CHRYSTIAN ARRIEL AMARAL

ANÁLISE COMPARATIVA DE FRAMEWORKS JAVA EM UM CLUSTER KUBERNETES: SPRING FRAMEWORK, QUARKUS E MICRONAUT

LAVRAS – MG

2023

CHRYSTIAN ARRIEL AMARAL

ANÁLISE COMPARATIVA DE FRAMEWORKS JAVA EM UM CLUSTER KUBERNETES: SPRING FRAMEWORK, QUARKUS E MICRONAUT

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. Dr. Rafael Serapilha Durelli
Orientador

LAVRAS – MG

2023

CHRYSTIAN ARRIEL AMARAL

ANÁLISE COMPARATIVA DE FRAMEWORKS JAVA EM UM CLUSTER KUBERNETES: SPRING FRAMEWORK, QUARKUS E MICRONAUT

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADA em 4 de Dezembro de 2023.

Prof. Dr. Rafael Serapilha Durelli	UFLA
Prof. Dr. Maurício Ronny de Almeida Souza	UFLA
Prof. Dr. Paulo Afonso Parreira Junior	UFLA

Prof. Dr. Rafael Serapilha Durelli
Orientador

LAVRAS – MG
2023

Dedico esse trabalho à toda minha família, em especial a minha mãe Luciana Aparecida Arriel Amaral, meu pai João Carlos Amaral, minha namorada Rafaela Rocha Nogueira e todos meus amigos e colegas.

AGRADECIMENTOS

Agradeço meus pais por todo apoio e incentivo.
Agradeço à minha namorada pelo apoio, compreensão e paciência.
Agradeço aos meus familiares e amigos, por palavras de afeto e apoio.
Agradeço ao Prof. Dr. Rafael Serapilha Durelli por todo o conhecimento e paciência no desenvolvimento desse projeto.
E agradeço à Universidade Federal de Lavras, o Departamento de Ciência da Computação e todos os professores que passaram pela minha trajetoria no curso.

RESUMO

Esta monografia propõe a construção de uma aplicação simples utilizando três *frameworks* Java: Spring Framework, Quarkus e Micronaut. Em seguida, será realizada uma comparação entre eles, utilizando o paradigma Goal Question Metric (GQM) para estruturar e definir objetivos e métricas. Serão estabelecidas métricas claras e objetivas, como tempo de inicialização, consumo de memória e desempenho em diferentes cargas de trabalho. O objetivo principal é avaliar o desempenho, eficiência e adequação de cada *framework* em um cluster Kubernetes.

Palavras-chave: devops - microservices - containers - docker - kubernetes - spring framework - quarkus - micronaut

ABSTRACT

This monograph proposes the development of a simple application using three Java frameworks: Spring Framework, Quarkus, and Micronaut. Subsequently, a comparison will be conducted among them, employing the Goal Question Metric (GQM) paradigm to structure and define objectives and metrics. Clear and objective metrics will be established, such as startup time, memory consumption, and performance under various workloads. The primary goal is to evaluate the performance, efficiency, and suitability of each framework in a Kubernetes cluster.

Keywords: devops - microservices - containers - docker - kubernetes - spring framework - quarkus - micronaut

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização e Motivação

O desenvolvimento de software está em constante evolução, trazendo mudanças e inovações na forma de conduzir seus processos. Como é possível analisar no trabalho de Grande, Vizcaíno e García (2024), o qual fez um Estudo de Mapeamento Sistemático, analisando 27 artigos e procurando analisar como o DevOps tem sido aplicado em ambientes distribuidos e globais, uma destas evoluções foi a adaptação à cultura de DevOps, sendo inicialmente mencionada nos anos 2000 e amplamente aceita, principalmente, por empresas de porte médio e grande. Esta cultura implica que os desenvolvedores se envolvam e sejam responsáveis pela implementação da aplicação em seu ambiente final, chamado de ambiente de produção.

No trabalho de TANZIL (2023), ao realizar um estudo empírico em postagens do StackOverflow e validando com profissionais de DevOps, foram descobertos 23 tópicos de DevOps agrupados em quatro categorias, sendo elas “Ferramenta de Cloud e CI/CD”, “Infraestrutura como Código”, “Contêiner e Orquestração” e “Garantia da Qualidade”. Nesse trabalho, o foco está em “Contêiner e Orquestração”, que têm como objetivo o uso do Kubernetes e do Docker.

Neste contexto, um ponto a ser analisado ao montar um *cluster* Kubernetes é o uso de recursos, como CPU e memória RAM, pois isso influencia diretamente nos custos de desenvolvimento da aplicação. Uma influência nesse ponto a ser analisada é o *framework*, pois cada um pode seguir caminhos diferentes no uso dos recursos para atender ao seu propósito. Por exemplo, o uso de reflexão em tempo de execução pode variar entre cada *framework*.

Considerando aplicações Java, três *frameworks* foram escolhidos pelos autores. O primeiro é o Spring Framework, que possui uma comunidade muito bem consolidada e muitos projetos realizados com ele. Já os outros dois são o Micronaut e o Quarkus, que se popularizaram no meio Java com a proposta de atender melhor ao ambiente de aplicações em nuvem. Isso porque propõem usar menos memória RAM e CPU.

Dessa forma, o trabalho procura responder se a escolha dentre os *framework* sitados anteriormente, poderia resultar em diferentes níveis de uso de recursos pelos containers orquestrados pelo Kubernetes. Essa resposta é valiosa para empresas que procuram desenvolver suas aplicações e ter custos menores com recursos em nuvem.

Para alcançar este objetivo, foi realizada uma abordagem Goal Question Metric (GQM) para definir o objetivo a ser atingido, as questões a serem respondidas e, por fim, as métricas a

serem obtidas. Essas métricas estão relacionadas ao uso de recursos do contêiner e à adaptabilidade ao contexto de um *cluster* Kubernetes. Serão apresentadas diferentes variantes de uma aplicação Java, todas seguindo a Arquitetura Hexagonal e desenvolvidas em um dos *frameworks* selecionados. Essas versões serão configuradas e implantadas em um *cluster* Kubernetes, que também incluirá instâncias do Prometheus e do Grafana para a coleta de métricas.

1.2 Objetivo

Este trabalho tem como objetivo realizar uma análise comparativa dos *frameworks* Spring Framework, Quarkus e Micronaut quando implementados em um *cluster* Kubernetes. Isso será feito utilizando o paradigma Goal Question Metric (GQM) para definir objetivos e métricas, visando mensurar o uso de recursos do contêiner e a adequação ao contexto de escalabilidade. Para atingir esse objetivo, serão desenvolvidas três aplicações Java, cada uma utilizando um dos *frameworks* mencionados. Os resultados obtidos serão utilizados para argumentar e responder às questões levantadas, proporcionando insights sobre qual *framework* é mais compatível e adequado para implementação em um *cluster* Kubernetes.

1.3 Organização

Este trabalho está organizado como descrito a seguir. No Capítulo 2 são apresentadas as tecnologias utilizadas, os conceitos e fundamentações teóricas deste trabalho. No Capítulo 3 é detalhada a metodologia conduzida. No Capítulo 4 é descrito a implementação da aplicação de teste. No Capítulo 5 são discutidos os resultados obtidos. E por fim, no Capítulo 6 é explicitado as considerações finais com sugestões de trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta as tecnologias e conceitos utilizados no desenvolvimento do projeto em questão. Ele está dividido em duas seções principais. A seção 2.1 aborda os conceitos e princípios utilizados e a seção 2.2 descreve as técnicas e tecnologias empregadas na criação do projeto.

2.1 Conceitos e Princípios Utilizados

Nesta seção, são apresentados os conceitos e princípios utilizados na implementação do projeto. Inicia-se com a descrição do conceito de Containização na seção 2.1.1. Por fim, é apresentado o paradigma Goal Question Metric (GQM) na seção 2.1.2.

2.1.1 Conteinerização

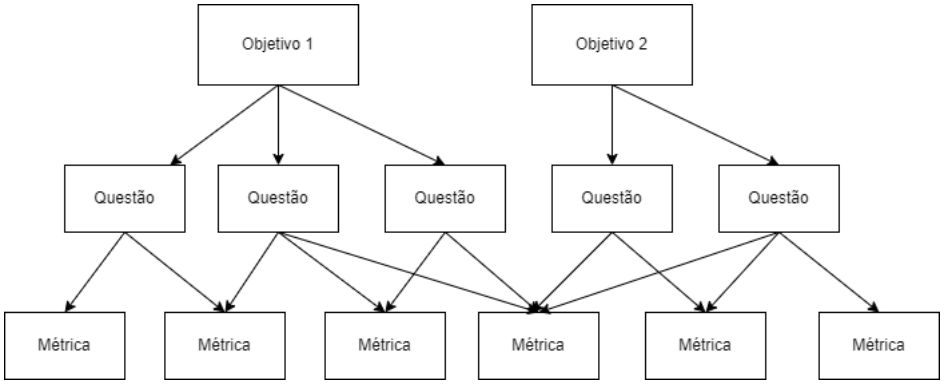
De acordo com Vitalino e Castro (2016), a contêinerização é o processo de encapsular aplicações em contêineres. Além disso, segundo Vitalino e Castro (2016), contêineres consistem no agrupamento da aplicação com suas dependências, compartilhando o *kernel* do sistema operacional do *host* onde estão em execução. Dessa forma, eles apresentam semelhanças com máquinas virtuais, mas, ao contrário destas, contêineres são mais leves e simples, aproveitando os recursos da máquina *host*. Os contêineres são portáteis para qualquer máquina que tenha uma plataforma de execução e implantação de aplicativos em contêineres, como o Docker, considerando, neste caso, contêineres gerados a partir de imagens Docker.

2.1.2 Goal Question Metric

De acordo com Caldiera e Rombach (1994), o Goal Question Metric (GQM) é um paradigma utilizado para estruturar a definição de métricas. Este método descreve que primeiro é necessário definir os objetivos, em seguida, rastrear dados que possam definir esses objetivos e, por fim, estabelecer uma estrutura para interpretar esses dados e atingir os objetivos. Em outras palavras, o GQM possui três níveis hierárquicos: o primeiro é o nível conceitual, o segundo é o nível operacional e, por fim, o nível quantitativo (métricas). A figura 2.1 representa esses níveis hierárquicos.

O nível conceitual está relacionado às metas gerais desejadas, representando o objetivo final e auxiliando na definição do propósito e direcionamento geral das métricas.

Figura 2.1 – Abordagem do GQM



Fonte: Autor

O nível operacional desdobra as metas gerais definidas no nível anterior em questões mais específicas e direcionadas. Essas questões são formuladas para compreender melhor como as metas podem ser alcançadas. Elas fornecem orientação sobre o que precisa ser investigado e respondido para atingir as metas estabelecidas.

O nível das métricas associa as questões levantadas no nível anterior a métricas específicas que podem ser medidas e analisadas para responder às perguntas formuladas. Elas podem ser quantitativas ou qualitativas e devem ser selecionadas com base na capacidade de responder às questões levantadas.

2.2 Tecnologias utilizadas

Nesta seção, são apresentadas todas as tecnologias utilizadas na implementação do projeto. Iniciando com a descrição da linguagem Java na Seção 2.2.1. Em seguida, uma descrição do gerenciador Maven na Seção 2.2.2. Os *frameworks* Spring Framework, Quarkus e Micronaut, são apresentados nas Seções 2.2.3, 2.2.4 e 2.2.5, respectivamente. Em seguida, as tecnologias Docker e Kubernetes são apresentadas nas Seções 2.2.6 e 2.2.7, respectivamente. Por fim, são apresentadas as tecnologias Prometheus e Grafana na Seção 2.2.8.

2.2.1 Java

Como descrito em Deitel (2017), Java é uma linguagem de programação orientada a objetos e plataforma de software, criada por James Gosling da Sun Microsystems em 1991. Sua criação teve como principal objetivo permitir que programadores desenvolvessem o código uma única vez e o executassem em qualquer dispositivo. Esta portabilidade é possível graças à Java Virtual Machine (JVM), que atua como intermediária entre a aplicação e o hardware.

Atualmente, é mantida e desenvolvida pela Oracle Corporation, com contribuições ativas da comunidade por meio de projetos de código aberto, como o OpenJDK (Java Development Kit). Atualmente já esta em sua versão 21, mas neste trabalho sera considerado a versão 17.

2.2.2 Maven

Como descrito pela documentação da ferramento, Apache Maven[1] é um *framework* criado pela Apache com o objetivo de facilitar a construção e automação de projetos Java. Ele oferece a possibilidade de gerenciar dependências e configurações do projeto em um arquivo XML. Dessa forma, o framework incentiva as boas práticas de organização de projetos.

2.2.3 Spring Framework

Como descrito em Walls (2018), o Spring Framework foi criado em 2003 e tem como objetivo facilitar a criação de aplicativos corporativos em Java, com suporte também para Groovy e Kotlin. Ele alcança esse objetivo tornando a construção de aplicativos mais leve, evitando a implementação de comportamentos desnecessários, o que é uma resposta à complexidade das primeiras especificações *J2EE*[2]. Dois contextos de relevância para este trabalho são o suporte nativo à inversão de controle e injeção de dependência, conhecido como os *Beans*.

Como descrito na sua documentação, o Spring Framework[3] possui diversos módulos que auxiliam nas necessidades de desenvolvimento, como: segurança, banco de dados e web APIs. Exemplos desses módulos incluem: Spring Boot, Spring Security, Spring Data, Spring Cloud e Spring Batch.

É um projeto de código aberto com uma comunidade ativa e muitos anos de evolução, o que o torna bem estabelecido.

2.2.4 Quarkus

Como descrito na documentação da ferramenta, o Quarkus[4] é um framework Java direcionado ao contexto de contêineres. Seu projeto é de código aberto sob a *Licença Apache versão 2.0*[5].

[1] Disponível em: <https://maven.apache.org/what-is-maven.html>. Acesso em 6 de Mar. 2023
[2] Disponível em: <https://en.wikipedia.org/wiki/Jakarta_EE>. Acesso em 6 de Mar. 2023
[3] Disponível em: <https://docs.spring.io/spring-framework/reference/overview.html>. Acesso em 6 de Mar. 2023
[4] Disponível em: <https://pt.quarkus.io/about/>. Acesso em 6 de Mar. 2023
[5] Disponível em: <https://www.apache.org/licenses/LICENSE-2.0>. Acesso em 6 de Mar. 2023

Como descrito em Bueno e Porter (2020), o Quarkus é proposto para ser otimizado, com baixo uso de memória e tempos de inicialização reduzidos com base em:

- Processamento máximo durante a compilação, reduzindo o tempo de execução apenas para as classes que serão usadas. Classes que não serão utilizadas em tempo de execução não chegam na JVM de produção.
- Redução no uso de reflexão por meio da substituição de chamadas de reflexão por chamadas regulares e impedindo proxies dinâmicos por meio da geração de proxies personalizados.
- Suporte ao GraalVM Native Executable, obtendo um tempo de inicialização menor e com uma heap menor do que em uma JVM padrão. Isso se dá pela incorporação de apenas partes da JVM e as classes absolutamente necessárias pela aplicação, pelo compilador nativo.
- Pré-inicialização do código de inicialização, reduz ainda mais o tempo de inicialização.
- Projetado para uso em Kubernetes, pois gera em tempo de compilação os metadados necessários para implantação no cluster Kubernetes. Por meio de uma única linha de comando, pode ser implementado em um cluster Kubernetes.

2.2.5 Micronaut

De acordo com a documentação da ferramenta, o Micronaut[6] foi criado sob a liderança de Graeme Rocher, criador do *framework* Java, Grails[7]. Também está sob a Licença Apache versão 2.0 e é mantido pela Object Computing Inc. (OCI)[8].

Também de acordo com a documentação da ferramenta, o Micronaut é um *framework* poliglota, atendendo às linguagens Java, Kotlin e Groovy, todas da JVM. Também possui diferentes possibilidades de aplicações, como, por exemplo, aplicações Serverless, ferramentas de CLI, aplicativos Android, IoT, entre outros. É também uma ferramenta *cloud-native*, com suporte para *service discovery*, *load balancing*, *circuit breaker*, entre outras.

De acordo com Singh, Dawood et al. (2021), o Micronaut, assim como o Quarkus, tem como propósito atender à nova realidade de aplicações de microsserviços e sem servidor. Para

[6] Disponível em: <https://docs.micronaut.io/index.html>. Acesso em 6 de Mar. 2023
[7] Disponível em: <https://grails.org/>. Acesso em 6 de Mar. 2023
[8] Disponível em: <https://objectcomputing.com/>. Acesso em 6 de Mar. 2023

alcançar esse objetivo, assim como o Quarkus, há uma redução significativa no uso de *reflection*, resolvendo a injeção de dependência em tempo de compilação, o que também o torna apto para ser compilado em código nativo pela Graal VM.

2.2.6 Docker

De acordo com o Vitalino e Castro (2016), Docker é uma solução que possibilita a criação de containeres Docker. Foi criado pela dotCloud[9], atualmente chamada de Docker, sendo um projeto open source desde 2013, obtendo ate o momento atual uma comunidade grande e colaborativa.

Como descrito por Vitalino e Castro (2016), o Docker trabalha com a ideia de imagens, as quais são geradas a partir de um arquivo chamado *Dockerfile*. Este arquivo descreve como será o container final. As imagens geradas carregam todas as dependências e arquivos da aplicação que serão executadas no container, incluindo também as dependências da imagem base escolhida. Uma distribuição Linux específica ou algo mais elaborado, como, por exemplo, uma distribuição com o Maven instalado, são exemplos de imagens que podem ser geradas. As imagens podem ser enviadas para uma conta no DockerHub, ficando disponível para qualquer usuário da comunidade.

