a utilização do OpenTelemetry não elimina a necessidade dos backends clássicos, mas antes os complementa, ao introduzir uma camada de padronização e abstração que reduz o acoplamento e aumenta a portabilidade da telemetria no ecossistema de observabilidade.

Tabela 3: Comparação: OpenTelemetry (OTel) vs Prometheus + Grafana + Jaeger

Característica	OpenTelemetry (OTel)	Prometheus + Grafana + Jaeger
Instrumentação	Unificada (traces, métricas, log)	Separada por ferramenta
Padronização	Convenções Semânticas (Sem-Conv)	Cada ferramenta define o seu padrão
Agnosticidade de backend	Sim, exporta para múltiplos destinos	Não, acoplado a cada backend
Collector centralizado	Sim, pipelines flexíveis	Não, coletores independentes
Auto-instrumentação	Suporte amplo	Limitado por ferramenta
Vendor lock-in	Baixo	Médio / Alto
Escalabilidade	Alta	Alta, mas com mais componentes
Curva de aprendizagem	Moderada (pipelines OTel)	Mais baixa para casos simples

4.5 Arquitetura Detalhada e Implementação da Solução

4.5.1 Visão Geral da Arquitetura

A imagem detalha o fluxo de dados de telemetria desde a sua origem nos micros-serviços até a visualização final no Grafana. A arquitetura é composta pelas seguintes camadas:

- **APIs e SDKs**

As APIs (Application Programming Interfaces) definem os contratos para a geração e correlação de dados de telemetria. Os SDKs (Software Development Kits) são implementações específicas de linguagem das APIs, fornecendo as ferramentas necessárias para instrumentar o código das

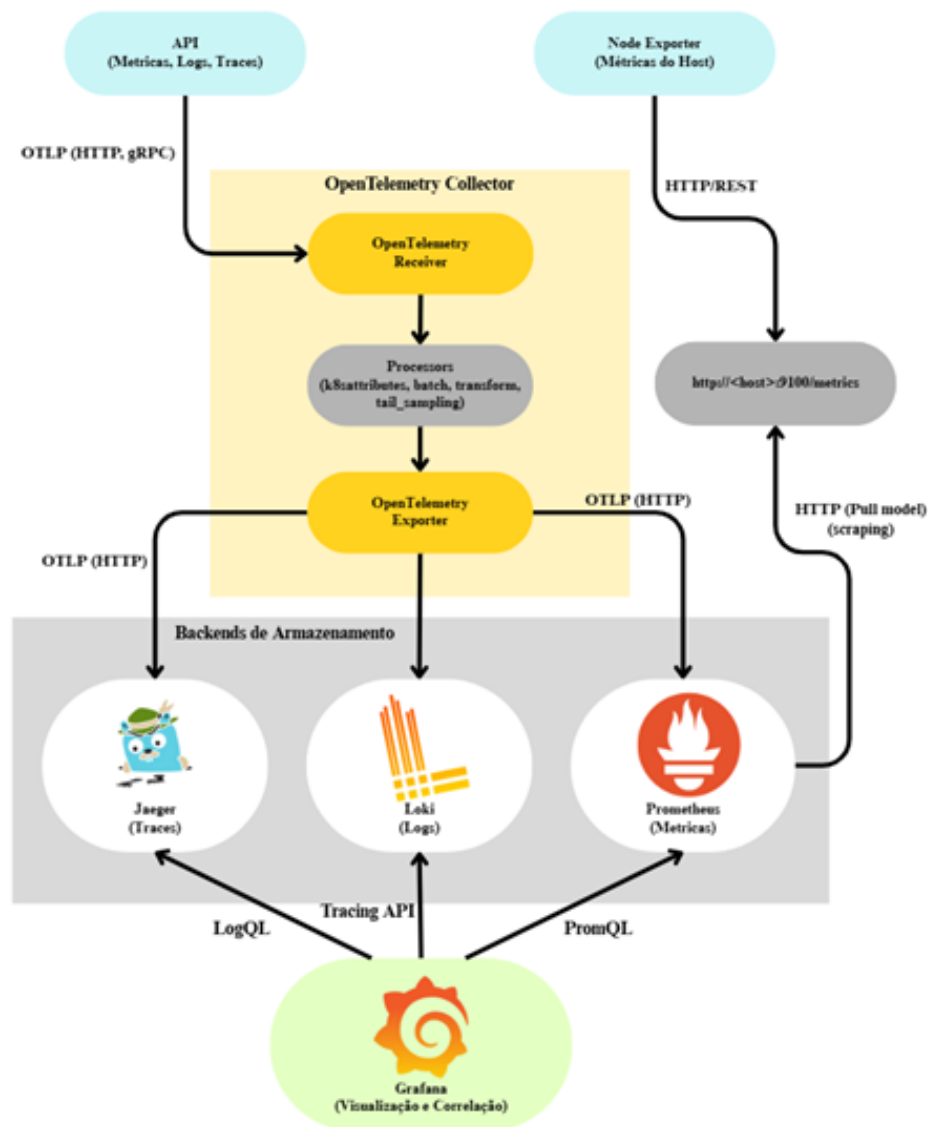


Figura 3: Arquitetura da Solução

aplicações. Estes SDKs permitem aos utilizadores gerar dados de telemetria na sua linguagem de programação escolhida e exportá-los para um backend preferencial. Incluem bibliotecas de instrumentação que geram dados relevantes a partir de bibliotecas e frameworks populares (por exemplo, requisições HTTP) e detetores de recursos que adicionam atributos contextuais (como nome do pod ou namespace em Kubernetes) aos dados de telemetria.(OpenTelemetry Authors, 2025; Thakur and Chandak (2022))

- **Collector**

O OpenTelemetry Collector é um componente independente e agnóstico de fornecedor, concebido para receber, processar e exportar dados de telemetria. Atua como um hub centralizado para gerir

pipelines de telemetria, recebendo dados em vários formatos (como OTLP, Prometheus, Jaeger) e encaminhando-os para um ou mais backends. A sua capacidade de processar e filtrar dados antes da exportação otimiza o fluxo de dados, reduzindo a sobrecarga nas aplicações e melhorando a eficiência geral. (OpenTelemetry Authors, 2025; [Thakur and Chandak \(2022\)](#))

- **Exportadores (Exporters)**

Os exportadores são componentes responsáveis por enviar os dados de telemetria (após serem gerados pelas aplicações ou processados pelo Collector) para ferramentas de backend específicas, como Grafana, Jaeger, Prometheus, Loki ou outros sistemas proprietários. A utilização de exportadores OTLP é considerada uma boa prática, pois são concebidos para emitir dados OpenTelemetry sem perda de informação e são amplamente suportados por diversas plataformas de observabilidade. (OpenTelemetry Authors, 2025; [Thakur and Chandak \(2022\)](#))

- **Node Exporter**

O Node Exporter é utilizado como agente de monitorização a nível do sistema operativo, responsável por expor métricas relacionadas com CPU, memória, disco e rede. Embora não esteja diretamente integrado no pipeline do OpenTelemetry Collector, este componente fornece ao Prometheus dados cruciais sobre a infraestrutura subjacente, permitindo complementar a observabilidade das aplicações com indicadores do ambiente de execução.

