

**UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO (2025)**

UM SISTEMA ESCALÁVEL E RESILIENTE PARA MONITORAMENTO DE SERVIÇOS WEB

Brenno Kevyn Maia de Souza¹
Paulo Henrique Lopes Silva²

RESUMO

Este trabalho apresenta o desenvolvimento do SentryWeb, um sistema escalável para o monitoramento da disponibilidade de serviços, utilizando *Kubernetes* e *RabbitMQ*. O objetivo foi criar uma plataforma modular que permita o agendamento e processamento de tarefas de monitoramento em um ambiente distribuído, escalável e resiliente a falhas. O projeto é composto por dois microsserviços principais: o *Scheduler*, responsável pelo agendamento das tarefas, e o *Worker*, que processa as tarefas. Para gerenciar o autoescalonamento horizontal o sistema utiliza métricas tradicionais como CPU e memória e de métricas de negócio personalizadas, extraídas do *RabbitMQ*. Os experimentos de validação realizados em diversos cenários e cargas de trabalho mostraram que a combinação entre *Kubernetes* e *RabbitMQ* oferece uma solução flexível e eficiente para o monitoramento de serviços, garantindo alta disponibilidade e desempenho.

Palavras-chave: *Kubernetes*, *RabbitMQ*, *Worker*, *Scheduler*, Tarefas.

1 INTRODUÇÃO

A demanda por aplicações robustas, escaláveis e distribuídas está em constante crescimento. Dessa forma, garantir que uma aplicação distribuída funcione corretamente vem se mostrando um desafio crítico, com sistemas de monitoramento e verificação cada vez mais necessários para acompanhar o funcionamento das aplicações, principalmente em contextos com múltiplas APIs, bancos de dados e outros serviços web (BURNS et al., 2022). O tempo de recuperação de uma aplicação aumenta consideravelmente quando não há mecanismos de detecção de falhas, o que pode resultar em interrupções no serviço, perda de dados ou degradação da experiência de uso.

Para endereçar essa necessidade, este trabalho propõe o desenvolvimento do SentryWeb, um sistema altamente escalável cuja arquitetura se baseia em duas ferramentas principais: *Kubernetes* e *RabbitMQ*. O *Kubernetes*, como um orquestrador de *containers* que oferece uma plataforma com diversos recursos para automação, escalabilidade e recuperação de falhas (BURNS et al., 2022; POSIELLO et al., 2021). Já o *RabbitMQ*, um sistema de

¹ Discente do Bacharelado em Ciência da Computação da UFERSA/Mossoró.

² Professor Associado do Departamento de Computação da UFERSA/Mossoró.

mensageria para permitir comunicação de forma desacoplada e assíncrona, proporcionando uma maior flexibilidade para implementar a lógica de outras partes do projeto (BURNS et al., 2022). Embora a aplicação prática desenvolvida seja a verificação de disponibilidade de *endpoints* — onde cada tarefa monitora o funcionamento de um componente, como uma API ou banco de dados através de uma URL —, a solução foi projetada como uma plataforma de agendamento de tarefas genérica, estando aberta para modelagem de diversos tipos de trabalhos.

Essa arquitetura modular do SentryWeb é composta por dois microsserviços principais: o *Scheduler*, responsável por agendar tarefas que devem ser executadas, e o *Worker*, que é responsável por processá-las. O desacoplamento da comunicação entre eles permite que ambos escalem independentemente, melhorando a capacidade do sistema. Um diferencial importante nesse projeto é o uso de métricas personalizadas para o autoescalonamento extraídas diretamente do *RabbitMQ*, já que em algumas situações foi constatado que medir apenas o uso de CPU e/ou memória não se mostrou eficaz em todos os casos.

Diante desse cenário, a combinação entre *Kubernetes* e *RabbitMQ* pode contribuir para o desenvolvimento de um sistema de verificação de serviços, pois reúnem características que promovem a escalabilidade, resiliência e interconexão, com flexibilidade para ampliação de suporte a diferentes tipos de serviços.

2 OBJETIVO GERAL

Desenvolver um sistema para verificar o funcionamento de serviços em ambientes distribuídos virtual, utilizando *Kubernetes* para orquestração e escalabilidade e *RabbitMQ* para distribuição de tarefas. O objetivo é monitorar endpoints de forma paralela, automatizada e tolerante a falhas, validando o estado real dos serviços.

2.1 OBJETIVOS ESPECÍFICOS

Para alcançar o propósito descrito na introdução, este trabalho foi guiado por uma série de objetivos específicos que detalham as etapas necessárias para sua concretização. Sendo eles:

- Escolher ferramentas e tecnologias adequadas;
- Projetar uma arquitetura de microsserviços desacoplada;
- Garantir a escalabilidade do *Worker* e *Scheduler*;
- Configurar métricas tradicionais e/ou personalizadas para autoescalonamento;
- Validar o comportamento do sistema em diferentes cenários.

A execução desses objetivos resultou no desenvolvimento da plataforma SentryWeb, com arquitetura, implementação e resultados serão apresentados nas próximas seções deste trabalho.

3 MÉTODO

O método adotado para o desenvolvimento deste trabalho pode ser classificado como uma pesquisa construtiva, cujo principal objetivo é a criação de um *software* funcional, o SentryWeb, como solução para o problema prático e objetivos definidos. Por linhas gerais o processo para construção do trabalho seguiu uma abordagem iterativa, baseada em uma fundamentação teórica com as principais ferramentas utilizadas, desenvolvimento com escolhas feitas para adequar às necessidades do projeto, realizar experimentos para validar o funcionamento do sistema e por fim apresentar as considerações finais com as conclusões alcançadas através da observação dos resultados.

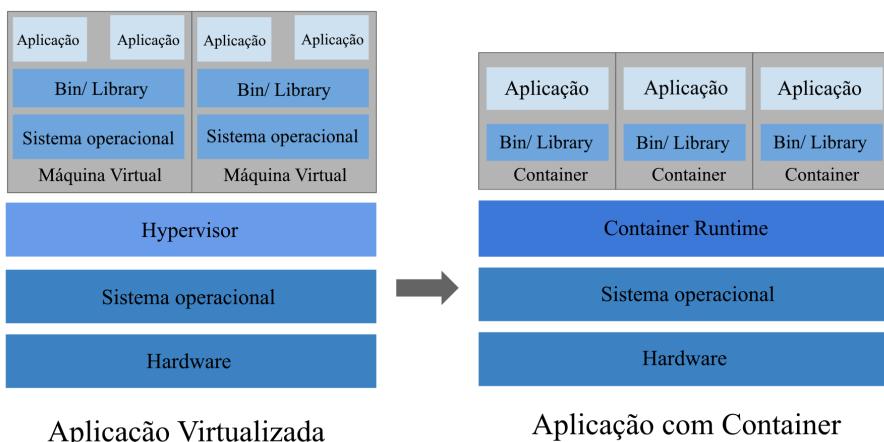
4 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, são apresentados os conceitos e tecnologias fundamentais que servem como pilar para o desenvolvimento do projeto SentryWeb. A compreensão destes tópicos é essencial para entender as decisões de arquitetura e implementação.

4.1 CONTAINER

Container é a tecnologia usada para empacotar aplicações junto com todas suas dependências, como bibliotecas e configurações. Diferem-se de máquinas virtuais (VMs), que virtualizam o sistema operacional inteiro, um container virtualiza apenas o necessário para funcionar e compartilha o *kernel* do sistema hospedeiro. Conforme ilustrado na Figura 1, vemos que ao utilizar *containers*, é excluída uma camada inteira de um sistema operacional virtualizado, isso os torna mais leves, rápidos para iniciar e gastam significativamente menos recursos (KUBERNETES, 2025).

Figura 1 - Comparação entre virtualizado e containerizado



Fonte: Adaptado de Kubernetes (2025).

Isso é possível graças às primitivas do *kernel* do *Linux*, como os *Namespaces*, que isolam a visão de um processo sobre outros recursos do sistema (outros processos, montagem de arquivos e rede), os *Control Groups* (*cgroups*), que limitam os recursos de processamento e memória que cada processo pode consumir e o comando *CHROOT*, que altera o diretório raiz de um processo, impedindo que ele consiga acesso a diretórios de níveis mais altos. Em

conjunto, essas primitivas podem criar um ambiente isolado para a aplicação, garantindo que ela se comporte da mesma forma independentemente do lugar em que esteja (KUBERNETES, 2025).

O pilar fundamental da portabilidade de um *container* é a sua imagem, ela é a representação de um software empacotado pronto para ser executado, ela contém tudo que é necessário para a aplicação (código, bibliotecas, configurações, etc). Por padrão, um *container* é imutável, não podemos mudar detalhes dentro de uma imagem, para efetuar alterações, uma nova imagem deve ser criada (KUBERNETES, 2025).

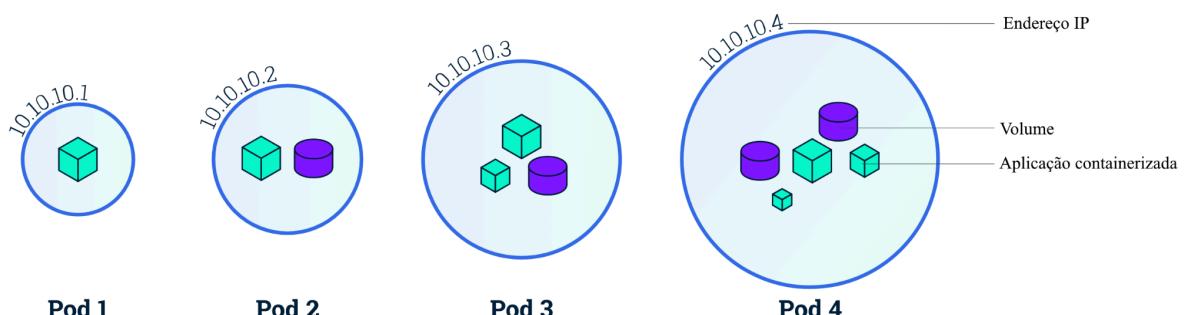
A tecnologia que popularizou o uso de *containers* e imagens de forma acessível é o *Docker*, um *container engine* amplamente utilizado. Ele permite a criação, empacotamento e execução de aplicações em *containers*, fornecendo ferramentas para construir, executar e gerenciar os *containers* em um sistema hospedeiro com base nas primitivas do *kernel Linux* (DOCKER, 2025). Embora o *Kubernetes* seja compatível com outras *engines*, o *Docker* é a tecnologia padrão utilizada pelo *Minikube* para executar os *containers* e é importante notar que o sucesso do *Docker* levou à criação da *Open Container Initiative (OCI)*, que padroniza os formatos de imagem e o *runtime* de *containers*, garantindo a interoperabilidade entre diferentes ferramentas do ecossistema, incluindo o próprio *Kubernetes* (OPEN CONTAINER INITIATIVE, 2025).