2.2.7 Kubernetes

Como descrito em Santos (2019), o Kubernetes tem como objetivo oferecer um ambiente capaz de automatizar, implantar, escalar e gerenciar contêineres, tarefas que compõem a *orquestração de contêineres*, de forma robusta e flexível. Seu desenvolvimento ocorreu pelo time da Google em 2014 e atualmente é um projeto open-source mantido pela Cloud Native Computing Foundation (CNCF)[10], com o apoio da comunidade.

A ferramenta proporciona um apoio significativo à construção de infraestrutura de aplicações, ao mesmo tempo em que já incorpora processos como escalabilidade, disponibilidade e portabilidade. No entanto, demanda um nível de conhecimento técnico maior dos desenvolvedores para configurar e gerenciar o cluster Kubernetes nos diferentes ambientes do processo de desenvolvimento de software.

[9] Disponível em: <https://www.docker.com/press-release/dotcloud-inc-now-docker-inc/>. Acesso em 6 de Mar. 2023

[10] Disponível em: <https://www.cncf.io/>. Acesso em 6 de Mar. 2023

A escalabilidade se dá pela sua capacidade de automaticamente ajustar a quantidade de nós necessária dado a carga recebida pela aplicação. A disponibilidade ocorre com a capacidade de retomar aplicações que entraram em estado de falha e distribuir a carga de forma equilibrada. Já a portabilidade ocorre com a capacidade da ferramenta de ser executada em diferentes ambientes de infraestrutura, como provedores de serviço em nuvem.

2.2.8 Prometheus e Grafana

De acordo com Brazil (2018), o Prometheus é um projeto de código aberto para monitoramento de métricas de sistemas, iniciado pela SoundCloud em 2012, gerando uma comunidade e um ecossistema em seu entorno. Ele é escrito em Go e licenciado sob a licença Apache 2.0. Linguagens e *frameworks* populares possuem bibliotecas que possibilitam a exportação de suas métricas no formato do Prometheus. Já o Kubernetes e o Docker possuem essas bibliotecas nativamente.

Também, de acordo com Brazil (2018), seu funcionamento está baseado em um processo de *scrape*, que consiste em uma requisição contínua às aplicações-alvo por meio do protocolo HTTP. Após realizar esses *scrapes*, os dados são armazenados e podem ser usados para montar dashboards e/ou criar alertas para gerenciadores de alerta. Os dados são armazenados e interpretados como séries temporais, formando assim o *time series database* (TSDB).

Os *dashboards* funcionam por meio da linguagem PromQL, que permite criar *queries* para serem usadas nos dados armazenados. Por meio dessas *queries*, pode-se filtrar tags, intervalos e diferentes informações, além de realizar operações básicas de matemática.

Para a exibição dos dashboards neste projeto, foi escolhido o uso do Grafana[11], uma ferramenta amplamente reconhecida de código aberto. O Grafana permite consultar, visualizar, alertar e explorar métricas, logs e rastreamentos, independentemente de onde estejam armazenados. Com ele, é possível transformar dados de bancos de dados de série temporal (TSDB) em gráficos e visualizações detalhadas, conforme descrito na documentação oficial.

[11] Disponível em: <https://grafana.com/docs/grafana/latest/>

3 METODOLOGIA DE DESENVOLVIMENTO

Neste capítulo, são apresentados os procedimentos seguidos para o desenvolvimento e obtenção dos resultados, na Seção 3.1. Também é discutida a definição das métricas de comparação na Seção 3.2. São apresentados os métodos de captação das métricas na Seção 3.3. Por fim, é discutido como é feita a análise das métricas na Seção 3.4.

3.1 Procedimentos

Dado o objetivo do trabalho, que é apresentar um comparativo entre os *frameworks* Spring, Quarkus e Micronaut, no contexto do Kubernetes, foram estabelecidos os seguintes passos:

1. Utilizando o paradigma GQM, definir as métricas que permitirão a comparação entre os *frameworks* em um ambiente Kubernetes. Dessa etapa, será possível determinar quais métricas devem ser analisadas, como devem ser obtidas e qual será a análise realizada, visando atender a cada meta estabelecida.

2. Elaboração de uma aplicação com conceito simples, a qual será implementada em três versões, uma para cada um dos *frameworks*, mantendo a mesma definição de arquitetura entre elas.

3. Com base nas aplicações obtidas na etapa anterior, realizar a implementação de cada versão no ambiente Docker e, posteriormente, no ambiente Kubernetes, para que seja possível obter as métricas estabelecidas na primeira etapa.

4. Após a implementação das aplicações e estando no cluster Kubernetes, realizar a coleta das métricas definidas por meio de ferramentas de análise de métricas e testes de carga.

5. Com as métricas obtidas, a última etapa é a análise, buscando responder às questões levantadas e, assim, atingir as metas definidas.

3.2 GQM para definição das métricas

Com o objetivo de definir as métricas, será utilizado o paradigma Goal Question Metric (GQM). A primeira etapa do GQM é chamada de nível conceitual, que consiste em definir os objetivos das métricas:

1. Mensurar para cada *framework*, características para escabilidade da aplicação, no cluster Kubernates.

2. Mensurar o uso de recursos do container por cada *framework*, no cluster Kubernetes.

A segunda etapa do GQM é chamada de nível operacional, que consiste em levantar questões específicas para atingir os objetivos:

Questões levantadas para o Objetivo 1 - Mensurar para cada *framework*, características para escabilidade da aplicação, no cluster Kubernates.:

1. Qual o tamanho da imagem Docker, gerado por cada *framework*?

2. E qual o tamanho do arquivo JAR, gerado por cada *framework*?

3. Quantas bibliotecas compartilhadas foram adicionadas nesse JAR, por cada *framework*?

4. Qual o tempo, que cada *framework* leva para realizar o processo de inicialização e estar pronto para receber requisições na API?

Questões levantadas para o Objetivo 2 - Mensurar o uso de recursos do container por cada *framework*, no cluster Kubernetes:

1. Com a aplicação rodando em um cluster Kubernetes, qual é a quantidade de CPU usada do container, por cada *framework*, assim que é inicializado?

2. Com a aplicação rodando em um cluster Kubernetes, qual é a quantidade de CPU usada do container, por cada *framework*, quando recebe uma alta carga de requisições?

3. Com a aplicação rodando em um cluster Kubernetes, qual é a quantidade de memória RAM usada do container, por cada *framework*, assim que é inicializado?

4. Com a aplicação rodando em um cluster Kubernetes, qual é a quantidade de memória RAM usada do container, por cada *framework*, quando recebe uma alta carga de requisições?

A terceira etapa do GQM é chamada de nível de métricas, que consiste na identificação das métricas e indicadores para responder às perguntas e avaliar o progresso em direção às metas:

- Tamanho da imagem Docker
  - É menos influente na escolha do *framework* para um projeto, mas pode ser relevante quando é necessário armazenar muitas imagens diferentes, especialmente em cenários com várias aplicações, como em arquiteturas de microserviços. Isso também influencia o tempo necessário para obter essas imagens, já que o download pode levar mais tempo quando a imagem é maior.
  - Mensurado em MB.
- Tamanho do arquivo JAR e quantidade de bibliotecas compartilhadas
  - Mensurado em MB.
- Tempo de inicialização da aplicação
  - Influencia diretamente no tempo de escalonamento da aplicação, o que pode afetar a disponibilidade dela para atender a muitas requisições.
  - Mensurado em segundos.
- Média do uso de memória RAM
  - Em particular, representa a maior diferença que um *framework* pode ter em relação a outro, devido às diferentes abordagens de cada um.
  - Esta relacionada ao consumo de recursos, implicando em custos de operação e, portanto, são importantes na escolha de um *framework*.
  - Mensurado em MB.
- Média do uso de CPU
  - Também pode indicar a maior diferença que um *framework* pode ter em relação a outro, devido às diferentes abordagens de cada um.
  - Esta relacionada ao consumo de recursos, implicando em custos de operação e, portanto, são importantes na escolha de um *framework*.
  - Mensurado em porcentagem de uso.

Tabela 3.1 – GQM para definição das métricas

Objetivos	**Questões**	**Métricas**
Mensurar para cada *framework*, características para escabilidade da aplicação, no cluster Kubernates.	Qual o tamanho da imagem Docker, gerado por cada framework?	Tamanho da imagem Docker
	E qual o tamanho do arquivo JAR, gerado por cada framework?	Tamanho do arquivo JAR e quantidade de bibliotecas compartilhadas
	Quantas bibliotecas compartilhadas foram adicionadas nesse JAR, por cada framework?	
	Qual o tempo, que cada *framework* leva para realizar o processo de inicialização e estar pronto para receber requisições na API?	Tempo de inicialização da aplicação
Mensurar o uso de recursos do container por cada framework, no cluster Kubernetes.	Qual é a quantidade de CPU usada do container, por cada framework, assim que é inicializado?	Média do uso de CPU
	Qual é a quantidade de CPU usada do container, por cada framework, quando recebe uma alta carga de requisições?	
	Qual é a quantidade de memória RAM usada do container, por cada framework, assim que é inicializado?	Média do uso de memória RAM
	Qual é a quantidade de memória RAM usada do container, por cada framework, quando recebe uma alta carga de requisições?	

3.3 Métodos de captação das métricas

Nessa seção, serão descritos os métodos definidos para a captação das métricas.

3.3.1 Uso de memória RAM e CPU

Para a captação das métricas de uso de memória RAM e CPU, haverá dois cenários de teste. O primeiro consiste na execução da aplicação, porém sem receber nenhuma requisição em sua API durante um intervalo de tempo de 10 minutos. O segundo cenário envolve o envio de diferentes requisições simultâneas à API, representando um teste de carga, por um intervalo de tempo de 60 segundos.

Para a obtenção das métricas relacionadas ao uso de recursos (memória RAM e CPU) em cada versão da aplicação de teste nos três *frameworks*, foi definido o uso do software Prometheus em conjunto com o software Grafana.

3.3.2 Teste de carga

Para realizar o teste de carga, foi utilizado um script em JavaScript que faz uso da biblioteca *K6*[1] como ferramenta auxiliar. O script realiza uma sequência de passos utilizando a API da aplicação, onde esses passos são:

1. Criação de uma nova Task - endpoint */task/createTask*
2. Busca da Task criada - endpoint */task/getTaskById*
3. Mudança de status da Task para completo - endpoint */task/completedTaskById*
4. Mudança de status da Task para incompleto - endpoint */task/uncompletedTaskById*
5. Mudança de título da Task - endpoint */task/changeTaskTitleById*
6. Mudança de descrição da Task - endpoint */task/changeTaskDescriptionById*
7. Mudança de data de entrega da Task - endpoint */task/changeTaskDeadlineById*
8. Deleção da Task - endpoint */task/deleteTask*

Por meio do *K6*, esse script é executado durante 60 segundos por 20 usuários virtuais simultâneos (20 threads assíncronas).

[1] Disponível em: <https://k6.io/>. Acesso em 6 de Mar. 2023

3.3.3 Dashboard de métricas no Grafana

Por meio das métricas geradas por cada aplicação e obtidas pelo Prometheus, será possível, através de um dashboard no Grafana, analisar os resultados deste trabalho em diferentes cenários.

O dashboard possui uma *variable* chamada *instance* que filtra toda a exibição do dashboard pela instância do cluster que deve ser visualizada. Também há outra *variable* chamada *pod* que filtra o pod do cluster que deve ser visualizado. Com isso, é possível referenciar esses valores nas demais queries do dashboard usando a anotação *"$instance"* ou *"$pod"*.

O dashboard está dividido em duas partes, sendo a primeira a análise de métricas em relação ao pod da aplicação, exibindo as métricas de seu contêiner. A segunda parte analisa as métricas obtidas pelas próprias aplicações, relacionadas à JVM.

3.3.3.1 Métricas relacionadas ao pod

A descrição da primeira parte do dashboard e suas *queries* estão a seguir:

- Container CPU Usage
  - rate(container_cpu_usage_seconds_total{pod="$pod"}[$__range])
- Container Memory Usage
  - container_memory_working_set_bytes{pod="$pod"}
  - container_memory_usage_bytes{pod="$pod"}

Os gráficos dessa parte do dashboard utilizam métricas expostas pelo *cAdvisor*[2], um daemon que coleta, agrega, processa e exporta informações sobre contêineres em execução, oferecendo suporte nativo para contêineres *Docker*. Conforme descrito na documentação do *Kubernetes*[3], o *kubectl* utiliza o *cAdvisor* para coletar e exportar métricas por meio de um endpoint do cluster.

A métrica *container_cpu_usage_seconds_total* visa informar o uso de CPU pelo contêiner, representando o tempo de CPU consumido pelo contêiner em segundos de núcleo. Como

[2] Disponível em: <https://prometheus.io/docs/guides/cadvisor/>. Acesso em 6 de Mar. 2023

[3] Disponível em: <https://kubernetes.io/docs/reference/instrumentation/cri-pod-container-metrics/>. Acesso em 6 de Mar. 2023

se trata de contadores, é necessário usar o operador de taxa (*rate*) na consulta (*query*) para obter o número de núcleos usados no intervalo definido (referenciado pelo *"$__range"*).

A métrica *container_memory_working_set_bytes* visa informar o uso de memória física do host pelo contêiner, incluindo a memória compartilhada com outros contêineres. Por outro lado, a métrica *container_memory_usage_bytes* visa informar o uso de memória do host pelo contêiner, incluindo a memória compartilhada e privada. A diferença entre ambas é que a primeira se concentra no uso da memória RAM pelo contêiner, enquanto a segunda indica toda a memória utilizada pelo contêiner. Ambas são em bytes.

Essas métricas podem ser encontradas na documentação de métricas do *Kubernetes*[4].

3.3.3.2 Métricas relacionadas a JVM

A descrição da segunda parte do dashboard e suas *queries* estão a seguir:

- CPU - Total
  - system_cpu_usage{instance="$instance"}
  - process_cpu_usage{instance="$instance"}
- Memory RAM - Total
  - sum(jvm_memory_used_bytes{instance="$instance"})
  - sum(jvm_memory_committed_bytes{instance="$instance"})
- Memory RAM - Heap
  - sum(jvm_memory_used_bytes{instance="$instance", area="heap"})
  - sum(jvm_memory_committed_bytes{instance="$instance", area="heap"})
- Memory RAM - Non Heap
  - sum(jvm_memory_used_bytes{instance="$instance", area="nonheap"})
  - sum(jvm_memory_committed_bytes{instance="$instance", area="nonheap"})

[4] Disponível em: <https://kubernetes.io/docs/reference/instrumentation/metrics/>. Acesso em 6 de Mar. 2023

Todas as métricas da segunda parte do dashboard são geradas pela própria aplicação, sendo captadas pelo Prometheus em seu processo de *scrap*, por meio de um endpoint específico. Em cada uma das versões da aplicação de teste, esse endpoint de métricas é gerado por uma biblioteca específica, sendo no Spring Framework a biblioteca *Actuator*[5], e no Quarkus e Micronaut a biblioteca *Micrometer*[6]. Ambas as bibliotecas possuem exportação de métricas para o *client* Java do Prometheus.

A métrica *system_cpu_usage* visa informar o uso de CPU do sistema em que o aplicativo está sendo executado. Já a métrica *process_cpu_usage* visa informar o uso de CPU pela JVM.

A métrica *jvm_memory_used_bytes* tem o objetivo de informar o uso de memória em uma determinada área da JVM, que incluem a *Heap* e a *Non Heap*. Por outro lado, a métrica *jvm_memory_committed_bytes* visa informar a quantidade de memória alocada pela JVM para uso futuro, abrangendo tanto a memória que está sendo utilizada no momento quanto a memória que ainda não foi utilizada até o momento. Ambas as métricas são expressas em bytes.

Essas métricas podem ser encontradas na documentação de métricas do *Prometheus Java Metrics Library*[7].

3.3.3.3 Análise da memória

Como descrito na documentação da JVM do Java SE 17, no site da *Oracle*[8], a estrutura da memória na JVM divide-se em duas categorias, *heap* e *non-heap*. A primeira trata-se da memória para alocação dinâmica de objetos, sendo dividida em três partes: *Young Generation* e *Old Generation*. Já a segunda trata-se da memória não relacionada com a alocação dinâmica de objetos, sendo chamado de *Metaspace*. Dessa forma, para definir a métrica de uso da memória RAM, vamos dividi-la em duas análises, sendo da memória *heap* e da *non-heap*.

3.3.4 Tamanho da imagem Docker

O tamanho da imagem Docker é obtido por meio da interface de linha de comando (CLI) do próprio *Docker*[9].

---

[5] Disponível em: <https://docs.spring.io/spring-boot/docs/current/actuator-api/htmlsingle/>. Acesso em 6 de Mar. 2023

[6] Disponível em: <https://micrometer.io/>. Acesso em 6 de Mar. 2023

[7] Disponível em: <https://prometheus.github.io/client_java/>. Acesso em 6 de Mar. 2023

[8] Disponível em: <https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html#jvms-2.5.3>. Acesso em 6 de Mar. 2023

[9] Disponível em: <https://docs.docker.com/engine/reference/commandline/cli/>. Acesso em 6 de Mar. 2023

3.3.5 Tamanho do arquivo JAR e quantidade de bibliotecas compartilhadas

O tamanho do arquivo JAR gerado ao realizar o *build* da aplicação, o qual é afetado pela quantidade de bibliotecas compartilhadas utilizadas pelo *framework*. Dessa forma, também é realizada uma análise das bibliotecas incluídas no JAR gerado e o tamanho do *fat JAR*[10] para comparação entre os *frameworks*. O número de bibliotecas compartilhadas é obtido utilizando a CLI do Maven, desconsiderando as bibliotecas do próprio projeto, pelo comando:

```
$ mvn dependency:list
```

Já o tamanho do *fat JAR* é obtido utilizando o comando, no Windows Powershell:

```
$ dir <path-JAR>
```

3.3.6 Tempo de inicialização

A métrica de tempo de inicialização foi obtida com base no valor exibido pelo *log* de cada um dos *frameworks* em questão.