- **Camada de Visualização e Análise**

O Grafana atua como a interface de usuário unificada para visualizar e analisar os dados. Ele se integra aos backends (Prometheus, Loki e Jaeger), permitindo a criação de dashboards interativos que mostram as métricas, logs e traces de maneira correlacionada.

A separação entre instrumentação (APIs e SDKs), processamento/exportação (Collector e Exportadores) e visualização confere maior flexibilidade e desacoplamento, permitindo que as aplicações permaneçam independentes da infraestrutura de observabilidade subjacente.

4.5.2 Fluxo de Telemetria

O fluxo de dados de telemetria segue um pipeline bem definido:

1. A aplicação ASP.NET Core emite dados de observabilidade (logs, métricas, traces) através do protocolo OTLP gRPC, e envia-os para o receiver do OpenTelemetry Collector;

2. As métricas do sistema operativo são recolhidas diretamente pelo Prometheus através do Node Exporter;
3. O OTel Receiver aceita entradas de diferentes fontes, incluindo OTLP e scraping Prometheus;
4. Os dados são então encaminhados para os processadores do Collector, que executam tarefas de transformação (transform), enriquecimento com diferentes atributos (attributes), agregação (batch) e amostragem de traces (tail_sampling);
5. Após o processamento, os dados são enviados pelo OTel Exporter para os seus respetivos destinos: Logs → Loki, Traces → Jaeger e Métricas → Prometheus;
6. O Grafana atua como ponto de agregação visual, utilizando as linguagens de consulta específicas de cada backend para gerar dashboards interativos e correlacionados.

4.6 Detalhes Técnicos da Implementação

4.6.1 Estratégia de Instrumentação em .NET

Do ponto de vista técnico, a instrumentação foi aplicada exclusivamente aos serviços criados internamente, evitando a recolha de dados de dependências externas. Desta forma, métricas, logs e traces refletem apenas o comportamento dos micro-serviços proprietários, reduzindo o ruído e a sobrecarga de dados. Esta abordagem garante que a telemetria recolhida é relevante para a análise e otimização da plataforma, focando a monitorização nos elementos críticos sob responsabilidade direta da equipa de desenvolvimento.

A instrumentação de aplicações é um passo crucial para gerar dados de telemetria significativos. Em ambientes de microserviços com múltiplas APIs, a instrumentação individual de cada serviço pode ser repetitiva e propensa a erros, para mitigar esta complexidade, foi adotada uma abordagem prática e reutilizável para a instrumentação de APIs ASP.NET Core, utilizando um pacote comum (DTX.Base.Common). Este pacote encapsula toda a configuração necessária para a emissão de traces distribuídos, métricas e logs estruturados, promovendo uma padronização e um desacoplamento eficaz entre o código da aplicação e a infra-estrutura de observabilidade. Esta estratégia centraliza a lógica de observabilidade num único ponto, reduzindo significativamente o código repetido. A configuração da observabilidade pode ser ativada e controlada dinamicamente através de variáveis de ambiente, o que confere grande flexibilidade e portabilidade entre diferentes ambientes (desenvolvimento, homologação e produção). Para instrumentar

novos serviços, a intervenção é mínima: basta adicionar o pacote `DTX.Base.Common`, invocar um método de extensão no `Program.cs` e definir as variáveis de ambiente necessárias. A lógica de observabilidade foi centralizada num único método de extensão, aplicada na inicialização de cada API .Net:

```
builder.ConfigureOpenTelemetry();
```

Este método ativa automaticamente os componentes do OpenTelemetry para instrumentação de tracing distribuído, coleta de métricas e exportação de logs estruturados. As configurações são controladas dinamicamente por variáveis de ambiente, o que facilita a portabilidade entre diferentes ambientes, sem a necessidade de recompilação ou reconfiguração manual.

Tabela 4: Variáveis de ambiente para configuração OTLP Exporter (colocar as em uso)

Variável	Descrição
OTEL_EXPORTER_OTLP_ENDPOINT	URL do collector OTLP
OTEL_EXPORTER_OTLP_PROTOCOL	Protocolo utilizado (grpc ou http/protobuf)
OTEL_EXPORTER_OTLP_HEADERS	Cabeçalhos opcionais no formato chave=valor (ex: Authorization=Bearer abc)

4.6.2 Vantagens da Abstração da Instrumentação OpenTelemetry

O encapsulamento da lógica de instrumentação no pacote `DTx.Base.Common` e a exposição via um método de extensão (`ConfigureOpenTelemetry()`) proporcionam benefícios significativos para o desenvolvimento e a operação de aplicações distribuídas:

- **Padronização da Instrumentação entre Múltiplas APIs:** Garante que todas as APIs sigam as mesmas convenções de observabilidade, resultando em dados de telemetria consistentes e facilmente comparáveis. Isso é fundamental para a correlação eficaz de dados em sistemas complexos;
- **Desacoplamento da Infraestrutura de Observabilidade:** O código da aplicação torna-se independente das ferramentas de backend utilizadas (Grafana, Jaeger, Tempo, etc.). Se houver uma mudança nas ferramentas de observabilidade, as modificações são minimizadas e confinadas à configuração do collector ou às variáveis de ambiente, não exigindo alterações no código da aplicação;
- **Facilidade de Configuração via Ambiente:** A ativação e o ajuste da observabilidade são feitos através de variáveis de ambiente, o que simplifica a implantação e a gestão em diferentes ambientes (desenvolvimento, homologação, produção), promovendo a portabilidade;

- **Redução Significativa de Código Repetido:** Ao centralizar a lógica de observabilidade num único ponto, evita-se a duplicação de código em cada novo serviço ou API, tornando o processo de instrumentação mais eficiente e menos propenso a erros.

Com esta abordagem, a instrumentação de novos serviços pode ser realizada com mínima intervenção, com a adição do pacote `DTX.Base.Common`, uma chamada simples ao método de extensão no `Program.cs` e a definição das variáveis de ambiente necessárias. Isso acelera o desenvolvimento e garante que a observabilidade seja uma parte integrante do ciclo de vida da aplicação desde o início.

4.6.3 Instrumentação Específica do ASP.NET Core

A instrumentação foi dividida nos três pilares da observabilidade, utilizando bibliotecas de instrumentação automática para cada um.

