|  |  |
| --- | --- |
| CURSO DE CIÊNCIA DA COMPUTAÇÃO – TCC | |
| (  ) PRÉ-PROJETO     ( X ) PROJETO | ANO/SEMESTRE: 2022/2 |

orquestração de soluções de monitoramento de microsserviços

Nestor Kammer

Marcel Hugo – Orientador

# Introdução

O assunto sobre microsserviços vem ganhando rapidamente um grande espaço em mídias sociais, blogs, discussões, conferências e meio corporativo (ROSSI, 2021). Embora os conceitos fundamentais por trás dos microsserviços não sejam novos, a aplicação da arquitetura de microsserviços é atual, e sua adoção tem sido motivada em parte pelos desafios de escalabilidade, falta de eficiência, velocidade lenta de desenvolvimento e dificuldades em adotar novas tecnologias que surgem quando sistemas complexos de software estão contidos e são implantados em uma grande aplicação monolítica (FOWLER, 2017).

Rossi (2021), basicamente define sistemas monolíticos como aplicações onde todos os seus componentes e funcionalidades estão combinados dentro de uma única aplicação e arquitetura. Opus Software (2021) classifica que os principais desafios da arquitetura monolítica são os seguintes:

1. aumento da complexidade e tamanho ao longo do tempo;
2. alta dependência de componentes de código;
3. escalabilidade limitada do sistema;
4. falta de flexibilidade;
5. dificuldade para colocar alterações em produção.

Para Fowler (2017), adotar a arquitetura de microsserviços, seja a partir do zero ou dividindo uma aplicação monolítica existente em microsserviços desenvolvidos e implantados independentemente, resolve esses problemas. Com a arquitetura de microsserviços, uma aplicação pode ser facilmente escalada tanto horizontalmente quanto verticalmente, a produtividade e a velocidade do desenvolvedor aumentam dramaticamente e tecnologias antigas podem facilmente ser trocadas por mais recentes (FOWLER, 2017).

Atualmente, existem vários modelos de como um ecossistema de microsserviços pode ser organizado. Um deles, é dividido em quatro camadas (OPUS SOFTWARE, 2021), sendo elas: hardware, comunicação, plataforma de aplicações e microsserviços.

Fowler (2017), destaca que existem alguns itens que são fundamentais para o sucesso das operações dos microsserviços. São eles: estabilidade, confiabilidade, escalabilidade, alta tolerância a falhas, desempenho, monitoramento e documentação.

Em se tratando de monitoramento, Fowler (2017) considera que um bom monitoramento tem quatro componentes: um adequado *logging* (registro) de todas as informações importantes e relevantes; interfaces gráficas úteis (*dashboards* de controle) que sejam facilmente compreendidas por todo desenvolvedor na empresa e que reflitam com precisão a saúde dos serviços; um sistema de alerta eficaz e acionável sobre as principais métricas; e implementar e manter uma rotação das equipes de prontidão.

Um microsserviço pronto para produção é adequadamente monitorado. Um monitoramento adequado é uma das partes mais importantes de construir um microsserviço pronto para produção e garantir maior disponibilidade do microsserviço (FOWLER, 2017).

Com base nisso muitas soluções começaram a surgir para suprir essa demanda não só arquitetural como de identificação de pontos críticos em sistemas e possíveis gargalos. Atualmente, entre as principais ferramentas utilizadas em *log* e monitoramento há: Fluentd, Prometheus, Grafana, Dynatrace, Stack ELK, Stack Prometheus e Kafka (BRANDÃO, 2020).

Diante deste cenário, o presente trabalho propõe orquestrar estas soluções para o monitoramento e disponibilização de informações sobre uma arquitetura de microsserviços. O monitoramento poderia ocorrer sobre uma solução já existente, mas optou-se por implementar uma arquitetura de protótipos de microsserviços baseada em contêineres para que os vários aspectos da operação e monitoramento de microsserviços estejam contemplados. Com a execução destes protótipos de microsserviços pretende-se aplicar os conceitos de monitoramento para coletar as informações dos *logs* distribuídos, centralizá-los em uma base, disponibilizá-los em *dashboards,* assim como, realizar o disparo de eventuais alertas para o público-alvo. Todas estas ações ocorrerão pela integração de diversas ferramentas disponíveis (orquestração), seja diretamente seja pela implementação de mecanismos de troca de dados.

## OBJETIVOS

Orquestrar diferentes ferramentas para monitorar protótipos de microsserviços disponibilizando os dados resultantes de monitoramento (métricas e *logs*) em um *dashboard*.

Neste contexto, este trabalho tem como objetivos específicos:

1. realizar o desenvolvimento de protótipos baseado no conceito de arquitetura de microsserviços;
2. implantar os protótipos desenvolvidos em um ecossistema baseado em contêineres;
3. gerar informações de métricas e *logs* dos microsserviços fazendo uso do conceito de *Domain-observed observability* na instrumentação dos mesmos*;*
4. apresentar os resultados do monitoramento em formato de *dashboard*, efetuando o disparo de alertas nas situações programadas.

# trabalhos correlatos

Nesta seção são apresentados alguns trabalhos correlatos com características semelhantes aos principais objetivos do estudo proposto. No primeiro trabalho, Rodrigues (2018) apresenta o embasamento e uma proposta de conversão de uma arquitetura de sistema monolítico em uma arquitetura de microsserviços. No segundo trabalho, Silva (2020) aborda um sistema de monitoramento de servidores, tendo neste caso, o foco mais específico no monitoramento da infraestrutura, que aborda os mecanismos da coleta destes dados e disponibilização em uma aplicação web para consulta por parte de seu público. Já no terceiro trabalho, Assad (2019) demonstra a investigação de soluções para monitoramento de aplicações orientadas a microsserviços, bem como, propõe uma solução para monitoramento baseado nas métricas principais de servidor e infraestrutura.