3.4 Análise das métricas

Nessa seção sera descrito como é realizado a analise dos dados obtidos.

3.4.1 Tamanho da imagem Docker, tamanho do arquivo JAR, quantidade de bibliotecas compartilhadas e tempo de inicialização

A análise do tamanho da imagem Docker, do tamanho do arquivo JAR, da quantidade de bibliotecas compartilhadas e do tempo de inicialização será realizada com base nos próprios valores obtidos, registrados em uma tabela. Cada linha representará os resultados de um *framework*. A comparação será feita utilizando essa tabela para identificar as diferenças nos valores entre os *frameworks*.

3.4.2 Uso de CPU e uso de memória RAM

Como descrito na seção 3.3.1, serão obtidos dados para dois cenários por meio do Prometheus e do Grafana, os quais geram os resultados em séries temporais. Com base nisso, a

[10] Disponível em: <https://dzone.com/articles/the-skinny-on-fat-thin-hollow-and-uber>. Acesso em 6 de Mar. 2023

análise dos dados será feita por meio da análise gráfica dos dashboards e pelo cálculo da média aritmética dos valores da série temporal, de acordo com a fórmula 3.1, onde *V* indica o conjunto de todos os valores da série temporal.

$$\text{Média da Métrica} = \frac{\sum_{i=1}^{\text{size (V)}} v_i}{\text{size (V)}} \tag{3.1}$$

Dessa forma, por meio das médias obtidas, será possível construir gráficos que indicam cada métrica obtida para cada cenário. Esses gráficos servirão de base para a análise comparativa dos *frameworks*.

4 RESULTADOS

Neste capitulo serão apresentados os resultados obtidos durante o desenvolvimento da aplicação de teste para cada *framework* em foco, como também os resultados obtidos das metricas e a comparação entre elas.

Todos os testes foram realizados em um computador com o sistema operacional Windows 11 Home Single Language de 64 bits, versão 22H2, equipado com 8,00 GB de memória RAM e um processador Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz.

A proposta da aplicação de teste desenvolvida, com o objetivo de auxiliar no entendimento e replicabilidade da mesma, mas que não é necessário para a compreensão dos resultados, é apresentada no anexo A.

4.1 Metricas obtidas

Nesta seção, serão apresentados os resultados obtidos em relação ao tamanho da imagem Docker, tamanho do arquivo JAR, tempo de inicialização, uso de memória RAM e uso de CPU, conforme definido no GQM apresentado na seção 3.2.

4.1.1 Tamanho da imagem Docker e tamanho do arquivo JAR

Com base nos valores indicados pela CLI do Docker e do Maven, foram obtidos os seguintes dados referentes ao tamanho da imagem Docker e à quantidade de bibliotecas incluídas no JAR, representados na tabela 4.1.

Tabela 4.1 – Dados obtidos dos arquivos JAR e imagens Docker

Framework	JAR	Bibliotecas Compartilhadas	Imagem Docker
Spring Framework	51,81 MB	99	462,02 MB
Quarkus	0,0127 MB	180	444,19 MB
Micronaut	35,32 MB	95	444,80 MB

Pode-se analisar pelos dados da tabela que o Spring Framework possui um arquivo JAR de tamanho maior que os demais, com um número de bibliotecas compartilhadas um pouco maior que o Micronaut e quase a metade da quantidade do Quarkus. Isso indica que o número de bibliotecas compartilhadas não influencia no tamanho do arquivo JAR final. O mesmo pode ser dito para a imagem Docker, já que o Spring Framework tem 18 MB a mais em relação aos demais.

Sem dúvida, o Quarkus se destaca ao apresentar um arquivo JAR de tamanho extremamente menor que os demais. No entanto, isso não trouxe uma grande diferença no tamanho da imagem Docker, pois sua diferença em relação ao Micronaut é menor que 1 MB.

Portanto, para atender ao primeiro objetivo do GQM, traçado na seção 3.2, a análise da imagem Docker é a métrica mais ideal a ser considerada. Ao realizar a comparação, é possível definir o Quarkus como o que possui menor imagem Docker, mostrando melhor aptidão para contextos de escalabilidade.

4.1.2 Tempo de inicialização

Todos os três *frameworks* exibem o tempo de inicialização por meio do log, ao final do processo. Com isso, os dados obtidos são descritos na tabela 4.2.

Tabela 4.2 – Tempo de inicialização em segundos

Framework	Resultado
Spring Framework	30,266
Quarkus	9,684
Micronaut	14,220

Ao analisar a tabela, é possível notar que o Spring Framework possui o maior tempo de inicialização, enquanto o Quarkus possui o menor tempo, e o Micronaut fica mais próximo do Quarkus. Com base no objetivo traçado pelo GQM, na seção 3.2, o Quarkus apresenta maior aptidão em contextos de escalabilidade, já que é inicializado mais rapidamente, deixando o Spring Framework em último lugar nesse quesito. O Micronaut, nesse caso, não está muito distante do Quarkus.

4.1.3 Uso de memória RAM e CPU

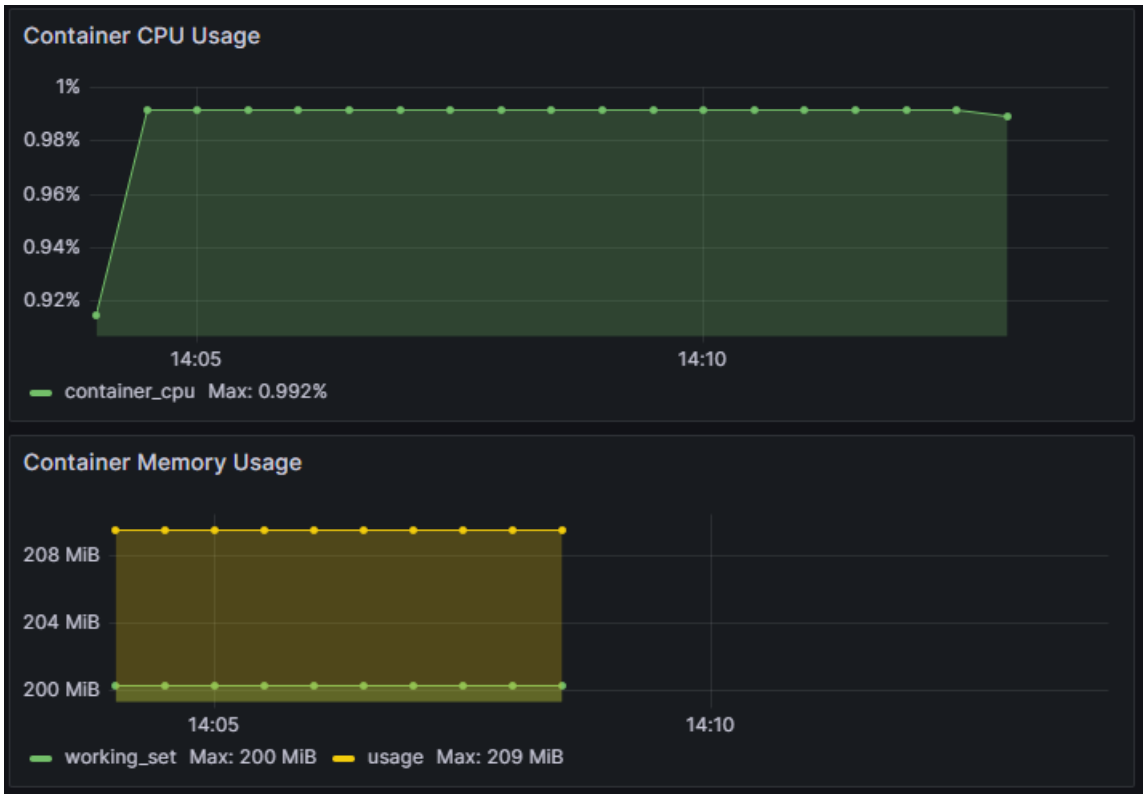
Os resultados referentes as métrica uso de memória RAM e CPU, foram obtidos por meio dos dashboards do Grafana, os quais representam cálculos com base nas séries temporais geradas pelo Prometheus.

4.1.3.1 Primeiro Cenário

4.1.3.1.1 Spring Framework

No primeiro cenário de teste, em que a aplicação apenas realiza o processo de inicialização, tem-se o resultado obtido representado nas figuras 4.1, 4.2 e 4.3. Nas figuras, a label dos valores do eixo *x* está indicando as minutos e segundos, respectivamente..

Figura 4.1 – Uso de CPU pelo container no primerio cenário - Spring Framework



Fonte: Autor

Na figura 4.1, é possível observar no gráfico *Container CPU Usage*, métrica *container_cpu*, que, ao iniciar a aplicação, há um rápido crescimento linear nos primeiros minutos, atingindo um pico de 0,992%. No entanto, logo em seguida, é estabelecido um valor quase constante próximo desse máximo, indicando que este é o consumo de CPU ao inicializar e permanecer em espera de requisições.

No gráfico *Container Memory Usage*, é notável que a memória alocada (métrica *usage*) se estabiliza desde o início em 209 MiB, com a memória em uso (métrica *working_set*) estabilizada em 200 MiB, sendo esse o valor necessário para a aplicação inicializar e permanecer no estado de espera de requisições.

Figura 4.2 – Uso de memória pela JVM no primerio cenário - Spring Framework

Fonte: Autor

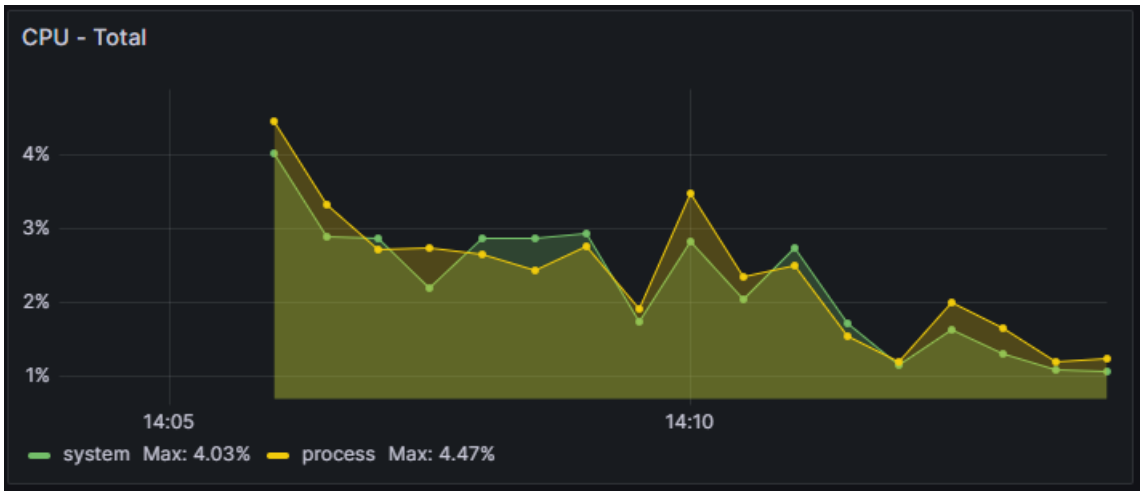
Na figura 4.2, é possível notar que, apesar dos valores serem menores, o padrão é semelhante ao do container, na figura 4.1. No gráfico *Memory RAM - Total*, métrica *sum - committed* (memória alocada), possui um valor praticamente constante de 138 MiB, sendo este o seu máximo. Já para a métrica *sum - used*, tem-se um máximo de 127 MiB, com oscilações, semelhante ao gráfico da memória *Heap*.

Quanto à memória *Heap*, no gráfico *Memory RAM - Heap*, métrica *sum - committed - heap* (memória alocada), possui um valor constante em 48,7 MiB, sendo este o seu máximo.

Já a métrica *sum - used - heap* (memória usada) possui um máximo de 41 MiB, demonstrando uma oscilação, que indica o funcionamento do *Garbage Collection*.

Já à memória Non Heap, no gráfico *Memory RAM - Non Heap*, em ambas as métricas, tem-se um crescimento constante, mas de baixa variação, atingindo um máximo de 89 MiB para a métrica *sum - committed - non heap* (memória alocada) e de 87 MiB para a métrica *sum - used - non heap* (memória usada).

Figura 4.3 – Uso de CPU pela JVM no primerio cenário - Spring Framework



Fonte: Autor

Na figura 4.3, é possível notar que o gráfico atinge seu máximo logo no início, coincidindo com a inicialização da aplicação, um momento de maior processamento. Posteriormente, os valores começam a diminuir gradualmente, estabilizando-se entre 1% e 2%.

É possível notar que as métricas relacionadas com a JVM iniciam após as 14:05, alguns segundos depois das métricas relacionadas com o contêiner. Isso se explica pelo fato do *cAdvisor* estar expondo métricas logo ao iniciar o contêiner, antes mesmo da aplicação finalizar seu processo de inicialização e disponibilizar os endpoints para os *scrapes* do Prometheus. O mesmo ocorre para os resultados do Quarkus e do Micronaut.

A média dos valores obtidos pelos gráficos 4.1, 4.2 e 4.3 está descrita nas tabelas 4.3 e 4.4. Essas médias foram calculadas utilizando a fórmula 3.1, detalhada na seção 3.4.

Tabela 4.3 – Médias do container do primeiro cenário - Spring Framework

Métrica	Tag	Média
Container CPU Usage	-	0,9877%
Container Memory	Working Set Usage	200,0000 MiB 209,0000 MiB

Tabela 4.4 – Médias da JVM do primeiro cenário - Spring Framework

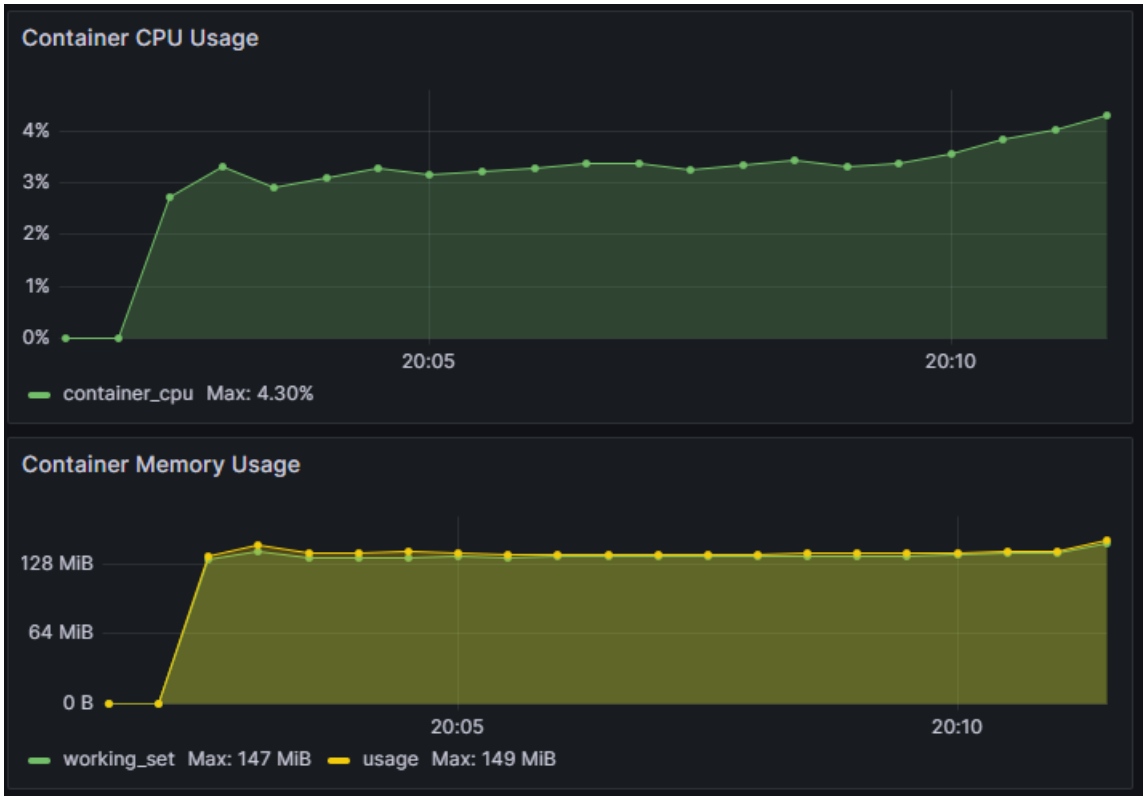
Métrica	Tag	Média
Memory RAM - Total	Used	120,8823 MiB
	Committed	136,8823 MiB
Memory RAM - Heap	Used	34,9176 MiB
	Committed	48,7000 MiB
Memory RAM - Non Heap	Used	86,0705 MiB
	Committed	88,1588 MiB
CPU - Total	System Usage	2,2388%
	Process Usage	2,3700%

As médias das tabelas 4.3 e 4.4 atendem às métricas "Média de uso de CPU"e "Média de uso de memória RAM"definidas no GQM, conforme a seção 3.2, respondendo às perguntas "Qual é a quantidade de CPU usada pelo container para cada framework, assim que é inicializado?"e "Qual é a quantidade de memória RAM usada pelo container para cada framework, assim que é inicializado?".

4.1.3.1.2 Quarkus

No primeiro cenário de teste, em que a aplicação apenas realiza o processo de inicialização, tem-se o resultado obtido representado nas figuras 4.4, 4.5 e 4.6. Nas figuras, a label dos valores do eixo *x* está indicando as minutos e segundos, respectivamente.