- **Instrumentação de Tracing Distribuído:** A instrumentação de rastreamento distribuído utiliza bibliotecas que monitoram requisições web, chamadas HTTP e interações com o banco de dados. Esses dados são exportados para o Collector. `.tracing .AddAspNetCoreInstrumentation() .AddHttpClientInstrumentation() .AddEntityFrameworkCoreInstrumentation() .AddNpgsql();`
- **Coleta de Métricas:** A solução também contempla a recolha de métricas relevantes tanto ao nível da aplicação como do ambiente de execução. As métricas são coletadas pelo SDK e enviadas para o Collector. `.metrics .AddAspNetCoreInstrumentation() .AddHttpClientInstrumentation() .AddRuntimeInstrumentation();`
- **Logs Estruturados:** A configuração de logs substitui os providers nativos de logging do .NET e ativa o OpenTelemetry como destino principal de logs estruturados. `builder.Logging.AddOpenTelemetry(logging => logging.IncludeFormattedMessage = true; logging.IncludeScopes = true; logging.AddOtlpExporter());`

A utilização de logs estruturados permite realizar pesquisas avançadas por atributos, correlacionar mensagens de log com spans e traces, e visualizar os logs num contexto de execução mais rico (e.g., por serviço, requisição ou erro).

4.7 Coleta de dados com o OpenTelemetry Collector

A fase de coleta é o ponto de entrada para os dados de telemetria no pipeline de observabilidade, é responsável por capturar logs, métricas e traces gerados pelas aplicações instrumentadas, agentes sidecar

ou ferramentas de terceiros. Esta coleta é feita principalmente através dos receivers do OpenTelemetry Collector, que atuam como ouvintes ou scrapers, aceitando dados em vários formatos. O OpenTelemetry Collector, um componente central do ecossistema, funciona como um hub de processamento centralizado, capaz de lidar com múltiplas fontes e destinos de dados, na implementação em questão, o Collector foi configurado para escutar em duas portas principais para o protocolo OTLP: 4317 para gRPC e 4318 para HTTP. Esta configuração permite que as aplicações enviem telemetria usando o protocolo de sua preferência, a fase de coleta é crítica para garantir que os dados cheguem de forma consistente e em tempo real ao pipeline de observabilidade. No ambiente Kubernetes, a implantação do OpenTelemetry Collector foi realizada como um DaemonSet. Esta escolha de implantação garante que o Collector seja executado como um agente em cada nó (node) do cluster, em vez de ser um serviço centralizado. Cada instância do DaemonSet Collector, que age como um agente local, recebe a telemetria dos pods que estão no mesmo nó. A principal vantagem desta abordagem é a redução de latência e a segurança. Ao coletar a telemetria localmente, os dados não precisam viajar pela rede do cluster, reduzindo a latência e o risco de congestionamento, além disso, essa arquitetura de agente local é mais resiliente, pois a falha de um agente afeta apenas a coleta de dados de um único nó, enquanto os outros nós continuam a operar normalmente. Essa configuração de DaemonSet, combinada com os receivers do Collector, otimiza o fluxo de telemetria e torna-o mais robusto e eficiente.

4.7.1 Protocolo OTLP (gRPC/HTTP) e Configuração

O OpenTelemetry Protocol (OTLP) é o protocolo padrão utilizado na plataforma de observabilidade para o transporte de dados de telemetria, como métricas, logs e tracing distribuído. Desenvolvido como parte do ecossistema OpenTelemetry, o OTLP é um protocolo aberto, extensível e eficiente, que permite a comunicação entre aplicações instrumentadas, agentes de coleta (como o OpenTelemetry Collector) e sistemas de backend como Grafana Loki (logs), Jaeger (traces) e Prometheus (métricas). O protocolo suporta os formatos gRPC e HTTP/protobuf, garantindo flexibilidade de integração com diversos ambientes e ferramentas. Num cluster Kubernetes, o uso do OTLP padroniza a coleta e exportação de dados de observabilidade entre os pods e os componentes da infraestrutura de monitoramento, gera assim portabilidade, interoperabilidade e escalabilidade da solução implementada.

4.7.2 OpenTelemetry Operator

Gerir a observabilidade em ambientes Kubernetes pode tornar-se complexo, especialmente quando é necessário configurar múltiplos Collectors, manter consistência de pipelines e aplicar boas práticas de

escalabilidade e segurança. Para simplificar esta gestão, a comunidade desenvolveu o OpenTelemetry Operator, um Custom Kubernetes Operator que automatiza o ciclo de vida dos Collectors e a sua configuração.

Conceito e Arquitetura

O OpenTelemetry Operator expande o Kubernetes através de Custom Resource Definitions (CRDs), introduzindo novos tipos de recurso que descrevem configurações de observabilidade de forma declarativa. O recurso mais importante é o OpenTelemetryCollector, no qual o utilizador define, em YAML, as características desejadas do Collector (receivers, processors, exporters e modo de execução). O Operator traduz automaticamente essa especificação em objetos nativos do Kubernetes, como Deployments, DaemonSets ou ConfigMaps.

Dessa forma:

1. O programador ou DevOps aplica um manifesto OpenTelemetryCollector;
2. O Operator valida e cria os recursos Kubernetes correspondentes;
3. O Collector é implantado de forma consistente e conforme as boas práticas definidas pela comunidade.

Modos de Deploy suportados

O CRD OpenTelemetryCollector permite escolher o modo de execução do Collector:

- DaemonSet: cada nó do cluster executa uma instância do Collector, recolhendo telemetria localmente;
- Deployment: uma instância centralizada atua como gateway, agregando e exportando dados;
- Sidecar: o Collector é implantado junto de cada Pod, garantindo isolamento máximo e baixo acoplamento;
- StatefulSet: utilizado em cenários que exigem persistência ou configuração estável.

Essa flexibilidade permite ajustar a arquitetura de observabilidade conforme a natureza da carga de trabalho.

Vantagens do Operator

- Automação: reduz a necessidade de escrever e manter manifestos Kubernetes complexos para cada Collector;
- Consistência: garante que todos os Collectors seguem uma configuração centralizada e padronizada;
- Integração com GitOps: por ser declarativo, integra-se facilmente em fluxos de CI/CD e ferramentas como ArgoCD ou Flux.

Evolução contínua: a comunidade mantém o Operator alinhado com novas versões do OpenTelemetry, reduzindo o risco de configuração obsoleta.

4.7.3 Abordagem Adotada: DaemonSet (Agente por nodo)

Justificação da Escolha

Optou-se por executar o OpenTelemetry Collector em DaemonSet, de forma a disponibilizar um agente local em cada nó do cluster.