## *MS-TRICK*: ARQUITETURA DE *MICROSERVICES* APLICADA

Rodrigues (2018) apresenta uma arquitetura de microsserviços. O objetivo do trabalho é reconstruir partes de uma aplicação existente de maneira que ela seja escalável, demonstrando os resultados obtidos com esta alteração. De maneira complementar, foi desenvolvido um aplicativo móvel para visualização da arquitetura. Os microsserviços desenvolvidos para contemplar a arquitetura, foram implementados na linguagem de programação Java, mais especificamente utilizando o SpringBoot. Para armazenamento de dados, foi utilizado o banco de dados relacional MySQL. O aplicativo móvel (interface visual disponibilizada aos usuários) foi desenvolvido utilizando o Ionic, *framework* responsável por gerar aplicativos para Android e iOS.

Rodrigues (2018) afirma que para o desenvolvimento do projeto foi necessário abordar determinados recursos tecnológicos, como servidores de aplicação e integração contínua. Segundo Spring (2017), Spring Boot é uma ferramenta pronta para produção, responsável por disponibilizar um ambiente de fácil configuração para desenvolvedores de software.

Outro recurso tecnológico utilizado diz respeito ao trabalho de integração contínua que está ligado ao conceito de DevOps. A integração e a entrega contínuas são práticas que automatizam o processo de lançamento de software, da fase de criação à fase de implantação (AWS, 2022).

Como resultados, Rodrigues (2018) destaca que o seu trabalho conseguiu alcançar as seguintes características: escalabilidade, integração contínua, disponibilidade, métricas exclusivas, modularidade, linguagem de programação Java e melhoria no processo de manutenção. Onde:

1. a escalabilidade foi alcançada com a possibilidade de execução do número de instâncias necessárias para suprir a necessidade dos usuários;
2. a integração contínua foi alcançada com a implementação de conceitos de DevOps; pela implementação de um script para Gitlab é possível realizar a compilação, o deploy e a avaliação do código da aplicação;
3. a disponibilidade está diretamente relacionada com a escalabilidade; no MS-Trick é possível aumentar o número de instâncias executantes, tornando a aplicação cada vez mais disponível;
4. as métricas exclusivas foram adquiridas pela implementação da arquitetura, estas foram disponibilizadas no aplicativo móvel implementado para melhor visualização da arquitetura;
5. a modularidade, foi alcançada pela implementação de módulos independentes das demais partes do sistema;
6. foi utilizada Java como linguagem de programação para adaptação do código existente e criação do novo código;
7. a melhoria no processo de manutenção, é trazida pela possibilidade de diferentes desenvolvedores trabalharem nos módulos individuais da aplicação, bem pela velocidade trazida pela integração contínua.

## WOLF MONITOR: SISTEMA DE MONITORAMENTO DE SERVIDORES

Em seu trabalho Silva (2020) apresenta um projeto para automatizar migrações e monitoramento de servidores, cujo objetivo é melhorar a forma como as empresas lidam com serviços e com arquivos de configuração de programas computacionais em seus servidores. Desta forma, não foi necessário efetuar uma conexão direta no servidor sendo feito todo o controle em um sistema.

O sistema é dividido em três fases. A primeira etapa, um serviço (agente) em execução no servidor da empresa. A segunda etapa Application Programming Interface ou Interface de Programação de Aplicações (API) da Web (sistema que pode receber informações via internet) que receberá as informações enviadas por esse servidor. Por último um sistema/aplicativo para computadores que será responsável por exibir as informações transmitidas em telas amigáveis para o usuário.

Seguem descritos abaixo os principais recursos utilizados:

1. webAPI implementada: na fase de implementação Silva (2020) utiliza os padrões de microsserviços para criação da API e *end-points.* Todo o sistema está nas conformidades de API *Gateway*, em que mesma é desenvolvida pensando no *front-end,* para que assim não sejam realizadas consultas desnecessárias e diminua a quantidade de requisições no servidor, permitindo até mesmo uma internet com menor velocidade acessar os dados;
2. banco de dados: como o sistema apresentado no trabalho utiliza de microsserviços, faz-se válido uma diferenciação no banco de dados, que foi separado em quatro partes para que dessa forma o fluxo de dados seja rápido e não ocorra lentidão no sistema nem congestionamento no fluxo de dados (SILVA, 2020);
3. desenvolvimento desktop*:* conforme Silva (2020), o desenvolvimento para desktop foi feito para a visualização das informações enviadas pelo serviço do sistema Windows/Linux para a Web API de dados. O sistema desktop se conecta com o servidor de informações de dados passando por autenticação no IdentityServer*.* Após ser autorizado o mesmo pode visualizar todos os dados que seu usuário tenha permissão para acessar;
4. agente – serviço do Linux/Windows: após esse cadastro o usuário solicitará os arquivos necessários para instalação do *agent* (agente) e deverá configurá-lo para utilização. Estes arquivos serão as DLLs (arquivos de sistema) e executável do programa de monitoramento, ao qual ficará responsável por verificar os dados de todos os itens cadastrados, sempre buscando as informações no servidor de dados da Web API(SILVA, 2020).

Silva (2020), conclui que apesar do presente sistema ser voltado a monitoramento apenas de arquivos e serviços, o mesmo abrange outras possibilidades como: banco de dados, serviços de IIS (Provedor de dados Windows), entre toda e qualquer informação que seja fundamental para o funcionamento correto dos servidores. A utilização deste sistema irá promover uma melhoria significativa na produção das empresas, para que sejam feitos procedimentos de forma rápida e com qualidade, garantindo a funcionabilidade correta dos servidores, mesmo após um serviço de migração ao qual tenha sido necessário o desligamento de todos os serviços.

## MANGUE BEAT! INVESTIGANDO SOLUÇÕES PARA MONITORAMENTO DE APLICAÇÕES ORIENTADAS A MICROSSERVIÇOS

O objetivo principal deste trabalho é investigar e expor algumas soluções possíveis e o melhor caso para a sua utilização no universo do Kubernetes, que é um orquestrador de contêineres popular (DOBIES; WOOD, 2020). Conforme Dobies e Wood (2020), ele reúne vários computadores em um grande recurso computacional e estabelece uma forma de administrar esse recurso por meio de sua API. Em seu trabalho Assad (2019) adota soluções como: a Stack EFK (Elasticsearch, Fluentd, Kibana), Beats do Elasticsearch e apresenta uma solução desenvolvida por ele próprio, o Mangue Beat.

