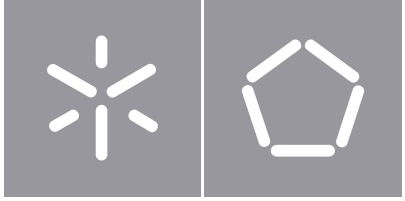
Universidade do Minho
Escola de Engenharia

Rui Pedro Chaves Silva Lousada Alves

Monitorização de uma Arquitetura de Micro Serviços

outubro 2025



Universidade do Minho
Escola de Engenharia

Rui Pedro Chaves Silva Lousada Alves

Monitorização de uma Arquitetura de Micro Serviços

Mestrado em Engenharia Informática

Dissertation supervised by
Professor Orlando Belo

outubro 2025

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

Lista de Figuras

Lista de Tabelas

Parte I

Material Introdutório

Capítulo 1

Introdução

1.1 Contextualização

As arquiteturas de microsserviços 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 microsserviços, 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 microsserviços, 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 microsserviços 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 microsserviços 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 microsserviços, 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 microsserviços. 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 microsserviços, 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 microsserviços, explorando a sua emergência como paradigma moderno no desenvolvimento de sistemas distribuídos. São discutidos os princípios que regem este modelo arquitetónico, 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 microsserviços.

- **Capítulo 5 – Visualização e Análise dos Dados**

  Este capítulo apresenta os mecanismos de visualização e análise de dados implementados para suportar a monitorização do sistema. São explorados os dashboards desenvolvidos no Grafana, evidenciando a forma como métricas, *logs* e *traces* são agregados e correlacionados para facilitar o diagnóstico de problemas e a avaliação do desempenho do sistema. Adicionalmente, demonstra-se o processo analítico seguido, desde a deteção de comportamentos anómalos até à identificação da causa-raiz, bem como a integração de alertas automáticos através do *Prometheus AlertManager* com notificação em canais operacionais (*Slack*). Este capítulo visa demonstrar a eficácia da solução

proposta na operacionalização da observabilidade, destacando a sua utilidade prática na supervisão contínua de ambientes baseados em microsserviços.

- **Capítulo 6 – 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 2

Microsserviços

Nos últimos anos, as arquiteturas de microsserviços 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 microsserviços, 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 microsserviços, 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 microsserviços. 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 microsserviços cada componente opera de forma independente, comunicando através de um API Gateway e podendo utilizar bases de dados próprias.

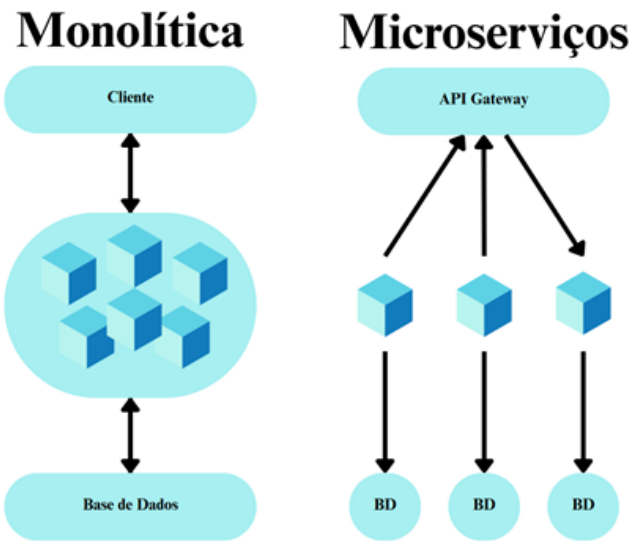


Figura 1: Comparação entre arquitetura monolítica e arquitetura de microsserviços

2.1.2 De SOA a Microsserviços

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.** Padrões como SOAP (Simple Object Access Protocol) e WS-*, um conjunto de especificações de *Web Services* que inclui funcionalidades como segurança, autenticação e transações distribuídas, requeriam a utilização de mensagens estruturadas em XML e contratos rígidos entre serviços. Embora estes protocolos garantissem interoperabilidade padronizada e mecanismos avançados de fiabilidade e segurança, introduziam também uma camada significativa de complexidade, tornando o processo de integração mais pesado, rígido e pouco ágil.

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.

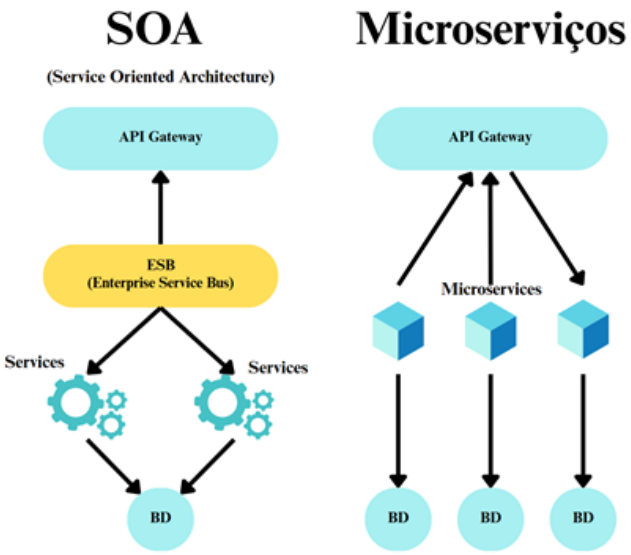


Figura 2: Comparação entre arquitetura SOA e arquitetura de microsserviços

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.

- **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 microsserviços 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 microsserviços.

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 microsserviços. 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 Microsserviços

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 especifica 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 microserviços 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 microserviços 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, momeadamente:

- **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 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 Microsserviços

Característica	**Arquitetura Monolítica**	**Arquitetura SOA**	**Arquitetura de Microserviços**
Âmbito funcional	Abrangente e integrado	Serviços de grande escala	Serviços pequenos e focados
Comunicação	Interna (memória local)	Middleware corporativo (ESB)	APIs leves (HTTP/-gRPC)
Deployment	Único e centralizado	Parcial, frequentemente acoplado ao ESB	Independente por serviço
Dados	Centralizados	Parcialmente descentralizados	Totalmente descentralizados
Autonomia no desenvolvimento	Reduzida	Moderada	Elevada

2.3 Os Desafios das Arquiteturas de Microsserviços

A arquitetura de microsserviços 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 microsserviços 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 microsserviços, bem como os desafios emergentes da sua adoção em larga escala.

2.3.1 Organização de Sistemas de Microsserviços

Os sistemas baseados em microsserviços 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 microsserviços 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 microsserviços 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 microsserviços 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 microsserviços 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 manu-

tenção 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 Microsserviços

A monitorização de sistemas de software é um pilar fundamental para garantir a disponibilidade, a performance e a evolução contínua das aplicações modernas. Com a emergência das arquiteturas baseadas em microsserviços, a importância da monitorização aumentou substancialmente, refletindo a complexidade e a natureza altamente distribuída destes sistemas. Ao contrário das arquiteturas monolíticas, nas quais a monitorização podia ser realizada através da análise de um conjunto reduzido de componentes centralizados, os microsserviços exigem uma abordagem distribuída, integrando métricas, *logs* e *tracing* distribuído para assegurar um elevado nível de visibilidade e controlo operacional sobre o sistema.

Neste capítulo analisa-se a importância estratégica da monitorização em sistemas de microsserviços, apresentam-se as principais abordagens e ferramentas utilizadas e discutem-se os desafios associados à recolha e correlação de dados em ambientes distribuídos. O objetivo é fornecer uma perspetiva crítica e fundamentada sobre o papel da monitorização na garantia de fiabilidade, desempenho e capacidade de adaptação destas arquiteturas.

3.1 A Importância da Monitorização

3.1.1 Monitorização de Arquiteturas Monolíticas e de Microsserviços

Em sistemas monolíticos tradicionais, a aplicação é geralmente executada num único processo ou num pequeno conjunto de processos homogéneos Villamizar et al. (2015). A monitorização desses sistemas pode, assim, centrar-se em métricas simples, como a utilização de CPU, o tempo de resposta global ou a disponibilidade de uma base de dados centralizada.

Em contraste, em arquiteturas de microsserviços, o sistema é composto por dezenas ou centenas de serviços autónomos, cada um com o seu próprio ciclo de vida, ambiente de execução e sistema de

armazenamento de dados Newman (2015). Além disso, cada interação entre serviços constitui uma potencial fonte de falha, e as comunicações distribuídas aumentam significativamente a complexidade de detetar e diagnosticar problemas.

Monitorizar um sistema baseado em microsserviços requer, por isso, uma abordagem holística e distribuída, na qual cada serviço deve ser instrumentado individualmente e os dados devem ser agregados de forma coerente para suportar a análise e o diagnóstico do comportamento global do sistema.

3.1.2 Impacto da Monitorização na Fiabilidade e na Escalabilidade

Uma estratégia de monitorização adequada é indispensável para assegurar o funcionamento correto de sistemas distribuídos baseados em microsserviços. A recolha contínua de métricas operacionais, registos de execução e informação sobre interações entre serviços permite antecipar problemas e garantir o desempenho esperado da plataforma.

A monitorização eficaz de sistemas distribuídos 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.
- **Apoiar a escalabilidade dinâmica.** Dados de utilização em tempo real permitem ajustar a capacidade dos serviços conforme a procura, tirando partido dos mecanismos elásticos disponíveis em plataformas de computação em nuvem, isto é, a capacidade de ajustar automaticamente os recursos disponíveis de acordo com a carga de trabalho, aumentando-os em períodos de maior procura e reduzindo-os quando a atividade diminui. Dragoni et al. (2017).

Sem uma infraestrutura de monitorização robusta, a operação de sistemas de microsserviços torna-se arriscada e dificilmente sustentável a longo prazo.

3.1.3 Principais Objetivos da Monitorização

Os principais objetivos da monitorização em microsserviços podem ser resumidos em três grandes áreas, conforme referido em 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.
- **Seguimento de requisições entre serviços (tracing).** Acompanhar o percurso das requisições entre serviços para identificar rapidamente o ponto de falha e a origem de erros.

3.2 Técnicas, Estratégias e Ferramentas de Monitorização

Existem três pilares clássicos de monitorização em microsserviços Soldani et al. (2018):

- **Logs centralizados.** Integram registos detalhados de eventos e exceções, estruturados e agregados num sistema de pesquisa e análise centralizado.
- **Métricas.** Indicadores quantitativos agregados, como taxas de erro, latência e *throughput*.
- **Tracing distribuído.** Registo do percurso de transações que atravessam múltiplos serviços.

Cada pilar fornece uma perspetiva complementar sobre o estado do sistema e, combinados, permitem uma análise completa e eficaz.

3.2.1 Logs Centralizados

Os *logs* são registos de eventos que ocorrem numa aplicação, detalhando o que aconteceu e quando ocorreu. Estes são essenciais para a resolução de problemas e para compreender o comportamento do sistema. Numa arquitetura de microsserviços, cada serviço gera os seus próprios *logs*; sem centralização, gerir e correlacionar estes dados torna-se extremamente difícil, dificultando o rastreamento e a depuração de falhas Soldani and Brogi (2022).

Um sistema de registo centralizado agrega os *logs* de múltiplas fontes, permitindo monitorização em tempo real, pesquisa avançada e análise visual. A utilização de registos estruturados (ex.: JSON) com campos consistentes (data/hora, nome do serviço, códigos de erro) facilita a análise automática. IDs de correlação propagados entre serviços permitem rastrear o ciclo completo de uma requisição, facilitando o diagnóstico de problemas Fu et al. (2012).