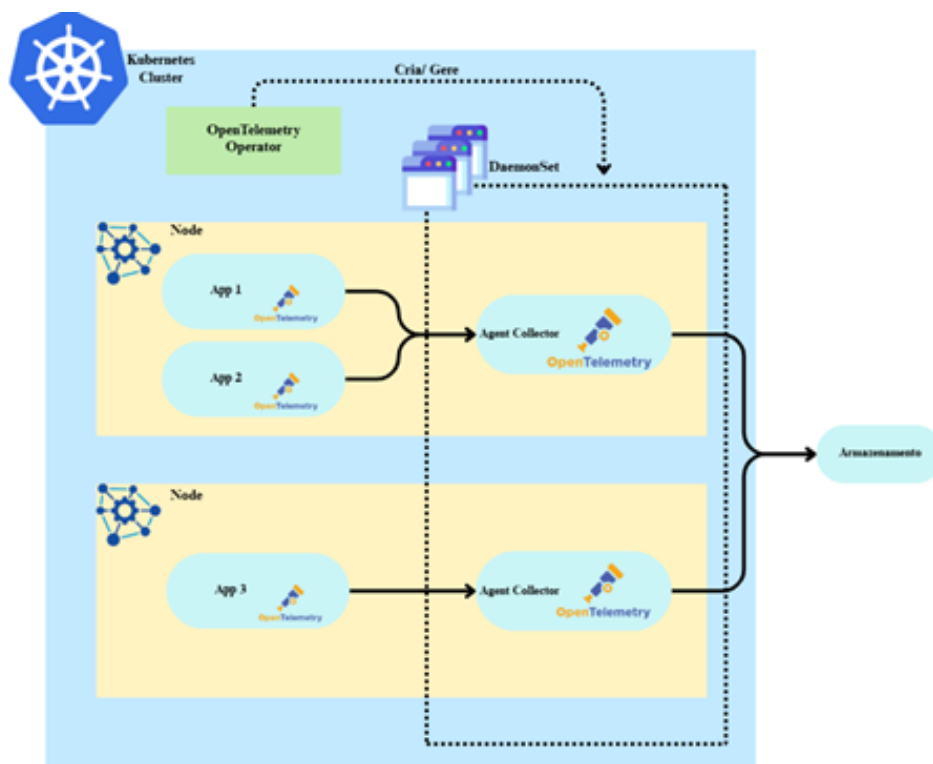


Figura 4: Implementação de DaemonSet Collector

Em cada nó, um Pod do Collector expõe receivers OTLP em gRPC (porta 4317) e HTTP/Protobuf (porta 4318), recebendo métricas, logs e traces das workloads residentes nesse nó.

Esta opção foi guiada por três objetivos principais:

- Baixa latência na entrega de telemetria, evitando saltos de rede desnecessários antes do primeiro processamento;
- Resiliência local, confinando o impacto de falhas ao nó afetado;
- Simplicidade de integração nas aplicações, que passam a publicar telemetria para um endpoint local único.

Benefícios Observados

A abordagem em DaemonSet resultou em menor variância de entrega dos sinais (coleta e pré-processamento locais), isolamento de falhas por nó, redução da dependência de rede intra-cluster e configuração simplificada do lado das aplicações (um único endpoint local, sem descoberta adicional).

4.8 Processamento e Transformação dos Dados

Após a recolha dos dados, estes passam por uma fase crítica de processamento dentro do OpenTelemetry Collector. Esta etapa, orquestrada por vários processadores, é essencial para modificar, enriquecer, agrupar ou filtrar os dados, garantindo a sua compatibilidade e utilidade para os sistemas subsequentes de armazenamento e análise.

4.8.1 Processadores e as suas Aplicações

Tabela 5: Processadores Essenciais do OpenTelemetry Collector

Processador	Função Primária	Aplicações-Chave	Significado para Observabilidade
<code>transform</code>	Modifica dados de telemetria usando OTTL.	Adiciona <code>service.name</code> , converte <i>timestamps</i> , normaliza severidades de <i>logs</i> , aplica regras de amostragem.	Garante consistência dos dados e conformidade com convenções semânticas.
<code>batch</code>	Agrupa dados de telemetria em lotes.	Otimiza a exportação de dados, reduzindo chamadas de rede e sobrecargas.	Melhora o débito e a eficiência do <i>pipeline</i> . Crucial para ambientes de alto volume.
<code>attributes</code>	Adiciona, modifica ou elimina atributos (metadados).	Injeta atributos de ambiente estáticos, enriquece com metadados do host.	Enriquece os dados com contexto crucial, melhorando a pesquisa, filtragem e correlação de dados.
<code>resource</code>	Modifica atributos de recurso.	Aplica consistentemente <code>service.name</code> e outros metadados fundamentais.	Assegura a uniformidade de identificadores em todos os dados de telemetria.
<code>memory_limiter</code>	Previne o consumo excessivo de memória.	Protege o <i>collector</i> contra falhas devido ao esgotamento de recursos.	Garante a estabilidade do <i>pipeline</i> .
<code>tail_sampling</code>	Amostra <i>traces</i> com base no contexto completo.	Reduz o volume de dados de <i>trace</i> , focando-se nos mais críticos (por exemplo, erros ou alta latência).	Otimiza custos de armazenamento e melhora o desempenho de consultas no <i>backend</i> .

- **Processador transform**

Este processador utiliza a OpenTelemetry Transformation Language (OTTL) para realizar modificações extensivas nos dados de telemetria. Na implementação atual, é empregado para adicionar o atributo `service.name` aos sinais de telemetria, converter timestamps para um formato padronizado, normalizar severidades de logs (por exemplo, mapeando vários níveis de log para um conjunto consistente como INFO, WARN, ERROR) e aplicar regras de amostragem dinâmicas a traces. O processador transform é fundamental para garantir a consistência dos dados e a adesão a convenções semânticas predefinidas, o que é primordial para uma análise precisa e uma correlação eficaz a jusante. Contudo, as suas poderosas capacidades exigem uma configuração cuidadosa para evitar "Transformações Inconsistentes" (Unsound Transformations) ou "Conflitos de Identidade" (Identity Conflicts) que possam comprometer a integridade dos dados;

- **Processador batch**

Este processador agrupa eficientemente os dados de telemetria em lotes antes de serem exportados. O batching reduz significativamente o número de chamadas de rede e a sobrecarga associada, melhorando assim o débito geral e a eficiência da exportação de dados, especialmente em ambientes de alto volume. Processador attributes: Concebido para adicionar, modificar ou eliminar atributos (metadados) em spans, logs ou métricas. Por exemplo, pode ser utilizado para injetar um atributo de ambiente estático (e.g., "produção", "desenvolvimento") em toda a telemetria de entrada ou enriquecer dados com metadados ao nível do host. Este processador é vital para enriquecer os dados de telemetria com informações contextuais cruciais, o que melhora a sua capacidade de pesquisa, filtragem e, em última análise, as suas capacidades de correlação entre diferentes sinais;

- **Processador resource**

Tem como alvo específico a modificação de atributos de recurso. Os atributos de recurso descrevem a entidade que produz a telemetria, como o serviço da aplicação, a máquina host ou o contentor. É crucial para anexar consistentemente metadados fundamentais como `environment`, `service.version` ou `region` às métricas e para enriquecer logs com informações detalhadas de recurso. Este processador garante que atributos de identificação comuns, notavelmente `service.name`, são aplicados uniformemente em todos os sinais de telemetria (logs, traces e métricas). Esta consistência é fundamental para alcançar uma correlação perfeita entre sinais no Grafana;

- **Processador memory limiter**

Este processador é implementado para evitar que o OpenTelemetry Collector consuma recursos de memória excessivos. Ao definir limites de memória, impede que o processo do Collector falhe devido ao esgotamento de recursos, garantindo assim a estabilidade e fiabilidade contínuas de todo o pipeline de observabilidade;

- **Processador tail sampling**

Este processador permite decisões de amostragem em traces com base no contexto completo de um trace, ou seja, depois de todos os spans relacionados com um trace terem sido recebidos. Suporta vários critérios de filtragem que podem ser encadeados, como amostragem baseada na latência do trace, taxas probabilísticas, códigos de status HTTP (por exemplo, apenas traces de erro) ou limitação de taxa. A amostragem de cauda é crítica para gerir o volume de dados de trace, particularmente em sistemas distribuídos de alto tráfego. Ajuda a otimizar os custos de armazenamento e a melhorar o desempenho da consulta no backend de tracing sem sacrificar a capacidade de capturar e analisar traces críticos ou anómalos.