Assad (2019) ressalta que existem dois pontos em comum entre todas as soluções que é o banco de dados, o Elasticsearch, e o Kibana um *framework open-source* empregado para visualização de dados do Elasticsearch. O Elasticsearch é um banco de dados não relacional, orientado a documentos, comumente utilizado para fazer consultas complexas em cima de grandes volumes de dados (COIMBRA, 2018).

Com a possibilidade de distribuir as aplicações em diferentes Máquinas Virtuais (VM) rodando em diferentes ambientes, a forma sugerida pela documentação original do Kubernetes é levantar um agente/operador em cada *node* onde existem aplicações rodando no *cluster* (KUBERNETES, 2022). Basicamente cada agente fica responsável por coletar os *logs* de cada contêiner que estejam rodando na VM e direcionar para um *backend,* que fica responsável por esta centralização dos *logs*.

Assad (2019), explica que o Fluentd é uma solução *open source* desenvolvida para centralização de *logs*, que busca armazenar os *logs* em forma de JavaScript Object Notation (JSON), como forma de facilitar as consultas e agrupamentos que pode ser feita no banco de dados. Assad (2019) ainda ressalta que sua configuração não é feita de forma trivial, tendo uma variedade grande de parâmetros que podem ser configurados.

Dentre as soluções apresentadas, a mais desafiadora é configurar em ambientes produtivos, devido a diversidade de integrações de *plugins* e a diferença na forma como são configurados. Por exemplo: configurar onde serão salvas as informações, a ferramenta de alerta e outros vários *plugins*. Já sobre o Beats (parte do grupo do Elasticsearch), que é responsável pela tecnologia de monitoramento de aplicações, Assad (2019) explica que é facilmente integrável com o Logstash e com o banco de dados Elasticsearch.

Dentro do agrupamento do Beats há o Metricbeat e o Filebeat. O Metricbeat é a solução voltada para coleta de métricas das aplicações, e o Filebeat é a solução utilizada para monitoramento dos *logs* dos contêineres. Ambas são *open source* e assim como o Fluentd armazenam os *logs* em forma de JSON (ASSAD, 2019). Um diferencial do Beats é que o Kibana já vem com um *dashboard* pronto para visualização das métricas coletadas. Basta apenas importá-lo na página inicial do Kibana. Na contramão das facilidades, Assad (2019) afirma que as ferramentas do Beats consomem cerca de 80 MB de memória a mais que o Fluentd por operador, o que em um servidor com 1000 máquinas pode chegar a 80GB de utilização deste recurso.

Segundo Assad (2019), ao contrário das demais soluções o MangueBeat integra-se com o servidor de métricas mantido pelo Kubernetes, não precisando de um operador rodando em cada *node* do *cluster*. O fato de não ser um operador reduz drasticamente seu consumo de recursos computacionais por *node* no *cluster*, visto que é necessária apenas uma instância do MangueBeat operacional para que o monitoramento seja feito (ASSAD, 2019).

Além disso, Assad 2019 ressalta que o serviço de monitoramento do MangueBeat possibilita o monitoramento de mais de um *cluster* por vez, podendo haver a centralização do agente de monitoramento, e salvando as informações de forma descentralizada. Como o MangueBeat integra-se apenas com o servidor de métricas, seu uso fica restrito apenas a coleta de informações de consumo de CPU, memória e rede das aplicações (ASSAD, 2019).

Um ponto interessante para a solução do MangueBeat é a facilidade de integração com outros microsserviços, para que sejam feitos por exemplo, alertas em caso de anomalias. Para isto, foi desenvolvida uma integração com um TelegramBot, onde são enviados alertas informando anomalias nos microsserviços e são possíveis a execução de algumas operações com base nas métricas coletadas.

Na análise das soluções apresentadas Assad (2019) evidencia que existem diversas formas para implementar e manter o monitoramento de aplicações no cenário sistemas distribuídos em Kubernetes. O autor destaca que, do ponto de vista de alocação dos sistemas de monitoramento em VMs é possível identificar duas possibilidades: a instalação de operadores em cada *worker* ou a integração do sistema de monitoramento ao servidor de métricas do Kubernetes.

Em sua conclusão, Assad (2019) salienta que para o desenvolvimento deste trabalho, focou na granularidade das informações coletadas aliando o consumo de recursos computacionais por microsserviços. Ficou claro que agentes de monitoramento em forma de operadores acabam por consumir muito mais recursos do *cluster*, do que, uma solução capaz de centralizar o monitoramento dos recursos de CPU, memória e rede no próprio servidor de métricas do Kubernetes.

# proposta do Protótipo

Nesta seção serão apresentadas as justificativas para o desenvolvimento do estudo proposto, bem como um quadro comparativo com os trabalhos correlatos, os requisitos funcionais e não funcionais e a metodologia utilizada no desenvolvimento da solução.

## JUSTIFICATIVA

É apresentado a seguir um comparativo de alguns trabalhos que abordam a questão dos microserviços e a ênfase do monitoramento, conforme o Quadro 1.

Quadro - Comparativo dos trabalhos correlatos

|  |  |  |  |
| --- | --- | --- | --- |
| Trabalhos Correlatos  Características | Rodrigues (2018) | Silva (2020) | Assad (2019) |
| Microsserviços | Sim | Sim | Sim |
| Escalabilidade | Sim | Sim | Sim |
| Contêiners | Sim | Não | Sim |
| Kubernetes | Sim | Não | Sim |
| Monitoramento | Não | Sim | Sim |
| Métricas | Não | Sim | Sim |
| Logging | Não | Sim | Não |

Fonte: elaborado pelo autor.

Conforme análise do Quadro 1, pode-se concluir que os trabalhos abordam de formas similares a utilização e implementação de microsserviços, bem como a utilização da estratégia do monitoramento voltadas para monitorar a arquitetura de microsserviços ou de uma arquitetura convencional de tecnologia da informação. Nesse contexto, o trabalho de Rodrigues (2018), intitulado MS-Trick*,* aborda a conversão e implementação de uma arquitetura de sistema monolítico em microsserviços. Já o trabalho de Silva (2020) aborda o tema voltando o foco para o monitoramento de infraestrutura de sistemas, sendo que se utiliza das vantagens da arquitetura de microsserviços para realizar o propósito de seu trabalho. Por fim, Assad (2019) aplica os conceitos de construção da arquitetura de microsserviços com o foco voltado também para o monitoramento, só que nesse caso, foca em investigar e expor soluções possíveis e o melhor caso para utilização no universo do Kubernetes.