Ferramentas como *Elastic Stack (ELK)* e *Grafana Loki* permitem armazenar, processar e visualizar *logs* de forma eficiente Bajer (2017). Boas práticas incluem formatos estruturados, IDs de transação e políticas de retenção e rotação bem definidas.

3.2.2 Monitorização de Métricas

As métricas são valores numéricos recolhidos ao longo do tempo e fornecem uma visão quantificável da saúde e tendências de desempenho de um sistema. Funcionam como indicadores do "painel de controlo", alertando para possíveis problemas antes de impactarem os utilizadores Burns (2015). Entre as métricas essenciais para microsserviços destacam-se:

- **Latência.** Tempo necessário para um serviço responder a um pedido.
- **Throughput.** Número de pedidos processados com sucesso por segundo.
- **Taxa de erros.** Percentagem de falhas em chamadas ou pedidos.

As métricas podem ser de infraestrutura (CPU, memória, disco), de aplicação (latência, número de pedidos) ou de utilizador final (tempo de carregamento). A recolha pode ocorrer por *push* ou *pull*. Ferramentas como o *Prometheus* recolhem métricas de séries temporais e permitem análises, criação de alertas e visualização de tendências Burns (2015).

3.2.3 Tracing Distribuído

O *tracing* distribuído é um processo essencial e único de monitorização, que rastreia pedidos individuais à medida que fluem através de um sistema complexo e distribuído Sambasivan et al. (2014). Este processo proporciona visibilidade de ponta a ponta, bem como revela o percurso de um pedido através de vários serviços, bases de dados e comunicações entre componentes 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 ou 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. Além disso, podem também conter etiquetas e registos para contexto adicional Sambasivan et al. (2014).
- **Traces.** São uma coleção de *spans* logicamente conectados, representando 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 incluem:

- **Identificação de gargalos**, permitindo identificar exatamente qual serviço ou operação está a causar atrasos ou degradação de desempenho.
- **Depuração de problemas em produção**, fornecendo o contexto necessário para depurar problemas complexos em produção, visualizando todo o fluxo do pedido.
- **Otimização de desempenho**, ajudando a identificar chamadas desnecessárias ou operações com elevada latência.
- **Compreensão de dependências**, permitindo mapear como os serviços se conectam e interagem, oferecendo *insights* sobre relações que podem não ser evidentes apenas através do código ou documentação arquitetónica.

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 anteriormente, 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 monitorização de *backend* específica, oferecendo flexibilidade e preparação para o futuro. Cada serviço propaga informações de *tracing* nos cabeçalhos HTTP ou RPC, permitindo reconstruir o fluxo completo de execução Sigelman et al. (2010).

Entre as ferramentas mais populares para *tracing* distribuído destacam-se o Jaeger, uma plataforma *open-source*, e o próprio OpenTelemetry, que promove a padronização da recolha de *logs*, métricas e *traces*. A Tabela 2 apresenta uma comparação resumida das principais ferramentas utilizadas em ecossistemas de microsserviços para monitorização e análise.

Tabela 2: Principais ferramentas para monitorização e análise em microsserviços

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

A Tabela 2 sintetiza as principais ferramentas utilizadas na monitorização de ambientes distribuídos, destacando o respetivo foco funcional e licenciamento.

3.2.4 Capacidade de Monitorização e Diagnóstico

A capacidade de monitorizar e diagnosticar um sistema corresponde à aptidão para inferir o seu estado interno a partir das informações externas que este expõe Kalman (1960). No contexto de microsserviços, isto implica:

- Disponibilizar *logs* detalhados e consistentes;
- Manter métricas ricas e acionáveis;
- Conseguir rastrear o percurso de uma requisição ponta-a-ponta.

Uma monitorização eficaz permite que as equipas identifiquem rapidamente a causa e a origem de problemas complexos.

3.3 Desafios na Monitorização de Microsserviços

A monitorização de arquiteturas baseadas em microsserviços apresenta diversos desafios que resultam da natureza distribuída e altamente dinâmica destes sistemas. Um dos principais problemas é a elevada cardinalidade de dados: microsserviços geram grandes volumes de métricas e *logs*, frequentemente contendo identificadores únicos, endereços IP e atributos dinâmicos. Este nível de cardinalidade pode sobrecarregar bases de dados de séries temporais e dificultar a construção de consultas de análise eficientes, exigindo uma instrumentação criteriosa e controlada. Ferramentas como o Prometheus podem apresentar limitações quando confrontadas com métricas de cardinalidade extremamente alta, tornando necessária uma estratégia seletiva de recolha e agregação de dados.

Outro desafio relevante é a correlação de eventos distribuídos. A identificação da causa raiz de falhas implica cruzar informação proveniente de múltiplas fontes - métricas, *logs* e *traces*. Sem mecanismos como *tracing* distribuído ou *logs* estruturados com IDs de correlação, esta tarefa torna-se extremamente complexa Sigelman et al. (2010). Entre as boas práticas encontram-se a propagação consistente de identificadores entre serviços e a inclusão automática de metadados relevantes em *logs* e métricas.

De igual modo, a latência constitui um fator crítico. Em sistemas distribuídos, atrasos podem surgir em diversos pontos, como chamadas entre serviços, acessos a bases de dados ou comunicação pela rede. Sem mecanismos de visibilidade adequados, torna-se difícil identificar rapidamente a origem da latência e agir para mitigar o problema Railic and Savic (2021). Adicionalmente, os mecanismos de monitorização devem ser concebidos de forma a não introduzir impacto significativo no desempenho da aplicação, evitando acrescentar sobrecarga que agrave ainda mais eventuais problemas de desempenho.

3.4 Estudos de Caso: Exemplos Práticos de Monitorização em Microsserviços

A monitorização de microsserviços é um desafio significativo para as organizações que implementam esta arquitetura, dada a sua natureza distribuída e complexa. Empresas como a Netflix, a Amazon ou a Uber são exemplos de sucesso na implementação de sistemas de monitorização em grande escala. Estes casos demonstram como é possível manter a fiabilidade e a escalabilidade dos serviços enquanto se lida com a complexidade inerente aos microsserviços.

3.4.1 Netflix: Monitorização em Escala Global

A Netflix, pioneira na adoção de microsserviços, 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 microsserviços 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 conceito de *Chaos Engineering*, uma prática que envolve a injeção intencional de falhas no sistema para testar a sua resiliência Basiri et al. (2019). Testes como a suspensão de serviços ou o aumento de latência permitem identificar pontos de fragilidade e garantir que a plataforma é capaz de se recuperar rapidamente de falhas imprevistas. A utilização de ferramentas de monitorização como o *Atlas* e o *Eureka*, em conjunto com práticas de *Chaos Engineering*, contribui para a fiabilidade do sistema, permitindo à Netflix operar em escala global com elevada 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 enorme de transações e dados. A monitorização da sua arquitetura de microsserviços 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 otimização 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 microsserviços. 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 para Escalabilidade Global

A Uber, com operação em escala global, necessita de uma plataforma de monitorização eficaz para gerir milhões de transações por minuto. A empresa utiliza uma combinação de ferramentas *open-source* para alcançar visibilidade operacional completa Newman (2015).

O *Jaeger*, uma plataforma de *tracing* distribuído, é a peça central para rastrear a jornada de cada requisição através dos seus microsserviços, permitindo 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 para suportar a procura 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 microsserviços sejam significativos, podem ser superados com a adoção de uma abordagem multifacetada. A combinação dos três pilares da monitorização - *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 elevada exigência Dragoni et al. (2017).

3.5 Futuro da Monitorização de Microsserviços

3.5.1 Tendências Futuras na Monitorização de Microsserviços

À medida que as arquiteturas de microsserviços continuam a evoluir, as ferramentas e práticas de monitorização acompanham esta evolução. O futuro da monitorização em microsserviços está associado à integração de novas tecnologias e à melhoria contínua dos mecanismos de visibilidade operacional, com o objetivo de garantir um funcionamento ainda mais eficiente, resiliente e autónomo dos sistemas distribuídos.

3.5.2 Utilização de Inteligência Artificial e Machine Learning

Uma das áreas mais promissoras na monitorização de microsserviços é a aplicação de Inteligência Artificial (IA) e *Machine Learning* (ML). Estas tecnologias podem ser utilizadas para detetar anomalias, prever falhas e otimizar o desempenho dos sistemas distribuídos. A monitorização preditiva permite identificar padrões de comportamento e ajustar automaticamente os recursos para prevenir interrupções Khan et al. (2022).

A integração de AIOps (*Artificial Intelligence for IT Operations*) nas plataformas de monitorização está a transformar a forma como os problemas são identificados e resolvidos em tempo real. Ao utilizar algoritmos de ML para analisar grandes volumes de dados de telemetria, é possível automatizar o diagnóstico e a correção de falhas, tornando a operação de microsserviços mais eficiente e autónoma Khan et al. (2022).

3.5.3 Monitorização Preditiva e Automação

A monitorização preditiva está a tornar-se uma tendência relevante na gestão de microsserviços. Ao analisar dados históricos e padrões de comportamento, os sistemas poderão prever falhas e otimizar a alocação de recursos antes mesmo de estas ocorrerem, reduzindo tempos de indisponibilidade e melhorando a resposta a eventos Kusuma and Oktiawati (2022).

A automação na instrumentação dos serviços será igualmente determinante. Ferramentas como o *OpenTelemetry*, que permitem a recolha unificada de métricas, *logs* e *traces*, deverão tornar-se cada vez mais comuns, assegurando que os dados de telemetria podem ser agregados e analisados de forma consistente e eficiente Kusuma and Oktiawati (2022).

3.5.4 Tecnologias Emergentes para Monitorização de Microsserviços

Além da IA e do ML, outras tecnologias emergentes irão influenciar o futuro da monitorização de microsserviços, nomeadamente o 5G e o *edge computing*. O 5G permitirá uma comunicação mais rápida e eficiente entre componentes distribuídos, enquanto o *edge computing* possibilitará a descentralização do processamento, reduzindo a latência e aumentando a performance das aplicações Dragoni et al. (2017).

Adicionalmente, abordagens como *serverless computing* e a evolução dos *containers* continuarão a moldar os requisitos de monitorização. A integração de ferramentas especializadas com plataformas como *Kubernetes* e *Docker* será fundamental para garantir que as aplicações escalem e funcionem de forma consistente, independentemente do modelo de execução ou orquestração Dragoni et al. (2017).

Parte II

Corpo da Dissertação

Capítulo 4

Monitorização de Serviços com OpenTelemetry

4.1 Introdução e Caracterização do Sistema Monitorado

O cenário atual do desenvolvimento de software é marcado pela crescente adoção de arquiteturas de microsserviços e ambientes *cloud-native*, suportados por plataformas de orquestração como o Kubernetes. Embora esta abordagem promova agilidade, escalabilidade e resiliência, também introduz uma complexidade significativa, especialmente no monitorização e na depuração de sistemas distribuídos. A proliferação de serviços independentes, cada um com a sua própria lógica e ciclo de vida, bem como a comunicação assíncrona entre eles, torna o seguimento de uma única requisição de ponta a ponta uma tarefa desafiadora.