Figura 4.4 – Uso de CPU pelo container no primerio cenário - Quarkus

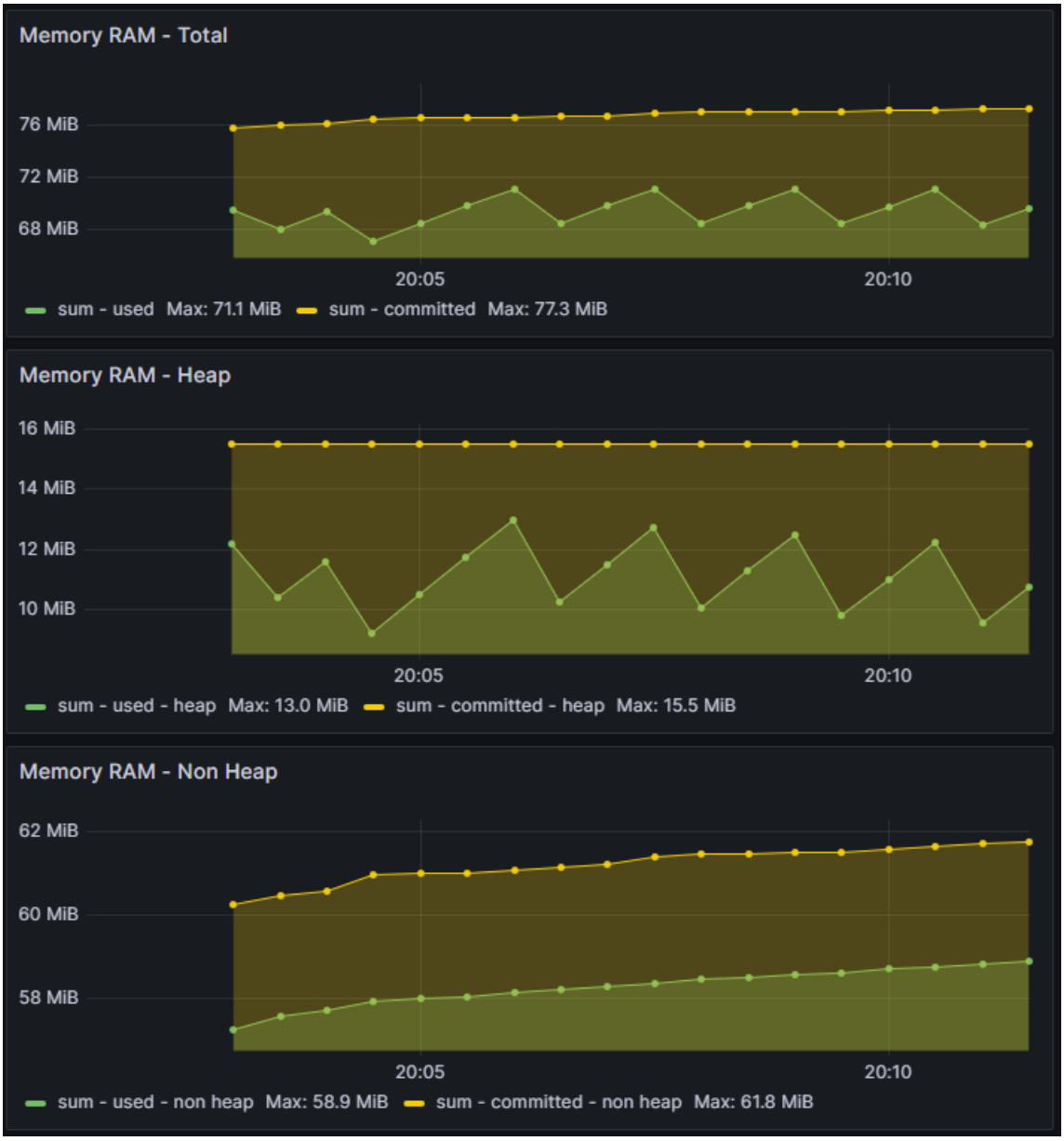


Fonte: Autor

Diferentemente do resultado obtido pelo Spring Framework, na figura 4.1, o uso de CPU do contêiner pelo Quarkus na figura 4.4, no gráfico *Container CPU Usage*, métrica *container_cpu*, é maior e apresenta um desenvolvimento diferente da série temporal. Não mostra um aumento totalmente linear e há algumas oscilações pequenas. Mesmo após atingir os 3%, seus valores permanecem relativamente constantes, com um aumento no final, atingindo seu máximo de 4,3%, sendo maior que o máximo de 0,992% do Spring Framework.

Já no gráfico *Container Memory Usage*, da figura 4.4, os valores são consideravelmente menores em comparação com os observados no Spring Framework, na figura 4.1, com um uso de memória do contêiner constante, em ambas as métricas (*working_set* e *usage*), de aproximadamente 149 MiB, sendo menor que os 209 MiB do Spring Framework.

Figura 4.5 – Uso de memória pela JVM no primerio cenário - Quarkus



Fonte: Autor

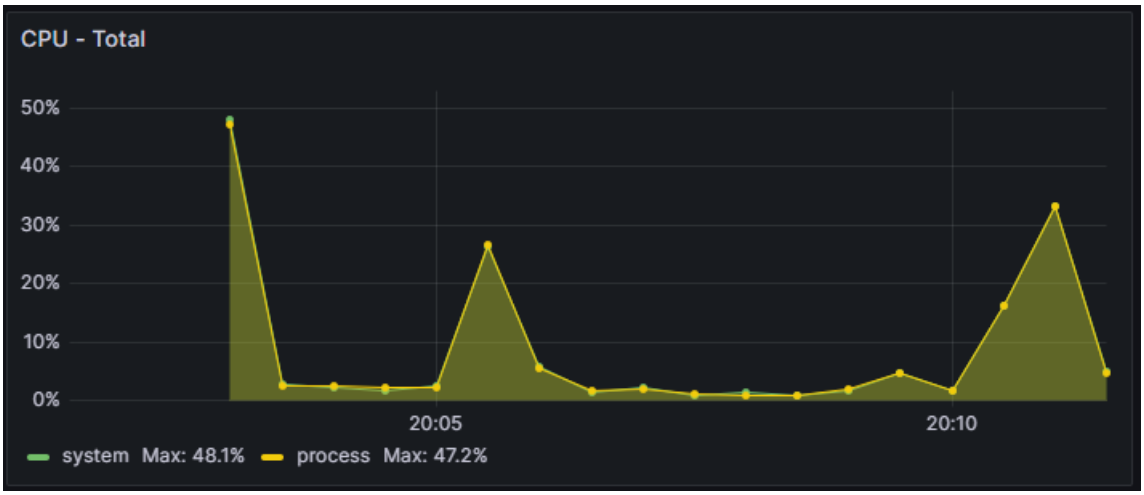
Igualmente ocorre nos resultados do Spring Framework, na figura 4.2, os valores de uso de memória da JVM são menores que os do contêiner na figura 4.4. No entanto, o desenvolvimento da série temporal indica uma maior oscilação da memória usada de fato, tanto no gráfico *Memory RAM - Total*, métrica *sum - used*, como também no gráfico *Memory RAM - Heap*, métrica *sum - used - heap*, sugerindo um uso mais frequente do *Garbage Collection* para limpar a memória em comparação com o Spring Framework.

Outro ponto a se notar é que a proporção de memória *Heap* é menor, sendo de aproximadamente 20% da memória total, em comparação com o Spring Framework, na figura 4.2, que é aproximadamente 35% da memória total, considerando a memória alocada (métricas *sum*

- *committed* e *sum - committed - heap*). Isso indica o menor uso de reflexão que o Quarkus propõe, como visto na seção 2.2.4.

Já o gráfico "Memory RAM - Non Heap"possui um desenvolvimento da série temporal parecido com o do Spring Framework, na figura 4.2, mas também com valores menores.

Figura 4.6 – Uso de CPU pela JVM no primerio cenário - Quarkus



Fonte: Autor

Na figura 4.6, é possível notar que o gráfico atinge seu máximo logo no início, coincidindo com a inicialização da aplicação, um momento de maior processamento, com um valor de 48,1%. No entanto, logo é reduzido significativamente para valores abaixo de 10%, com duas oscilações maiores acima de 20%.

Diferente do Spring Framework, os valores atingiram máximos consideravelmente maiores.

A média dos valores obtidos pelos gráficos 4.4, 4.5 e 4.6 está descrita nas tabelas 4.5 e 4.6. Essas médias foram calculadas utilizando a fórmula 3.1, detalhada na seção 3.4.

Tabela 4.5 – Médias do container do primeiro cenário - Quarkus

Métrica	Tag	Média
Container CPU Usage	-	3.0552%
Container Memory	Working Set Usage	122.7916 MiB 125.1744 MiB

Tabela 4.6 – Médias da JVM do primeiro cenário - Quarkus

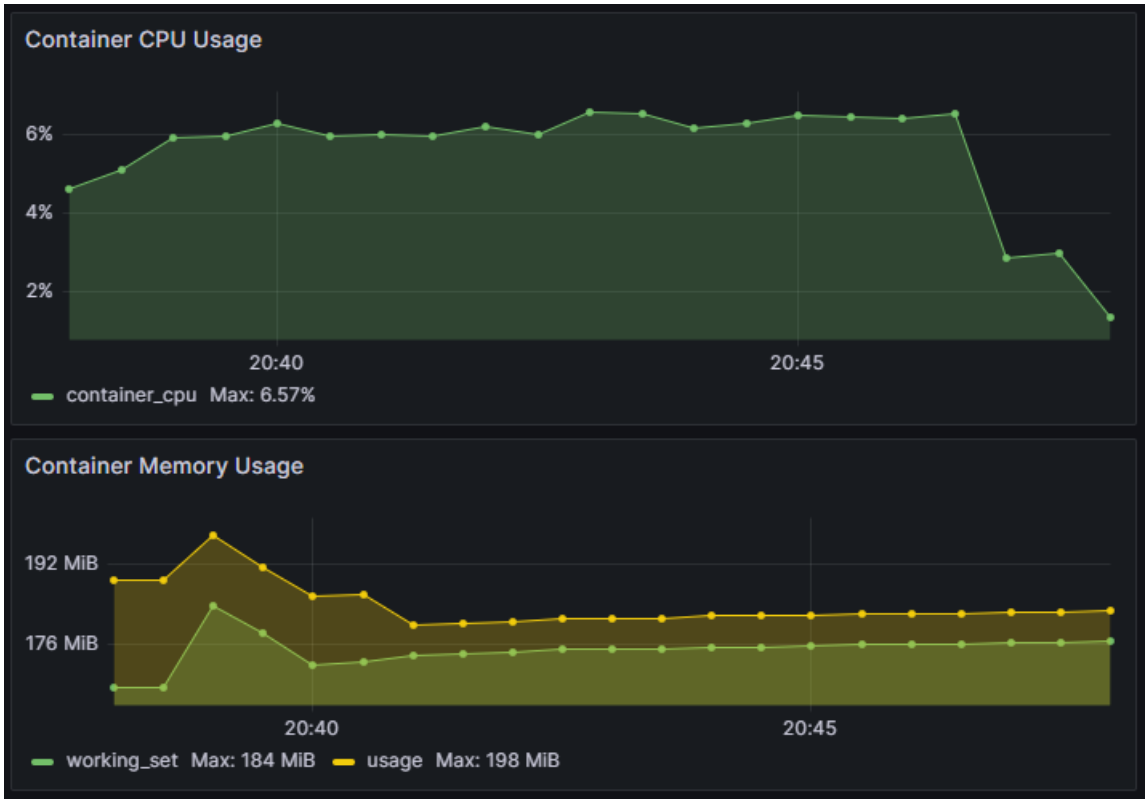
Métrica	Tag	Média
Memory RAM - Total	Used	69.3722 MiB
	Committed	69.3722 MiB
Memory RAM - Heap	Used	11.1161 MiB
	Committed	15.5000 MiB
Memory RAM - Non Heap	Used	58.2666 MiB
	Committed	61.1944 MiB
CPU - Total	System Usage	8.7916%
	Process Usage	8.7387%

As médias das tabelas 4.5 e 4.6 atendem às métricas "Média de uso de CPU"e "Média de uso de memória RAM"definidas no GQM, conforme a seção 3.2, respondendo às perguntas "Qual é a quantidade de CPU usada pelo container para cada framework, assim que é inicializado?"e "Qual é a quantidade de memória RAM usada pelo container para cada framework, assim que é inicializado?".

4.1.3.1.3 Micronaut

No primeiro cenário de teste, em que a aplicação apenas realiza o processo de inicialização, tem-se o resultado obtido representado nas figuras 4.7, 4.8 e 4.9. Nas figuras, a label dos valores do eixo *x* está indicando as minutos e segundos, respectivamente.

Figura 4.7 – Uso de CPU pelo container no primerio cenário - Micronaut



Fonte: Autor

Diferentemente do resultado obtido pelo Spring Framework, na figura 4.1, e do resultado obtido pelo Quarkus, na figura 4.7, o uso de CPU do contêiner pelo Micronaut na figura 4.7, no gráfico *Container CPU Usage*, métrica *container_cpu*, é ainda maior e apresenta um desenvolvimento diferente da série temporal. Já inicia com valores acima de 4%, atingindo uma maxima de 6,7%, demonstrando pequenas ocilações, mas com valores proximos a 6%, demonstrando assim valores maiores que ambos os *frameworks*. Os valores reduzem significamente ao final, cainda para menos de 2%.

Já no gráfico *Container Memory Usage*, da figura 4.7, os valores são um pouco menores em comparação com os observados no Spring Framework, na figura 4.1, com um uso de memória do contêiner em grande parte constante, em ambas as métricas (*working_set* e *usage*), de aproximadamente 198 MiB, sendo menor que os 209 MiB do Spring Framework. Em contrapartida, fica consideravelmente atrás do Quarkus.

Figura 4.8 – Uso de memória pela JVM no primerio cenário - Micronaut

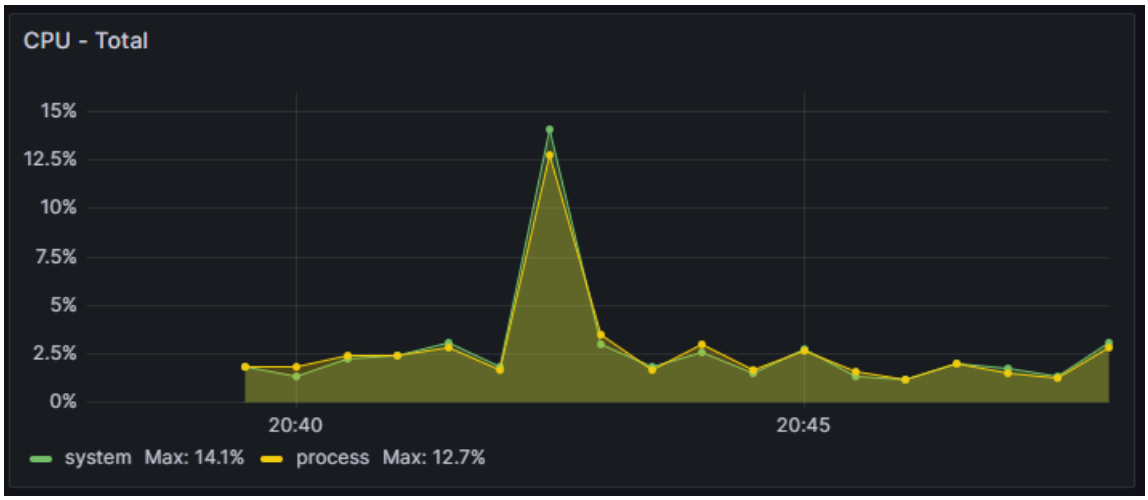
Fonte: Autor

Igualmente ocorre nos resultados do Spring Framework, na figura 4.2, onde os valores de uso de memória da JVM são menores que os do contêiner na figura 4.4. No entanto, a análise da série temporal indica uma maior oscilação da memória usada, tanto no gráfico *Memory RAM - Total*, na métrica *sum - used*, como também no gráfico *Memory RAM - Heap*, na métrica *sum - used - heap*, sugerindo um uso mais frequente do *Garbage Collection* para limpar a memória em comparação com o Spring Framework, mas com uma oscilação menor que o Quarkus.

Outro ponto a se notar é que, diferentemente do Quarkus na figura 4.5, a proporção da memória *Heap* em relação à memória *Total* é semelhante ao Spring Framework, sendo aproximadamente 33% para o Micronaut.

Quanto ao gráfico "*Memory RAM - Non Heap*", ele possui um desenvolvimento da série temporal parecido com o do Spring Framework na figura 4.2 e do Quarkus na figura 4.5. No entanto, os valores do Micronaut são ligeiramente menores do que os do Spring Framework e consideravelmente maiores do que os do Quarkus.

Figura 4.9 – Uso de CPU pela JVM no primerio cenário - Micronaut



Fonte: Autor

Na figura 4.6, é possível notar que os valores oscilam, mas se mantêm próximos a 2.5%, com um único pico de 14,1%.

Ao contrário do Quarkus, os valores são bem menores, uma vez que o Quarkus apresentou mais oscilações grandes. Em comparação com o Spring Framework, o uso de CPU é inferior.

A média dos valores obtidos pelos gráficos 4.7, 4.8 e 4.9 está descrita nas tabelas 4.7 e 4.8. Essas médias foram calculadas utilizando a fórmula 3.1, detalhada na seção 3.4.