A capacidade do OpenTelemetry Collector de modificar todos os aspetos da telemetria, incluindo a remoção de informações sensíveis através do processador transform, ou o enriquecimento consistente de dados com atributos específicos, posiciona-o como um ponto de controlo estratégico para a governação de dados. Esta centralização do controlo significa que os requisitos de conformidade podem ser geridos e aplicados ao nível do Collector, reduzindo a necessidade de alterações individuais ao nível da aplicação ou de configurações complexas específicas do backend. Esta abordagem centraliza significativamente o controlo de dados e minimiza a superfície de ataque para fugas acidentais de dados, além disso, a capacidade de filtrar, agrupar e reduzir a cardinalidade dos dados antes de serem exportados para o backend traduz-se diretamente em poupanças substanciais nos custos de ingestão e armazenamento de dados, especialmente para serviços de observabilidade geridos. Ao reduzir o volume e a complexidade dos dados na origem, o Collector melhora o desempenho da consulta nos backends e mitiga potenciais problemas de "Crise de Identidade" em métricas, levando a dashboards mais fiáveis. Isto torna o Collector como um componente crítico para gerir tanto a eficiência económica como operacional de toda a pilha de observabilidade.

4.9 Persistência e Armazenamento de Dados

4.9.1 Armazenamento de Logs com o Loki

O Loki foi escolhido como sistema de armazenamento de logs devido à sua arquitetura otimizada para consultas baseadas em etiquetas (labels) em vez de full-text search. Esta abordagem torna-o mais eficiente e menos dispendioso em termos de recursos quando comparado com soluções tradicionais, como o Elasticsearch. Os logs estruturados emitidos pelas APIs .NET são enviados ao OpenTelemetry Collector através do protocolo OTLP e, em seguida, exportados para o Loki. A integração nativa com o Grafana permite que os logs sejam visualizados e correlacionados facilmente com métricas e traces, proporcionando uma análise unificada.

4.9.2 Armazenamento de Traces com o Jaeger

O Jaeger foi adotado como backend de armazenamento e análise de traces distribuídos, dada a sua capacidade de oferecer visibilidade ponta a ponta sobre o ciclo de vida de uma requisição. Através da identificação do serviço (atributo service.name), é possível segmentar e filtrar os traces de forma eficaz, identificando gargalos e pontos de falha no percurso entre microserviços. Os dados são exportados pelo OpenTelemetry Collector via protocolo OTLP/gRPC para o endpoint do Jaeger, onde ficam persistidos para consulta e análise detalhada no Grafana.

4.9.3 Armazenamento de Métricas com o Prometheus

O Prometheus foi selecionado como sistema de armazenamento de métricas pela sua robustez no tratamento de séries temporais e pela linguagem de consulta PromQL, que possibilita análises avançadas. As métricas da aplicação .NET são recolhidas pelo OpenTelemetry Collector e exportadas para o Prometheus, centralizando a coleta e evitando a exposição direta das aplicações. Complementarmente, métricas de infraestrutura provenientes do Node Exporter são recolhidas diretamente pelo Prometheus através de scraping HTTP. Esta combinação assegura visibilidade tanto a nível da aplicação como da infraestrutura, permitindo uma visão abrangente do desempenho do sistema.

Capítulo 5

Visualização e Análise de Dados

5.1 Visualização centralizada no Grafana

A visualização dos dados de telemetria assume um papel crucial para a compreensão e monitorização eficaz de sistemas distribuídos. Embora as ferramentas Prometheus, Jaeger e Loki disponham de interfaces próprias para análise independente de métricas, traces e logs, a fragmentação das informações pode dificultar a correlação rápida entre estes dados. Por esse motivo, optou-se pelo Grafana como camada de visualização unificada, visando uma experiência integrada e eficiente.

Entre as principais vantagens destacam-se:

- Centralização das métricas, logs e traces num único painel interativo;
- Capacidade avançada de correlação entre diferentes tipos de dados, facilitando a identificação de causas-raiz nas anomalias;
- Configuração unificada de alertas que abrange todas as fontes de dados;
- Interface intuitiva e personalizável, acessível para diversos perfis técnicos;
- Linguagens de consulta especializadas (PromQL, LogQL) diretamente integradas.

Assim, o Grafana habilita uma abordagem de "single pane of glass", essencial para o acompanhamento consolidado do desempenho e saúde do sistema e traces de chamadas entre serviços, com visualização temporal.

5.2 Organização dos Dashboards

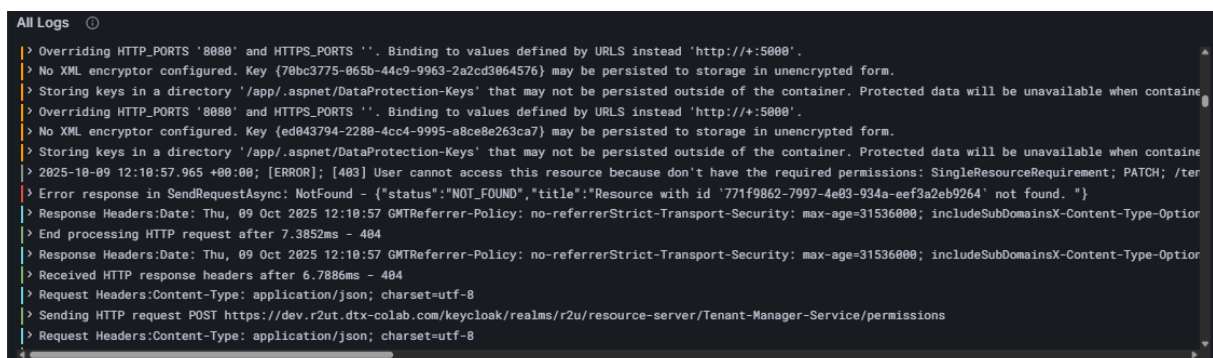
Os dashboards no Grafana foram estruturados em três categorias principais:

- **Métricas da Aplicação:** monitorização do número de requisições HTTP por segundo, latência média e percentis (p95 e p99), taxas de erro 4xx e 5xx, além do uso de memória da aplicação .NET;
- **Infraestrutura Kubernetes:** dados provenientes do Node Exporter relativos à utilização de CPU por nó e pod, carga do sistema (load average), e espaço de disco e memória física utilizados;
- **Logs e Traces:** visualização de logs estruturados filtrados por nível, mensagem e nome do serviço, análise de spans por rota, duração e código de estado, bem como visualização temporal das chamadas entre serviços através dos traces.