Neste contexto, a monitorização avançada torna-se uma disciplina fundamental para garantir a fiabilidade, o desempenho e a resiliência destes sistemas distribuídos 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 com o objetivo de garantir flexibilidade, escalabilidade e resiliência na gestão do ciclo de vida da construção modular. Esta arquitetura é composta por diversos componentes (ou módulos), cada um responsável por uma funcionalidade específica. A estrutura modular facilita a manutenção, o desenvolvimento e a integração de novas capacidades sem impactar a operação dos restantes serviços.

A solução está organizada em camadas, sendo que cada serviço é executado de forma isolada num *container* Docker, posteriormente gerido e orquestrado pelo Kubernetes. Esta abordagem permite que os serviços operem de forma autónoma, escalando conforme necessário de acordo com a carga e as necessidades operacionais. A comunicação entre os serviços é assegurada através de *REST APIs*, *gRPC* e mensagens via *Kafka*, permitindo interoperabilidade eficiente tanto em cenários síncronos como assín-

cronos.

Além disso, a comunicação entre os diferentes componentes é realizada através de interfaces bem definidas, assegurando um elevado nível de desacoplamento entre os microsserviços e promovendo uma arquitetura extensível, robusta e facilmente evolutiva.

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 gerir os *containers* dos microserviços e garantindo, entre outras coisas:

- **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 utilização sem comprometer o seu desempenho, como no caso de um aumento de utilizadores que acedem simultaneamente aos serviços.

- **Alta disponibilidade (High Availability — HA).** O Kubernetes garante que, caso um *pod* falhe, outro seja automaticamente reiniciado num nó diferente do cluster, minimizando o impacto de falhas no sistema. Isso permite assegurar que os serviços essenciais, como autenticação ou autorização de utilizadores e dispositivos, ou mesmo a gestão de dispositivos IoT, continuem operando sem interrupção.

- **Gestão de containers.** O Kubernetes permite gerir a execução de *containers*, garantindo que todos os microsserviços estejam devidamente implantados e funcionando corretamente. Além disso, facilita a atualização e a implementação de novas versões dos serviços, assegurando a inexistência de tempos de inatividade por meio de *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 microsserviços sejam rápidas e confiáveis.

4.1.3 Arquitetura de Microsserviços no Kubernetes

A plataforma R2UT é composta por diversos serviços que interagem entre si, cada um implementado como um microsserviço em *containers*. Esses serviços são organizados num cluster Kubernetes, no qual a comunicação entre os componentes é feita através de APIs e de mensagens assíncronas via Kafka. Isto permite que o sistema seja altamente escalável e eficiente. De seguida, destacam-se alguns dos componentes chave desta arquitetura:

- **Base de dados global.** Uma base de dados partilhada por todos os serviços que necessita de escalabilidade horizontal. O Kubernetes facilita a gestão de bases de dados distribuídas, como as suportadas por PostgreSQL com Citus, garantindo elevada performance nas consultas e capacidade de escala.

- **Middleware e plataforma em nuvem.** Serviços responsáveis pela interoperabilidade entre os sistemas internos e pela gestão da infraestrutura na nuvem. O Kubernetes gere os recursos necessários para escalar estes serviços conforme as necessidades de tráfego, facilitando a implementação contínua das aplicações.

- **Módulos específicos.** *Tenant Management*, *Ticket Management*, *PDFBuilder*, *IoT Manager*, *Rules Engine*, *DAE Authentication* são módulos da arquitetura. Cada um opera de forma independente em *containers*, com comunicação entre serviços mediada através de APIs e filas de mensagens (Kafka).

- **Cluster Kafka.** O Kafka é utilizado como *broker* de mensagens para garantir a comunicação assíncrona entre os microsserviços, especialmente quando é necessário gerir grandes volumes de dados e eventos gerados por dispositivos IoT ou interações de utilizadores. O Kubernetes permite a escalabilidade do Kafka através de clusters e réplicas dos *brokers*, assegurando alta disponibilidade e tolerância a falhas.

4.1.4 Desafios da Monitorização em Sistemas Distribuídos

A implementação de sistemas distribuídos, como o ambiente Kubernetes com microsserviços, cria desafios significativos em relação à monitorização distribuída. A complexidade do sistema aumenta devido à comunicação assíncrona entre os serviços, à escalabilidade dinâmica dos *pods* e à necessidade de correlacionar eventos gerados por diferentes componentes. A deteção 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.

Uma monitorização eficaz exige que o sistema seja capaz de capturar, correlacionar e analisar as interações entre os microsserviços, além de acompanhar de forma contínua 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 uma visibilidade completa sobre o comportamento do sistema, oferecendo informação em tempo real sobre a saúde e o desempenho da aplicação. O objetivo foi criar uma solução de monitorização avançada 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 monitorizar métricas, *logs* e *traces* de forma centralizada. Isto permite a deteção imediata de anomalias, minimizando o impacto no desempenho e na experiência do utilizador.

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 identificar rapidamente a sua causa.

3. **Otimização de desempenho.** Capacitar a equipa de desenvolvimento a identificar gargalos e áreas de melhoria da solução antes que estes afetem os utilizadores finais. O uso de alertas e *dashboards* permite otimizar o sistema, ajustando componentes e recursos de acordo com a carga e as necessidades operacionais.

Nesta fase, importa referir que o foco da solução de monitorização foi deliberadamente delimitado aos microsserviços desenvolvidos internamente pela equipa do projeto, nomeadamente os módulos responsáveis pela gestão de *tickets*, gestão de utilizadores, dispositivos IoT, autenticação, regras e serviços de apoio à plataforma. 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.

4.3 O Papel do OpenTelemetry na Monitorização de Microsserviços

Em arquiteturas de microsserviços, nas quais diversos serviços isolados cooperam para servir uma única requisição, rastrear o comportamento global do sistema torna-se um desafio. Problemas como latência entre serviços, falhas silenciosas ou dependências ocultas exigem que a monitorização vá além das abordagens tradicionais. 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 *API*, *SDK* e o *OpenTelemetry Collector* para instrumentar e transportar os três sinais de telemetria - *traces*, métricas e *logs* - de forma agnóstica ao fornecedor e independente do *backend*. O objetivo é padronizar a geração e exportação de telemetria, permitindo encaminhar os dados para sistemas como *Prometheus*, *Jaeger*, *Loki* e outros, sem necessidade de 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 como *OTLP*, *Prometheus* e *Jaeger*, entre outros.

4.3.2 Principais Componentes

Embora os detalhes sejam explicados nas secções seguintes, vale apresentar aqui os blocos conceptuais do *OpenTelemetry*. São eles:

- **APIs / SDKs / Instrumentação.** As *API* definem o contrato genérico para rastreamento, métricas e *logs*, enquanto os *SDKs* concretizam esse contrato em cada linguagem, permitindo tanto a instrumentação manual como automática.

- **Exporters / Receivers.** Componentes responsáveis por enviar ou receber telemetria em formatos como *OTLP*, *Jaeger* e *Prometheus*.

- **Collector.** Componente neutro que organiza *pipelines* (*receivers - processors - exporters*), permitindo filtragem, enriquecimento, amostragem e distribuição (*fan-out*) de telemetria.

Todos estes componentes trabalham em conjunto 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 as ações de monitorização em três sinais principais:

- **Traces**, que descrevem operações distribuídas através de *spans* correlacionados;

- **Métricas**, que representam valores quantitativos observados ao longo do tempo;

- **Logs estruturados**, que contêm registos detalhados de eventos com contexto adicional.

Para permitir comparações entre serviços e linguagens, o *OTel* define convenções semânticas (*Semantic Conventions - SemConv*), um vocabulário padronizado para atributos relacionados com *HTTP*, bases de dados, envio de mensagens 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* - `service.name` é um dos atributos obrigatórios.

4.3.4 Vantagens, Limitações e Desafios do OpenTelemetry

A adoção do *OpenTelemetry* traz benefícios substanciais no contexto de arquiteturas distribuídas, sobretudo ao padronizar a recolha de telemetria e ao oferecer uma integração unificada para diversos tipos de sinais e plataformas. No entanto, tal como qualquer tecnologia emergente, implica também alguns desafios que precisam ser considerados.

Vantagens:

- **Padronização e portabilidade.** Uma única camada de instrumentação permite recolher telemetria de diferentes serviços e enviá-la para múltiplas ferramentas, reduzindo dependência de fornecedores e minimizando esforços em processos de migração ou integração.

- **Flexibilidade através do Collector.** O *OpenTelemetry Collector* oferece grande capacidade de adaptação, suportando exportação para múltiplos destinos, mecanismos de *buffering* e *retry*, amostragem inteligente e enriquecimento de dados antes do armazenamento, tudo num ponto central da arquitetura.

- **Aderência ao ecossistema cloud-native.** A ferramenta foi concebida para ambientes modernos, com suporte maduro para Kubernetes, serviços distribuídos e *auto-instrumentation* disponível para várias linguagens, incluindo .NET.

A adoção do *OpenTelemetry*, contudo, não está isenta de desafios. A complexidade de sistemas distribuídos faz com que a instrumentação, operação e gestão da telemetria requeiram planeamento cuidadoso e boas práticas consolidadas.

Limitações e Desafios:

- **Curva de configuração e operação.** Pipelines mal definidas podem provocar perda de dados, latências adicionais ou sobrecarga em recursos. O próprio Collector necessita de configuração cuidada e monitorização contínua.

- **Maturidade heterogénea entre sinais.** Embora *tracing* e métricas estejam altamente estabilizados, a componente de logs continua em evolução e pode depender mais de integrações com ferramentas externas, como Loki ou ELK.

- **Necessidade de stack complementar.** O *OpenTelemetry* fornece coleta e processamento, mas não inclui armazenamento ou visualização, tornando obrigatória a utilização de ferramentas adicionais como Prometheus, Jaeger, Loki ou Grafana para consulta e análise.

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

Antes de procedermos à comparação das diferentes alternativas, importa clarificar que a análise estabelecida entre *OpenTelemetry* e a *stack* composta por Prometheus, Grafana, Jaeger e Loki não deve ser interpretada como uma avaliação entre ferramentas equivalentes ou mutuamente exclusivas. O *OpenTelemetry* não é um *backend* de armazenamento ou visualização de dados. O seu papel situa-se na camada de instrumentação, recolha, normalização e encaminhamento de telemetria (métricas, rastreamentos e registos).

Ou seja, o *OpenTelemetry* atua como um padrão unificador para a geração e transporte de telemetria, permitindo que diferentes linguagens, bibliotecas e serviços emitam dados num formato consistente (OTLP), que podem ser processados e exportados através do *Collector*. Por contraste, a *stack* tradicional composta por Prometheus, Grafana, Jaeger e Loki cobre essencialmente funções de armazenamento persistente, consulta e visualização dos dados recolhidos.

- **Prometheus** realiza a recolha e o armazenamento de séries temporais de métricas, com consultas via PromQL.
- **Grafana** fornece visualização e gestão de alertas sobre métricas, logs e outros dados operacionais.
- **Jaeger** armazena e permite consultar rastreamentos distribuídos.
- **Loki** gere a ingestão e consulta de logs.

Assim, mesmo com *OpenTelemetry*, continuam a ser necessários sistemas de *backend* que garantam a persistência e exploração dos dados. A comparação, portanto, reflete abordagens arquiteturais distintas:

- No **modelo tradicional**, cada ferramenta exige o seu próprio *agent* ou *exporter*, resultando numa coleta fragmentada.
- Com **OpenTelemetry**, a instrumentação e coleta são unificadas, e os mesmos dados podem ser enviados para múltiplos *backends*.