Tabela 4.7 – Médias do container do primeiro cenário - Micronaut

Métrica	Tag	Média
Container CPU Usage	-	5.5533%
Container Memory	Working Set Usage	174.9523 MiB 183.9047 MiB

Tabela 4.8 – Médias da JVM do primeiro cenário - Micronaut

Métrica	Tag	Média
Memory RAM - Total	Used	99.3777 MiB
	Committed	120.3333 MiB
Memory RAM - Heap	Used	21.3333 MiB
	Committed	39.8999 MiB
Memory RAM - Non Heap	Used	78.0500 MiB
	Committed	80.3555 MiB
CPU - Total	System Usage	2.7238%
	Process Usage	2.6861%

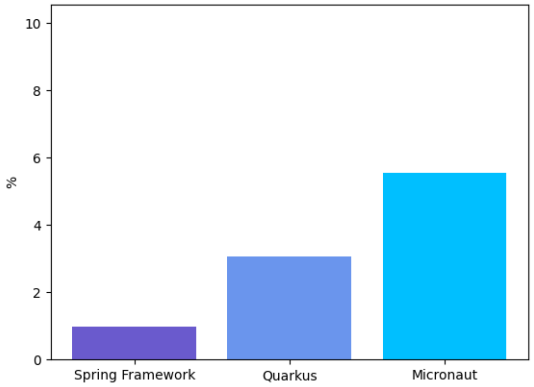
As médias das tabelas 4.7 e 4.8 atendem às métricas "Média de uso de CPU"e "Média de uso de memória RAM"definidas no GQM, conforme a seção 3.2, respondendo às perguntas "Qual é a quantidade de CPU usada pelo container para cada framework, assim que é inicializado?"e "Qual é a quantidade de memória RAM usada pelo container para cada framework, assim que é inicializado?".

4.1.3.1.4 Comparativo geral do primeiro cenário

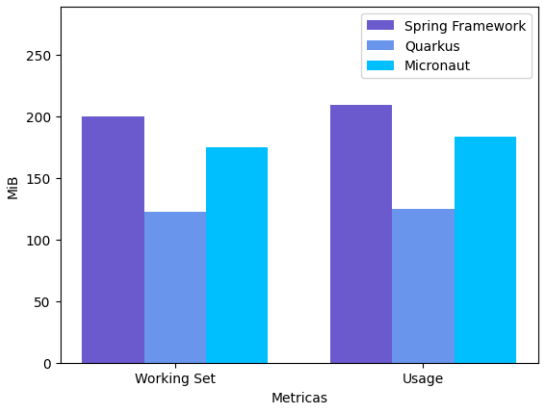
A figura 4.10 tem como objetivo expor graficamente as diferenças entre as médias obtidas para cada métrica em cada *framework*, no primeiro cenário. No primeiro gráfico, encontra-se o *Container CPU Usage*, demonstrando as médias obtidas para cada *framework*, onde o eixo x representa os *frameworks* e o eixo y mostra a porcentagem de uso. Nos demais gráficos, o eixo x representa as métricas obtidas, sendo que cada *framework* é representado por uma barra de cor específica, conforme descrito na legenda de cada gráfico.

Figura 4.10 – Comparativo das médias no primeiro cenário

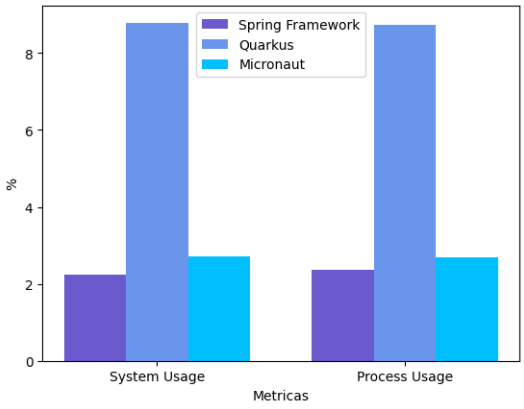
((a)) Container CPU Usage



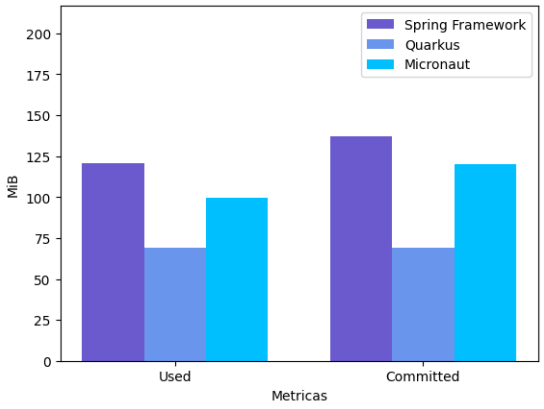
((b)) Container Memory



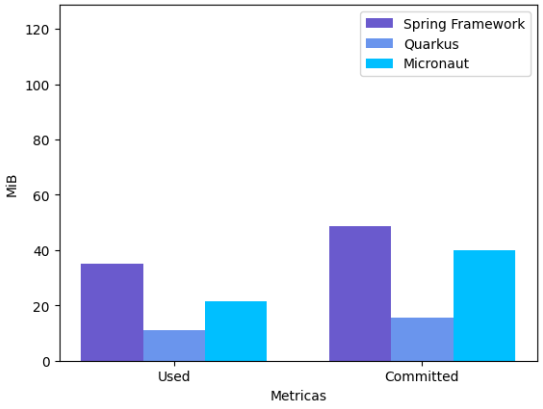
((c)) JVM - CPU Total



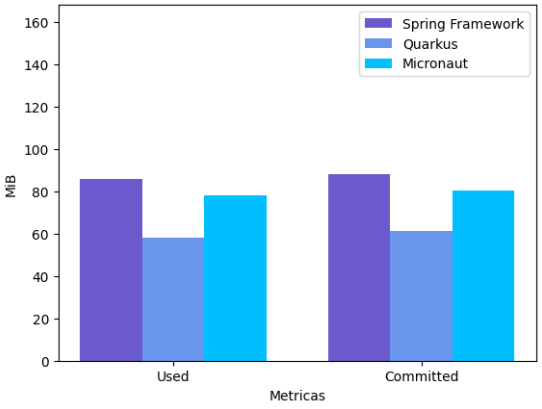
((d)) JVM - Memory RAM Total



((e)) JVM - Memory RAM Heap



((f)) JVM - Memory RAM Non Heap



Fonte: Autor

Como é possível observar na figura 4.10(a), com relação à métrica *Container CPU Usage*, o uso é menor com o Spring Framework e maior com o Micronaut, apresentando uma diferença de 462.25%. O Quarkus fica entre os dois, com uma diferença de 209.32% em comparação com o Spring Framework. Já em relação às métricas *System Usage* e *Process Usage*

da JVM, na figura 4.10(c), o Quarkus apresenta um valor significativamente maior de uso de CPU, com uma diferença superior a 200% em relação aos demais *frameworks*. Por outro lado, o Spring Framework e o Micronaut apresentam uma diferença pequena de aproximadamente 13%.

Quanto às métricas do *Container Memory*, na figura 4.10(b), o Spring Framework possui o maior uso de memória, enquanto o Quarkus apresenta o menor uso, com uma diferença de 62.88% para o *Working Set* e 66.97% para o *Usage*. Em relação ao Micronaut, também há um uso maior de memória, embora seja menor que o Spring Framework. A diferença entre o Micronaut e o Quarkus é de 42.48% para o *Working Set* e 46.92% para o *Usage*.

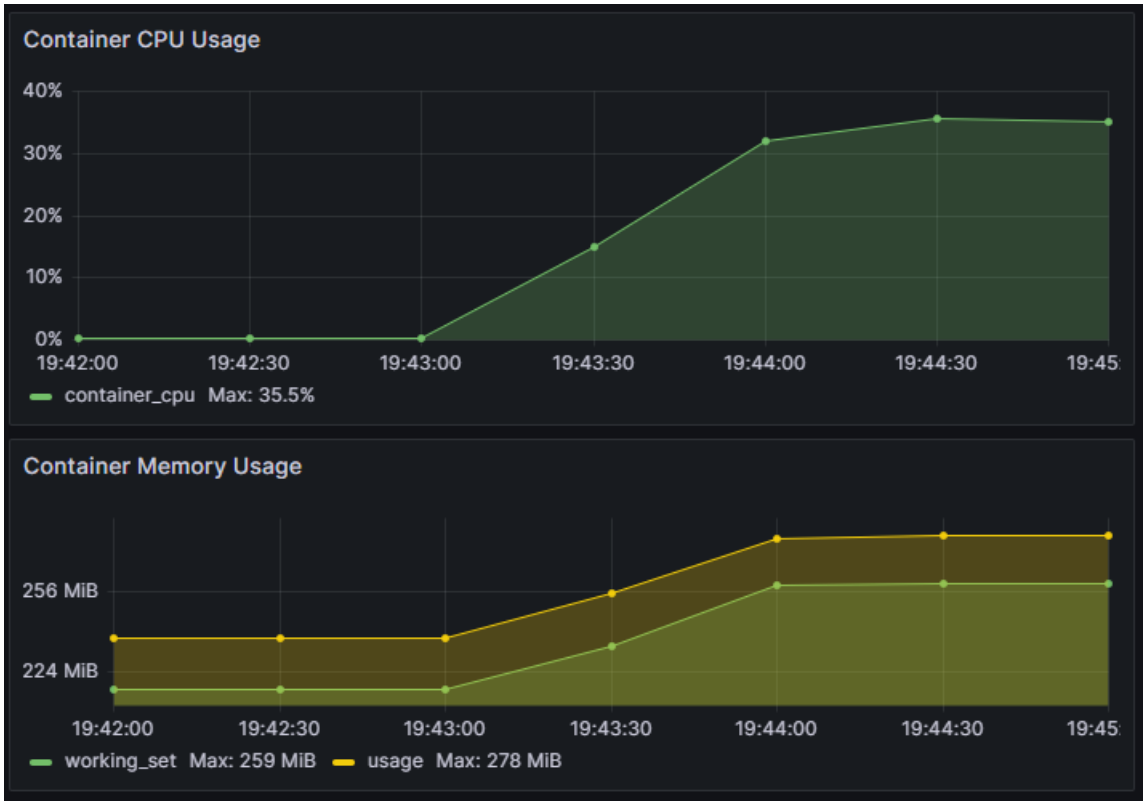
Já em relação às métricas de *JVM - Memory RAM*, nas figuras 4.10(d), 4.10(e) e 4.10(f), a situação é semelhante à métrica anterior. O Spring Framework apresenta o maior uso de memória, o Quarkus o menor uso e o Micronaut fica entre ambos, com uma diferença menor em relação ao Spring Framework.

4.1.3.2 Segundo Cenário

4.1.3.2.1 Spring Framework

Já no segundo cenário, onde é realizado o teste de carga, tem-se o resultado obtido representado nas figuras 4.11, 4.12 e 4.13. Nas figuras, a label dos valores do eixo x está indicando as horas, minutos e segundos, respectivamente.

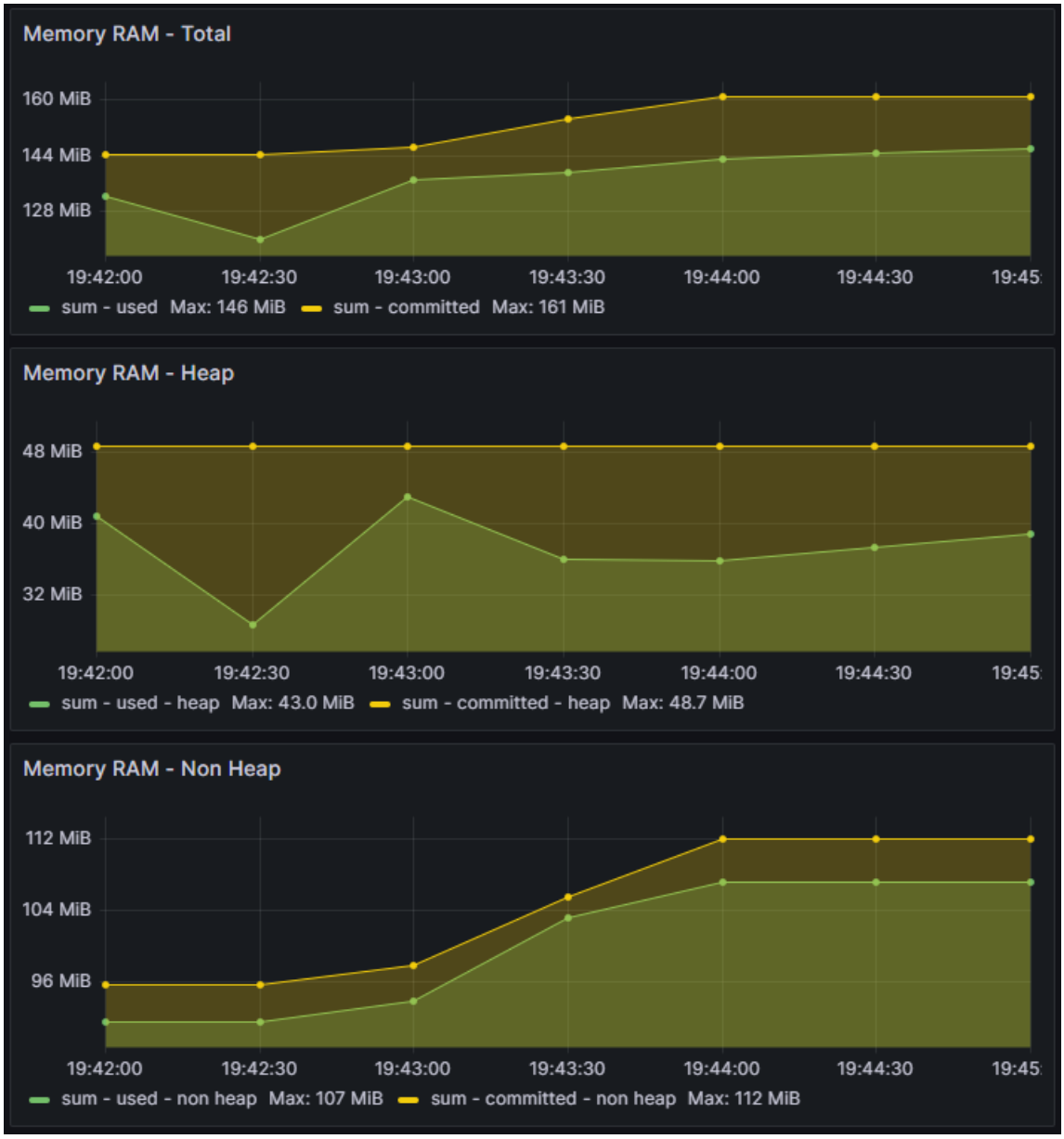
Figura 4.11 – Uso de CPU pelo container no segundo cenário - Spring Framework



Fonte: Autor

Na figura 4.11, é evidente um padrão em ambos os gráficos, demonstrando o aumento do uso de CPU e de memória RAM que o teste de carga causou no intervalo de 19:43 a 19:44. No gráfico *Container CPU Usage*, observa-se que o uso de CPU aumentou de aproximadamente 0% para mais de 30%, atingindo o máximo de 35,5%. Já no gráfico *Container Memory Usage*, o aumento do uso de memória RAM atinge o máximo de 278 MiB. O Kubernetes mantém esses valores alocados por um tempo, mesmo após as requisições terem sido finalizadas.

Figura 4.12 – Uso de memória pela JVM no segundo cenário - Spring Framework



Fonte: Autor

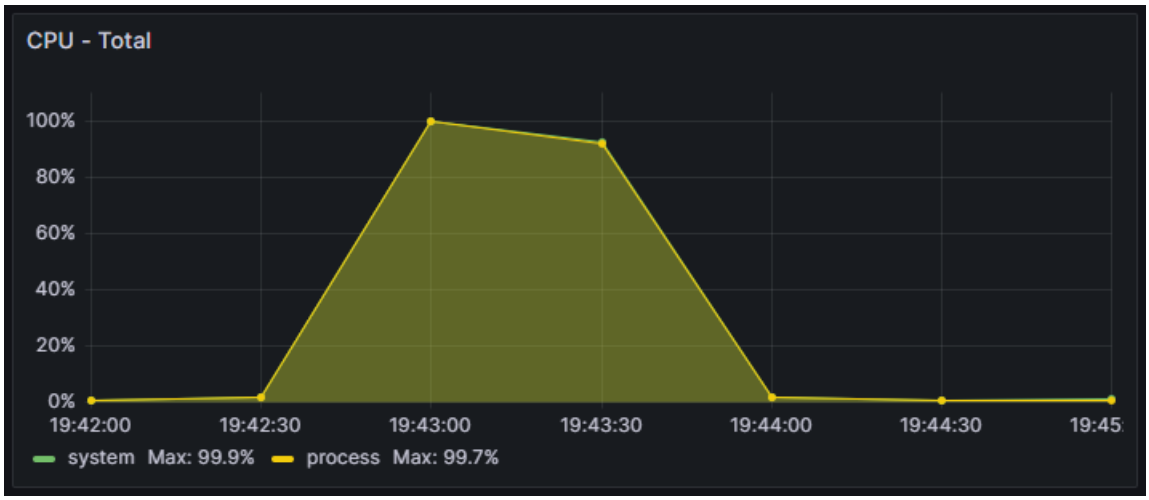
Na figura 4.12, é possível notar que, apesar dos valores serem menores, o padrão é semelhante ao do contêiner na figura 4.1. No gráfico *Memory RAM - Total*, a métrica *sum - committed* (memória alocada) apresenta um crescimento de aproximadamente 144 MiB até atingir cerca de 160 MiB, sendo o máximo atingido de 161 MiB. Já para a métrica *sum - used*, observa-se uma queda no início, mas que rapidamente é seguida por um crescimento linear, alcançando o máximo de 146 MiB.

Quanto especificamente à memória *Heap*, no gráfico *Memory RAM - Heap*, a métrica *sum - committed - heap* (memória alocada) mantém um valor constante em 48,7 MiB, representando seu máximo. Já a métrica *sum - used - heap* (memória usada) atinge um máximo de

43 MiB, exibindo uma oscilação no início que sugere o funcionamento do *Garbage Collection*. Em seguida, observa-se um crescimento contínuo, impulsionado pelo teste.