Esta organização aprimora a navegação e facilita a análise rápida e contextualizada.

5.2.1 Paineis e Dashboards

Para efeito de apresentação e análise, os dashboards apresentados neste capítulo focam-se exclusivamente nos dados relativos ao serviço específico TicketsManagement. Embora fosse perfeitamente possível incluir visualizações de outros serviços da arquitetura, tal abordagem tenderia a revelar informações redundantes, uma vez que os padrões e métricas observados seriam semelhantemente aplicáveis. Assim, esta escolha visa proporcionar uma análise mais clara e exemplar, evitando a dispersão do foco e privilegiando a profundidade sobre a generalidade.



```
All Logs ⓘ
> Overriding HTTP_PORTS '8080' and HTTPS_PORTS ''. Binding to values defined by URLS instead 'http://+:5000'.
> No XML encryptor configured. Key {78bc3775-065b-44c9-9963-2a2cd3864576} may be persisted to storage in unencrypted form.
> Storing keys in a directory '/app/.aspnet/DataProtection-Keys' that may not be persisted outside of the container. Protected data will be unavailable when containe
> Overriding HTTP_PORTS '8080' and HTTPS_PORTS ''. Binding to values defined by URLS instead 'http://+:5000'.
> No XML encryptor configured. Key {ed943794-2288-4cc4-9995-a8ce8e263ca7} may be persisted to storage in unencrypted form.
> Storing keys in a directory '/app/.aspnet/DataProtection-Keys' that may not be persisted outside of the container. Protected data will be unavailable when containe
> 2025-10-09 12:10:57.965 +00:00; [ERROR]; [403] User cannot access this resource because don't have the required permissions: SingleResourceRequirement; PATCH; /ter
> Error response in SendRequestAsync: NotFound - {"status":"NOT_FOUND","title":"Resource with id '771f9862-7997-4e03-934a-eef3a2eb9264' not found. "}
> Response Headers:Date: Thu, 09 Oct 2025 12:10:57 GMTReferrer-Policy: no-referrerStrict-Transport-Security: max-age=31536000; includeSubDomainsX-Content-Type-Optior
> End processing HTTP request after 7.3852ms - 404
> Response Headers:Date: Thu, 09 Oct 2025 12:10:57 GMTReferrer-Policy: no-referrerStrict-Transport-Security: max-age=31536000; includeSubDomainsX-Content-Type-Optior
> Received HTTP response headers after 6.7886ms - 404
> Request Headers:Content-Type: application/json; charset=utf-8
> Sending HTTP request POST https://dev.r2ut.dtx-colab.com/keycloak/realms/r2u/resource-server/Tenant-Manager-Service/permissions
> Request Headers:Content-Type: application/json; charset=utf-8
```