Deste modo, o *OpenTelemetry* não substitui os *backends* clássicos; complementa-os ao introduzir uma camada padronizada e de abstração que reduz o acoplamento e aumenta a portabilidade da telemetria.

A Tabela 3 resume as principais diferenças entre as duas abordagens, destacando aspetos como instrumentação, padronização e flexibilidade arquitetural.

Tabela 3: Comparação entre OpenTelemetry e a *stack* tradicional de monitorização

Característica	**OpenTelemetry (OTel)**	**Prometheus + Grafana + Jaeger + Loki**
Instrumentação	Unificada (traces, métricas, logs)	Separada por ferramenta
Padronização	Convenções Semânticas (*SemConv*)	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
Dependência de fornecedor	Baixa	Média / Alta
Escalabilidade	Elevada	Elevada, mas com mais componentes dedicados
Curva de aprendizagem	Moderada (pipelines OTel)	Mais baixa em cenários simples

4.5 Arquitetura Detalhada e Implementação da Solução

4.5.1 Visão Geral da Arquitetura

A arquitetura do sistema de monitorização (Figura 3) é concebida com uma abordagem modular, na qual cada componente desempenha um papel claro e isolado. Esta separação favorece a flexibilidade, a escalabilidade e o desacoplamento entre a aplicação e a infraestrutura responsável pela recolha e análise da telemetria.

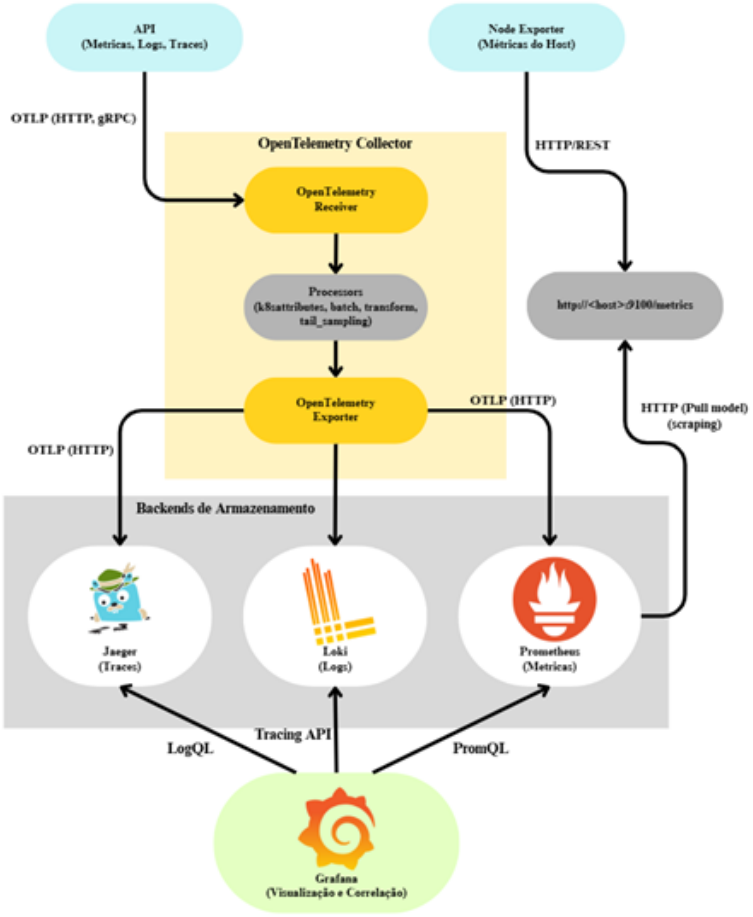


Figura 3: Arquitetura da solução de monitorização com OpenTelemetry, Prometheus, Loki, Jaeger e Grafana

A Figura 3 ilustra o fluxo de dados de telemetria desde a sua origem nos microsserviços até à visualização final no Grafana. Nesse fluxo, os dados são primeiro gerados e enriquecidos nas aplicações, depois enviados para o *Collector*, onde são processados, transformados e encaminhados para diferentes sistemas de armazenamento, conforme o seu tipo (métricas, logs ou *traces*). Finalmente, o Grafana agrega estes dados e proporciona uma visualização unificada para análise operacional e diagnóstico. Esta divisão

clara entre instrumentação, processamento e visualização confere ao sistema maior flexibilidade e evita o acoplamento entre os serviços da aplicação e a infraestrutura de monitorização.

APIs e SDKs

As API definem os contratos para a geração e correlação de dados de telemetria. Os SDK são implementações específicas de linguagem das APIs, fornecendo as ferramentas necessárias para instrumentar o código das aplicações. Estes SDK 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 & 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 & 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 monitorização (OpenTelemetry Authors, 2025; Thakur & Chandak, 2022).

Node Exporter

O Node Exporter é utilizado como agente de monitorização ao 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 monitorização das aplicações com indicadores do ambiente de execução.

Inicialmente, tentou-se recolher estas métricas através do *OpenTelemetry Collector*, de forma a centralizar todo o processo de monitorização, incluindo as métricas de infraestrutura num único pipeline de recolha e exportação. A intenção era utilizar o Collector como ponto único de integração, encaminhando todos os dados para os respetivos backends (Prometheus, Loki e Jaeger).

Contudo, durante a implementação, verificou-se que a configuração do Collector para recolher métricas do sistema operativo exigia componentes e *receivers* adicionais, cuja compatibilidade e maturidade ainda se encontram limitadas. Para além disso, o nível de detalhe e granularidade obtido não correspondia ao fornecido pelo Node Exporter, tornando essa abordagem menos prática.

Por este motivo, optou-se por manter o Node Exporter como fonte dedicada para métricas de sistema, uma vez que a sua integração direta com o Prometheus é simples, amplamente documentada e de implementação imediata. Assim, o *OpenTelemetry Collector* permanece responsável pela recolha e encaminhamento dos dados das aplicações (métricas, logs e *traces*) para os seus destinos (Prometheus, Loki e Jaeger). Esta abordagem revelou-se mais simples, estável e adequada ao objetivo de obter uma visão completa tanto da infraestrutura como das aplicações.

Camada de Visualização e Análise

O Grafana atua como uma interface de utilizador unificada para visualizar e analisar os dados. Esta ferramenta integra os backends (Prometheus, Loki e Jaeger), permitindo a criação de dashboards interativos que mostram as métricas, logs e traces de maneira correlacionada, facilitando o diagnóstico e a tomada de decisões operacionais.

4.5.2 Fluxo de Telemetria

O fluxo de dados de telemetria segue um *pipeline* bem definido que separa instrumentação, recolha, processamento e armazenamento/visualização, conforme ilustrado na Figura 4. Esta organização permite correlação ponta a ponta com baixo acoplamento aos *backends*.

1. A aplicação *ASP.NET Core* emite telemetria (logs, métricas, *traces*) via *OTLP gRPC* e envia-a 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* do *Prometheus*.

4. Os dados são encaminhados para os *processors* do *Collector*, que executam transformação (*transform*), enriquecimento de atributos (*attributes*), agregação (*batch*) e amostragem de *traces* (*tail_sampling*).

5. Após o processamento, os dados são enviados pelos *exporters* para os 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* (PromQL, LogQL, *Tracing API*) para gerar *dashboards* interativos e correlações.

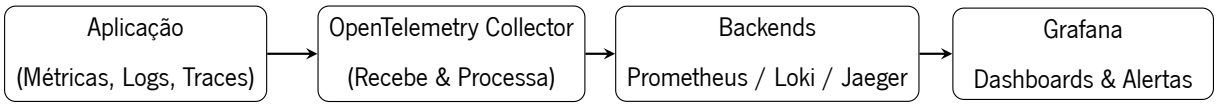


Figura 4: Pipeline simplificado de telemetria na arquitetura implementada

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 microsserviços 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 R2UT, 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 microsserviços 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 infraestrutura de monitorização.

Esta estratégia centraliza a lógica de monitorização num único ponto, reduzindo significativamente o código repetido. A configuração da telemetria pode ser ativada e controlada dinamicamente através de variáveis de ambiente, conferindo flexibilidade e portabilidade entre diferentes ambientes (desenvolvimento, homologação e produção). Para instrumentar novos serviços, a intervenção necessária é mínima: basta adicionar o pacote *DTX.Base.Common*, invocar um método de extensão no `Program.cs` e definir as variáveis de ambiente correspondentes.

A lógica de monitorização foi centralizada num único método de extensão, aplicado na inicialização de cada API NET:

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

Este método ativa automaticamente os componentes do OpenTelemetry responsáveis pela instrumentação de *tracing* distribuído, recolha de métricas e exportação de logs estruturados. As opções de configuração são controladas dinamicamente por variáveis de ambiente, evitando alterações no código ou recompilação ao migrar entre ambientes.

As variáveis de ambiente listadas na Tabela 4 são responsáveis por configurar a camada de exportação OTLP em cada microsserviço. Em vez de definir endpoints ou credenciais diretamente no código, estes parâmetros são injetados no ambiente de execução, permitindo separar configuração operacional da lógica de aplicação.

A variável `OTEL_EXPORTER_OTLP_ENDPOINT` determina o endereço do Collector para onde os dados serão enviados, garantindo flexibilidade na migração entre ambientes (por exemplo, ambiente local, cluster de homologação ou produção em Kubernetes). A variável `OTEL_EXPORTER_OTLP_PROTOCOL` define o protocolo de comunicação a utilizar, assegurando compatibilidade com diferentes configurações de rede e segurança. Por fim, `OTEL_EXPORTER_OTLP_HEADERS` permite incluir cabeçalhos adicionais, como tokens de autenticação ou etiquetas de contexto, possibilitando uma integração segura e multi-tenant quando necessário.

Este modelo de configuração facilita a automação de *deploys*, promove maior segurança operacional e reduz a necessidade de alterações no código ao longo do ciclo de vida do sistema.

Variável	Descrição
`OTEL_EXPORTER_OTLP_ENDPOINT`	URL do *endpoint* 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`).

Tabela 4: Variáveis de ambiente utilizadas para configuração da exportação OTLP

Embora a instrumentação explícita via `DTX.Base.Common` fosse o padrão adotado, também foi avaliada a possibilidade de implementação da instrumentação *zero-code* no ambiente .NET, utilizando agentes e configurações automáticas do *OpenTelemetry*. No entanto, foram identificadas algumas limitações no controlo da granularidade dos dados e na integração com determinadas bibliotecas utilizadas internamente. Por esse motivo, optou-se por encapsular a instrumentação num pacote comum, garantindo padronização e flexibilidade.

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. De referir:

- **Padronização da Instrumentação entre Múltiplas APIs**, que 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 Monitorização**, que faz com que o código da aplicação se torne independente das ferramentas de backend utilizadas (*Grafana*, *Jaeger*, *Tempo*, etc.). Se houver uma mudança nas ferramentas de monitorização, 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**, que assegura que a ativação e o ajuste da observabilidade sejam 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**, que, ao centralizar a lógica de observabilidade num único ponto, evita 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 monitorização seja uma parte integrante do ciclo de vida da aplicação desde o início.

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 monitorização. É responsável por capturar logs, métricas e traces gerados pelas aplicações instrumentadas, agentes *sidecar* ou ferramentas de terceiros. A 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, uma vez que é ela que garante que os dados cheguem de forma consistente e em tempo real ao pipeline de monitorização.