Quanto à memória Non Heap, no gráfico *Memory RAM - Non Heap*, em ambas as métricas, há um crescimento rápido no início do teste, seguido por uma manutenção de valor constante. Seu crescimento atinge um máximo de 107 MiB na métrica *sum - used - non heap* (memória usada) e 112 MiB na métrica *sum - committed - non heap* (memória alocada).

Figura 4.13 – Uso de CPU pela JVM no segundo cenário - Spring Framework



Fonte: Autor

Na figura 4.13, fica evidente o aumento do uso de CPU, atingindo seu máximo de 99,9%, demonstrando que a aplicação requisitou o máximo de processamento disponível. Isso é corroborado pela figura 4.11, onde é observado que o container disponibilizou mais recurso de processamento logo em seguida.

A média dos valores obtidos pelos gráficos 4.11, 4.12 e 4.13 está descrita nas tabelas 4.9 e 4.10. Essas médias foram calculadas utilizando a fórmula 3.1, detalhada na seção 3.4.

Tabela 4.9 – Médias do container do segundo cenário - Spring Framework

Métrica	Tag	Média
Container CPU Usage	-	16,9340%
Container Memory	Working Set	237,2857 MiB
	Usage	257,0000 MiB

Tabela 4.10 – Médias da JVM do segundo cenário - Spring Framework

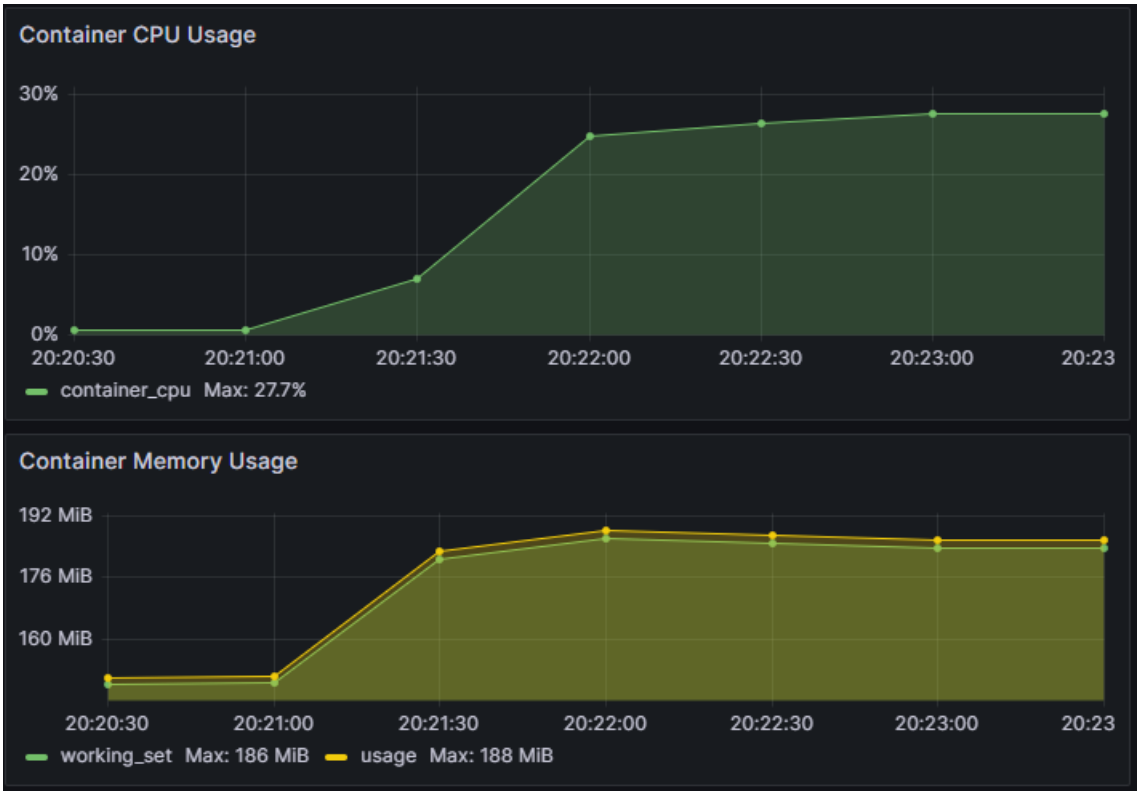
Métrica	Tag	Média
Memory RAM - Total	Used	137,2857 MiB
	Committed	153,1428 MiB
Memory RAM - Heap	Used	37,2714 MiB
	Committed	48.6999 MiB
Memory RAM - Non Heap	Used	100,1285 MiB
	Committed	104,4571 MiB
CPU - Total	System Usage	28,2801%
	Process Usage	28,1694%

As médias das tabelas 4.9 e 4.10 atendem às métricas "Média de uso de CPU"e "Média de uso de memória RAM"definidas no GQM, conforme a seção 3.2, respondendo às perguntas "Qual é a quantidade de CPU usada do container, por cada framework, quando recebe uma alta carga de requisições?"e "Qual é a quantidade de memória RAM usada pelo container para cada framework, assim que é inicializado?".

4.1.3.2.2 Quarkus

Já no segundo cenário, onde é realizado o teste de carga, tem-se o resultado obtido representado nas figuras 4.14, 4.15 e 4.16. Nas figuras, a label dos valores do eixo *x* está indicando as horas, minutos e segundos, respectivamente.

Figura 4.14 – Uso de CPU pelo container no segundo cenário - Quarkus



Fonte: Autor

Na figura 4.14, é evidente um padrão em ambos os gráficos, demonstrando o aumento do uso de CPU e de memória RAM que o teste de carga causou no intervalo de 60 segundos, das 20:21:00 às 20:22:00. No gráfico *Container CPU Usage*, observa-se que o uso de CPU aumentou de aproximadamente 0% para quase 30%, atingindo o máximo de 27,7%. Já no gráfico *Container Memory Usage*, o aumento do uso de memória RAM atinge o máximo de 188 MIB. Apesar de uma leve redução após o pico no momento 20:22:00, o Kubernetes mantém boa parte dos valores alocados por um tempo, mesmo após as requisições terem sido finalizadas.

Diferente do Spring Framework, os máximos atingidos para o uso de CPU foram menores, abaixo de 30%, diferente dos quase 40% do Spring Framework. Igualmente ao uso de CPU, o uso de memória também teve valores menores, com uma diferença significativa, atingindo um máximo de 188 MiB, diferente dos 278 MiB do Spring Framework.

Figura 4.15 – Uso de memória pela JVM no segundo cenário - Quarkus

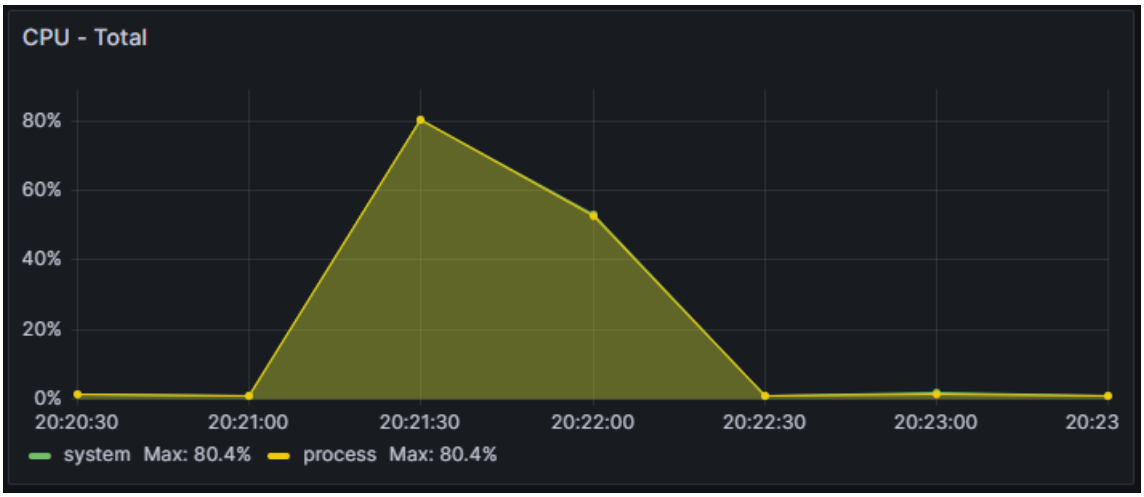
Fonte: Autor

Na figura 4.15, é possível notar que, apesar dos valores serem menores, o padrão é semelhante ao container, na figura 4.14. A memória RAM total (*Memory RAM - Total*) alocada (*sum - used*) atinge um máximo de 161 MiB. Quanto à memória Heap, é notável o seu crescimento na memória alocada (*sum - committed - heap*) e algumas oscilações na memória usada (*sum - used - heap*), com um máximo de 21,5 MiB alocada. Por fim, quanto à memória Non Heap, os valores da memória alocada iniciam abaixo de 64 MiB e crescem até 83 MiB, com uma diferença pequena para a memória usada.

Igualmente ao uso de memória do container, os valores obtidos pela JVM são menores em comparação ao Spring Framework, também obtidos pela JVM, o qual teve um máximo de

161 MiB de memória RAM total alocada, contra os 105 MiB de memória RAM total alocada do Quarkus.

Figura 4.16 – Uso de CPU pela JVM no segundo cenário - Quarkus



Fonte: Autor

Na figura 4.16, fica evidente o aumento do uso de CPU, assim como no Spring Framework, no entanto, com um máximo menor de 80,4%.

A média dos valores obtidos pelos gráficos 4.14, 4.15 e 4.16 está descrita nas tabelas 4.11 e 4.12. Essas médias foram calculadas utilizando a fórmula 3.1, detalhada na seção 3.4.

Tabela 4.11 – Médias do container do segundo cenário - Quarkus

Métrica	Tag	Média
Container CPU Usage	-	16.3789%
Container Memory	Working Set Usage	173.4285 MiB 175.7142 MiB

Tabela 4.12 – Médias da JVM do segundo cenário - Quarkus

Métrica	Tag	Média
Memory RAM - Total	Used Committed	88.7000 MiB 96.2571 MiB
Memory RAM - Heap	Used Committed	15.1428 MiB 19.7857 MiB
Memory RAM - Non Heap	Used Committed	73.5428 MiB 76.2857 MiB
CPU - Total	System Usage Process Usage	19.8474% 19.8722%

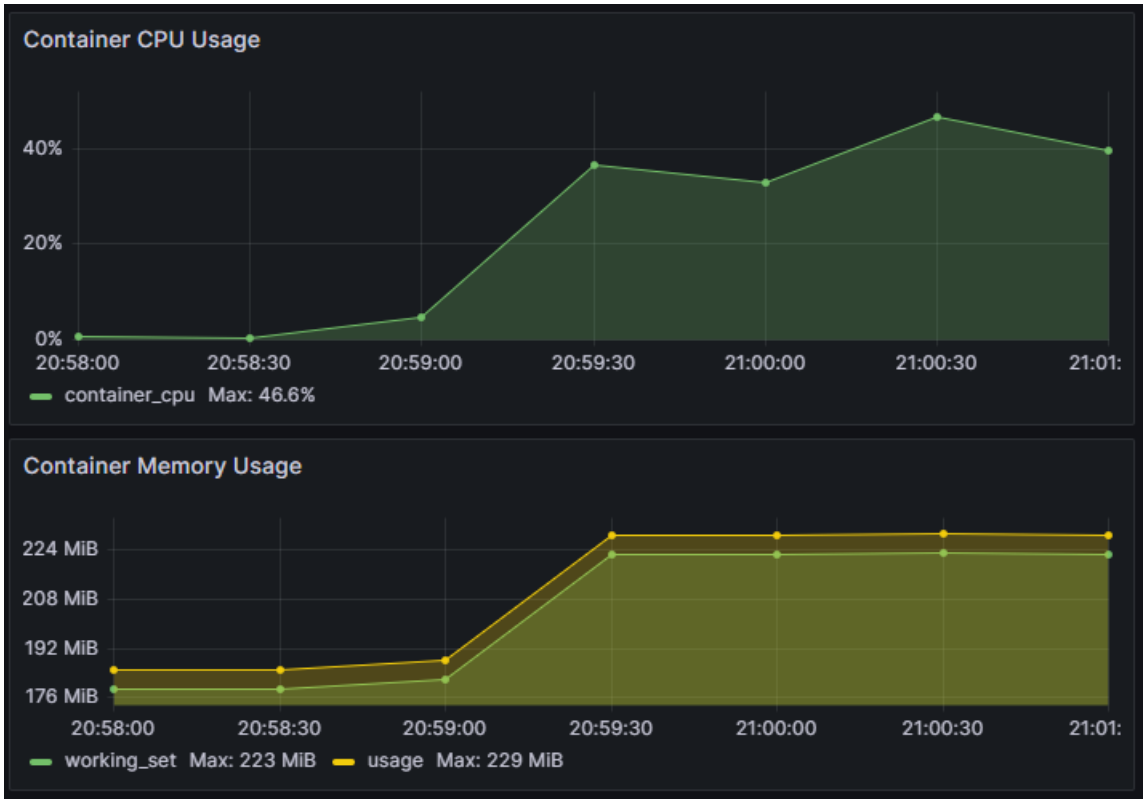
As médias das tabelas 4.11 e 4.12 atendem às métricas "Média de uso de CPU"e "Média de uso de memória RAM"definidas no GQM, conforme a seção 3.2, respondendo às perguntas

"Qual é a quantidade de CPU usada do container, por cada framework, quando recebe uma alta carga de requisições?"e "Qual é a quantidade de memória RAM usada pelo container para cada framework, assim que é inicializado?".

4.1.3.2.3 Micronaut

Já no segundo cenário, onde é realizado o teste de carga, tem-se o resultado obtido representado nas figuras 4.17, 4.18 e 4.19. Nas figuras, a label dos valores do eixo *x* está indicando as horas, minutos e segundos, respectivamente.

Figura 4.17 – Uso de CPU pelo container no segundo cenário - Micronaut



Fonte: Autor

Na figura 4.17, é evidente um padrão em ambos os gráficos, demonstrando o aumento do uso de CPU e de memória RAM causado pelo teste de carga no intervalo de 60 segundos, das 20:59:00 às 21:00:00. No gráfico *Container CPU Usage*, observa-se que o uso de CPU aumentou de aproximadamente 0% para mais de 40%, atingindo o máximo de 46,6%. Já no gráfico *Container Memory Usage*, o aumento do uso de memória RAM atingiu o máximo de 229 MiB. O Kubernetes mantém boa parte dos valores alocados por um tempo, mesmo após as requisições terem sido finalizadas, igualment aos demais frameworks.

Os valores atingidos foram semelhantes e até maiores do que os obtidos com o Spring Framework, estando assim significativamente maiores que os do Quarkus.

Figura 4.18 – Uso de memória pela JVM no segundo cenário - Micronaut

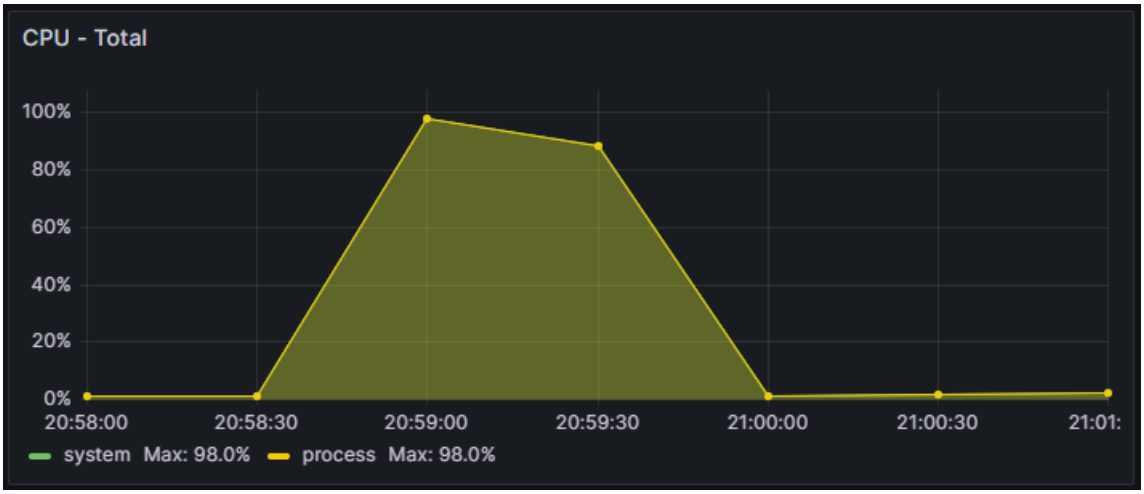
Fonte: Autor

Na figura 4.18, é possível notar que, apesar dos valores serem menores, o padrão é semelhante ao do container na figura 4.17. A memória RAM total (*Memory RAM - Total*) alocada (*sum - used*) atinge um máximo de 123 MiB. Quanto à memória Heap, é notável que a memória alocada (*sum - committed - heap*) se manteve constante em 39,9 MiB. Já na memória usada (*sum - used - heap*), há poucas oscilações, com um máximo de 27,1 MiB. Por fim, quanto

à memória Non Heap, os valores da memória alocada iniciam próximos a 80 MiB e crescem até 102 MiB, com uma diferença pequena para a memória usada.

Assim como o uso de memória do container, os valores obtidos pela JVM são semelhantes e até maiores aos obtidos pelo Spring Framework, também executado pela JVM, e consideravelmente maiores do que os obtidos pelo Quarkus.