Figura 5: Painel global de Logs

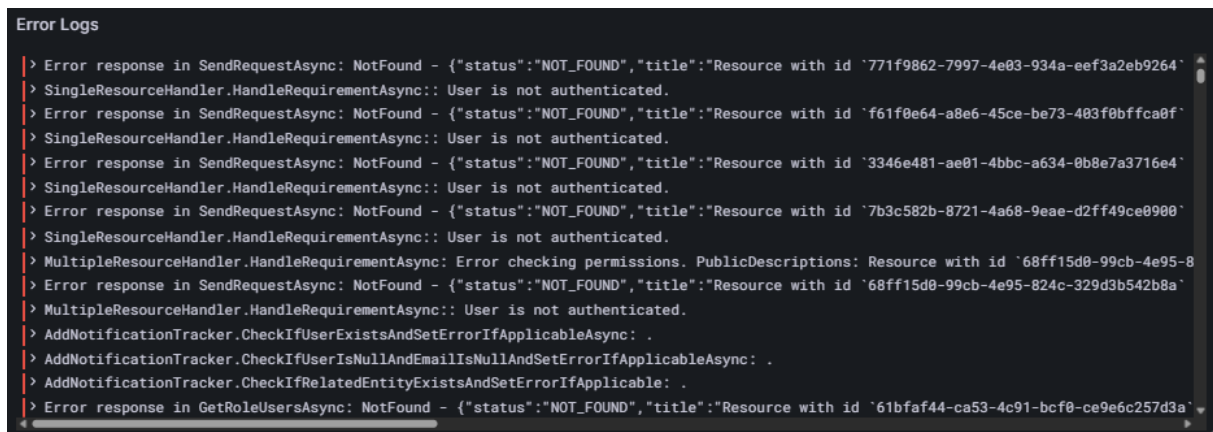


Figura 6: Painel de Logs de Erro

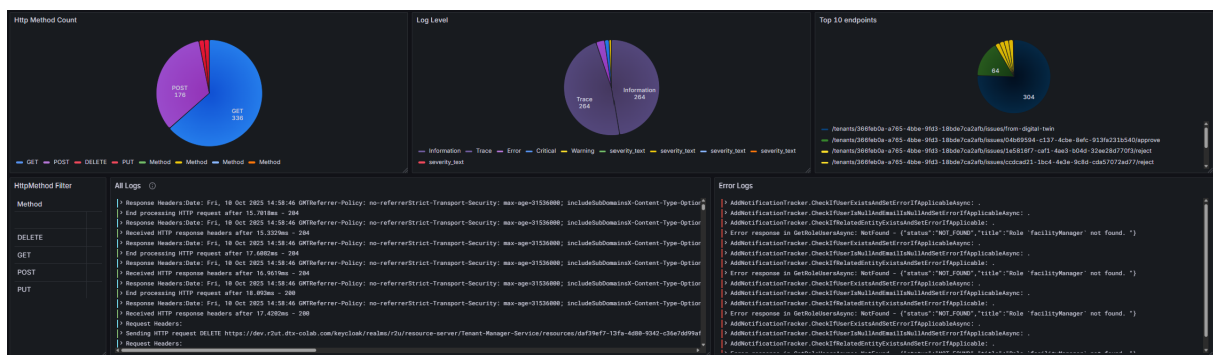


Figura 7: Dashboard de Logs

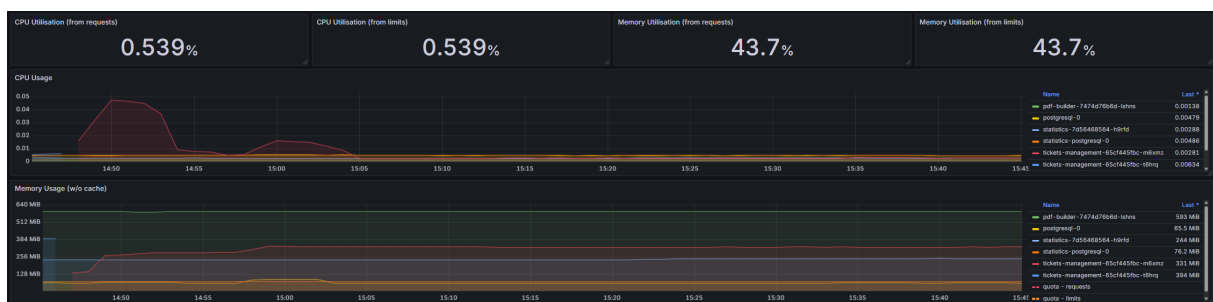


Figura 8: Uso de CPU e Memória



Figura 9: Dashboard de métricas: p95, Taxa de Erro, Pedidos por segundo

5.3 Correlação entre Logs e Traces

Um dos principais ganhos trazidos pelo Grafana é a correlação direta entre logs e traces. Com um dashboard dedicado, foi possível implementar múltiplos filtros, incluindo o `service_name`, que permitem visualizar os logs emitidos por diferentes serviços e vinculá-los diretamente às requisições (traces) correspondentes que os originaram.

Cada entrada de log disponibiliza um link direto para o trace associado, simplificando significativamente a depuração e o diagnóstico de problemas. Este recurso proporciona uma visão holística do estado do sistema e das suas interações, facilitando a identificação rápida de causas de falhas ou degradação do serviço.

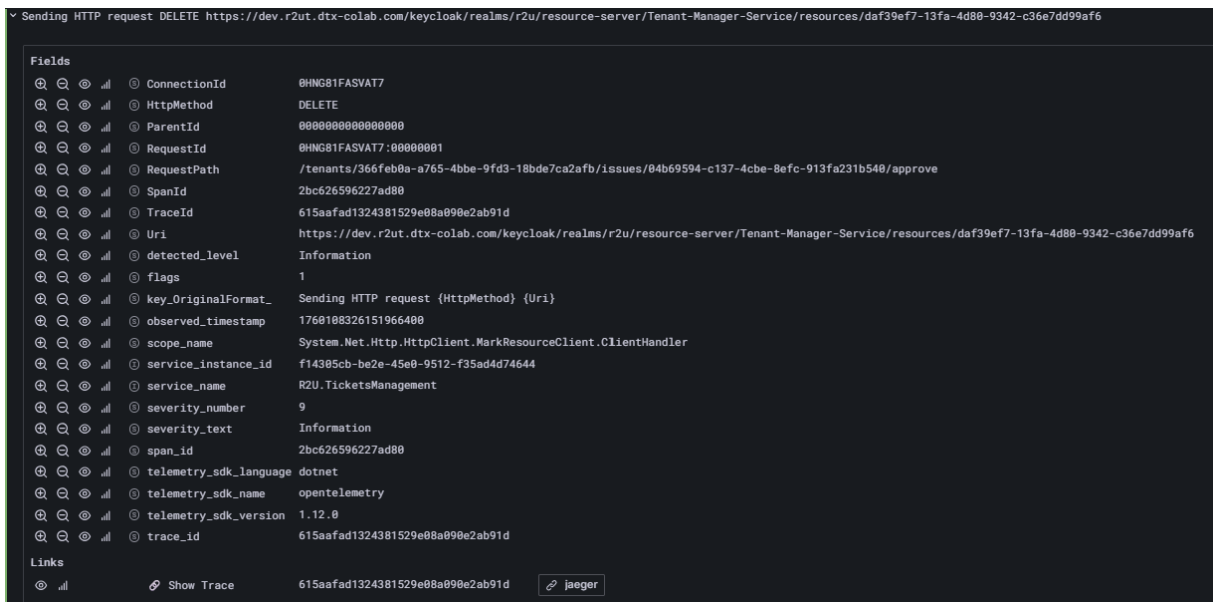


Figura 10: Log: Correlação Log e Trace

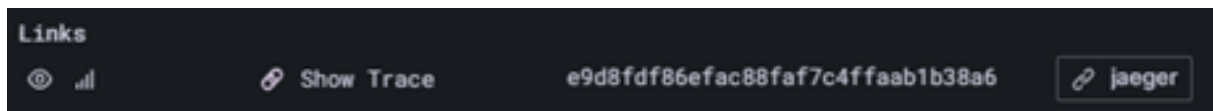


Figura 11: Link para Trace

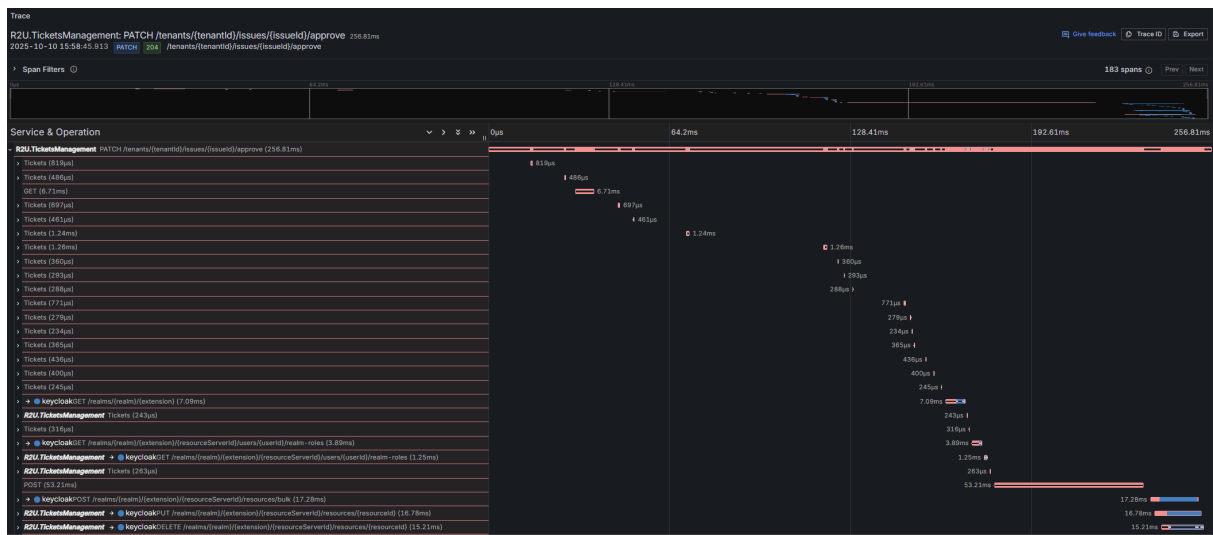


Figura 12: Trace: Correlação Log e Trace

5.4 Alertas

todo: falar um pouco sobre os alertas, ainda nao implementados, mas de configuracao muito simples

Capítulo 6

Conclusões e Trabalho Futuro

6.1 Conclusões

O trabalho desenvolvido permitiu conceber e implementar uma solução de monitorização e observabilidade para o projeto *R2UT*, reforçando a visibilidade e a resiliência da sua infraestrutura de microserviços. A integração das ferramentas *open source* *Prometheus*, *Grafana*, *Loki*, *Jaeger* e *OpenTelemetry Collector* revelou-se eficaz para recolher, centralizar e correlacionar métricas, *logs* e *traces* de forma padronizada.