4.2 KUBERNETES

O *Kubernetes* é a ferramenta para orquestração de *containers* mais usada atualmente. Com ela, é possível automatizar implantação, dimensionamento e gerenciamento de aplicações em *containers* de forma distribuída com o uso de vários novos conceitos, sendo os mais relevantes para o projeto: *namespace*, *pod*, *volumes*, *replicaset*, *deployment*, *statefulset*, *service*, *horizontal pod autoscaler* (KUBERNETES, 2025).

A plataforma opera através de um conjunto de objetos que descrevem o estado desejado da aplicação. A unidade fundamental de execução é o *Pod*, que é a menor unidade de componente que é possível criar em um *cluster*, é com *Pod* que o *kubernetes* interage diretamente e não com um *container* individualmente. Conforme visto na Figura 2, cada *Pod* possui um endereço na rede interna do *kubernetes* e pelo menos a imagem de um *container*, mas pode conter múltiplos *containers* e *volumes* — diretórios de armazenamento que são acessíveis aos containers do *Pod* em questão e são atrelados ao seu ciclo de vida.

Figura 2 - Pods do Kubernetes



Fonte: Adaptado de Kubernetes (2025).

A comunicação entre diferentes *Pods* é crucial para aplicações distribuídas e, em ambientes maiores, essa organização é facilitada pelo uso de *namespaces* — que funcionam como “*clusters* virtuais” dentro de um *cluster* físico, permitindo o isolamento lógico da comunicação, onde, por padrão, um *Pod* não se comunica com um *Pod* de um *namespace* diferente (KUBERNETES, 2025). É importante ressaltar que o *Pod* é efêmero, seu ciclo de vida é curto, um *Pod* é descartável, eles são criados e apagados sempre que necessário dentro de um *cluster*, por essa razão, os *Pods* são gerenciados por controladores de nível superior que garantem a resiliência e o estado desejado da aplicação (KUBERNETES, 2025).

O principal controlador do *Kubernetes* é o *ReplicaSet* cuja principal função é garantir que um número especificado de réplicas de um *Pod* esteja sempre em execução. Ele atua como um processo de auto-recuperação, substituindo *Pods* que falham ou são excluídos. Embora seja um componente fundamental, geralmente não interagimos diretamente com ele, pois o *Deployment* o utiliza e gerencia de forma abstrata para orquestrar atualizações e o ciclo de vida das réplicas (KUBERNETES, 2025). Um *Deployment* é um objeto de alto nível que gerencia o ciclo de vida de aplicações *Stateless* (sem estado). Ele permite atualizações declarativas para *Pods* e *ReplicaSets*, garantindo que um número desejado de réplicas esteja sempre em execução e orquestrando atualizações graduais (*rolling updates*) sem tempo de inatividade. No projeto SentryWeb, o componente *Worker* é gerenciado por um *Deployment*, pois cada instância é idêntica e descartável, podendo ser criada ou destruída a qualquer momento sem perda de estado (KUBERNETES, 2025).

Enquanto *Deployments* são para aplicações *Stateless*, um *StatefulSet* é o controlador ideal para aplicações *stateful*. Ele fornece garantias sobre a ordem e a unicidade dos *Pods*, oferecendo identidades de rede estáveis e únicas (ex: *scheduler-0*, *scheduler-1*) e armazenamento persistente estável. Esta característica é fundamental para o *Scheduler* do SentryWeb. A identidade estável de cada *Pod* do *Scheduler* é utilizada diretamente pelo algoritmo de particionamento para determinar quais tarefas pertencem a qual instância, tornando o *StatefulSet* a escolha correta para este componente (KUBERNETES, 2025).

A comunicação entre esses componentes é viabilizada pelo *Service*, um objeto que expõe um conjunto de *Pods* como um serviço de rede, fornecendo um único ponto de acesso estável (um nome DNS e um endereço IP virtual). Ele desacopla os clientes dos *Pods*, que podem ser criados e destruídos dinamicamente. No SentryWeb, um *Service* é utilizado para expor o *PostgreSQL* e o *RabbitMQ*, permitindo que os *Pods* do *Scheduler* e *Worker* se conectem a eles através de um nome de serviço estável (ex: *sentryweb-rabbitmq*), sem precisar conhecer os IPs individuais dos *Pods* do banco de dados ou do *broker* de mensagens (KUBERNETES, 2025).

Por fim, temos o *Horizontal Pod Autoscaler* (HPA), o componente que gerencia a escalabilidade do sistema. O HPA é o mecanismo do *Kubernetes* que ajusta automaticamente o número de réplicas em um *Deployment* ou *StatefulSet* com base em métricas observadas. Embora tradicionalmente utilize métricas de CPU e memória, sua versão v2 permite o uso de métricas customizadas e externas. Esta funcionalidade é um pilar do SentryWeb, onde o HPA do *Worker* é configurado para escalar não com base em recursos, mas sim em métricas de negócio extraídas do *RabbitMQ* (como *rabbitmq_queue_messages_ready*), permitindo um escalonamento muito mais inteligente e alinhado com a carga de trabalho real (KUBERNETES, 2025).

4.3 MINIKUBE

O *Minikube* é uma ferramenta que configura um *cluster Kubernetes* de forma local, ou seja, em uma única máquina (MINIKUBE, 2025). Ele foi projetado para facilitar boa parte da configuração do cluster, fornecendo o ambiente completo de forma simplificada. Para o desenvolvimento do SentryWeb, o *Minikube* foi a ferramenta escolhida para criar o ambiente de validação local. Pois permite testar todos os componentes reais de um *cluster* como a interação entre os microsserviços, persistência de dados e estratégias de autoescalonamento.

4.4 RABBITMQ

RabbitMQ é um intermediário de mensagens (*message broker*) de código aberto muito popular amplamente utilizado para implementar sistemas de comunicação assíncrona. Ele se baseia no protocolo AMQP (*Advanced Message Queuing Protocol*) e permite que diferentes partes de uma aplicação se comuniquem de forma indireta, através da publicação e consumo de mensagens (RABBITMQ, 2025).

No SentryWeb, o *RabbitMQ* tem como papel central o desacoplamento da arquitetura. Ele atua como uma camada intermediária entre o *Scheduler* (que publica tarefas) e o *Worker* (que consome). A principal vantagem dessa abordagem é programar os microsserviços de maneira independente, tornando o sistema mais resiliente e escalável, ao remover a necessidade de comunicação direta entre eles (RABBITMQ, 2025).

A arquitetura de comunicação do *RabbitMQ* é baseada em três componentes:

- *Exchange*: É o ponto de entrada para as mensagens publicadas por um produtor. Uma *exchange* recebe a mensagem e a roteia para uma ou mais filas, com base em regras definidas pelo seu tipo (como *direct*, *topic* ou *fanout*).
- *Queue* (fila): É a estrutura de dados que armazena as mensagens, funcionando como um *buffer*. As mensagens permanecem na fila até que um consumidor esteja pronto para processá-las.
- *Binding*: É a regra que conecta uma *exchange* a uma *queue*.

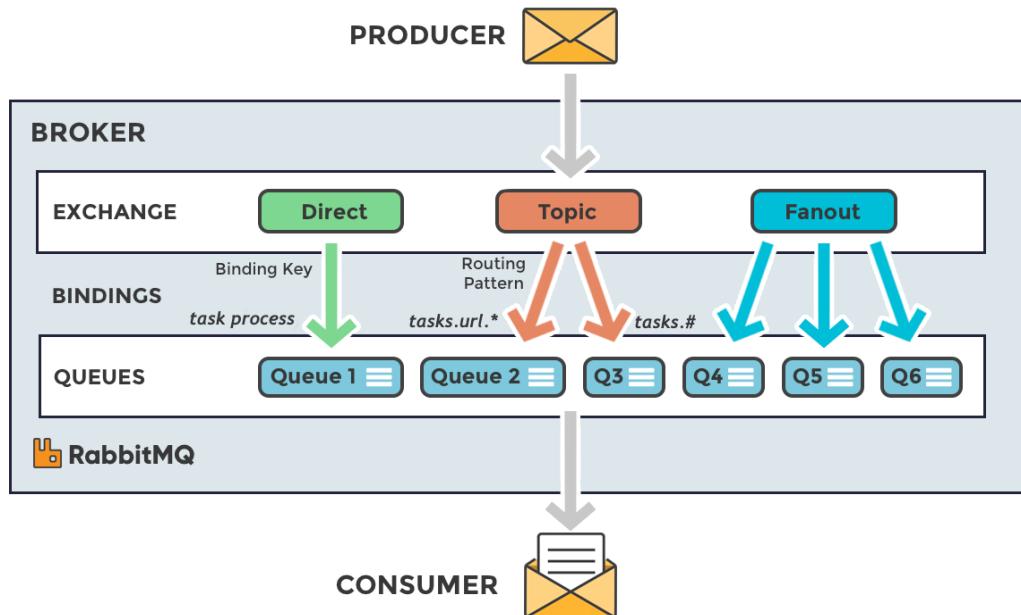
A flexibilidade do *RabbitMQ* reside nos diferentes tipos de *exchanges*, que permitem a implementação de diversos padrões de comunicação. Cada tipo utiliza uma lógica de roteamento distinta para distribuir as mensagens para as filas. Sendo os tipos:

- ***Direct***: Roteia uma mensagem para a fila cujo *binding key* é idêntico à *routing key* da mensagem. É o tipo utilizado implicitamente no SentryWeb para a fila de trabalho “*tasks*”.
- ***Fanout***: Roteia uma mensagem para todas as filas que estão conectadas a ela, ignorando a *routing key*.
- ***Topic***: Realiza o roteamento com base em uma correspondência de padrões entre a *routing key* (ex: *tasks.url.check*) e o *binding key* (ex: *tasks.url.**). Este tipo, embora não implementado no protótipo atual, representa um caminho natural para a evolução do SentryWeb. Por exemplo, seria possível implementar *Workers* especializados que assinam apenas *tasks.url.** para

receber todas as tarefas relacionadas à verificação de serviços *web*, enquanto um outro *Worker* poderia assinar *tasks.#* para receber todas as tarefas, independentemente do tipo.