Figura 4.19 – Uso de CPU pela JVM no segundo cenário - Micronaut



Fonte: Autor

Na figura 4.19, fica evidente o aumento do uso de CPU, assim como no Spring Framework e no Quarkus; no entanto, atinge um máximo de 80,4%, sendo ligeiramente menor que o Spring Framework, mas consideravelmente maior do que o do Quarkus.

A média dos valores obtidos pelos gráficos 4.17, 4.18 e 4.19 está descrita nas tabelas 4.13 e 4.14. Essas médias foram calculadas utilizando a fórmula 3.1, detalhada na seção 3.4.

Tabela 4.13 – Médias do container do segundo cenário - Micronaut

Métrica	Tag	Média
Container CPU Usage	-	22.7997%
Container Memory	Working Set	204.5714 MiB
	Usage	210.5714 MiB

Tabela 4.14 – Médias da JVM do segundo cenário - Micronaut

Métrica	Tag	Média
Memory RAM - Total	Used	115.7142 MiB
	Committed	115.7142 MiB
Memory RAM - Heap	Used	22.7428 MiB
	Committed	39.9000 MiB
Memory RAM - Non Heap	Used	93.1571 MiB
	Committed	95.6142 MiB
CPU - Total	System Usage	27.6152%
	Process Usage	27.5917%

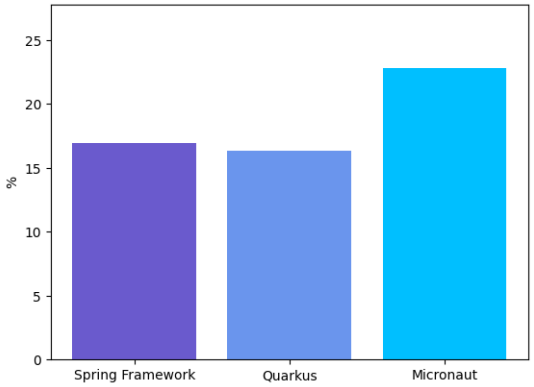
As médias das tabelas 4.13 e 4.14 atendem às métricas "Média de uso de CPU"e "Média de uso de memória RAM"definidas no GQM, conforme a seção 3.2, respondendo às perguntas "Qual é a quantidade de CPU usada do container, por cada framework, quando recebe uma alta carga de requisições?"e "Qual é a quantidade de memória RAM usada pelo container para cada framework, assim que é inicializado?".

4.1.3.2.4 Comparativo geral do segundo cenário

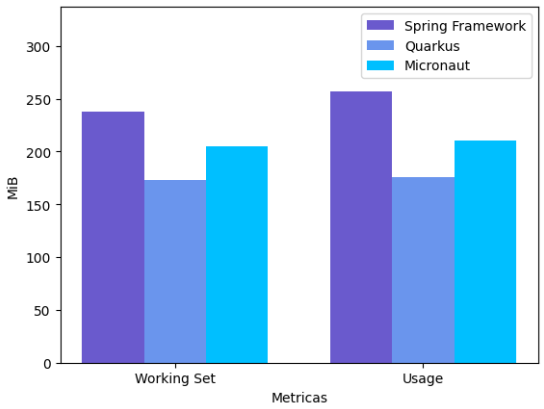
A figura 4.20 tem o mesmo objetivo que a figura 4.10 citada anteriormente, mas para o segundo cenário.

Figura 4.20 – Comparativo das médias no segundo cenário

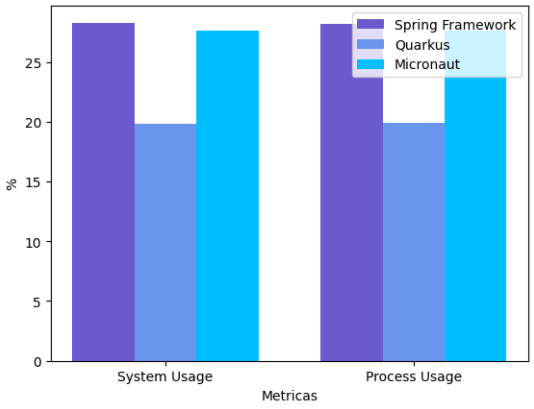
((a)) Container CPU Usage



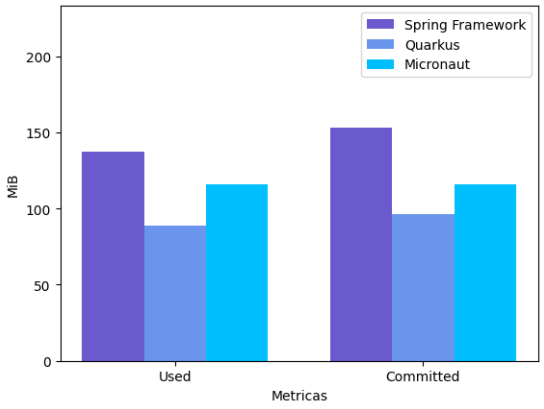
((b)) Container Memory



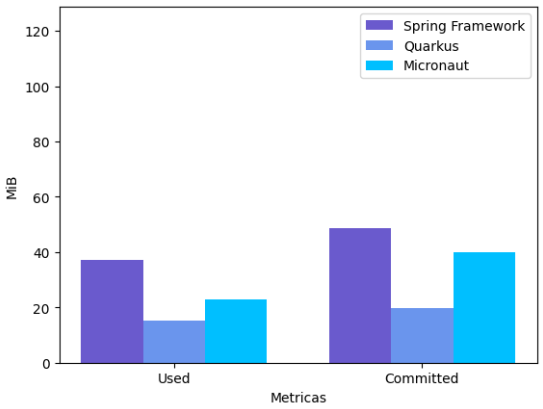
((c)) JVM - CPU Total



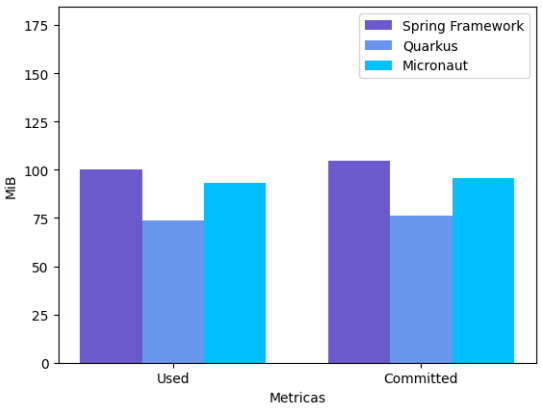
((d)) JVM - Memory RAM Total



((e)) JVM - Memory RAM Heap



((f)) JVM - Memory RAM Non Heap



Fonte: Autor

Como é possível observar na figura 4.20(a), com relação à métrica *Container CPU Usage*, o uso é menor com o Quarkus, com uma pequena diferença para o Spring Framework de 3.39%. Já o Micronaut possui o maior uso, com uma diferença de 39.20% em relação ao Quarkus e de 34.64% em relação ao Spring Framework. No entanto, em relação às métricas

System Usage e *Process Usage* da JVM, na figura 4.20(c), o Spring Framework possui o maior uso de CPU, aproximadamente 42% em relação ao Quarkus, mas próximo ao Micronaut, com uma diferença de aproximadamente 2%.

Quanto às métricas do *Container Memory*, na figura 4.20(b), o Spring Framework possui o maior uso de memória, enquanto o Quarkus apresenta o menor uso, com uma diferença de 36.82% para o *Working Set* e 46.26% para o *Usage*. Em relação ao Micronaut, também há um uso maior de memória, embora seja menor que o Spring Framework. A diferença entre o Micronaut e o Quarkus é de 17.96% para o *Working Set* e 19.84% para o *Usage*.

Já em relação às métricas de *JVM - Memory RAM*, nas figuras 4.20(d), 4.20(e) e 4.20(f), a situação é semelhante à métrica anterior. O Spring Framework apresenta o maior uso de memória, o Quarkus o menor uso e o Micronaut fica entre ambos, com uma diferença menor em relação ao Spring Framework.

5 CONSIDERAÇÕES FINAIS

O presente trabalho teve como objetivo realizar uma análise comparativa entre os *frameworks* Spring Framework, Quarkus e Micronaut no contexto de Kubernetes. Durante a pesquisa e desenvolvimento do projeto, foram realizadas etapas de revisão literária e técnica, definição da arquitetura para a aplicação de teste, configuração de um cluster Kubernetes, instalação da ferramenta Prometheus e Grafana no cluster, além da análise de métricas por meio de testes entre os *frameworks* em cenários distintos.

5.1 Comparativo geral

Com base nos resultados obtidos, é possível indicar o *framework* Quarkus como mais indicado para a implementação de projetos que utilizem Docker e Kubernetes, considerando seu menor uso de CPU - em um dos cenários - e memória RAM pelo container, tempo de inicialização e tamanho da imagem Docker.

No entanto, o *framework* Micronaut obteve bons resultados, tornando-se uma opção próxima ao Quarkus. Caso a escolha leve em consideração a facilidade de implementação e a curva de aprendizado, o *framework* Spring Framework também se torna uma boa escolha.

5.2 Principais desafios enfrentados

Durante a realização deste trabalho, enfrentamos alguns desafios significativos. Um deles foi a implementação da aplicação de teste com o *framework* Micronaut, especialmente no gerenciamento de dependências, onde ocorreram conflitos em mais de uma ocasião. As mudanças nos domínios de algumas dependências entre versões do Micronaut geraram problemas ao tentar construir uma aplicação do zero, sem utilizar o inicializador oficial[1].

Outro desafio foi a configuração do cluster Kubernetes com o Prometheus e o Grafana, o que pode ser mais desafiador quando não se possui um conhecimento aprofundado da ferramenta Kubernetes.

Outro desafio foi encontrar exemplos de implementações da Arquitetura Hexagonal no contexto de Quarkus e Micronaut. Isso confere valor ao código-fonte desenvolvido neste trabalho.

[1] Disponível em: <https://micronaut.io/launch/>. Acesso em 6 de Mar. 2023

5.3 Trabalhos futuros

Como recomendações para trabalhos futuros, sugere-se a realização de análises comparativas com novas métricas, incluindo aspectos qualitativos como facilidade de implementação, curva de aprendizado, qualidade da documentação e estabilidade das versões. Além disso, considerar implementações usando imagens "native"com a GraalVM pode potencialmente oferecer melhores resultados para ambos os *frameworks*. Outra sugestão é analisar o tempo de escalabilidade de cada *framework* no cluster Kubernetes e verificar se há uma influência direta no tempo de inicialização da aplicação.

REFERÊNCIAS

BRAZIL, B. **Prometheus: Up & Running: Infrastructure and Application Performance Monitoring**. [S.l.]: "O'Reilly Media, Inc.", 2018.

BUENO, A. S.; PORTER, J. **Quarkus Cookbook**. [S.l.]: O'Reilly Media, 2020.

CALDIERA, V. R. B. G.; ROMBACH, H. D. The goal question metric approach. **Encyclopedia of software engineering**, p. 528–532, 1994.

COCKBURN, A. Hexagonal architecture. **https://alistair.cockburn.us/hexagonal-architecture/. html. Ultimo acesso em: 07 de nov. de 2023**, 2005.

DEITEL, H. D. P. **Java: como programar**. [S.l.]: Pearson Universidades, 2017. ISBN 978-8543004792, 8543004799.

GRANDE, R.; VIZCAÍNO, A.; GARCÍA, F. O. Is it worth adopting devops practices in global software engineering? possible challenges and benefits. **Computer Standards & Interfaces**, Elsevier, v. 87, p. 103767, 2024.

MARTIN, R. C. **Arquitetura Limpa**. [S.l.]: Alta Books, 2019. ISBN 9788550808161, 8550808164.

SANTOS, L. **Kubernetes - Tudo sobre orquestração de contêineres**. [S.l.]: Casa do Código, 2019. ISBN 9788572540254.

SINGH, N.; DAWOOD, Z. et al. **Building Microservices with Micronaut®: A quick-start guide to building high-performance reactive microservices for Java developers**. [S.l.]: Packt Publishing Ltd, 2021.

TANZIL, M. H. e. a. A mixed method study of devops challenges. **Information and Software Technology, v. 161, p. 107244**, 2023.

VALENTE, M. T. **Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade**. [S.l.]: Independente, 2020. ISBN 978-6500019506, 6500019504.

VITALINO, J. F. N.; CASTRO, M. A. N. **Descomplicando o Docker**. [S.l.]: Brasport, 2016. ISBN 9788574527970.

WALLS, C. **Spring in Action**. [S.l.]: Manning, 2018. ISBN 978-1617294945, 1617294942.

A APLICAÇÃO DE TESTE

Neste anexo, é apresentada a proposta da aplicação de teste desenvolvida, com o objetivo de auxiliar no entendimento e replicabilidade da mesma, mas que não é necessário para a compreensão dos resultados apresentados no Capítulo 4.

Na seção A.1 é apresentado como acessar o código-fonte do projeto, na seção A.1. Na seção A.2 é apresentado o conceito da Arquitetura Hexagonal. Na seção A.3, é apresentada a proposta da aplicação de teste desenvolvida e na seção A.4, é apresentada a arquitetura da aplicação. Por fim, é apresentada a implementação da aplicação de teste no ambiente Kubernetes, na seção A.5.

A.1 Replicabilidade

Em caso de interesse em replicar os testes com o mesmo código-fonte e configuração do cluster, é possível acessar o repositório do projeto, disponivel no link a seguir. Siga o arquivo README como guia para entender o código e os passos necessários para instalação e execução do projeto.

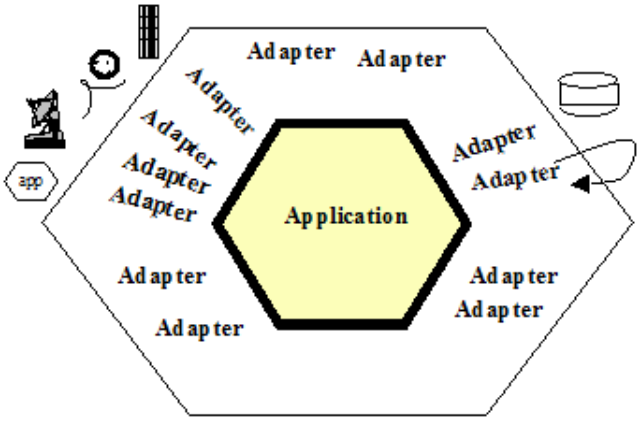
<https://github.com/chrystian9/comparativo-spring-quarkus-micronaut-em-kubernetes>

A.2 Conceito da Arquitetura Hexagonal

Apresentada por Cockburn (2005), a Arquitetura Hexagonal (também conhecida como Ports And Adapters) tem como objetivo blindar a aplicação de situações onde a lógica de negócio se mistura e cria dependência a drivers externos, que são tecnologias externas à aplicação, como *frameworks* e bancos de dados. Logo, a arquitetura resolve o problema propondo que uma camada interna do hexágono, onde fica o domínio e a lógica de negócio da aplicação (casos de uso), não tenha um acoplamento com camadas externas, mas sim contratos (interfaces) que definem as regras de cada porta.

De forma menos abstrata, imaginando uma aplicação com uma entidade A. Um dos casos de uso desta aplicação é a alteração do campo *title* da entidade A, definido pela interface *UpdateTitleA* com o método *update*. Essa interface fica na camada interna da aplicação. Para que esse caso de uso seja utilizado, a aplicação possui uma classe que implementa uma API (que pode ser uma REST, por exemplo). Essa API permite a edição dos campos da entidade A, relacionando-se por meio do campo ID da entidade. A classe da API importa a interface

Figura A.1 – Arquitetura Hexagonal - Exemplo básico

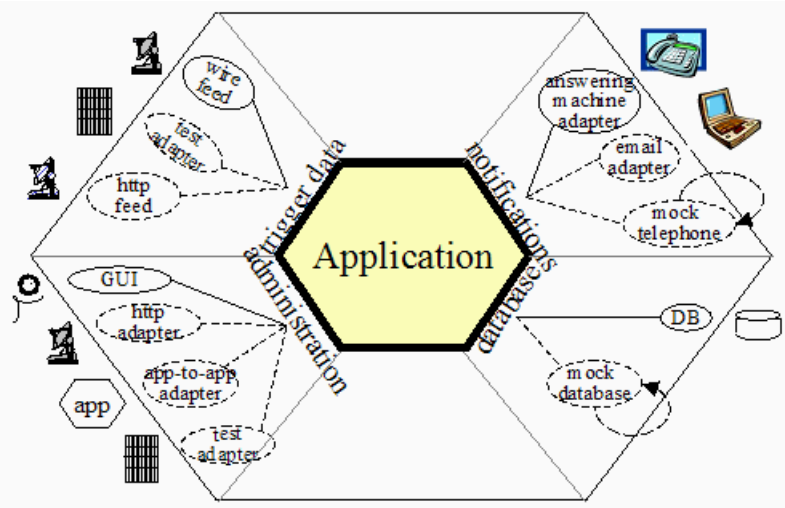


Fonte: Cockburn (2005)

UpdateTitleA e chama seu método *update*, passando o novo valor a ser atribuído. Assim, a API não invoca uma implementação concreta, mas sim um contrato. Neste contexto, caso ocorram mudanças na API, elas não afetarão a camada interna da aplicação.

Para que isso funcione, uma classe *factory* deve definir qual classe concreta implementará o contrato. Em conjunto, é utilizado a Inversão de Dependência e o Inversão de Controle para que classes de implementação sejam vinculadas aos seus contratos.

Figura A.2 – Arquitetura Hexagonal - Exemplo complexo



Fonte: Cockburn (2005)

Martin (2019) descreve essa arquitetura como tendo objetivos semelhantes à Arquitetura Limpa, sendo uma predecessora. O principal objetivo refere-se às dependências, as quais devem sempre apontar apenas para as camadas internas (MARTIN, 2019).