Assim o presente trabalho se torna relevante no sentido em que pretende coletar, tratar e analisar informações de estado de um conjunto de protótipos de microsserviços em tempo real e permitir que o responsável por eles atue antes ou logo após ocorrer alguma falha no sistema monitorado.

Sobre os protótipos de microsserviços em questão: o primeiro irá executar uma funcionalidade de autenticação de acesso; o segundo irá ter as funcionalidades de CRUD (*create*,*read*,*update*,*delete*) para criação de usuários; o terceiro irá ter as funcionalidades de CRUD para criação de alunos; o quarto enviará notificações por e-mail; e o quinto enviará notificações por *push*.

O volume de informações a ser tratado neste tipo de problema pode variar muito de acordo com o tipo de monitoramento desejado. Por exemplo, caso se deseje monitorar apenas informações básicas de estado, como uso de memória, processador ou disco, a quantidade de dados a ser enviada por recurso monitorado é relativamente pequena, dependendo da frequência que estas informações foram programadas para serem enviadas. No entanto, este volume pode crescer rapidamente quando se deseja monitorar muitas máquinas ou quando se deseja monitorar ativos que produzam uma grande quantidade de dados por unidade de tempo, como por exemplo, roteadores que enviam metadados (IP de origem, IP de destino, porta de origem e de destino) de cada pacote roteado (TAVEIRA, 2015).

Para tanto buscou-se uma ferramenta que permita realizar as integrações necessárias de forma assíncrona e em tempo real, possibilitando atualizações distribuídas e entregando mais taxa de transferência (*throughput*): Kafka. Ele é uma plataforma distribuída de *streaming* que atua como uma fila de mensagens de publicação / assinatura, recebendo dados de vários sistemas de origem e disponibilizando-os para vários sistemas e aplicativos em tempo real (KAMILLA PIMENTEL, 2020).

Nesse contexto as principais vantagens da utilização do Kafka são que ele fornece armazenamento durável, o que significa que os dados armazenados nele não podem ser facilmente violados e são altamente escalonáveis, para que possam lidar com um grande aumento de usuários, cargas de trabalho e transações quando necessário (KAMILLA PIMENTEL, 2020).

## REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os principais requisitos funcionais e não funcionais estão apresentados nos tópicos seguintes, sendo divididos entre as especificações dos protótipos de microsserviços e da organização do *dashboard* de monitoramento a partir do Kafka.

## Protótipos de microsserviços

Os requisitos funcionais são:

1. o protótipo deve permitir o usuário realizar *login*;
2. o protótipo deve permitir a inclusão, alteração e remoção de usuários do sistema;
3. o protótipo deve permitir a inclusão de usuários em massa;
4. o usuário deve permitir a inclusão, alteração e remoção de alunos do sistema;
5. o protótipo deve permitir a inclusão de alunos em massa;
6. o protótipo deve emitir mensagens de erro caso algum dado esteja incompleto;
7. o protótipo deve permitir a inclusão de avisos para os alunos e usuários;
8. o protótipo deve enviar mensagens via *push notification* para alunos e usuários;
9. o protótipo deve enviar mensagens via e-mail para alunos e usuários.

Os requisitos não funcionais são:

1. implementar os microsserviços utilizando a ferramenta Spring Boot;
2. utilizar Github para repositório de fontes e desenvolver a integração contínua;
3. utilizar Docker para execução das instâncias dos microsserviços;
4. utilizar Ionic para implementação da interface do protótipo;
5. utilizar MySQL como ferramenta de banco de dados;
6. utilizar o conceito de *Domain-observed observability* na instrumentação dos *logs* dos microsserviços.

## Monitoramento

Os requisitos funcionais são:

1. gerar *logs* para tópicos do Kafka;
2. gerar métricas para tópicos do Kafka;
3. as mensagens de *log* devem ser indexadas;
4. os *logs* devem ser visualizados no *dashboard*;
5. as métricas devem ser visualizadas no *dashboard*.

Os requisitos não funcionais são:

1. utilizar Docker, Docker Composer como contêiner para execução das ferramentas;
2. utilizar Kafka para obter os *logs* e métricas;
3. utilizar Logstash e ElasticSearch para indexação dos *logs*;
4. utilizar Kibana para visualização dos *dashboards*.

## METODOLOGIA

O trabalho será desenvolvido observando as seguintes etapas:

1. realizar levantamento bibliográfico com relação aos assuntos abordados: arquitetura de microsserviços, estratégias de monitoramento, ferramentas utilizadas (Spring Boot, Kafka, Stask ELK - Elasticsearch, Logstash, Kibana), Ionic e das metodologias aplicadas (DEVOPS, Domain-observed observability);
2. realizar levantamento de requisitos: nesta etapa será feito o refinamento dos requisitos tomando como base a pesquisa realizada;
3. realizar especificação da arquitetura dos microsserviços e base de dados;
4. realizar implementação dos microsserviços: será realizada a implementação dos microsserviços utilizando a linguagem Java e Spring Boot. Será desenvolvida com base na arquitetura especificada no passo anterior;
5. realizar instalação e configuração das ferramentas de monitoramento;
6. realizar definição dos indicadores e métricas a serem gerados e ou coletados;
7. realizar configuração da interface para a visualização dos *dashboards* e métricas dos serviços monitorados;
8. realizar os testes dos protótipos de microsserviços, aplicando o monitoramento em situações diversas.

As etapas serão realizadas nos períodos relacionados no Quadro 2.