A Figura 3 ilustra o funcionamento desses três tipos de *exchanges*, exemplificando o fluxo da publicação de uma mensagem feita pelo *Producer* e o recebimento pelo *Consumer*.

Figura 3 - Componentes do RabbitMQ



Fonte: Adaptado de CloudAMQP (2025).

No projeto, os *Workers* utilizam uma *exchange* do tipo *direct* (a *exchange* padrão) e consomem mensagens da fila “*tasks*”, onde cada *Worker* conectado recebe uma tarefa. Já os *Schedulers*, publicam essas mensagens, mas também são consumidores da *exchange* do tipo *fanout*, que foi criada para notificar a criação de novas tarefas a serem agendadas, isso garante que cada *Scheduler* ativo receba a mesma mensagem e se responsabilize do agendamento, dessa forma evitando que eles sejam reiniciados para lerem novas tarefas no banco de dados.

5 DESENVOLVIMENTO

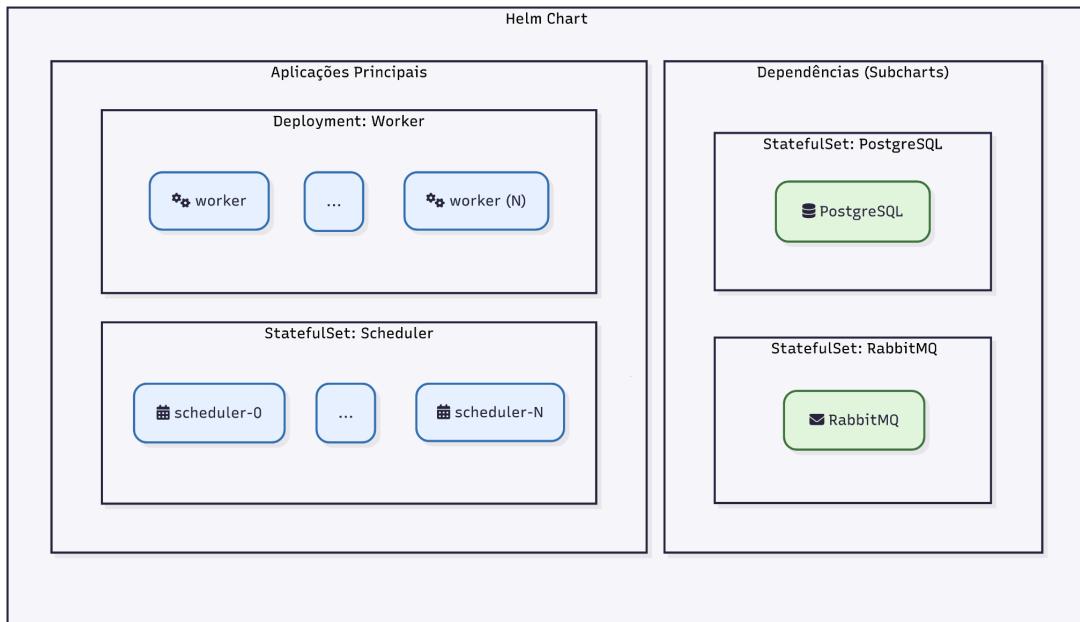
Essa seção detalha o desenvolvimento do projeto SentryWeb, serão abordadas a arquitetura geral do sistema, implementação do *Scheduler* e *Worker*, e configuração das estratégias de autoescalonamento.

5.1 ARQUITETURA GERAL

A arquitetura do SentryWeb foi pensada para seguir o padrão de microsserviços para criar um sistema desacoplado, resiliente e escalável. A organização do projeto está contida em um *Chart* gerenciado pelo *Helm*, um gerenciador de pacotes para aplicações *Kubernetes* que facilita a aplicação de novos manifestos e novas versões dos componentes da aplicação

(HELM, 2025). Os componentes principais são o *Worker* e *Scheduler*, e os *Subcharts* do *RabbitMQ* e *PostgreSQL*, conforme mostrado na Figura 4.

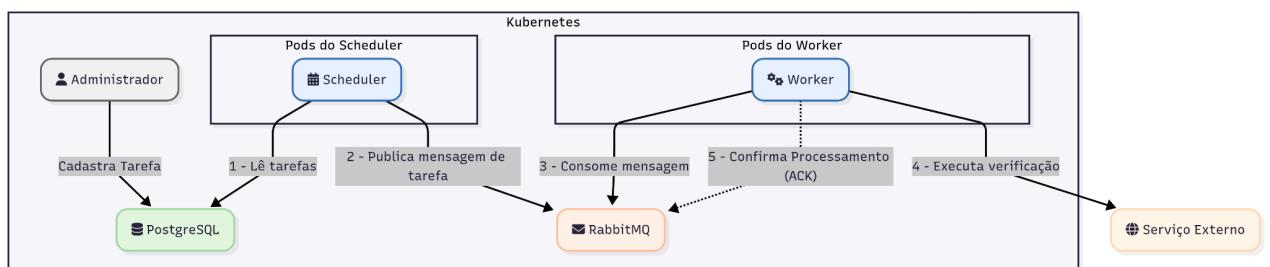
Figura 4 - Arquitetura dos microsserviços no projeto



Fonte: Autoria própria (2025)

Objetivando compreender o fluxo de trabalho temos, como primeira ação, o *Scheduler* consultando o *PostgreSQL* para obter todas as tarefas que estão inseridas. Em seguida, o *Scheduler* publica na fila de mensagens no *RabbitMQ* cada tarefa com base em um intervalo de tempo definido em cada uma delas. Do outro lado está o *Worker*, que consome a fila de mensagens contendo a tarefa, executa a verificação no *endpoint* de destino e em seguida confirma (*ACK*) o processamento da mensagem para o *RabbitMQ*, assim concretizando o consumo da mensagem. A Figura 5 mostra esse funcionamento de forma simplificada.

Figura 5 - Fluxo de funcionamento



Fonte: Autoria própria (2025)

Outro ponto importante do projeto, mas que não está incluído no pacote do *Helm* é o *Prometheus*, um coletor de métricas do *cluster*. Ele é gerenciado como uma aplicação independente no *ArgoCD* e foi configurado para coletar métricas específicas do *RabbitMQ*. Com apoio do *Prometheus-adapter*, que atua como uma ponte para a *API* do *Kubernetes*, essas métricas serão expostas ao *cluster* para que o *HPA* seja capaz de ler e entender essas

novas informações externas (PROMETHEUS, 2025). O uso de métricas personalizadas é um ponto opcional do projeto, sendo configurável de acordo com a necessidade, mas suas implicações de uso serão abordadas no trabalho.

5.2 IMPLEMENTAÇÃO DOS MICROSERVIÇOS

Os componentes foram desenvolvidos em *Python*, com foco no rápido desenvolvimento e facilidade na montagem de imagem do *container* por ser uma linguagem interpretada (PYTHON, 2025).

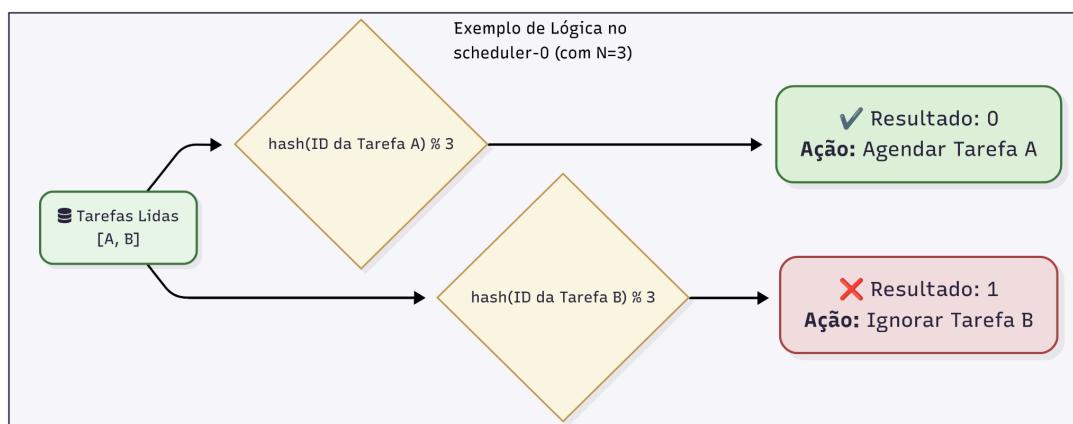
5.2.1 SCHEDULER

O *Scheduler* foi implementado como um serviço *Stateful*, gerenciado pelo *StatefulSet* do *Kubernetes*. Isso garante identidades estáveis de rede, na prática isso significa que cada *Scheduler* terá um nome fixo e serão criados em ordem (*Scheduler-0*, *Scheduler-1*...*Scheduler-N*). Esse comportamento é fundamental para o funcionamento da arquitetura distribuída do componente, que se baseia em três processos principais: a sincronização periódica com o banco de dados e *Kubernetes*, comunicação com uma exchange especial do *RabbitMQ* para receber novas tarefas em tempo de execução e o ciclo principal responsável por publicar mensagens na fila. Para que possamos adentrar no detalhamento desses processos nas seções a seguir, primeiro é necessário explicar o particionamento de tarefas.