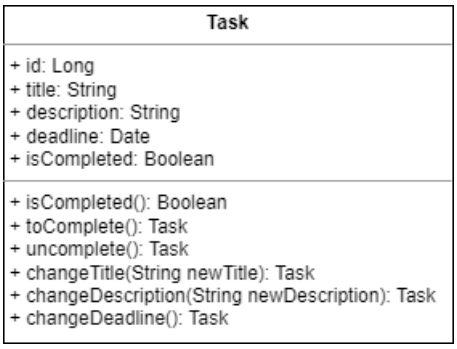
Como Valente (2020) descreve, a Arquitetura Hexagonal separa suas classes em dois grupos, sendo elas de domínio e relacionadas com a infraestrutura, tecnologias e sistemas ex-

ternos. Ele esclarece o objetivo principal, que é que as classes de domínio nunca dependam de classes relacionadas com tecnologias, infraestrutura e sistemas externos.

A.3 Proposta da aplicação de teste

Buscando simplicidade, a aplicação de teste se resume a um conceito de *To Do List*, onde existe uma lista de objetos do tipo *Task*. Cada *Task* é uma entidade com título, descrição, *deadline* (data de entrega) e um status de completo ou não. A figura A.3 descreve o *diagrama de classes*[1] da entidade *Task*.

Figura A.3 – Entidade Task



Fonte: Autor

Como validações para os dados, temos que os campos *title* e *description* devem ser diferentes de *null* e diferentes de uma *string* vazia, e o campo *deadline* também deve ser diferente de *null*.

Por fim, os casos de uso da aplicação se resumem à criação, remoção, alteração e obtenção da entidade Task, resultando em uma lista de Tasks - uma "To Do List". Esses casos de uso serão realizados por meio de uma API REST que receberá requisições do usuário.

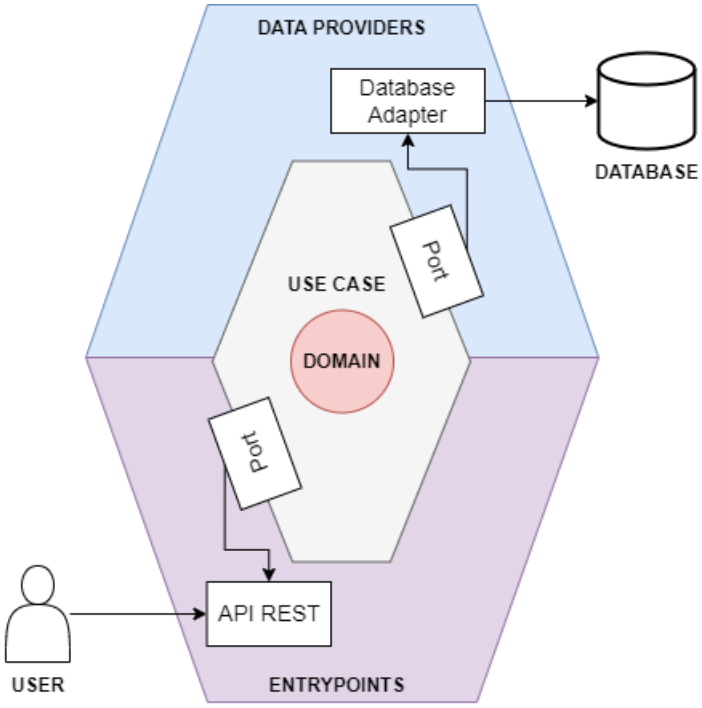
A.4 Arquitetura da aplicação de teste

A arquitetura a ser utilizada foi definida com base na Arquitetura Hexagonal, apresentado na seção **??**. Essa escolha de arquitetura tem dois objetivos neste trabalho, sendo o primeiro objetivo adicionar um pouco mais de complexidade e contexto de aplicações reais do mercado. O segundo está descrito adiante. A figura A.4 representa a arquitetura da aplicação.

Na parte interna da arquitetura, têm-se as camadas *Domain* e *Use Case*. Logo acima, encontram-se as camadas *Data Provider* e *Entrypoints*, que estão no mesmo nível da Arquitetura Hexagonal. Seguindo a Arquitetura Hexagonal, apresentada na seção **??**, as camadas

[1] Disponível em: <https://pt.wikipedia.org/wiki/Diagrama_de_classes>. Acesso em 6 de Mar. 2023

Figura A.4 – Arquitetura da aplicação de teste



Fonte: Autor

Domain e *Use Case* não possuem dependência de tecnologias, infraestrutura e sistemas externos. Portanto, essas camadas do projeto de teste serão os mesmos conjuntos de classes Java para todas as três versões da aplicação, de cada *framework*, sendo este o segundo objetivo da escolha dessa arquitetura.

Dadas as características simples da aplicação, o sistema gerenciador de banco de dados (SGBD) escolhido foi o *PostgreSQL*[2]. Trata-se de um SGBD do tipo relacional que utiliza memória em disco. Essa escolha leva em consideração a sua popularidade, com uma comunidade ativa e contribuições significativas, o que facilita a adaptação para qualquer um dos três *frameworks* em foco neste trabalho.

A.4.1 Camada Domain

A camada *Domain* tem como objetivo principal abrigar a regra de negócio, que neste trabalho se resume à entidade *Task* e suas validações, definidas na seção A.3, que devem ser realizadas durante a criação da mesma.

[2] Disponível em: <https://www.postgresql.org/>. Acesso em 6 de Mar. 2023

A.4.2 Camada Use Case

A camada *Use Case* contém as classes responsáveis por implementar os casos de uso definidos na camada *Domain*. Neste trabalho, esses casos de uso incluem a criação, alteração, recuperação e remoção da entidade *Task*. Para realizar essas funcionalidades, todas as classes de casos de uso possuem dependência de um contrato (interface) chamado *TaskOutputPort*, o qual abstrai o acesso ao banco de dados. Este contrato será implementado por um adaptador em uma camada mais externa, seguindo assim a Arquitetura Hexagonal. Também nesta camada, são definidos os contratos para cada caso de uso, permitindo que a camada de entrada mais externa os utilize.

A.4.3 Camada Data Providers

A camada *Data Providers* já está em um nível da arquitetura que possui contato com tecnologias e conexões externas. Portanto, encontraremos aqui dependências diretas ao *framework* em questão. Seu conceito principal é ser a camada que implementa os adaptadores definidos na camada mais interna, *Use Case*. Consequentemente, é nessa camada que temos a conexão com sistemas externos, que neste trabalho se resume ao banco de dados. Por meio das bibliotecas do *framework* voltadas para a conexão com banco de dados, essa camada será capaz de realizar a persistência da aplicação.

A.4.4 Camada Entrypoint

A camada *Entrypoints* é o ponto de entrada para o usuário final da aplicação; é aqui que a API REST é implementada. Essa API consiste em uma classe *controller* responsável por receber as requisições e utilizar os contratos da camada *Use Case* para alcançar seus objetivos. Além disso, nesta camada, encontra-se o ponto de *runtime* da aplicação, onde o *framework* é inicializado com suas bibliotecas compartilhadas.

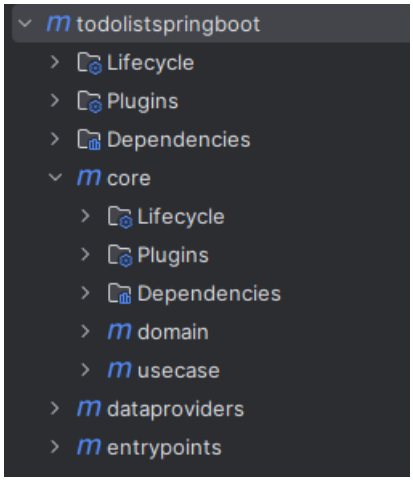
A.4.5 Organização do código-fonte

Com o objetivo de seguir os conceitos da Arquitetura Hexagonal, cada camada representa um módulo da aplicação, tendo assim seu próprio processo de compilação, onde a camada mais interna é importada pela camada mais externa. Dessa forma, obtemos módulos com possi-

bilidade de portabilidade entre aplicações, como é o caso dos módulos que contêm as camadas *Domain* e *Use Case*.

Para obter esse resultado, foi utilizada a ferramenta Maven, descrita na seção 2.2.2, a qual fornece a configuração de projetos multimódulo. Isso é possível utilizando a funcionalidade de módulos pai e filho, onde o arquivo mais externo é o pai principal, que define os demais módulos.

Figura A.5 – Arquitetura do Maven com multimódulos



Fonte: Autor

Logo, tem-se que o arquivo de gerenciamento de dependência, POM, do módulo *entrypoints*, importa os módulos *dataproviders* e *core*. O módulo *core* desempenha o papel de pai dos módulos *domain* e *usecase*, facilitando a portabilidade desse módulo entre as três versões da aplicação, com cada *framework*.

A.4.6 Injeção de dependência

As camadas *Data Providers* e *Entrypoints* farão uso dos contratos da camada *Use Case* para implementar a persistência e receber as requisições da API REST, executando os casos de uso da aplicação, como mencionado anteriormente.

Para que esses contratos estejam devidamente relacionados com suas implementações concretas, uma classe *AppConfig* foi implementada nas três versões. Essa classe tem o objetivo de utilizar a API de injeção de dependência de cada *framework* para configurar qual classe concreta "A"implementa o contrato "B".

No Spring Framework, foi utilizada a anotação *@Configuration* para definir a classe *AppConfig* como uma classe que deve ser lida durante a inicialização da aplicação. Seus métodos são responsáveis pela criação de instâncias das implementações de cada contrato necessário.

Para que essas instâncias sejam atribuídas aos *Beans* (injeções de dependências do tipo singleton), foi utilizado o método *@Bean*.

No Quarkus, o conceito é o mesmo, porém a anotação da classe é *@Dependent* da biblioteca *jakarta*. Para os métodos, são definidas as anotações *@DefaultBean* e *@Produces*.

No Micronaut, o conceito também é o mesmo. A anotação da classe é *@Factory*, própria do Micronaut, e para os métodos utiliza-se *@Singleton*.

A.5 Implementação da aplicação de teste no ambiente Kubernetes

Para implementar as versões da aplicação de teste no ambiente Kubernetes, foi necessário criar um cluster Kubernetes. Isso foi feito utilizando o *Docker Desktop*[3], o qual é uma plataforma que possibilita o gerenciamento de imagens e contêineres Docker, contendo nele a funcionalidade nativa de criação de um cluster Kubernetes, o que torna prático o desenvolvimento em máquinas com o sistema operacional Windows. Em sequencia, foi necessário containerizar as aplicações, o que foi feito por meio do Docker. Com as imagens construidas, foram implementados os manifestos de Deployment e Service para cada versão da aplicação, em cada *framework*. Também foram criados manifestos para cada versão do banco PostgreSQL.

A.5.1 Containerização da aplicação

Em busca de containerizar cada versão da aplicação, foi criado um arquivo Dockerfile para cada uma. Este arquivo Dockerfile opera em dois estágios, sendo o primeiro destinado à compilação da aplicação e o segundo à execução da mesma.

No primeiro estágio, destacado na figura A.6, a imagem base utilizada é a *maven:3.8.3-openjdk-17*, contendo tanto o Maven quanto o JDK 17 instalado. O processo inicia buscando os arquivos de configuração POM para, em seguida, utilizar o Maven para resolver as dependências. Com as dependências resolvidas e armazenadas no cache do Maven, o código-fonte do projeto é buscado e, em seguida, compilado utilizando o Maven.

Já no estágio 2, é realizado o processo de execução da aplicação, utilizando o JAR compilado no estágio anterior. O estágio 2 varia entre os *frameworks*. No Spring Framework e no Micronaut, destacado na figura A.7, é mais simples, somente executando o JAR a partir de uma imagem simples que contenha a JDK 17. Já no Quarkus, destacado na figura A.8,

[3] Disponível em: <https://www.docker.com/products/docker-desktop/>. Acesso em 6 de Mar. 2023

Figura A.6 – Estágio 3 do arquivo Dockerfile

```
# Estágio 1: Construção do projeto Java
FROM maven:3.8.3-openjdk-17 AS DEPS

WORKDIR /build

# Copiar arquivos de configuração Maven
COPY pom.xml pom.xml
COPY entrypoints/pom.xml ./entrypoints/
COPY dataproviders/pom.xml ./dataproviders/
COPY core/pom.xml ./core/
COPY core/domain/pom.xml ./core/domain/
COPY core/usecase/pom.xml ./core/usecase/

# Resolve as dependências
RUN mvn -B -e -C org.apache.maven.plugins:maven-dependency-plugin:3.1.2:go-offline

# Copie as dependências do estágio DEPS com a vantagem
# de usar o cache de camadas do Docker. Se algo der errado a partir desta
# linha, todas as dependências do estágio DEPS já foram baixadas e
# armazenadas nas camadas do Docker.
FROM maven:3.8.3-openjdk-17 AS BUILDER

WORKDIR /build

COPY --from=deps /root/.m2 /root/.m2
COPY --from=deps /build /build
COPY entrypoints/src /build/entrypoints/src
COPY dataproviders/src /build/dataproviders/src
COPY core /build/core

# Compila o projeto
RUN mvn -B -e -o clean install -DskipTests=true
```

Fonte: Autor

é necessário utilizar a imagem *registry.access.redhat.com/ubi8/openjdk-17:1.16* e utilizar de comandos de inicialização diferentes, para funcionar corretamente.

Ponto importante a ser destacado é que o Quarkus gera arquivos Dockerfile automaticamente para cada tipo de imagem (nativa ou normal), sendo necessário apenas executar o comando de build do Docker. No entanto, para os propósitos deste trabalho, foi escolhido construir um novo arquivo Dockerfile, ajustando-o com base naqueles gerados pelo *framework* e considerando a abordagem não nativa.

Figura A.7 – Estágio 2 do arquivo Dockerfile - Spring Framework e Micronaut

```
# Compila o projeto
RUN mvn -B -e -o clean install -DskipTests=true

# Estágio 2: Usa uma imagem JRE mínima para executar o aplicativo
FROM openjdk:17-jdk-slim

WORKDIR /app

# Copia o arquivo JAR gerado do primeiro estágio
COPY --from=builder /build/entrypoints/target/entrypoints-1.0-SNAPSHOT.jar .

# O comando a ser executado quando o contêiner iniciar
CMD ["java", "-jar", "entrypoints-1.0-SNAPSHOT.jar"]
```

Fonte: Autor

Para gerar as imagens Docker, foi utilizado o comando:

```
$ docker build -t chrystian9/<image-name> .
```

Em seguida, para enviar a imagem para o DockerHub, foi utilizado o comando:

Figura A.8 – Estágio 2 do arquivo Dockerfile - Quarkus

```
# Estágio 2: Usa uma imagem JRE mínima para executar o aplicativo
FROM registry.access.redhat.com/ubi8/openjdk-17:1.16

# Copia os arquivos JAR gerados do primeiro estágio
COPY --from=builder /build/entrypoints/target/quarkus-app/lib/ /deployments/lib/
COPY --from=builder /build/entrypoints/target/quarkus-app/*.jar /deployments/
COPY --from=builder /build/entrypoints/target/quarkus-app/app/ /deployments/app/
COPY --from=builder /build/entrypoints/target/quarkus-app/quarkus/ /deployments/quarkus/

EXPOSE 8080
ENV JAVA_OPTS="-Dquarkus.http.host=0.0.0.0 -Djava.util.logging.manager=org.jboss.logmanager.LogManager"
ENV JAVA_APP_JAR="/deployments/quarkus-run.jar"

# O comando a ser executado quando o contêiner iniciar
ENTRYPOINT [ "/opt/jboss/container/java/run/run-java.sh" ]
```

Fonte: Autor

```
$ docker push chrystian9/<image-name>
```

A.5.2 Manifestos da aplicação

Os manifestos de cada aplicação consistem em quatro arquivos YAML, incluindo o *deployment*, o *service*, o *service-monitor* e o *postgres*.

O arquivo *deployment* define o manifesto de Deployment da aplicação no cluster Kubernetes, especificando o namespace, as labels, o nome do app e do container, as variáveis de ambiente do container, os limites de recursos (CPU e memória) e a porta do container, que é configurada para *8080*.

O arquivo *service* define o manifesto de Service do container definido no *deployment*, expondo a rede da aplicação na porta *8080*.

O arquivo *service-monitor* define o manifesto customizado ServiceMonitor, disponibilizado pela API *monitoring.coreos.com/v1* ao instalar o Prometheus. Seu objetivo é criar automaticamente as configurações necessárias para que o Prometheus possa extrair as métricas da aplicação.

Por fim, o arquivo *postgres* define os manifestos de ConfigMap, PersistentVolumeClaim, Deployment e Service relacionados ao *PostgreSQL*, possibilitando a criação de um container para o banco de dados.

A.5.3 Instalação do Prometheus e do Grafana

Para a utilização do Prometheus e do Grafana, foi utilizado o repositório *Helm*[4] *prometheus-community*[5]. Esse repositório possui o projeto *kube-prometheus-stack*[6], o qual disponibiliza o *Prometheus Operator*, um conjunto de configurações automáticas da pilha de monitoramento Prometheus, que inclui o Grafana.

Para instalar o repositório Helm localmente, foi utilizado o comando:

```
$ helm repo add prometheus-community https://prometheus-community.
    github.io/helm-charts
```

Com o repositório instalado localmente, foi necessário instalar o prometheus-operator com o comando:

```
$ helm install prometheus-operator prometheus-community/kube-
    prometheus-stack -n monitoring
```

[4] Disponível em: <https://helm.sh/>. Acesso em 6 de Mar. 2023

[5] Disponível em: <https://github.com/prometheus-community/helm-charts>. Acesso em 6 de Mar. 2023

[6] Disponível em: <https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack>. Acesso em 6 de Mar. 2023