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 garantia de 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 O Protocolo OTLP (gRPC/HTTP)

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 o *Grafana Loki* (logs), o *Jaeger* (traces) ou *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 monitorização, garantindo 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 de 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.

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.

O recurso `OpenTelemetryCollector`, disponibilizado pelo *OpenTelemetry Operator*, permite definir, de forma declarativa, o modo de execução do Collector, conferindo elevada flexibilidade na adaptação da arquitetura de monitorização às características do sistema. Os principais modos de operação são:

- **DaemonSet.** Executa uma instância do Collector em cada nó do cluster, recolhendo a telemetria localmente e reduzindo a latência e o tráfego de rede. Este modo é particularmente indicado para clusters de grande dimensão ou aplicações com elevado volume de métricas e logs.

- **Deployment.** Executa o Collector como um serviço centralizado, atuando como *gateway* e agregando telemetria antes de a reenviar para os backends definidos. É uma opção recomendada para ambientes de desenvolvimento ou para arquiteturas onde a centralização seja operacionalmente vantajosa.

- **Sidecar.** Executa o Collector em conjunto com cada *pod*, garantindo o nível máximo de isolamento e controlo sobre a instrumentação de cada serviço. Este modelo oferece monitorização altamente granular, embora com maior custo de recursos.

- **StatefulSet.** Utilizado em cenários onde é necessário garantir estado persistente ou uma configuração estável e ordenada, útil em pipelines de telemetria mais complexos ou ambientes de auditoria.

Além desta flexibilidade arquitetural, a utilização do *OpenTelemetry Operator* oferece benefícios adicionais:

- **Automação.** Elimina a necessidade de escrever e manter manualmente manifestos Kubernetes complexos para cada instância do Collector, simplificando significativamente a gestão operacional e reduzindo o risco de erros de configuração.

- **Consistência e padronização.** Garante que todas as instâncias do Collector são configuradas de forma homogénea e alinhadas com boas práticas, assegurando uniformidade na recolha e processamento da telemetria em todo o cluster.

- **Integração nativa com GitOps.** Por ser baseado em definições declarativas, integra-se facilmente com pipelines de CI/CD e ferramentas GitOps, como ArgoCD e Flux, facilitando a gestão versionada e auditável das configurações de monitorização.

- **Evolução contínua e alinhamento com a comunidade.** O Operator é mantido pela comunidade OpenTelemetry, garantindo compatibilidade com novas versões, boas práticas atualizadas e reduzindo o risco de configurações obsoletas ao longo do tempo.

Em síntese, o *OpenTelemetry Operator* simplifica substancialmente a adoção de monitorização em Kubernetes, promovendo maior automação, padronização e escalabilidade, ao mesmo tempo que reduz o esforço manual e a probabilidade de erros operacionais.

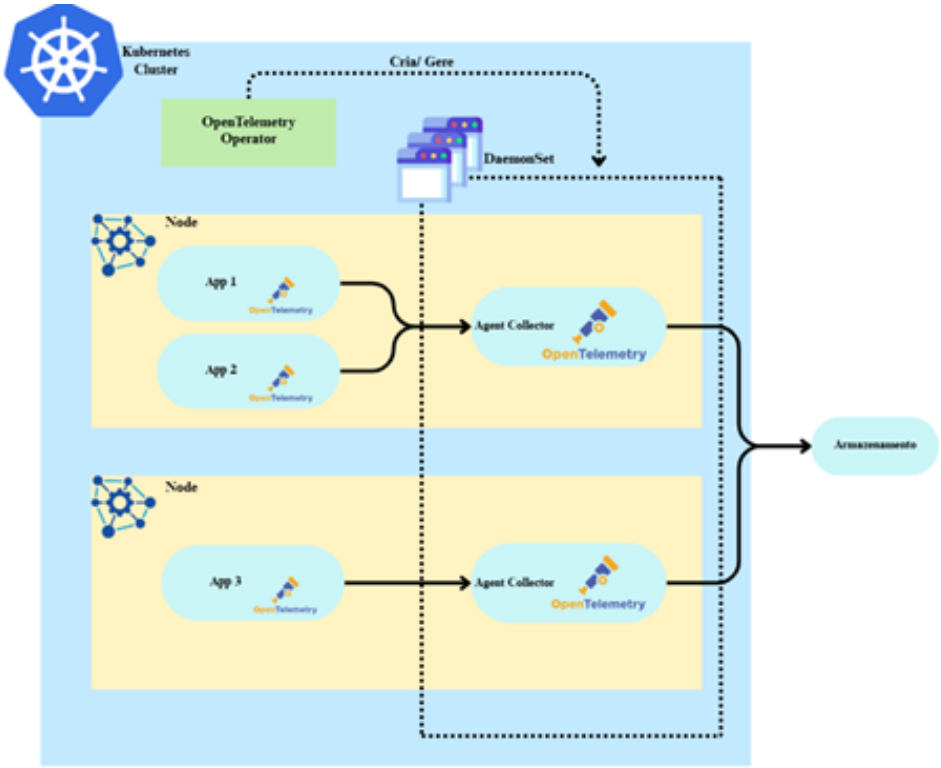


Figura 5: Arquitetura de coleta local com *OpenTelemetry Collector* em modo *DaemonSet*.

A abordagem adotada neste trabalho foi a execução do *OpenTelemetry Collector* como um **DaemonSet**, disponibilizando um agente local em cada nó do cluster Kubernetes (Figura 5). Em cada nó, um *Pod* do Collector expõe *receivers* OTLP por gRPC (porta 4317) e HTTP/Protobuf (porta 4318), recebendo métricas, logs e traces das *workloads* residentes nesse nó.

Esta decisão foi guiada por três objetivos principais:

- **Baixa latência na entrega de telemetria.** Evitando transmissões de rede adicionais antes do primeiro processamento;
- **Resiliência local.** Confinando o impacto de eventuais falhas ao nó específico, sem afetar o resto do cluster;
- **Simplicidade de integração.** Permitindo que as aplicações publiquem telemetria para um único *endpoint* local, sem a necessidade de mecanismos adicionais de descoberta.

A adoção do padrão *DaemonSet* resultou numa menor variabilidade na entrega dos sinais, com coleta e pré-processamento a nível local, isolamento de falhas por nó, redução da dependência de rede intra-cluster e uma configuração simplificada do lado das aplicações, uma vez que estas comunicam com um *endpoint* local único.

4.8 Processamento e Transformação dos Dados

Após a sua recolha, os dados de telemetria passam por uma fase crítica de processamento dentro do *OpenTelemetry Collector*. Esta etapa, orquestrada por diversos *processors*, é essencial para modificar, enriquecer, agrupar ou filtrar os dados, assegurando a sua consistência, eficiência e utilidade para os sistemas subsequentes de armazenamento e análise.

4.8.1 Processadores e as suas Aplicações

Após a fase de recolha, os dados de telemetria passam por um estágio fundamental de processamento dentro do *OpenTelemetry Collector*. Este processamento é realizado por um conjunto de *processors*, responsáveis por transformar, enriquecer, agrupar e filtrar os dados. Esta etapa assegura que a telemetria seja consistente, eficiente e alinhada com as necessidades de análise, conformidade e desempenho da solução implementada.

De forma a consolidar o papel de cada componente, a Tabela 5 apresenta os principais *processors* utilizados, a sua função primária e respetivas aplicações práticas.

Tabela 5: Principais *processors* do OpenTelemetry Collector utilizados

Processor	**Função Primária**	**Aplicações Chave**	**Benefício para Monitorização**
`transform`	Modifica dados com OTTL.	Adiciona `service.name`, normaliza severidade, converte timestamps.	Consistência semântica e refinamento da telemetria.
`batch`	Agrupa dados em lotes.	Reduz chamadas de rede e overhead.	Melhora o débito e otimiza a exportação.
`attributes`	Adiciona ou altera atributos.	Injeta metadados como ambiente e host.	Melhora correlação, filtragem e contexto.
`resource`	Modifica atributos de recurso.	Garante aplicação uniforme de `service.name`, versão, ambiente.	Uniformidade e correta identificação de origem dos dados.
`memory_limiter`	Controla consumo de memória.	Evita saturação e falha do Collector.	Estabilidade operacional.
`tail_sampling`	Amostragem com base no trace completo.	Seleciona traces críticos (ex.: erros, alta latência).	Reduz volume e custo mantendo visibilidade de incidentes.

- **Processador** `transform`

  O processador `transform` utiliza a *OpenTelemetry Transformation Language* (OTTL) para realizar modificações avançadas nos dados de telemetria. Na implementação atual, é utilizado para adicionar o atributo `service.name` aos sinais de telemetria, converter *timestamps* para um formato padronizado, normalizar níveis de severidade de logs (por exemplo, mapeando diferentes níveis para categorias consistentes como `INFO`, `WARN` e `ERROR`), e aplicar regras de amostragem dinâmica a *traces*. No entanto, devido ao seu elevado grau de flexibilidade e poder expressivo, a sua configuração deve ser cuidadosamente validada para evitar *transformações inconsistentes* (*Unsound Transformations*) ou *conflitos de identidade* (*Identity Conflicts*), que podem comprometer a integridade e a fiabilidade da telemetria.

- **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`

  O processador `attributes` permite adicionar, modificar ou remover atributos (metadados) associados a *spans*, logs ou métricas. Na prática, este processador é utilizado para enriquecer os dados de telemetria com informação contextual relevante - por exemplo, injetando um atributo estático que identifica o ambiente de execução (como *produção* ou *desenvolvimento*) ou acrescentando metadados ao nível do *host*. Este enriquecimento é fundamental para consultas mais eficientes, filtragem precisa e correlação robusta entre diferentes sinais, contribuindo para uma análise mais clara e completa do comportamento do sistema distribuído.

- **Processador** `resource`

  Este processador tem como objetivo a modificação de atributos de recurso, os quais descrevem a entidade responsável pela geração da telemetria, como o serviço da aplicação, a máquina *host* ou o *container*. É essencial para anexar de forma consistente metadados fundamentais, tais como *environment*, *service version* ou *region* às métricas e para enriquecer logs com informação contextual detalhada. O processador `resource` assegura que atributos de identificação comuns, nomeadamente `service.name`, são aplicados uniformemente a todos os sinais de telemetria (logs, *traces* e métricas). Esta consistência é crucial para permitir uma correlação eficiente e precisa entre sinais no Grafana.

- **Processador** `memory_limiter`

  Este processador é utilizado para evitar que o OpenTelemetry Collector consuma uma quantidade excessiva de memória. Ao definir limites explícitos de utilização, previne que o processo do Collector falhe devido ao esgotamento de recursos, garantindo assim a estabilidade e a fiabilidade contínuas

de todo o *pipeline* de monitorização. Esta salvaguarda é especialmente importante em ambientes de elevada carga, onde picos de telemetria podem ocorrer de forma imprevisível.

- **Processador** `tail_sampling`

  Este processador permite tomar decisões de amostragem com base no contexto completo de um *trace*, ou seja, apenas após todos os *spans* associados terem sido recebidos. Suporta múltiplos critérios de filtragem que podem ser combinados, incluindo amostragem baseada na latência observada, taxas probabilísticas, códigos de estado HTTP (por exemplo, apenas *traces* com erro) ou limites de taxa.