5.2.1.1 PARTICIONAMENTO DE TAREFAS

A principal função de um *Scheduler* é determinar quais e quando as tarefas devem ser executadas. Para evitar um único ponto de falha e permitir o escalonamento, foi implementada a lógica de particionamento, onde cada tarefa possui um identificador exclusivo, esse valor passa por um algoritmo de *hash* (*SHA-256*) e em seguida é dividido pelo valor que representa a quantidade total de *Schedulers* ativos naquele momento, o resto dessa divisão resulta no índice do *Scheduler* que é responsável por agendar a tarefa em questão. A Figura 6 exemplifica a lógica de particionamento.

Figura 6 - Particionamento de tarefas



Fonte: Autoria própria (2025)

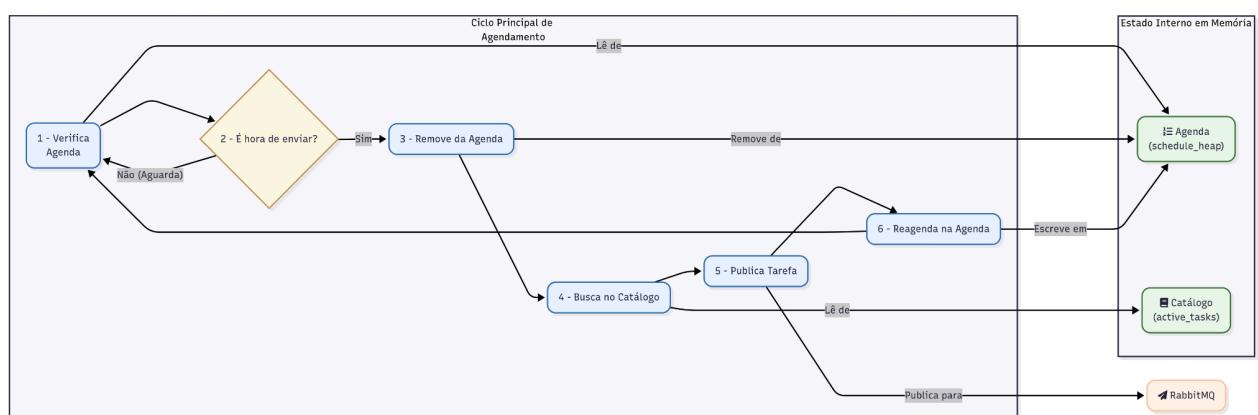
Na Figura 6 é mostrado um exemplo onde o total de *Scheduler* ativos é 3 e duas tarefas A e B foram recentemente inseridas. Considerando o ponto de vista do *Scheduler-0*, será de sua responsabilidade as tarefas onde o resto da divisão do hash do identificador pelo total de *Schedulers* que resultar em 0. No caso da Tarefa B, o resultado foi 1, isso significa que o *Scheduler-1* é responsável.

5.2.1.2 CICLO PRINCIPAL DE AGENDAMENTO

O ciclo de agendamento é responsável pelo funcionamento básico do componente, é aqui onde as tarefas são enviadas para a fila de mensagem. Inicialmente, no protótipo do *Scheduler*, ele iria criar uma thread para cada tarefa de sua responsabilidade, na maior parte do tempo as threads ficariam inativas esperando o tempo de intervalo entre o envio das tarefas. Essa abordagem inicial mostrou-se um problema apenas quando uma quantidade massiva de tarefas eram atribuídas, pois facilmente o código alcançava o limite de criação de novas *threads* e parava de escalar, já que nunca ficaria sobrecarregado o suficiente para o HPA decidisse aumentar o número de réplicas do *Scheduler*.

Surgiu então a ideia de reduzir drasticamente o número de *threads* do *Scheduler*, antes existia uma para cada tarefa, agora apenas uma thread seria responsável por agendar todas tarefas. Isso foi possível com o apoio de uma fila de prioridade, onde é armazenado o identificador da tarefa e uma *timestamp* contendo o horário onde o próximo envio daquela tarefa deve acontecer, vale destacar que a principal função de uma fila de prioridade é ordenar os elementos com base em algum critério, que nesse caso é a *timestamp*. Outra variável guardada na memória é o dicionário, que contém detalhes de todas tarefas que são responsabilidade do *Scheduler* em que estão, é apenas uma estrutura de dados comum que possui o identificador como chave e todos outros dados da tarefa como valor. A Figura 7 mostra o fluxo do ciclo de agendamento.

Figura 7 - Ciclo de agendamento



Fonte: Autoria própria (2025)

O ciclo principal funciona de acordo com a Figura 7, primeiro é lido da fila de prioridade, que chamaremos de Agenda, a tarefa com o menor tempo até a próxima execução, então o código precisa esperar chegar o momento certo e faz uma sequência de ações: remove da Agenda a tarefa, lê os outros detalhes da tarefa no dicionário, enviar a mensagem no

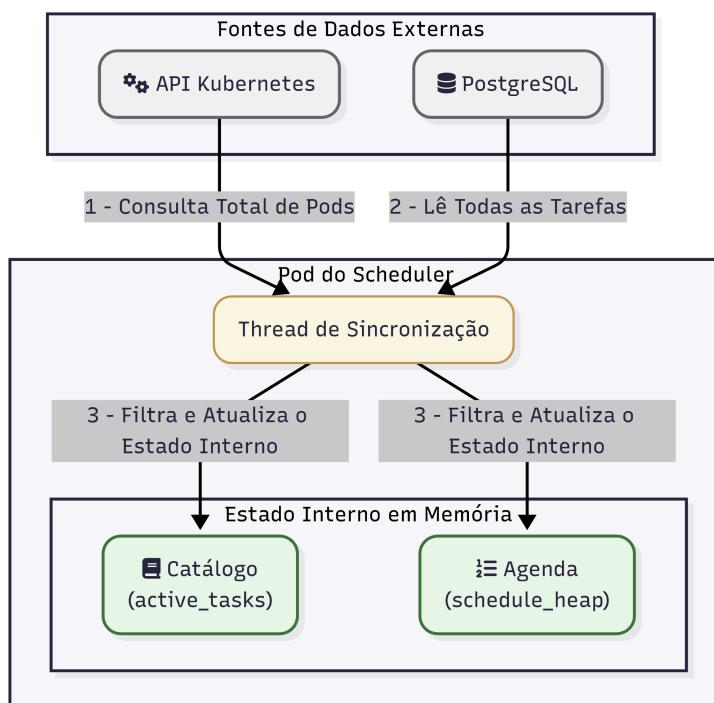
RabbitMQ e calcula a próxima execução para reinserir na Agenda. Esse ciclo se repete durante toda vida útil do *Pod*. Graças a abordagem de unificar todas tarefas em uma única *thread*, o *Scheduler* ficou otimizado ao ponto de raramente ser necessário mais de uma réplica em funcionamento no *Cluster*.

5.2.1.3 SINCRONIZAÇÃO PERIÓDICA

Para que o particionamento funcione corretamente, cada *Scheduler* precisa ter conhecimento da quantidade de outros *Schedulers* ativos no cluster e também quais são todas tarefas existentes. Essa responsabilidade está atribuída a uma *thread* em segundo plano, que é executada em um intervalo definido de tempo. Esse processo é crucial para garantir a consistência no agendamento de tarefas entre os *Schedulers*, é o pilar da escalabilidade e adaptação do sistema.

O fluxo de sincronização mostrado na Figura 8 realiza duas ações principais: primeiro consulta a API do *Kubernetes* para obter o número atual de *pods* ativos do *Scheduler*, a segunda ação acontece quando for detectado uma alteração na variável que representa a quantidade de *pods* ativos em relação a quantidade anterior antes da checagem. Com esse fato em mãos, a *thread* precisa consultar o banco de dados e reavaliar a posse de cada tarefa aplicando a lógica de particionamento ilustrada na Figura 6. Tarefas que não pertencerem mais à instância do *Scheduler* serão removidas do catálogo e da agenda, e novas tarefas de sua responsabilidade serão adicionadas e agendadas.

Figura 8 - Sincronização entre Schedulers



Fonte: Autoria própria (2025)

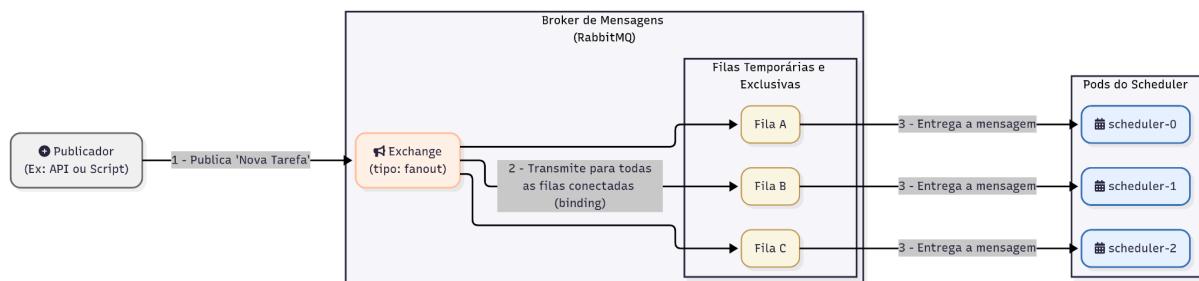
Esse mecanismo garante que o sistema se recupere de falhas e se adapta dinamicamente a eventos de escalonamento, mantendo a consistência em todo o *cluster*.