Quadro 2 – Cronograma

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2023 | | | | | | | | | |
|  | fev. | | mar. | | abr. | | maio | | jun. | |
| etapas / quinzenas | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |
| Realizar levantamento bibliográfico |  |  |  |  |  |  |  |  |  |  |
| Realizar levantamento de requisitos |  |  |  |  |  |  |  |  |  |  |
| Realizar especificação da arquitetura dos microsserviços |  |  |  |  |  |  |  |  |  |  |
| Realizar implementação dos microsserviços |  |  |  |  |  |  |  |  |  |  |
| Realizar instalação e configuração das ferramentas de monitoramento |  |  |  |  |  |  |  |  |  |  |
| Realizar definição dos indicadores e métricas |  |  |  |  |  |  |  |  |  |  |
| Realizar configuração da interface para a visualização dos dashboards |  |  |  |  |  |  |  |  |  |  |
| Realizar os testes dos protótipos de microsserviços e do monitoramento |  |  |  |  |  |  |  |  |  |  |

Fonte: elaborado pelo autor.

# REVISÃO BIBLIOGRÁFICA

Esta seção apresentará fundamentadas bibliografias sobre sistemas monolíticos, microsserviços, *logging* e monitoramento, *stream* de dados com Kafka e Stack ELK.

## Sistemas monolíticos

Quase todas as aplicações de software escritas atualmente podem ser divididas em três elementos distintos: um lado *frontend* (ou lado do cliente), um lado de *backend* e algum tipo de armazenamento de dados. As solicitações são feitas à aplicação a partir do lado do cliente, o código de *backend* faz o trabalho pesado e quaisquer dados relevantes que tiverem de ser armazenados ou acessados são enviados para o local em que os dados estão armazenados. Esta divisão ficou conhecida como arquitetura de três camadas (FOWLER, 2017).

Nos primeiros estágios, quando uma empresa é jovem, suas aplicações são simples e o número de desenvolvedores que contribuem para o código-base é pequeno. À medida que a empresa cresce, mais desenvolvedores são contratados, novos recursos são acrescentados à aplicação, com três consequências significativas:

1. aumento da carga de trabalho operacional;
2. aumento do número de linhas de código e complexidade da aplicação;
3. necessidade de dimensionamento horizontal ou vertical da aplicação.

À medida que a empresa cresce e o número de engenheiros ultrapassa a ordem de grandeza das dezenas ou centenas e tudo começa a ficar mais complicado. O desenvolvimento e a implantação tornam-se um pesadelo, o teste passa a ser um fardo e a implantação das correções mais cruciais é adiada, aumentando rapidamente a defasagem técnica. Aplicações cujo ciclo de vida se encaixa nesse padrão são conhecidas pela comunidade de software como monólitos (FOWLER, 2017).

Claro que nem toda aplicação monolítica é ruim ou sofre dos problemas listados, mas são muito raros os monólitos que não apresentam esses problemas em algum ponto de seu ciclo de vida. A razão pela qual a maioria dos monólitos é suscetível a esses problemas é a natureza de um monólito ser diretamente oposta à escalabilidade em termos mais gerais possíveis. Escalabilidade requer simultaneidade e segmentação: duas condições que são difíceis de ser obtidas com um monólito (FOWLER, 2017).

## microsserviços

Para Fowler (2017) a adoção da arquitetura de microsserviços resolve os desafios mais prementes apresentados pela arquitetura de aplicação monolítica. Os microsserviços não são assolados pelos mesmos desafios de escalabilidade, pela falta de eficiência ou pelas dificuldades de adotar novas tecnologias: eles são otimizados para escalabilidade, eficiência e velocidade de desenvolvimento.

Conforme descreve Ibm Cloud Education (2021), a arquitetura de microsserviços é uma abordagem na qual um único aplicativo é composto de muitos serviços menores que são implementáveis de forma independente e têm acoplamento fraco. O acoplamento fraco dos microsserviços também fornece um nível de isolamento de falhas, além de melhor resiliência para os aplicativos.

Segundo Fowler (2017), observamos este padrão surgir em empresas que executam aplicações em milhares, até mesmo em centenas de milhares de servidores e cujas aplicações tornaram-se monólitos e passaram a enfrentar desafios de escalabilidade. Os desafios que elas enfrentaram foram superados abandonando-se a arquitetura de aplicação monolítica em favor de microsserviços. Esses serviços normalmente:

1. têm a própria *stack* de tecnologia, incluindo o modelo de banco de dados e gerenciamento de dados;
2. comunicam-se sobre uma combinação de APIs de REST, fluxos de eventos e *brokers* de mensagens;
3. são organizados por recurso de negócios, com a linha separando os serviços que, muitas vezes, são chamados de contexto delimitado.

Além de suas características arquitetônicas, seu valor pode ser mais facilmente compreendido por meio de benefícios corporativos e de negócios bastante simples (IBM CLOUD EDUCATION, 2021):

1. o código pode ser atualizado mais facilmente, novos recursos ou funcionalidades podem ser incluídos sem mudar nenhuma parte do aplicativo;
2. as equipes podem usar *stacks* e linguagens de programação diferentes para componentes diferentes;
3. é possível ajustar a escala dos componentes de forma independente, reduzindo o desperdício e o custo associado ao ajuste de escala de aplicativos inteiros, como nos casos em que há um único recurso sobrecarregado.

O ambiente no qual os microsserviços são construídos, executados e interagem é onde eles vivem. Considerar este ambiente como um ecossistema é benéfico quando se adota a arquitetura de microsserviços (FOWLER, 2017). O ecossistema de microsserviços pode ser dividido em quatro camadas (Figura 1), embora as fronteiras entre elas não sejam sempre muito claras: alguns elementos da infraestrutura tocarão todas as partes da pilha. As três camadas inferiores são as camadas de infraestrutura: na parte inferior da pilha temos a camada de hardware e sobre ela a camada de comunicação (que chega até a quarta camada), seguida da plataforma de aplicação. A quarta camada (na parte superior) é onde vivem todos os microsserviços individuais.

Figura - Modelo de quatro camadas do ecossistema de microsservicos



Fonte: Fowler (2017, p. 35).

Fowler (2017) destaca que precisamos definir padrões e requisitos para nossos microsserviços e eles precisam ser genéricos o suficiente para ser aplicados a todos os microsserviços, porém específicos o suficiente também para ser quantificáveis e produzir resultados mensuráveis. É aí que entra o conceito de disponibilidade de produção: um conjunto de oito princípios que, quando adotados em conjunto, atendem a estes critérios. Cada um destes princípios é quantificável, resulta em um conjunto de requisitos acionáveis e produz resultados mensuráveis.