  Esta abordagem de amostragem baseada no contexto completo do trace é fundamental para controlar o volume de dados gerados por sistemas distribuídos de elevado tráfego, permitindo reter apenas os *traces* mais relevantes para diagnóstico e análise. O seu uso contribui para otimizar custos de armazenamento e melhorar o desempenho de consultas no *backend* de *tracing*, mantendo a capacidade de capturar e investigar *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 na 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 nas aplicações ou de configurações complexas específicas dos *backends*.

Esta abordagem não só centraliza significativamente o controlo de dados, como também minimiza a superfície de ataque para fugas acidentais de informação. Além disso, a capacidade de filtrar, agrupar e reduzir a cardinalidade dos dados antes da sua exportação para os *backends* traduz-se diretamente em poupanças substanciais nos custos de ingestão e armazenamento, especialmente em serviços de observabilidade geridos. Ao reduzir o volume e a complexidade dos dados na origem, o Collector melhora o desempenho das consultas nos *backends* e mitiga potenciais problemas de "crise de identidade" em métricas, resultando em *dashboards* mais fiáveis. Assim, o Collector assume-se como um componente crítico para assegurar tanto a eficiência económica como a eficiência operacional de toda a *stack* de monitorização.

4.9 Persistência e Armazenamento de Dados

A seleção dos sistemas de *backend* para armazenar *logs*, *traces* e métricas foi realizada de acordo com os requisitos definidos pela empresa, que recomendou soluções maduras e amplamente adotadas no ecossistema de monitorização. Estas ferramentas foram utilizadas exclusivamente como *backends*, tendo como principal objetivo centralizar o armazenamento e a análise dos dados exportados pelos microsserviços, sem adicionar dependências diretas ao código das aplicações.

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 pesquisa textual completa (*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, posteriormente, exportados para o Loki. A integração nativa com o Grafana permite que os *logs* sejam visualizados e correlacionados com métricas e *traces*, proporcionando uma análise centralizada e coerente do sistema.

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 comprovada de fornecer visibilidade *end-to-end* sobre o ciclo de vida de uma requisição. Com base no atributo `service.name`, é possível segmentar e filtrar os *traces*, identificando gargalos e potenciais pontos de falha durante a interação entre microsserviços.

Os dados são exportados pelo *OpenTelemetry Collector* via OTLP/*gRPC* para o *endpoint* do Jaeger, onde ficam persistidos para consulta e inspeção detalhada, sendo posteriormente visualizados através do Grafana.

4.9.3 Armazenamento de Métricas com o Prometheus

O Prometheus foi selecionado como sistema de armazenamento de métricas devido à sua robustez no processamento de séries temporais e à linguagem de consulta PromQL, que possibilita análises avançadas e precisas. As métricas das aplicações .NET são recolhidas pelo *OpenTelemetry Collector* e exportadas

para o Prometheus, evitando exposição direta dos serviços e centralizando a captura de telemetria.

Complementarmente, as métricas de infraestrutura provenientes do *Node Exporter* são recolhidas diretamente pelo Prometheus através de *scraping* HTTP. Esta abordagem combinada assegura visibilidade tanto ao nível da aplicação como da infraestrutura, proporcionando uma visão abrangente e consistente 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 da utilização do Grafana destacam-se:

- A centralização das métricas, *logs* e *traces* num único painel interativo;
- A capacidade avançada de correlação entre diferentes tipos de dados, facilitando a identificação de causas-raiz em anomalias;
- A configuração unificada de alertas abrangendo todas as fontes de dados;
- A interface intuitiva e personalizável, acessível a diferentes perfis técnicos;
- Linguagens de consulta especializadas (*PromQL*, *LogQL*) diretamente integradas na ferramenta.

O Grafana habilita uma abordagem de "*single pane of glass*", essencial para o acompanhamento consolidado do desempenho e da saúde do sistema. Além disso, permite seguir o percurso completo de uma requisição entre microsserviços e analisar a sua evolução temporal, proporcionando uma visão detalhada do comportamento distribuído da aplicação.

5.2 Organização dos Dashboards

5.2.1 Estrutura Lógica dos Dashboards

Para garantir uma análise sistemática e eficiente dos dados recolhidos, a plataforma de visualização foi estruturada em diferentes painéis temáticos no Grafana. Esta organização permite um fluxo analítico claro, desde a observação de métricas de alto nível até à inspeção detalhada de eventos específicos, facilitando o diagnóstico rápido de anomalias e a compreensão do comportamento global do sistema.

De modo a assegurar uma exploração coerente e eficiente dos sinais de telemetria, os dashboards foram agrupados em três categorias principais:

- **Métricas da Aplicação.** Focadas no comportamento das APIs .NET, onde são monitorizados indicadores como o número de requisições HTTP por segundo, latência média e percentis (p95 e p99), taxas de erro (4xx e 5xx), bem como o consumo de CPU e memória por serviço. Estes painéis permitem identificar padrões de carga e avaliar a eficiência operacional dos microsserviços.

- **Infraestrutura Kubernetes.** Dedicados à monitorização dos recursos do cluster, através dos dados expostos pelo Node Exporter. Incluem métricas como utilização de CPU e memória por nó e por *pod*, carga média do sistema (*load average*), e capacidade e utilização de armazenamento. Estes dashboards fornecem uma visão sobre a saúde da infraestrutura e permitem antecipar situações de saturação ou falhas ao nível dos recursos físicos.

- **Logs e Traces.** Concebidos para análise detalhada de eventos e interações entre serviços. Nesta secção é possível filtrar logs estruturados por nível de severidade, serviço ou mensagem, inspecionar *spans* individuais e observar o encadeamento de operações entre microsserviços ao longo do tempo. Esta correlação entre logs e rastreamento distribuído suporta a identificação de pontos de falha, atrasos inesperados e comportamentos anómalos na comunicação entre componentes.

Esta estrutura modular facilita a navegação entre diferentes perspetivas operacionais e acelera o processo de diagnóstico e resolução de problemas. Para além disso, os dashboards incluem gráficos de séries temporais, indicadores numéricos e filtros dinâmicos, permitindo uma interpretação prática e contextual dos dados. Assim, a solução não apenas centraliza a telemetria, mas também proporciona aos engenheiros uma visão unificada e acionável sobre a saúde, desempenho e fiabilidade do sistema distribuído.

5.2.2 Painéis de Dashboards e Metodologia de Análise

A visualização de dados representa uma camada fundamental na estratégia de monitorização para sistemas distribuídos. Após a instrumentação e recolha dos sinais de telemetria, métricas, *logs* e *traces*, torna-se necessário disponibilizar uma interface capaz de sintetizar esta informação de forma clara, permitindo identificar rapidamente anomalias, diagnosticar problemas e interpretar o comportamento operacional das aplicações.

Nesta secção, apresentam-se os dashboards desenvolvidos no Grafana para demonstrar a eficácia da solução proposta. O objetivo é ilustrar o percurso analítico completo, desde a deteção de um evento anómalo até à identificação da sua causa, evidenciando o valor prático da integração entre métricas, *logs* e rastreamento distribuído.

Para efeitos de consistência e clareza, os exemplos apresentados focam-se exclusivamente no microsserviço *TicketsManagement*.

Embora fosse possível incluir visualizações de outros serviços da arquitetura, tal abordagem tenderia a revelar informações redundantes, uma vez que os padrões e métricas observados seriam aplicáveis de forma semelhante. Assim, com esta escolha conseguimos proporcionar uma análise mais clara da *pipeline* de monitorização em produção, evitando a dispersão do foco e privilegiando a profundidade sobre a generalidade.

1) Visão geral de logs (ponto de partida). A Figura 6 apresenta uma visão agregada de *logs* estruturados emitidos pelo serviço *TicketsManagement*. Este painel permite observar o volume total de eventos, distribuição por severidade e principais *endpoints* envolvidos. Este é o ponto de partida do processo analítico: permite perceber rapidamente se existem picos de erros, mensagens recorrentes ou padrões anómalos.

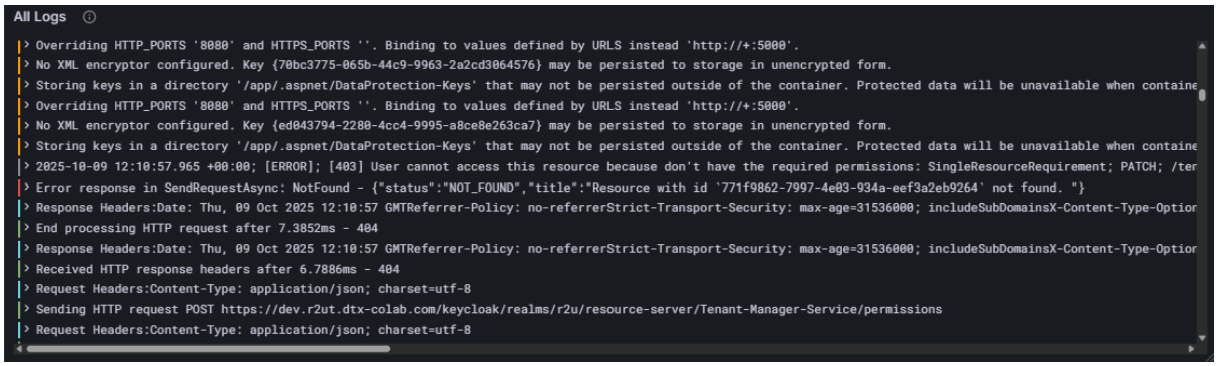


Figura 6: Visão geral dos *logs* estruturados do serviço *TicketsManagement*.

Transição: Identifica-se um pico de erros ou eventos suspeitos e aplica-se filtragem por `service.name = TicketsManagement` e severidade (`ERROR`/`WARN`), avançando para o foco em erros.

2) Foco exclusivo nos erros. A Figura 7 mostra um painel dedicado exclusivamente a mensagens de erro, permitindo identificar *endpoints* afetados, códigos HTTP e mensagens predominantes. Este painel acelera a identificação de falhas e reduz o *MTTR* ao tornar evidentes os padrões de erro mais frequentes.

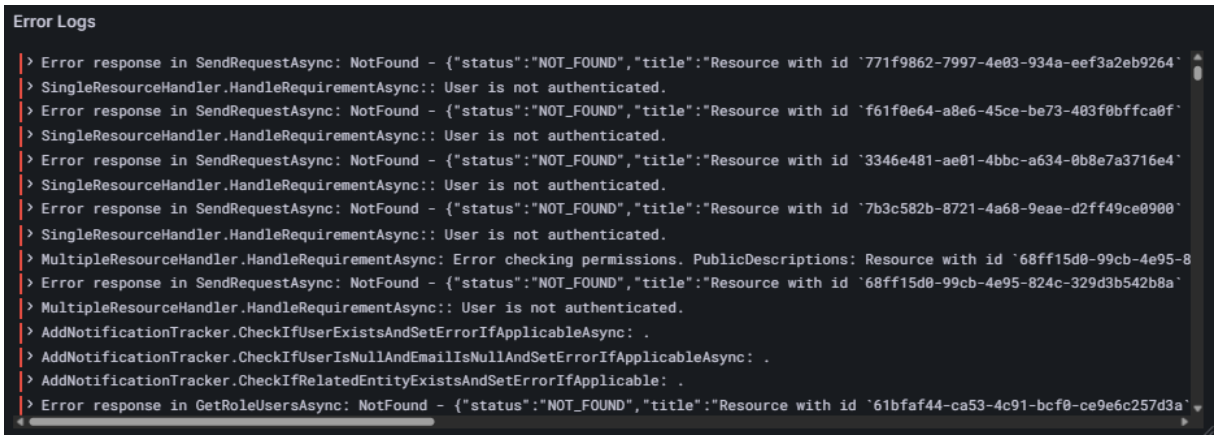


Figura 7: Painel focado em *logs* de erro do serviço.