Durante o desenvolvimento, foram superados diversos desafios técnicos relacionados com a orquestração dos componentes, a instrumentação da aplicação e a gestão de recursos em ambiente *cloud-native*. Estes obstáculos contribuíram para um melhor entendimento das boas práticas de observabilidade e consolidaram a robustez da solução final.

Os resultados obtidos demonstram uma redução significativa dos tempos médios de deteção e resolução de incidentes (*MTTD* e *MTTR*), bem como uma melhoria da estabilidade operacional do sistema. Em síntese, foi validada a viabilidade de uma pilha de observabilidade escalável e aberta, totalmente baseada em tecnologias de utilização livre, alinhada com os objetivos do projeto *R2UT*.

6.2 Trabalho Futuro

Como continuidade deste trabalho, prevê-se a expansão da solução de observabilidade através da integração de mecanismos de automação e inteligência artificial para análise preditiva de métricas e deteção de anomalias. A aplicação de técnicas de *machine learning* poderá permitir a antecipação de falhas e a geração de alertas inteligentes com base em padrões históricos de comportamento.

Adicionalmente, pretende-se aprofundar a integração com ferramentas de orquestração em larga escala, avaliando o desempenho do sistema em ambientes *multi-tenant* e cenários de elevada carga. Também se prevê a criação de painéis avançados de *dashboards* e relatórios dinâmicos, bem como o

estudo de políticas de *autoscaling* e de recuperação automática de serviços.

Por fim, a disseminação dos resultados e a integração desta solução no ecossistema de produção da plataforma *R2UT* representam uma oportunidade de validação real do seu impacto, contribuindo para uma infraestrutura mais inteligente, resiliente e observável.

Bibliografia

- Omer Aziz, Muhammad Shoaib Farooq, Adnan Abid, Rubab Saher, and Naeem Aslam. Research trends in enterprise service bus (esb) applications: A systematic mapping study. *IEEE Access*, 8, 2020. ISSN 21693536. doi: 10.1109/ACCESS.2020.2972195.
- Sabrina E. Bailey, Sneha S. Godbole, Charles D. Knutson, and Jonathan L. Krein. A decade of conway's law: A literature review from 2003-2012. In *Proceedings - 2013 3rd International Workshop on Replication in Empirical Software Engineering Research, RESER 2013*, 2013. doi: 10.1109/RESER.2013.14.
- Saman Barakat. Monitoring and analysis of microservices performance. *Journal of Computer Science & Control Systems*, 10:19–22, 2017. ISSN 18446043. URL <http://ezp.waldenulibrary.org/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=iih&AN=124588981&site=ehost-live&scope=site>.
- Grzegorz Blinowski, Anna Ojdowska, and Adam Przybylek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10, 2022. ISSN 21693536. doi: 10.1109/ACCESS.2022.3152803.
- Brendan Burns. *Designing Distributed Systems*, volume 53. 2015.
- Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59, 2016. ISSN 15577317. doi: 10.1145/2890784.
- Marcello Cinque, Raffaele Della Corte, and Antonio Pecchia. Microservices monitoring with event logs and black box execution tracing. *IEEE Transactions on Services Computing*, 15, 2022. ISSN 19391374. doi: 10.1109/TSC.2019.2940009.
- Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, today, and tomorrow*. 2017. doi: 10.1007/978-3-319-67425-4_12.

- Jezz Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 2010.
- Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonca, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead, 2018. ISSN 07407459.
- Muhammad Shoaib Khan, Abudul Wahid Khan, Faheem Khan, Muhammad Adnan Khan, and Taeg Keun Whangbo. Critical challenges to adopt devops culture in software organizations: A systematic review. *IEEE Access*, 10, 2022. ISSN 21693536. doi: 10.1109/ACCESS.2022.3145970.
- Guntoro Yudhy Kusuma and Unan Yusmaniar Oktiawati. Application performance monitoring system design using opentelemetry and grafana stack. *Journal of Internet and Software Engineering*, 3, 2022. doi: 10.22146/jise.v3i1.5000.
- Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. Microservices. *IEEE Software*, 35, 2018. ISSN 07407459. doi: 10.1109/MS.2018.2141030.
- James Lewis. Microservices - a definition of this new architectural term, 2014.
- Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou, and Zhang Li. Microservices: Architecture, container, and challenges. In *Proceedings - Companion of the 2020 IEEE 20th International Conference on Software Quality, Reliability, and Security, QRS-C 2020*, 2020. doi: 10.1109/QRS-C51114.2020.00107.
- Benjamin Mayer and Rainer Weinreich. A dashboard for microservice monitoring and management. In *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, 2017. doi: 10.1109/ICSAW.2017.44.
- Sam Newman. *Building Microservices*. 2015.
- M. NYGARD. *Release It!: Design and Deploy Production-Ready Software. The Pragmatic Bookshelf*. 2018.
- Njegos Railic and Mihajlo Savic. Architecting continuous integration and continuous deployment for microservice architecture. In *2021 20th International Symposium INFOTEH-JAHORINA, INFOTEH 2021 - Proceedings*, 2021. doi: 10.1109/INFOTEH51037.2021.9400696.
- Chris Richardson. Microservices patterns. *Manning Publications*, 2018.

- Fabio Gomes Rocha, Michel S. Soares, and Guillermo Rodriguez. Patterns in microservices-based development: A grey literature review. In *CibSE 2023 - XXVI Ibero-American Conference on Software Engineering*, 2023. doi: 10.5753/cibse.2023.24693.
- David S. Rogers. Implementing domain-driven design (by v. vernon). *ACM SIGSOFT Software Engineering Notes*, 47, 2022. ISSN 0163-5948. doi: 10.1145/3539814.3539822.
- Jacopo Soldani and Antonio Brogi. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Computing Surveys*, 55, 2022. ISSN 15577341. doi: 10.1145/3501297.
- Jacopo Soldani, Damian Andrew Tamburri, Willem-Jan Van, and Den Heuvel. The pains and gains of microservices the journal of systems and software the pains and gains of microservices: A systematic grey literature review. *The Journal of Systems and Software*, 146, 2018.
- Salman Taherizadeh and Marko Grobelnik. Key influencing factors of the kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. *Advances in Engineering Software*, 140, 2020. ISSN 18735339. doi: 10.1016/j.advengsoft.2019.102734.
- Aadi Thakur and M. B. Chandak. review on opentelemetry and http implementation. *International journal of health sciences*, 2022. ISSN 2550-6978. doi: 10.53730/ijhs.v6ns2.8972.
- Mario Villamizar, Oscar Garces, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Colombian Computing Conference, 10CCC 2015*, 2015. doi: 10.1109/ColumbianCC.2015.7333476.

Parte II

Apêndices