A força motora por trás destes princípios é que, juntos, eles contribuem para e impulsionam a disponibilidade de um microsserviço. Eles são: estabilidade, confiabilidade, escalabilidade, tolerância a falhas, prontidão para catástrofes, desempenho, monitoramento e documentação (FOWLER, 2017).

Os requisitos para construir um microsserviço estável são: um ciclo de vida de desenvolvimento estável; um processo de implantação estável; e procedimentos estáveis de introdução e descontinuação. Já os requisitos para construir um microsserviço confiável são: um processo confiável de implantação; planejar, atenuar e proteger-se contra falhas e dependências; e roteamento e descoberta confiáveis. Já os requisitos para construir um microsserviço escalável são: escalas de crescimento quantitativas e qualitativas bem definidas; identificação de gargalos e requisitos de recursos; planejamento de capacidade cuidadoso e preciso; tratamento escalável do tráfego; o escalamento das dependências; e armazenamento escalável dos dados. Já os requisitos para construir um microsserviço tolerante a falhas e preparado para qualquer catástrofe são: identificar e se planejar para os potenciais cenários de catástrofe e falhas; identificar e resolver os pontos únicos de falhas; estratégias para detecção e correção de falhas em funcionamento; teste de resiliência por meio de teste do código, teste de carga e teste de caos; o tráfego deve ser gerenciado cuidadosamente em preparação para falhas; e incidentes e interrupções devem ser tratados de forma adequada e produtiva. Já os requisitos para construir um microsserviço de alto desempenho são: SLAs (*service-level-agreement*) adequados de disponibilidade; adequado tratamento e processamento de tarefas; e utilização eficiente de recursos. Já os requisitos para construir um microsserviço adequadamente monitorado são: *logging* e rastreamento adequados em toda a pilha; *dashboards* bem projetados que sejam fáceis de entender e reflitam com precisão a saúde do serviço; alertas eficazes e acionáveis acompanhados de roteiros; e implementar e manter uma rotação das equipes de prontidão. E por fim, os requisitos para construir um microsserviço bem documentado são: documentação completa, atualizada e centralizada contendo todas as informações relevantes e essenciais sobre o microsserviço; e compreensão organizacional nos níveis de desenvolvedor, equipe e ecossistema (FOWLER, 2017).

## *logging* e monitoramento

Para Fowler (2017), um microsserviço pronto para produção é adequadamente monitorado. Um monitoramento adequado é uma das partes mais importantes de construir um microsserviço pronto para produção e garantir maior disponibilidade do microsserviço. Alguns dos requisitos para construir um microsserviço adequadamente monitorado são:

1. *logging* e rastreamento adequado em toda a pilha;
2. *dashboards* bem projetados que sejam fáceis de entender e reflitam com precisão a saúde do serviço;
3. alertas eficazes e acionáveis acompanhados de roteiros;
4. implementar e manter uma rotação das equipes de prontidão.

Todos os microsserviços devem manter um *logging* em nível de microsserviço de todas as solicitações feitas ao microsserviço (incluindo todas as informações relevantes e importantes) e suas respostas. Devido à natureza acelerada de desenvolvimento de microsserviços, geralmente é impossível reproduzir os *bugs* no código, pois é impossível reconstruir o estado do sistema no momento da falha. Um bom *logging* em nível de microsserviço oferece aos desenvolvedores as informações necessárias para que eles entendam completamente o estado de seu serviço em certo momento do passado ou presente. O monitoramento em nível de microsserviço de todas as principais métricas dos microsserviços é essencial por razões semelhantes: um monitoramento preciso e em tempo real permite que os desenvolvedores sempre conheçam a saúde e o status de seu serviço (FOWLER, 2017).

No caso dos *dashboards*, deve apresentar informações das métricas principais para microsserviços, conforme destaca Fowler (2017), cada microsserviço deve ter pelo menos um *dashboard* no qual todas as métricas principais (como utilização de hardware, conexões a databases, disponibilidade, latência, respostas e o *status* de *endpoints* de API) são coletadas e exibidas. Um *dashboard* é uma visualização gráfica que é atualizada em tempo real para refletir todas as mais importantes informações sobre um microsserviço. *Dashboards* devem ser facilmente acessíveis, centralizados e padronizados em todo o ecossistema de microsserviços. O essencial é identificar quais propriedades de um microsserviço são necessárias e suficientes para descrever seu comportamento e então determinar o que as mudanças nessas propriedades nos dizem sobre o status geral e a saúde do microsserviço. Conforme Fowler (2017), essas propriedades são denominadas de métricas principais. Dentre as métricas principais de microsserviços, podemos ter:

1. métricas específicas da linguagem; disponibilidade;
2. *Service Level Agreement* (SLA);
3. latência;
4. sucesso do *endpoint*;
5. respostas do *endpoint*;
6. tempos de resposta do *endpoint*;
7. clientes;
8. erros e exceções;
9. dependências.

Outro componente do monitoramento de um microsserviço pronto para produção são os alertas em tempo real. A detecção de falhas, assim como a detecção de mudanças nas métricas principais que possam levar a uma falha, é efetuada por meio de alertas. Para isso, todas as métricas principais – métricas de servidor, métricas de infraestrutura e métricas específicas de microsserviço – devem gerar alertas, e os alertas devem ser configurados em vários limites. Alertas eficazes e acionáveis são essenciais para preservar a disponibilidade de um microsserviço e evitar o *downtime* (FOWLER, 2017).

## Stream DE DADOS com kafka

Coelho (2020) destaca que em uma realidade onde o microsserviço é a palavra de ordem, acabamos com tantos componentes de variados tipos e funcionalidades espalhados pelos ambientes que nosso novo e principal desafio passou a ser a gestão destes componentes.

Ainda conforme Coelho (2020), o padrão bastante conhecido e usado atualmente para fazer a comunicação entre os microsserviços é o REST, só que as chamadas HTTP podem ser custosas, e por padrão, sempre se espera uma resposta do serviço consumidor (mesmo que seja apenas um HTTP 200). Para resolver isto é, em geral, adotada uma arquitetura de mensageria para que os componentes possam ser tornar independentes entre si e também 100% assíncronos (aqueles que não precisam esperar nenhum tipo de resposta imediata).