Transição: Seleciona-se um evento concreto para análise detalhada do registo.

3) Expansão do registo e metadados. A Figura 8 apresenta o detalhe de um evento de *log*, incluindo `trace_id`, `span_id`, rota, código de estado e contexto adicional.

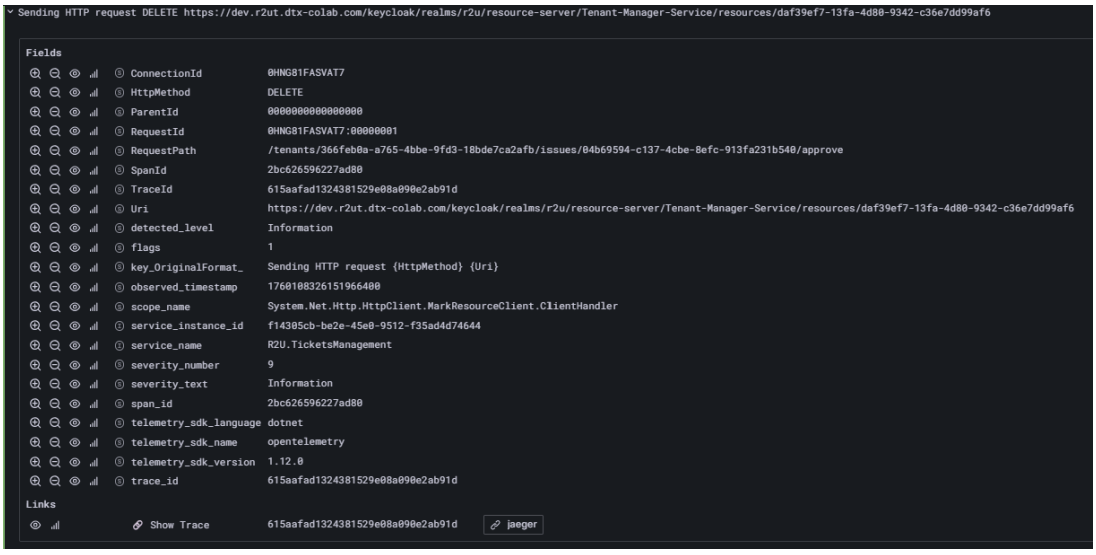


Figura 8: Detalhe de *log* com metadados e referência direta ao trace.

Transição: A partir do `trace_id`, o analista segue para o rastreamento associado.

4) Métricas da aplicação. A Figura 9 apresenta indicadores operacionais da API: pedidos por segundo (RPS), distribuição por códigos HTTP, latência média e percentis (p95/p99). Este painel contextualiza o erro no panorama global do serviço: variações de latência ou picos de erros tendem a refletir-se aqui.

Figura 9: Métricas da aplicação: RPS, códigos HTTP, latência média e percentis.

Transição. Se houver degradação, o analista volta aos eventos e segue para o *trace* para localizar gargalos. Em paralelo, valida se há limitação de recursos na infraestrutura.

5) Recursos da infraestrutura Kubernetes. A Figura 10 consolida métricas do cluster: utilização de CPU e memória por nó e por *pod*, *load average* e espaço de disco. Serve para confirmar ou excluir causas relacionadas com recursos (e.g., *throttling*, saturação, *out-of-memory*).

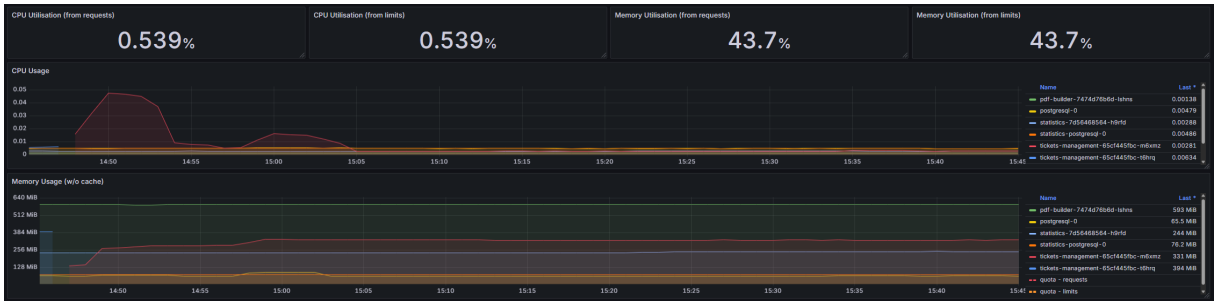


Figura 10: Métricas de infraestrutura Kubernetes.

Transição. Com os recursos estáveis, a investigação regressa ao encadeamento distribuído das chamadas para localizar a origem funcional do problema.

6) Rastreio distribuído. A Figura 11 ilustra um rastreamento completo associado ao incidente analisado. Observam-se as chamadas entre microsserviços, o tempo gasto em cada *span* e a identificação de *bottlenecks* (e.g., chamadas a *databases* ou serviços externos).

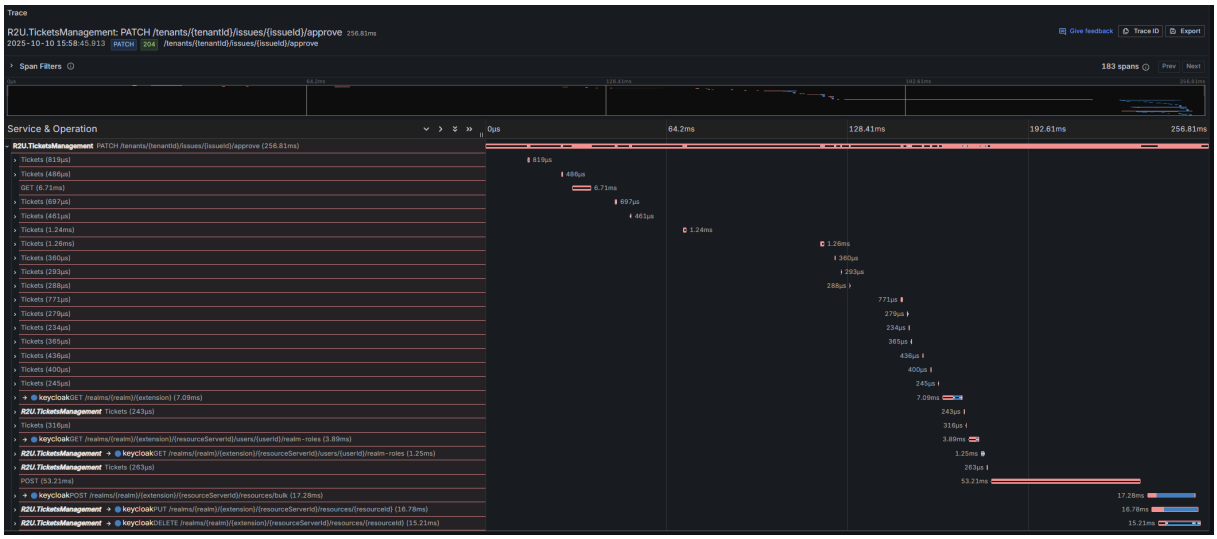


Figura 11: Rastreamento distribuído associado ao evento analisado.

5.2.3 Correlação entre Logs e Traces

Um dos principais benefícios da solução implementada reside na correlação direta entre *logs* e *traces*. A Figura 12 apresenta um painel que liga cada registo de *log* ao *trace* correspondente, permitindo navegar rapidamente entre eventos e fluxos de execução.

Figura 12: Ligação direta do *log* ao *trace* correspondente.

Esta funcionalidade acelera significativamente a deteção da causa raiz de incidentes, uma vez que permite cruzar mensagens de erro com a linha temporal e contexto completo da execução distribuída.

Nota sobre a migração da plataforma. Apesar da eficácia demonstrada, importa mencionar que a plataforma R2UT encontra-se ainda em fase de migração gradual para Kubernetes. Assim, apenas os serviços já migrados emitem telemetria compatível com o modelo adotado. À medida que a migração progride, a monitorização será estendida a toda a plataforma, permitindo rastreamento ponta-a-ponta.

Síntese do fluxo analítico. O processo recomendado segue a sequência:

1. Iniciar em *logs* gerais (Figura 6) e identificar picos;
2. Focar erros (Figura 7);
3. Expandir registo e seguir o `trace_id` (Figura 8);
4. Validar métricas globais (Figura 9);
5. Confirmar estado da infraestrutura (Figura 10);
6. Inspecionar *trace* completo (Figura 11);
7. Consolidar correlação (Figura 12).

Este método reduz o tempo de análise e materializa o Grafana enquanto *single pane of glass* para operação e diagnóstico.

5.3 Alertas e Notificações Operacionais

A monitorização eficaz não se limita à visualização passiva de métricas, exige mecanismos de alerta que permitam atuação rápida perante degradações de desempenho, falhas de serviço ou comportamentos anómalos. Para esse fim, foi configurado o *Prometheus AlertManager*, responsável por receber regras de alerta definidas no Prometheus e encaminhar notificações para canais operacionais pré-definidos.

Os alertas constituem um elemento crítico em ambientes *cloud-native*, onde a natureza distribuída e dinâmica dos microsserviços pode levar a falhas rápidas e efeitos em cascata. Assim, foram definidas regras de monitorização orientadas a três objetivos principais: (i) assegurar disponibilidade, (ii) detetar degradação de desempenho e (iii) prevenir saturação de recursos.

A título ilustrativo, foram implementados alertas para as seguintes condições:

- Taxa de erros HTTP 5xx superior a 5% durante 5 minutos;
- Latência p95 das requisições superior a 1 s;
- Utilização de CPU por *pod* superior a 90% durante período prolongado.

Embora o Grafana também suporte a configuração de alertas diretamente sobre *dashboards*, a opção por utilizar o *AlertManager* reflete as melhores práticas para ambientes de produção, privilegiando uma abordagem declarativa, versionável e escalável. As regras de alerta podem ser geridas em ficheiros YAML, permitindo a sua integração em pipelines CI/CD e garantindo consistência operacional.

No contexto prático deste trabalho, foi integrada uma notificação para o *Slack*, garantindo que as equipas técnicas recebem alertas em tempo real num canal dedicado. Esta abordagem assegura visibilidade imediata sobre incidentes operacionais e permite resposta rápida, contribuindo para a redução do *Mean Time to Detect* (MTTD) e do *Mean Time to Recover* (MTTR).

A Figura 13 exemplifica dois alertas gerados pelo sistema: um associado a uma taxa de erros elevada e outro à utilização excessiva de CPU num *pod* Kubernetes, ilustrando a capacidade do sistema em detetar e notificar eventos críticos.