5.2.1.4 SINCRONIZAÇÃO DE NOVAS TAREFAS

Além da sincronização periódica entre *Schedulers*, o sistema precisava de uma forma de reagir dinamicamente à criação de novas tarefas, sem precisar esperar uma nova alteração na quantidade de réplicas e sem precisar consultar todo banco de dados desnecessariamente. Para resolver isso, foi implementado a segunda *thread* em segundo plano, que atua como ouvinte de notificações em tempo real. Este processo é semelhante ao que acontece com o *Worker*, que consome a fila de mensagens populada pelo *Scheduler*.

Isso faz com que todo *Scheduler* também seja consumidor de uma fila do *RabbitMQ*, mas não a mesma fila onde eles publicam. Para esta finalidade foi criada uma exchange do tipo *fanout* onde cada *Scheduler*, ao iniciar, cria uma fila temporária e exclusiva conectada nessa *exchange*. Quando uma mensagem é transmitida nessa *exchange*, ela é repassada para todas as filas conectadas, ou seja, cada *Scheduler* recebe uma cópia da mesma mensagem. Dessa forma cada um deles pode aplicar o algoritmo de particionamento, isso garante com que o *Scheduler* responsável pela tarefa seja notificado com a mensagem e adicione imediatamente à sua agenda, se a mensagem fosse de uma *exchange direct*, haveria o risco da mensagem se perder e nunca chegar no *Scheduler* correto, ocasionando uma inconsistência no agendamento das tarefas. O fluxo completo deste processo de notificação é ilustrado na Figura 9.

Figura 9 - Recebimento dinâmico de novas tarefas



Fonte: Autoria própria (2025)

É importante notar a flexibilidade desta abordagem. No estado atual do projeto, a publicação desta mensagem de notificação é realizada por um *script* auxiliar, no entanto, em um sistema de produção, essa tarefa seria tipicamente acionada por meio de uma API de gerenciamento de tarefas. Esse mesmo padrão poderia ser estendido para funcionar com outras operações além da criação de novas tarefas, como a remoção ou a modificação nas tarefas, assim complementando o CRUD (*CREATE*, *READ*, *UPDATE* e *DELETE*), dessa forma aperfeiçoando o sistema e garantindo consistência em tempo real dos *Schedulers* em todo *cluster*.

5.2.2 WORKER

Enquanto *Scheduler* agenda as tarefas na fila de mensagens, o *Worker*, do outro lado do sistema, atua como consumidor e representa a força de trabalho, sendo o componente responsável pela execução real das tarefas. Para entender como o *Worker* funciona, é

importante definir a estrutura de dados que ele processa: a tarefa. Cada tarefa no sistema é representada por um conjunto de campos padronizados, conforme mostrado no Quadro 1.

Quadro 1 - Estrutura de dados da tarefa

Campo	Tipo de Dado	Descrição
<i>id</i>	<i>INTEGER</i>	Identificador único sequencial no banco de dados.
<i>task_uuid</i>	<i>UUID</i>	Identificador único universal, usado para o particionamento.
<i>task_name</i>	<i>TEXT</i>	Nome descritivo da tarefa.
<i>task_type</i>	<i>TEXT</i>	Define o tipo de operação a ser executada pelo <i>Worker</i>
<i>payload</i>	<i>JSONB</i>	Contém os dados específicos necessários para a execução da tarefa.
<i>interval_seconds</i>	<i>INTEGER</i>	O intervalo em segundos entre cada execução da tarefa.

Fonte: Autoria própria (2025)

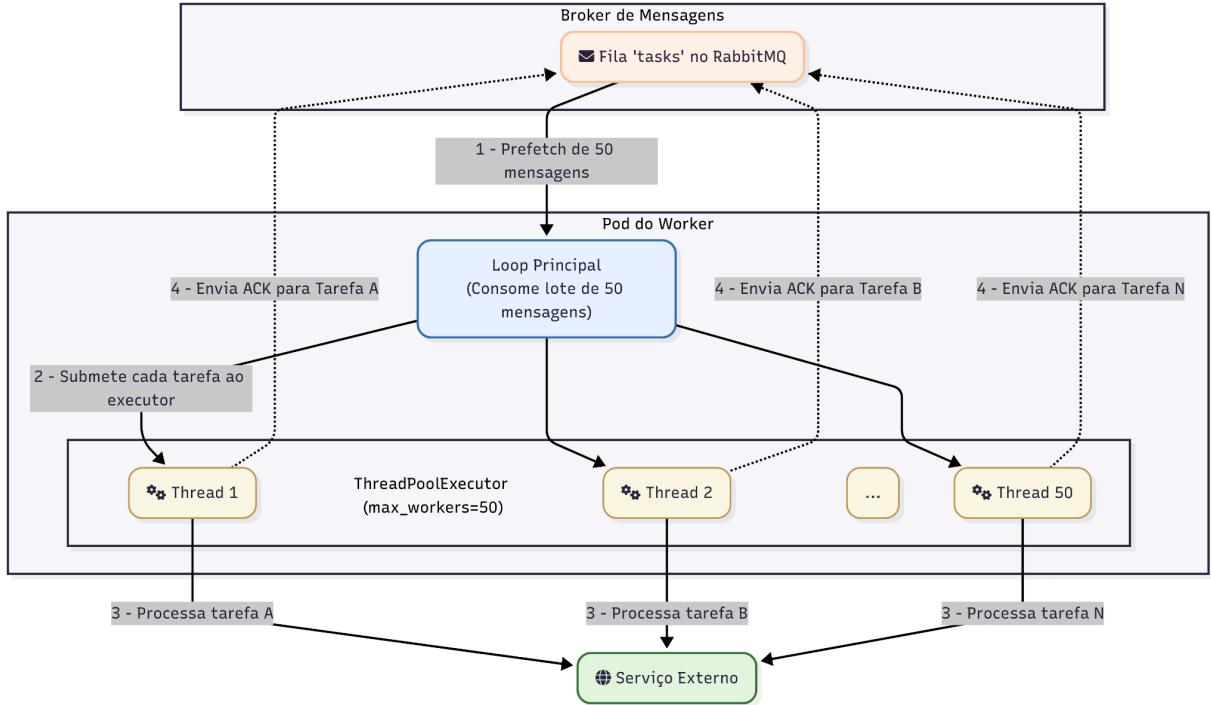
É crucial notar que a arquitetura do SentryWeb foi pensada para ser genérica e extensível. O campo *task_type* nas tarefas serve para permitir um polimorfismo entre as tarefas, onde cada *Worker* pode ser especializado em um tipo específico de tarefa e lidar com operações totalmente distintas. Futuramente seria possível, por exemplo, implementar esses *Workers* especializados e separar filas de mensagens para cada tipo de tarefa, onde através do uso de uma *exchange* de tópicos, separar o envio dessas mensagens de acordo com a necessidade atual. No entanto, para o escopo do trabalho atual e como forma de validar a arquitetura de um lado ao outro, foi implementado um único tipo de tarefa: a verificação de disponibilidade de um *endpoint*. Nesse tipo de tarefa, o *task_type* recebe o valor “*url_check*” e o campo *payload* contém o endereço a ser verificado. e o *Worker* executa a requisição HTTP (*Hypertext Transfer Protocol*) retornando sucesso se o código de *status* for 200, que significa que OK. A Figura 10 exemplifica o funcionamento de um *Worker*.

Com a estrutura de tarefas bem definida, o *Worker* foi implementado com foco apenas em processar o maior número de mensagens contendo as tarefas. Diferente do *Scheduler*, o *Worker* é um serviço *stateless* (sem estado), isso significa que ele não é gerado seguindo um padrão e ordem de nomes que é necessário no *Scheduler*. Os *Workers* são gerenciados por um *Deployment*, cada um deles funciona independentemente dos outros, o que facilita o seu escalonamento já que todos eles podem ser considerados iguais e não precisam fazer nada além de se conectar a fila de mensagens e consumir.

Conforme a Figura 10, que mostra o funcionamento interno do *Worker*, podemos notar que ele é bem mais simples que um *Scheduler*, possui apenas o ciclo principal. Primeiramente o fluxo consome mensagens da fila, podendo armazenar simultaneamente até 50 delas. Em seguida, divide as mensagens recebidas com o auxílio de um *ThreadPoolExecutor* configurado com 50 *threads*, assim sendo capaz de processar múltiplas

tarefas em paralelo. Cada *thread*, ao receber a mensagem, vai extrair os dados e executar a tarefa de acordo com *task_type* e o conteúdo do *payload*, que em todos cenários atuais serão o de verificar o endereço, que na Figura 10 é representada pelo acesso a um serviço externo. Depois de executar a tarefa, a *thread* confirma manualmente (*ACK/NACK*) o recebimento da mensagem diretamente com o *RabbitMQ* para garantir que nenhuma tarefa seja perdida em caso de falha na comunicação.

Figura 10 - Funcionamento interno do Worker



Fonte: Autoria própria (2025)

Atualmente, o resultado de cada verificação é registrado nos logs do *Worker* visando a observabilidade. Porém, a arquitetura é projetada para ser um ponto de extensão para outras funcionalidades. O campo *payload* pode conter instruções para ações a serem tomadas com base no resultado da verificação. Por exemplo, no caso de falha da checagem, o *Worker* poderia publicar uma mensagem em outra fila do *RabbitMQ* para acionar um sistema de alertas.

5.3 ESTRATÉGIA DE AUTOESCALONAMENTO

Nesta subseção, o funcionamento do *Horizontal Pod Autoscaler* (HPA) será detalhado, mostrando como ele foi aplicado no *Scheduler* e no *Worker* para gerenciar o autoescalonamento horizontal de *pods*. É nesse ponto do projeto que localiza-se um dos principais objetivos, tornar o sistema auto escalável, isso faz com que o sistema seja eficiente no uso de recursos do *cluster*, pois controla as réplicas dinamicamente conforme a necessidade atual. Para compreender as estratégias adotadas, é necessário primeiro entender os conceitos de gerenciamento de recursos e o comportamento de escalonamento no *Kubernetes*.