Por fim, Coelho (2020) descreve o Apache Kafka como uma solução assíncrona, tolerante a falhas e escalável. A ferramenta foi desenvolvida inicialmente pelo LinkedIn com o objetivo de solucionar problemas como a ingestão de baixa latência de grandes quantidades de dados de eventos do site e de sua infraestrutura, em uma arquitetura que utiliza o Hadoop e os sistemas de processamento de eventos em tempo real (COELHO, 2020).

Kafka ou Apache Kafka é um dos modelos *Publish-Subscribe* (Figura 2) mais populares do momento que o mundo de *Big Data* tem adotado em larga escala. Basicamente as desenvolvedoras do Linkedin gostariam de capturar o comportamento do usuário nas páginas (visualizar um perfil de usuário, pesquisar uma empresa, recomendar um usuário, concorrer para uma vaga, ou seja, qualquer ação realizada nas páginas) (SOUZA, 2018).

Para Souza (2018), fazer um monitoramento operacional de aplicações distribuídas e agregação de logs também pode ser um espaço para aplicação dessa ferramenta. Por exemplo, centralizar os logs de aplicações distribuídas e metrificar com alguma ferramenta estatística consumidora. A agregação de logs geralmente coleta arquivos de log físico de servidores e os coloca em um local central para processamento. Kafka abstrai os detalhes dos arquivos e fornece uma abstração mais limpa do log como um fluxo de mensagens. Isso permite um processamento de baixa latência e suporte mais fácil para várias fontes de dados e consumo de dados distribuídos.

Souza (2018) explica ainda que o Kafka possui os seguintes componentes arquiteturais: mensagem, *offset*, tópico, *partition* e *broker*. Mensagem é a informação que sua aplicação quer publicar ou consumir. As mensagens são armazenadas nos tópicos. Logo, tópicos podem ser vistos como *collections* de mensagem. *Partition* é o recurso de paralelização dentro dos tópicos e que possibilita a alta escalabilidade e disponibilidade dos dados. Um tópico tem que estar vinculado a pelo menos uma *partition*. Por fim, *broker* são os organizadores de t*opics/partitions*, que podem ser vistos como nós dentro de um *cluster* (Figura 3).

Para Ciência e Dados (2016), essencialmente, o Kafka age como uma espécie de “sistema nervoso central”, que coleta dados de alto volume como por exemplo a atividade de usuários (clicks em um web site), logs, cotações de ações, entre outros; e torna estes dados disponíveis como um fluxo em tempo real para o consumo por outras aplicações. O Kafka vem ganhando cada vez mais popularidade em *Big Data* pois além de ser um projeto *open-source* de alta qualidade, possui a capacidade de lidar com fluxos de alta velocidade de dados, característica cada vez mais procurada para uso em Internet das Coisas, por exemplo.

Figura - Modelo Produtor/Consumidor do Apache Kafka

Diagrama

Descrição gerada automaticamente

Fonte: Ciência e Dados (2016).

Figura - Tópicos, partições e broker

Diagrama, Desenho técnico

Descrição gerada automaticamente

Fonte: Doughty (2020).

## KUBERNETES

Ellingwood (2018) demonstra que Kubernetes, em seu nível mais básico, é um sistema para executar e coordenar aplicações em *container* através de um *cluster* de máquinas. É uma plataforma desenhada para gerenciar completamente o ciclo de aplicações e serviços em *container* utilizando métodos que fornecem previsibilidade, escalabilidade, e alta disponibilidade.

Segundo Ellgingwood (2018), como usuário do Kubernetes, você pode definir como as suas aplicações devem rodar e as maneiras pelas quais elas devem ser capazes de interagir com outras aplicações ou com o mundo exterior. Você pode escalar seus serviços para cima ou para baixo, executar atualizações contínuas elegantemente, e trocar tráfego entre diferentes versões de suas aplicações para testar recursos ou reverter *deployments* problemáticos. O Kubernetes fornece interfaces e primitivas de plataformas combináveis que lhe permitem definir e gerenciar suas aplicações com alto grau de flexibilidade, potência e confiabilidade.

Vertigo Tecnologia (2021) relaciona os seis principais componentes da arquitetura que compõem o Kubernetes, são eles: *container*, *pod*, *node*, *cluster*, *control plane* e armazenamento de dados. Os *containers* são os blocos em que os códigos das aplicações são armazenados em nuvem ou data center privado. O Kubernetes foi criado justamente para gerenciar de forma efetiva esses blocos de construção. Já os *pods* de K8s (sigla em inglês para Kubernetes) são uma coleção de *containers* que compartilham recursos, como *namespace* (para isolar recursos por projetos ou funcionalidades), armazenamento e rede. Os *pods* são dinâmicos e quando terminam as suas tarefas, são excluídos. Já um *node* é uma instância de servidor que executa outros componentes de nível superior no Kubernetes. Já um *cluster* consiste em um conjunto de servidores de processamento, os *nodes* responsáveis por executarem as aplicações conteinerizadas. Todo *cluster* tem pelo menos um servidor de processamento para hospedar os *pods*. Já o *control plane* ou plano de controle é o local onde o *cluster* é gerenciado. Existem muitos componentes que fazem parte do plano de controle, entre eles temos: APIserver, controller manager, etcd e scheduler. E Por fim, faz-se necessário o armazenamento de dados no Kubernetes, pois existem dados que fatalmente seriam perdidos quando um *pod* ou *container* tem erro ou é excluído, dessa forma, o Kubernetes utiliza os chamados volumes de armazenamento. É possível ter volumes de armazenamento externo para Kubernetes e esses volumes podem ser provisionados por você ou dinamicamente pelo sistema.

## stack elk

Elastic Stack (2022), explica que "ELK" é o acrônimo para três projetos open source: Elasticsearch, Logstash e Kibana. O Elasticsearch é um mecanismo de busca e análise. O Logstash é responsável por coletar os *logs* de várias fontes, processá-los e em seguida, enviá-los para o Elasticsearch que por sua vez faz as buscas desses *logs* de forma mais performática possível, utilizando um banco de dados noSQL (MongoDB ou Cassandra) para persistir os dados. Já o Kibana é uma interface que facilita a visualização desses dados em forma de listas e gráficos (BRANDÃO, 2020). E por fim, para ligar os pontos, 2Much 2Learn (2020), sugere uma solução usando o Docker para configurar Kafka com Stack ELK para habilitar recursos de log centralizados (Figura 4).