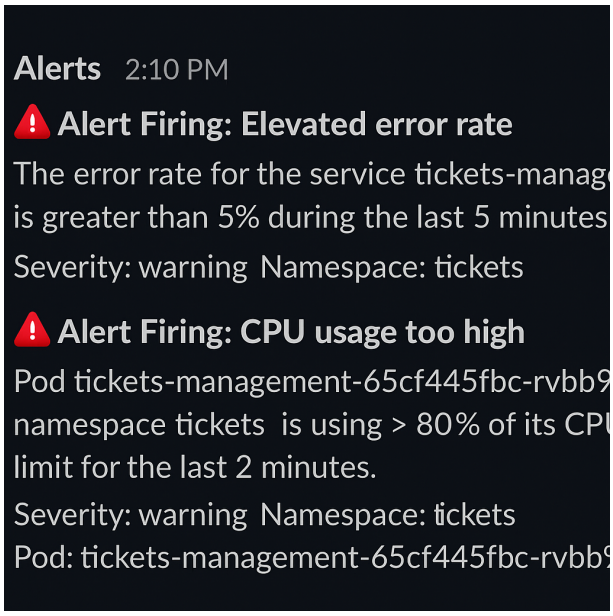


Figura 13: Notificações enviadas para o Slack contendo alertas gerados pelo *AlertManager*, incluindo taxa de erros elevada e utilização excessiva de CPU.

Capítulo 6

Conclusões e Trabalho Futuro

6.1 Conclusões

Nesta dissertação começámos por realizar uma análise crítica e aprofundada da literatura disponível sobre arquiteturas de microsserviços e sistemas de monitorização em ambientes distribuídos. Esse trabalho visou construir uma base teórica sólida que suportasse o desenvolvimento de uma plataforma de monitorização eficaz e alinhada com os objetivos do projeto R2UT, destacando-se o uso de ferramentas *open-source* amplamente reconhecidas, como Prometheus, Grafana, ELK Stack e Jaeger.

A revisão bibliográfica incidiu particularmente nos desafios associados à monitorização de sistemas distribuídos dinâmicos, procurando identificar padrões de resiliência e boas práticas que assegurassem a fiabilidade, a escalabilidade e a eficiência operacional dos microsserviços. Foram igualmente aprofundados conceitos fundamentais como *distributed tracing*, *alerting* e observabilidade, analisando como estas técnicas podem ser combinadas para otimizar a gestão e operação de infraestruturas baseadas em microsesrviços. A partir desta fundamentação teórica, foi possível estabelecer os requisitos funcionais e tecnológicos que orientaram a implementação prática.

No contexto do projeto R2UT, foi selecionado um conjunto de serviços representativos do ecossistema da plataforma, com o objetivo de demonstrar o funcionamento da solução num ambiente real e alinhado com o fluxo operacional do sistema. A escolha destes serviços permitiu validar a abordagem tomada, avaliar a capacidade de escalabilidade da solução e aferir a eficácia na deteção e investigação de falhas, garantindo simultaneamente a continuidade operacional da plataforma durante a fase de migração para Kubernetes.

A implementação de uma solução de monitorização baseada em OpenTelemetry, integrando ferramentas como Prometheus, Loki, Jaeger e Grafana, proporcionou uma visão completa e em tempo real do comportamento da aplicação distribuída. A arquitetura modular e escalável oferece flexibilidade, portabilidade e facilidade de manutenção. A correlação sinérgica entre *logs*, métricas e *traces*, potenciada

pela padronização do OpenTelemetry, transformou a forma como a equipa de desenvolvimento e operações lida com problemas de desempenho, bem como permitiu a deteção e correção proativa de falhas e *bottlenecks* no fluxo de execução dos microsserviços. A capacidade de navegar de um alerta de métrica diretamente para os *traces* e *logs* correspondentes em poucos segundos demonstra claramente o valor operacional desta abordagem.

Importa ainda referir que, durante o desenvolvimento, foi avaliada a adoção de *zero-code instrumentation* para aplicações .NET, recorrendo a agentes automáticos de OpenTelemetry. Embora esta abordagem apresente vantagens em termos de rapidez de integração e eliminação de alterações diretas no código, verificaram-se limitações relevantes ao nível do controlo de granularidade, compatibilidade com bibliotecas utilizadas no ecossistema da aplicação e flexibilidade na definição de atributos específicos de negócio. Estas restrições levaram à opção por uma solução híbrida, baseada num pacote comum de instrumentação (`DTX.Base.Common`) e num método de extensão centralizado, garantindo simultaneamente padronização, flexibilidade e governância sobre os sinais de telemetria emitidos. Assim, a solução final alcançou um equilíbrio entre automatização e controlo fino da instrumentação, assegurando consistência técnica e alinhamento com as necessidades operacionais da plataforma.

Do ponto de vista prático, a solução revelou-se eficaz na redução do tempo médio de diagnóstico (*MTTR*) e na identificação rápida de anomalias, quer ao nível da aplicação, quer ao nível da infraestrutura. Entre os principais benefícios observados destacam-se a padronização da telemetria entre serviços, a capacidade de correlação entre diferentes sinais e a redução da dependência de ferramentas proprietárias. Esta abordagem contribuiu igualmente para uma maior transparência no funcionamento interno dos serviços, permitindo apoiar decisões informadas em fases de desenvolvimento, testes e produção.

Contudo, a implementação também evidenciou desafios relevantes. A configuração inicial do *pipeline* de telemetria requer conhecimento técnico especializado, nomeadamente na definição de *pipelines* e recursos do OpenTelemetry Collector, de modo a evitar perdas de dados, degradação de desempenho ou consumo excessivo de memória. Adicionalmente, a instrumentação introduz algum *overhead* computacional, pelo que se torna essencial aplicar boas práticas de controlo de cardinalidade, *sampling* e políticas de retenção. A coexistência de múltiplos componentes *open-source* com diferentes ciclos de maturidade, destacando-se a componente de *logs*, que ainda depende de integrações externas como Loki ou ELK, implica um esforço contínuo de manutenção e atualização. Acresce que nem todos os serviços da plataforma R2UT se encontram ainda migrados para Kubernetes, limitando temporariamente a visibilidade transversal e a correlação total de telemetria, embora tal limitação esteja a ser mitigada pelo plano de migração progressiva em curso. Por fim, a eficácia da solução depende da literacia técnica da equipa e

da adoção de processos operacionais maduros para análise e atuação sobre os indicadores de monitorização. Ainda assim, os benefícios práticos obtidos superam largamente estas limitações, demonstrando a robustez e a aplicabilidade desta abordagem em ambientes *cloud-native*.

Em suma, neste trabalho conseguimos demonstrar que é possível construir um sistema de observabilidade robusto e eficiente em um ambiente *cloud-native* utilizando ferramentas open-source. Essa abordagem não apenas melhora a confiabilidade e o desempenho da aplicação, mas também capacita as equipas a tomar decisões e a inovar com mais agilidade.

6.2 Trabalho Futuro

Embora a solução desenvolvida tenha demonstrado a sua eficácia na monitorização e correlação de métricas, *logs* e *traces* em ambiente *cloud-native*, existem diversas oportunidades de evolução e consolidação que poderão potenciar significativamente o seu impacto operacional e científico.

Uma primeira linha de desenvolvimento consiste na **integração completa da telemetria em toda a plataforma R2UT**, acompanhando o plano de migração gradual para Kubernetes. A plena instrumentação de todos os serviços permitirá alcançar visibilidade ponta-a-ponta e uma correlação completa entre os fluxos de execução, tornando possível uma análise operacional verdadeiramente holística. Este esforço deverá incluir não apenas a instrumentação de novas APIs, mas também a definição de *standards* internos formais para telemetria, garantindo consistência semântica e metodológica em toda a plataforma.

Outra vertente relevante consiste na **exploração da instrumentação automática (zero-code instrumentation)**, aproveitando os avanços contínuos na comunidade OpenTelemetry. Embora tenham sido identificadas limitações nesta implementação inicial, espera-se que a evolução do suporte nativo para .NET e a maturação dos agentes de instrumentação possam permitir a adoção parcial ou total desta abordagem no futuro, reduzindo o esforço de manutenção e eliminando a necessidade de inclusão de bibliotecas adicionais nos serviços.

A implementação de **dashboards avançados e automatização do alerting** constitui também uma área de expansão crítica. A definição de *Service Level Indicators* (SLI) e *Service Level Objectives* (SLO) formais permitirá monitorizar níveis de serviço associados à disponibilidade, latência e fiabilidade, alinhando a observabilidade com a governança de serviço e as metas operacionais da plataforma. Complementarmente, a configuração de estratégias de alerta baseadas em anomalias e tendências (em vez de simples *thresholds*) contribuirá para uma resposta mais proativa a falhas.

Outra frente de investigação com elevado potencial consiste na **integração de técnicas de Inte-**

ligência Artificial e Machine Learning para deteção preditiva de anomalias e suporte à tomada de decisão. A análise automática de padrões de degradação e a implementação de mecanismos de correlação assistida poderão reduzir tempos de diagnóstico, potenciar respostas automáticas e aproximar a plataforma de um modelo de operação AIOps.

A adoção de mecanismos de **tracing unificado em ambientes híbridos e de múltiplos clusters** representa também um caminho promissor, possibilitando a observação integrada de componentes que residam em infraestruturas distintas (por exemplo, máquinas físicas, serviços externos e múltiplos clusters Kubernetes). Esta abordagem pode ser suportada por *OpenTelemetry Gateways*, arquiteturas *mesh* de observabilidade e mecanismos de encriptação e roteamento seguro de telemetria.

Por fim, uma linha de evolução natural reside na **automação do ciclo de vida da observabilidade com práticas GitOps**, garantindo que configurações do OpenTelemetry Collector, dashboards Grafana e regras de alerta são versionadas, auditáveis e sincronizadas com o estado desejado do cluster. Tal abordagem reforçará a consistência, segurança e fiabilidade do sistema ao longo do tempo.

Em suma, o trabalho futuro deverá focar-se na expansão da cobertura, no aumento do grau de automação, na integração de capacidades inteligentes e na consolidação de práticas de observabilidade como parte integrante do ciclo de vida de desenvolvimento e operação da plataforma. Ao perseguir estas linhas de evolução, será possível não só ampliar o valor prático desta solução no contexto do R2UT, como também contribuir para o avanço do estado da arte em observabilidade aplicada a sistemas distribuídos e ambientes industriais.

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.

Marcin Bajer. Building an iot data hub with elasticsearch, logstash and kibana. In *Proceedings - 2017 5th International Conference on Future Internet of Things and Cloud Workshops, W-FiCloud 2017*, volume 2017-January, 2017. doi: 10.1109/FiCloudW.2017.101.

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.

Ali Basiri et al. Chaos engineering. *IEEE Software*, 36(1):35–41, 2019. doi: 10.1109/MS.2018.2874323.

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.

Xiaoyu Fu, Rui Ren, Jianfeng Zhan, Wei Zhou, Zhen Jia, and Gang Lu. Logmaster: Mining event correlations in logs of large-scale cluster systems. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 2012. doi: 10.1109/SRDS.2012.40.

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.

R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 1960. doi: 10.1115/1.3662552.

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.

Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions, ICITST 2016*, 2017. doi: 10.1109/ICITST.2016.7856721.

Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. So, you want to trace your distributed system? key design insights from years of practical experience. *Carnegie Mellon University Parallel Data Lab Technical Report*, 2014.

Benjamin H Sigelman, Luiz Andr, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper , a large-scale distributed systems tracing infrastructure. *Google Research*, 2010. ISSN <null>.

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.

Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing edge-cases in distributed systems. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023*, 2023.