5.3.1 GERENCIAMENTO DE RECURSOS: REQUESTS E LIMITS

Para que o *Kubernetes* possa gerenciar aplicações de forma eficaz, é fundamental informar ao orquestrador os recursos que cada *container* necessita. Isso é feito através de duas diretivas no manifesto de um *Pod*: *requests* e *limits*. Para o uso de CPU, é usada a medida em *milicores*, onde 1000m (1000 *milicores*) representa 1 núcleo de CPU. O uso de memória é medido em *Mebibytes*, por exemplo 256Mi (256 *Mebibytes*).

- *Requests*: Define a quantidade mínima de recursos (CPU e memória) que o *Kubernetes* garante para aquele *container*. O escalonador do *Kubernetes* utiliza esse valor para decidir em qual nó do *cluster* um novo *Pod* será alocado. Para o autoescalonamento, o HPA utiliza o valor de *requests* como base para calcular a utilização percentual de CPU ou memória (KUBERNETES, 2025).
- *Limits*: Ao contrário das *requests*, *limits* definem a quantidade máxima de recursos que um *container* pode consumir. Um *container* não consegue exceder o limite de CPU, seu desempenho será limitado sempre que atingir esse valor. No caso do uso de memória, o *pod* sofrerá uma morte súbita.

Essas definições são cruciais para que o HPA possa trabalhar e calcular um número de réplicas ideal. Um cuidado especial é necessário com o uso de memória, pois não é ideal que *pods* sejam reiniciados constantemente por conta de um mau dimensionamento de recursos.

5.3.2 AJUSTE DO ESCALONAMENTO: BEHAVIOR

O HPA possui o campo *behavior*, que permite manter um controle granular sobre a taxa de escalonamento. O principal objetivo é evitar oscilações bruscas e constantes no número de réplicas, por exemplo, o número de réplicas aumenta rapidamente para diminuir o peso do trabalho e logo em seguida oscila para baixo como resposta ao alívio concedido pela réplica recém criada.

Com o campo *behavior*, é possível ajustar políticas de tempo separadas de *scaleUp* e *scaleDown*, aumento e diminuição de réplicas. Em cada uma delas, o principal parâmetro é o *stabilizationWindowSeconds*, ela funciona como uma “memória” para o HPA. Para uma ação de *scaleUp*, por exemplo, o HPA espera que as métricas sugiram o aumento de réplicas em todos momentos da janela de estabilização, caso o comportamento dos pods não seja constante, o HPA não irá agir. O mesmo princípio se aplica ao *scaleDown*, o HPA irá monitorar para ter certeza de que o nível da carga de trabalho se mantém estável antes de reduzir as réplicas (KUBERNETES, 2025).

5.3.3 APLICAÇÃO DA ESTRATÉGIA NO SCHEDULER

Depois da otimização que reduziu o uso de *threads*, o *Scheduler* passou a trabalhar com o agendamento de um número maior de tarefas e com um consumo menor de recursos. Isso permitiu que o HPA com métricas tradicionais, como uso de CPU e memória, se mostrasse uma estratégia eficiente. O ciclo principal do *Scheduler* se resume em agendar as tarefas, boa parte do tempo ele passa boa parte do tempo aguardando o horário de cada uma

das tarefas que estão armazenadas na memória. Logo, é evidente que quanto maior o número de tarefas atribuídas a um *Scheduler*, mais recursos ele irá consumir para gerenciar a fila de prioridades e o dicionário de tarefas.

Dessa forma, o HPA do *Scheduler* foi configurado com alvos de utilização de CPU e memória. Por exemplo, considerando que o alvo de utilização de CPU foi definido em 80%, e a requisição desse recurso está 150m de CPU (15% de um núcleo) por *Scheduler*, quando essa instância ultrapassar o uso de 120m de CPU, o HPA irá automaticamente iniciar o processo de escalonamento, criando uma nova réplica para reduzir o consumo médio de CPU por *Scheduler*.

Seu principal objetivo é garantir que o número de tarefas gerenciadas não cresça a um ponto que sobrecarregue uma única instância do *Scheduler*, antes disso ele deve ajustar o número de réplicas para que finalmente, elas possam redistribuir a responsabilidade de tarefas entre si, de acordo com o que é mostrado nas Figuras 6 e 8.

5.3.4 APLICAÇÃO DA ESTRATÉGIA NO WORKER

Diferente do *Scheduler*, a estratégia de autoescalonamento do *Worker* não respondeu adequadamente com o uso das métricas tradicionais de CPU e memória. Pelo fato de o *Worker* passar a maior parte de seu tempo em inatividade aguardando respostas das requisições HTTP para concluir as tarefas, o uso de CPU permanecia baixo, principalmente em momentos de instabilidade da rede, o que aumentava ainda mais o tempo de espera. Esse comportamento causava um aumento de mensagens acumuladas na fila de mensagens, pois o HPA constantemente reduzia o número de réplicas por conta do baixo uso de CPU, tendo o comportamento contrário do desejado.

A solução foi desacoplar o escalonamento de recursos com base em CPU e ligá-los diretamente na métrica que importa, a fila de mensagem, mais especificamente ao número de mensagens acumuladas nela. Para isso, foi necessário expor as métricas do *RabbitMQ* e implementar uma estratégia personalizada utilizando o *Prometheus*, uma ferramenta que monitora e coleta métricas do *cluster*. Em conjunto do *Prometheus Adapter*, que traduz os dados de forma que o HPA entenda, foi possível utilizar as métricas externas *rabbitmq_queue_messages_ready* e *rabbitmq_queue_messages_unacked*, que representam respectivamente o número de mensagens prontas para serem lidas e o número de mensagens que foram entregues, mas que não tiveram o processamento confirmado pelo *Worker*. Com isso, o HPA aumenta o número de réplicas quando a fila está com acúmulo de mensagens.

6 RESULTADOS

Nesta seção, são apresentados os resultados práticos obtidos com a implementação do SentryWeb. O objetivo é validar o funcionamento da arquitetura, analisar a eficiência da estratégia de autoescalonamento e discutir os resultados observados durante os testes com cargas simuladas.

6.1 AMBIENTE DE VALIDAÇÃO

Todos os testes e validações foram realizados em um ambiente de desenvolvimento local, configurado para simular um *cluster Kubernetes*. As ferramentas e versões utilizadas foram:

- **Orquestrador:** *Minikube* v1.36.0 (Executando *Kubernetes* v1.33.1).
- **Container Runtime:** *Docker* 28.1.1.
- **Gerenciador de Pacotes:** *Helm* v3.18.0.
- **Stack de Monitoramento:** *Kube-Prometheus-Stack*.
 - **Prometheus:** v2.51.2.
 - **Prometheus Adapter:** v0.11.2.

A máquina física usada neste ambiente para desenvolvimento possui as seguintes especificações:

- Processador: 13th Gen Intel(R) Core(TM) i5-13420H.
- Memória: 2x 8GB dddr5 5200MT/s.
- Disco: SSD NVMe SKHynix_HFS512GEJ9X115N 477GB.
- Gráficos: Intel(R) UHD Graphics e NVIDIA GeForce RTX 3050 6GB Laptop.

Todo uso de recursos foram reservados para o *cluster Kubernetes* e utilização da máquina não teve outros fins durante os experimentos.

6.2 VALIDAÇÃO FUNCIONAL DO FLUXO DE TAREFAS

O primeiro passo é validar o fluxo de trabalho básico do sistema, garantindo que uma tarefa possa ser agendada por um *Scheduler* e executada por um *Worker*. Para isso, uma tarefa de verificação de URL foi inserida no banco de dados *PostgreSQL* e na exchange *fanout* da qual o *Scheduler* é consumidor.

A Figura 11 é o *log* do *Scheduler* do momento em que o ciclo principal inicia. Nela vemos o recebimento dinâmico da tarefa via *RabbitMQ* e o agendamento para fila dos *Workers* logo em seguida às 20:54:10.

Figura 11 - Log do scheduler recebimento e envio da tarefa

```
[2025-07-15 20:52:31] [scheduler-0] [INFO] Iniciando o loop de agendamento principal.  
[2025-07-15 20:52:50] [scheduler-0] [INFO] Recebeu nova tarefa c1d70d4e-3e8f-42e3-93f6-472914bc2049 via RabbitMQ – é minha, agendando...  
[2025-07-15 20:54:10] [scheduler-0] [DEBUG] Tarefa c1d70d4e-3e8f-42e3-93f6-472914bc2049 agendada para execução em 90 segundos.  
[2025-07-15 20:54:10] [scheduler-0] [INFO] Agora gerenciando 1 tarefas ativas.  
[2025-07-15 20:54:10] [scheduler-0] [INFO] Enviando tarefa: Nome: url_example_single, uuid: c1d70d4e-3e8f-42e3-93f6-472914bc2049, id (db): 1
```

Fonte: Autoria própria (2025)

Na Figura 12 vemos o *log* do recebimento da tarefa, ele contém algumas informações da tarefa e o resultado da checagem no *endpoint*.

Figura 12 - Log do worker recebimento da tarefa

```
[2025-07-15 20:54:11] [worker] ===== Tarefa Recebida =====
  Nome: url_example_single
  Tipo: url_checker
  ID (uuid): c1d70d4e-3e8f-42e3-93f6-472914bc2049
  ID (db): 1
  Payload: {'url': 'https://example.com'}
  Resultado: True
  Tempo de execução: 1.32 segundos
  Finalizou tarefa.
```

Fonte: Autoria própria (2025)

Através da observação desses *logs* dos pods do *Scheduler* e *Worker* mostrados respectivamente nas Figuras 11 e 12, foi possível confirmar o ciclo completo onde o *Scheduler* recebe a responsabilidade de agendar a tarefa e insere na fila de mensagens logo em seguida, depois o *Worker* do outro lado da fila, recebe a mensagem e executa a tarefa.

6.3 VALIDAÇÃO DA ESTRATÉGIA DE AUTOESCALONAMENTO

Nesta seção será validado o autoescalonamento dos *Pods Scheduler* e *Worker*, onde cada um deles será submetido a condições semelhantes para demonstrar a capacidade de adaptação do sistema.