Figura - Orquestrando Kafka e ELK Stack para centralização de logs

Diagrama

Descrição gerada automaticamente

Fonte: 2Much 2Learn (2020).

Referências

2MUCH 2LEARN. **Configuring Centralized logging with Kafka and ELK stack**. 2020. Disponível em: https://2much2learn.com/centralized-logging-with-kafka-and-elk-stack/. Acesso em: 04 dez. 2022.

ASSAD, Lucas Serra da Cunha. **Mangue Beat! Investigando soluções para Monitoramento de Aplicações Orientadas a Microsserviços**. 2019. 46 f. TCC (Graduação) - Curso de Bacharelado em Sistemas de Informação, Centro de Informática, Universidade Federal de Pernambuco, Recife, 2019.

BRANDÃO, Daniel. **Log e Monitoramento de Microsserviços**: estado da arte, problemas e soluções com autoscaling multi-nuvem. 2020. Disponível em: https://medium.com/@dcruzb/log-e-monitoramento-de-microsserviços-11c465349a77. Acesso em: 19 set. 2022.

CIÊNCIA E DADOS. **Apache Kafka e Big Data**. 2016. Disponível em: https://www.cienciaedados.com/apache-kafka-e-big-data/. Acesso em: 01 dez. 2022.

COELHO, Rafael. **Apache Kafka, a evolução do stream de dados:** para que serve, qual o objetivo e seu funcionamento. 2020. Disponível em: https://www.allowme.cloud/conteudoallowme-apache-kafka-a-evolucao-do-stream-de-dados-para-que-serve-qual-o-objetivo-e-seu-funcionamento/. Acesso em: 01 dez. 2022.

COIMBRA, Vinicius. **Comparando: Elasticsearch vs MongoDB**. 2018. Disponível em: https://medium.com/data-hackers/comparando-elasticsearch-vs-mongodb-4b5932c613d9. Acesso em: 25 set. 2022.

DOBIES, Jason; WOOD, Joshua. **Operadores do Kubernetes**: automatizando a plataforma de orquestração de contêineres. São Paulo: Novatec O'Reilly, 2020. 160 p.

DOUGHTY, Matt. **How to Maximize Logging Performance with Kafka**. 2020. Disponível em: https://coralogix.com/blog/how-to-maximize-logging-performance-with-kafka/. Acesso em: 04 dez. 2022.

ELASTIC STACK. **O que é o ELK Stack?** 2022. Disponível em: https://www.elastic.co/pt/what-is/elk-stack. Acesso em: 04 dez. 2022.

ELLINGWOOD, Justin. **Uma Introdução ao Kubernetes**. 2018. Disponível em: https://www.digitalocean.com/community/tutorials/uma-introducao-ao-kubernetes-pt. Acesso em: 04 dez. 2022.

IBM CLOUD EDUCATION. **Microsserviços**. 2021. Disponível em: https://www.ibm.com/br-pt/cloud/learn/microservices. Acesso em: 10 out. 2022.

FOWLER, Susan J.. **Microsserviços prontos para produção**: construindo sistemas padronizados em uma organização de engenharia de software. São Paulo: Novatec O'Reilly, 2017. 218 p.

KUBERNETES. **Kubernetes the official documentation:** Logging Architecture. 2022. Disponível em: https://kubernetes.io/docs/concepts/cluster-administration/logging/#cluster-level-log. Acesso em: 25 set. 2022.

OPUS SOFTWARE. **Micro Serviços:** qual a diferença para a arquitetura monolítica?. Qual a diferença para a Arquitetura Monolítica?. 2021. Disponível em: https://www.opus-software.com.br/micro-servicos-arquietura-monolitica/. Acesso em: 19 set. 2022.

KAMILLA PIMENTEL (Brasil). **[TIMEBRA] MÉTRICAS IMPORTANTES PARA O MONITORAMENTO KAFKA**. 2020. Disponível em: https://blogac.me/timebra-metricas-importantes-para-o-monitoramento-kafka/. Acesso em: 26 set. 2022.

RODRIGUES, Ariel Rai. **MS-TRICK**: arquitetura de microservices aplicada. 2018. 85 f. TCC (Graduação) - Curso de Sistemas de Informação – Bacharelado, Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau, 2018.

ROSSI, Rodrigo. **Entrando no Mundo de Microsserviços**: parte 1. 2021. Disponível em: https://www.linkapi.solutions/blog/entrando-no-mundo-de-microsservicos-parte-1. Acesso em: 15 abr. 2022.

SILVA, Aléff Moura da. **WOLF MONITOR**: sistema de monitoramento de servidores. 2020. 39 f. TCC (Graduação) - Curso de Ciências da Computação, Centro Tecnológico, Centro Universitário Unifacvest, Lages, 2020.

SOUZA, Lorena de. **Kafka**: Quando usar, onde vive? e quais são seus componentes arquiteturais?. 2018. Disponível em: https://lorenadesouza.medium.com/kafka-o-que-%C3%A9-quando-usar-e-onde-vive-95345d410d8c. Acesso em: 01 dez. 2022.

SPRING. **Spring Boot**. 2017. Disponível em: https://spring.io/projects/spring-boot. Acesso em: 15 abr. 2022.

TAVEIRA, Luís Felipe Rabello. **Monitoramento de Ambientes Computacionais Distribuídos em Tempo Real**. 2015. 57 f. Monografia (Especialização) - Curso de Bacharelado em Engenharia da Computação, Departamento de Ciência da Computação, Universidade de Brasília, Brasília, 2015. Cap. 1. Disponível em: https://bdm.unb.br/bitstream/10483/10154/1/2015\_LuisFelipeRabelloTaveira.pdf. Acesso em: 26 set. 2022.

VERTIGO TECNOLOGIA. **Kubernetes**: 6 principais componentes da sua arquitetura. 2021. Disponível em: https://vertigo.com.br/kubernetes-6-principais-componentes-da-sua-arquitetura/. Acesso em: 04 dez. 2022.