6.3.1 COMPORTAMENTO DO SCHEDULER SEM HPA

Esse experimento tem o objetivo de observar como o consumo de recursos que o *Pod* do *Scheduler* se comporta com diferentes cargas de trabalho, mapeando qual é o recurso mais relevante para o desempenho do *Pod*. Com o resultado, será possível definir um bom alvo de utilização de recursos para o funcionamento do HPA.

Neste primeiro cenário, o *StatefulSet* do *Scheduler* está fixo com uma réplica (HPA desligado), as definições de *requests* são 150m de CPU e 256Mi de memória, os *limits* estão configurados como o dobro do *requests*. Foram atribuídas ao *Scheduler* cargas de trabalho crescentes, representando o número de tarefas sob sua responsabilidade, com cada tarefa configurada para um intervalo de 60 segundos. O consumo de CPU e memória foi monitorado para cada nível de carga e considerado o consumo médio após a estabilização do *Pod* em intervalos de 15 minutos entre o aumento das cargas de trabalho.

Conforme os dados da Tabela 1, podemos observar um *Scheduler* é capaz de gerenciar 25000 tarefas usando em média 115m de CPU, que representa cerca de 77% de consumo em relação aos 150m que foram definidos como *requests* do *Pod*. O uso de memória não demonstrou um crescimento significativo relacionado a carga de trabalho, o máximo atingido durante os testes foi o uso de 105Mi, distante do request de 256Mi, logo a métrica de CPU será mais eficaz para o autoescalonamento.

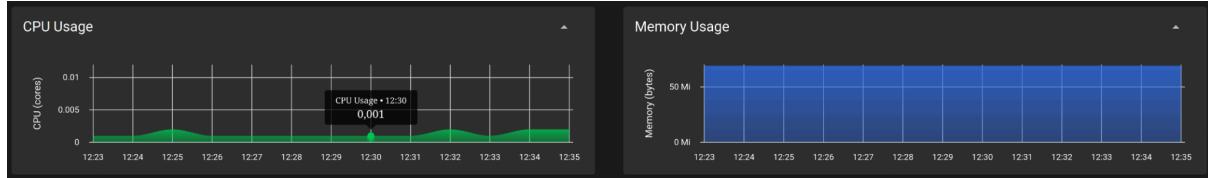
A Figura 13 representa consumo de recursos em relação ao tempo, neste momento o *Scheduler* estava com a carga de 100 tarefas, tendo um consumo de CPU que varia de 1m até 2m, este é um consumo mínimo, pois representa apenas 0,001% de um núcleo de CPU.

Tabela 1 - Consumo médio de recursos do Scheduler por carga de trabalho

Carga de tarefas	CPU Média (m)	Memória Média (MiB)
100 tarefas	2m	80Mi
500 tarefas	4m	80Mi
1000 tarefas	6m	80Mi
2000 tarefas	12m	80Mi
5000 tarefas	25m	90Mi
10000 tarefas	47m	90Mi
15000 tarefas	75m	100Mi
20000 tarefas	100m	100Mi
25000 tarefas	115m	105Mi

Fonte: Autoria própria (2025)

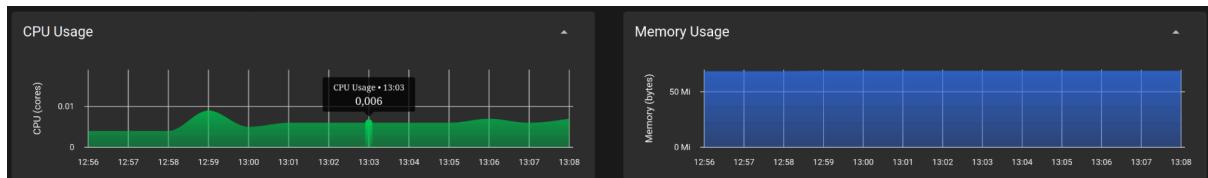
Figura 13 - Consumo de recursos do Scheduler-0 com 100 tarefas



Fonte: Autoria própria (2025)

Na Figura 14 podemos ver o histórico de consumo que aponta para o uso médio de 6m de CPU com 1000 tarefas. É possível notar um pico no uso de CPU às 12:59, esse pico é causado pelo recebimento repentino de um alto número de tarefas em relação a carga anterior, de 500 tarefas.

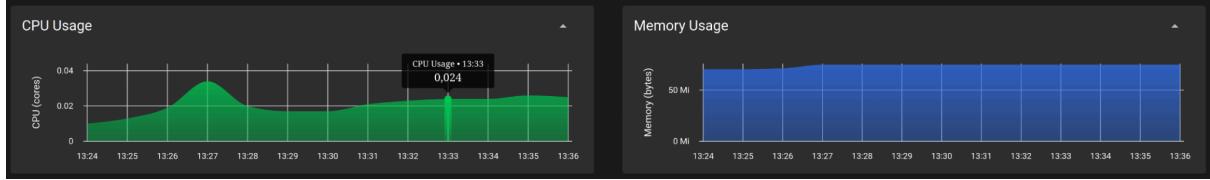
Figura 14 - Consumo de recursos do Scheduler-0 com 1000 tarefas



Fonte: Autoria própria (2025)

O mesmo padrão se repete na Figura 15, que mostra o momento que a carga de trabalho aumentou de 2000 tarefas para 5000 tarefas. Para cada nova tarefa recebida, o *Scheduler* aplica o algoritmo de particionamento de tarefas, que justifica o pico no uso de CPU às 13:27, mas que nos minutos seguintes tende a se estabilizar em 25m de CPU.

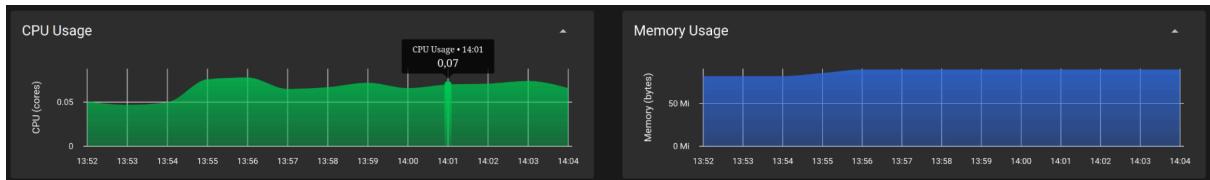
Figura 15 - Consumo de recursos do Scheduler-0 com 5000 tarefas



Fonte: Autoria própria (2025)

Com 15000 tarefas, o consumo médio de CPU se estabiliza em 75m, conforme mostrado na Figura 16.

Figura 16 - Consumo de recursos do Scheduler-0 com 15000 tarefas



Fonte: Autoria própria (2025)

É notório o padrão que se seguiu durante todo o experimento, onde o consumo de CPU sofre picos de uso nos momentos que a carga aumenta e na estabilização que vem em seguida. Essa estabilização foi usada como a média para o consumo de cada nível de carga mostrados na Tabela 1.

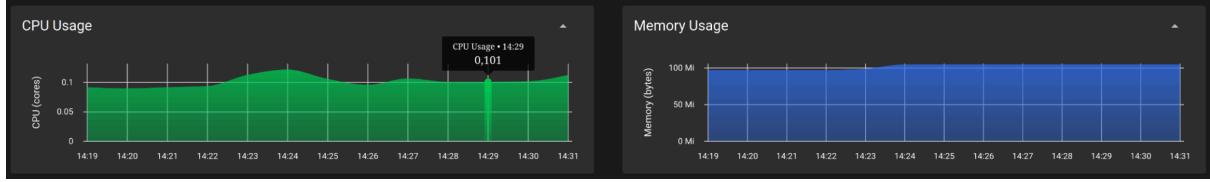
6.3.2 COMPORTAMENTO DO SCHEDULER COM HPA

O segundo experimento visa validar o funcionamento do HPA, observando sua capacidade de ajustar dinamicamente o número de réplicas do *Scheduler* em resposta à carga de trabalho, e como isso impacta a distribuição do consumo de recursos entre as instâncias.

Foi definido então como alvo do HPA o monitoramento de CPU na faixa de 70% de uso com 2 minutos de estabilização, isso significa que durante essa janela de tempo o consumo se manter acima de 70% (105m de CPU) definidos no request do *Pod*, ele irá agir para que o número de réplicas aumente.

Neste segundo cenário foi seguido o mesmo padrão do primeiro experimento, com a diferença que o HPA está ativado, ou seja, ele deve aumentar o número de réplicas se o consumo ultrapassar 105m de CPU constantemente por mais de 2 minutos. Na Figura 17 podemos observar o consumo dos recursos ao longo do tempo, um leve pico no uso de CPU acontece às 14:23, esse momento representa o aumento do nível na carga de trabalho do *Scheduler-0*. Antes desse pico existiam 20000 tarefas em sua responsabilidade e outras 5000 foram inseridas, o aumento repentino são os cálculos de particionamento sendo aplicados em cada tarefa.

Figura 17 - Consumo de recursos do Scheduler-0 com 25000 tarefas



Fonte: Autoria própria (2025)

Ao longo do tempo o consumo de CPU do *Scheduler-0* variou entre 100m e 120m de CPU. A Figura 18 captura o momento onde essa variação se manteve estável durante 2 minutos, e logo em seguida acontece uma queda no consumo que se estabiliza em cerca de 80m de CPU, a diminuição aconteceu por conta da criação de um novo *Pod*, o *Scheduler-1*. Cada *Scheduler* assumiu responsabilidade de gerenciar uma porção diferente de tarefas, aliviando o consumo de recursos que estavam concentrados no *Scheduler-0*.

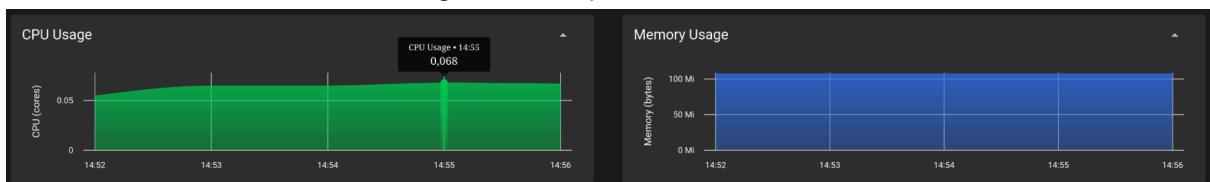
Figura 18 - Estabilização no consumo de recursos do Scheduler-0 com 25000 tarefas



Fonte: Autoria própria (2025)

Na Figura 19 podemos ver o consumo de CPU do *Scheduler-1* crescendo e eventualmente estabilizando e confirmando o funcionamento correto do HPA, que aumentou o número de réplicas como resposta ao consumo de recursos da carga de trabalho com 25000 tarefas e dividiu a responsabilidade entre os dois *Schedulers*.

Figura 19 - Criação do Scheduler-1



Fonte: Autoria própria (2025)

Com isso, o segundo experimento alcançou o objetivo de validar o funcionamento do HPA configurado para o *Scheduler*.

6.3.3 COMPORTAMENTO DO WORKER SEM HPA

O terceiro experimento tem como objetivo mapear o comportamento de um *Worker* com diferentes cargas de trabalho aplicadas sobre ele de forma semelhante ao primeiro experimento com o *Scheduler*. Cada tarefa do *Worker* é checar um *endpoint* externo ao *cluster* e verificar o resultado da requisição, ou seja, requer conexão com internet e fica evidente que o desempenho do *Pod* está ligado ao tempo de resposta do *endpoint*.

Durante todo experimento a máquina estará conectada via cabo a internet com 800Mbps (800 Megabits de por segundo) de *download* e 200Mbps de *upload* para evitar oscilações extras no tempo de resposta provindas de uma conexão *wireless*. As especificações de um Pod *Worker* de *requests* e *limits* de CPU serão respectivamente 500m e 1000m, para *requests* e *limits* de memória serão 256Mi e 512Mi respectivamente.

As cargas de trabalho serão medidas pela quantidade de tarefas inseridas por segundo na fila de mensagem. Além disso, este experimento estará dividido em três etapas:

- ***endpoint local***: todas tarefas serão fixas verificar um endereço local hospedado na mesma máquina dentro do *cluster* para tirar a variação da conexão com a internet.
- ***endpoint fixo***: todas tarefas serão fixadas em um único endereço externo, deixando a variação da conexão acontecer, mas limitada a uma única hospedagem.
- ***endpoints variados***: tarefas terão endereços externos variados, tornando o comportamento mais imprevisível.

Essas abordagens foram adotadas para testar o desempenho máximo de um único *Worker* em vários cenários e servir de base para um futuro experimento com o HPA.

6.3.3.1 ENDPOINT LOCAL

Começando pela primeira etapa do experimento, todas tarefas terão um *endpoint* local, a resposta da requisição HTTP será quase instantânea. Com isso, o *Worker* não será limitado pela velocidade da conexão à internet e todo seu potencial poderá ser explorado.

Na Figura 20 podemos confirmar o tempo de execução com a penúltima linha, que foi apenas 20 milissegundos. Também é possível ver o *endpoint* local, que está rodando em um *Pod* temporário com a imagem do *Nginx* na versão 1.25.3 (NGINX, 2025). O *Nginx* aqui está cumprindo o papel de um servidor *web* simples sem limitação de recursos na máquina.

Figura 20 - Log worker com tarefa de endpoint local no cluster

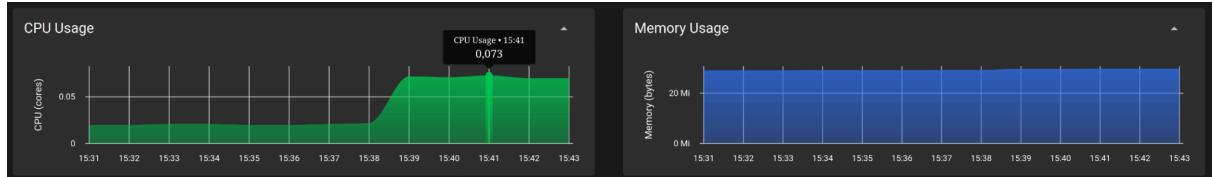
```
[2025-07-19 15:36:59] [worker] ===== Tarefa Recebida =====
Nome: check_endpoint_local
Tipo: url_checker
ID (uuid): ab9cb1f2-9392-4157-a5f3-2e0f0b4a69ca
ID (db): 5
Payload: {'url': 'http://test-endpoint-service.sentryk8s-dev.svc.cluster.local'}
Resultado: True
Tempo de execução: 0.02 segundos
Finalizou tarefa.
```

Fonte: Autoria própria (2025)

Para validar o desempenho do *Worker* foi feito o incremento da carga de trabalho de forma semelhante ao experimento do *Scheduler*. Porém agora temos o gráfico do fluxo de mensagens, representado na Figura 22, em adição do gráfico com consumo de recursos mostrado na Figura 21. O fluxo de mensagens foi resumido com duas linhas, a verde aponta para o número de mensagens que o *Worker* recebeu e processou, a linha amarela é o número

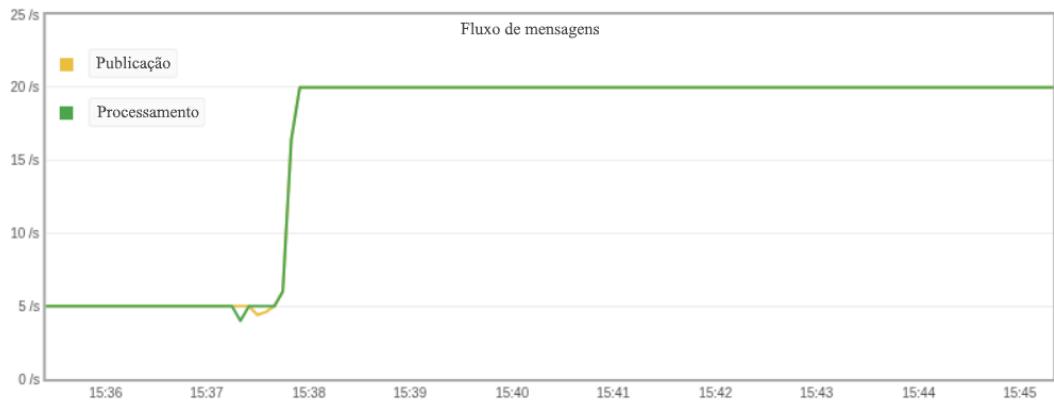
de mensagens que estão sendo publicadas pelo *Scheduler*. Na Figura 22 em questão, as duas linhas estão sobrepostas, indicando que o *Worker* está processando as tarefas ao mesmo tempo que são publicadas. O gráfico com o consumo de recursos na Figura 21 foi gerado simultaneamente ao gráfico anterior, nele vemos um padrão semelhante ao comportamento do *Scheduler*, onde o consumo de CPU sobe quando a carga de trabalho aumenta, com 20 tarefas por segundo esse consumo se estabiliza em cerca de 72m de CPU.

Figura 21 - Consumo de recursos do Worker com 20 tarefas por segundo



Fonte: Autoria própria (2025)

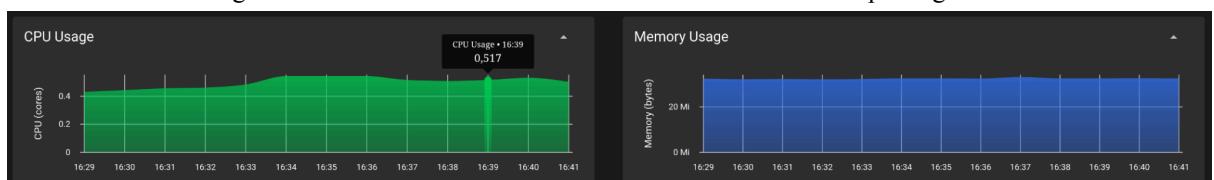
Figura 22 - Fluxo de mensagens com 20 tarefas por segundo



Fonte: Autoria própria (2025)

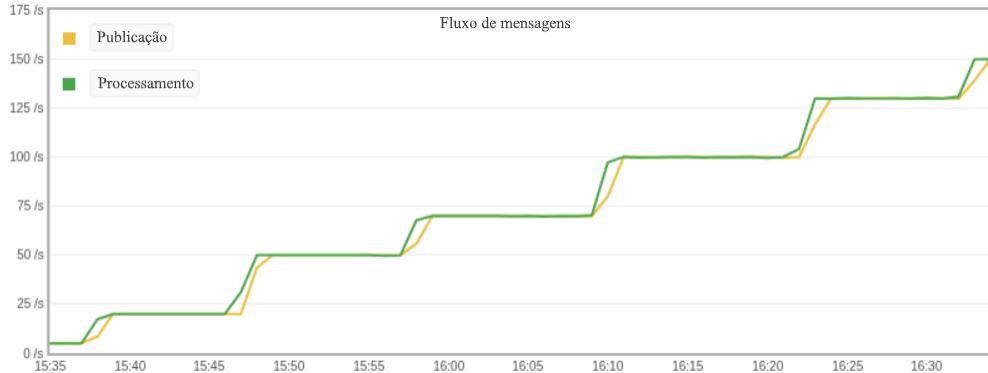
O mesmo processo foi feito incrementando a carga de trabalho em intervalos que duram em média 10 minutos para estabilização do fluxo de mensagens. Na Figura 24 confirmamos o histórico desse incremento dentro da janela de tempo de uma hora, podemos notar com mais clareza a linha amarela, que está sempre abaixo da linha verde, indicando que o processamento de tarefas pelo *Worker* ainda não alcançou o desempenho máximo. A Figura 23 confirma o consumo médio de CPU em 517m, consumo esse que já alcançou o *request* do *Pod*, que é de 500m de CPU e o fato do *limits* estar em 1000m (1 núcleo inteiro de CPU) aponta mais uma vez que ainda há potencial a ser extraído do *Worker*. O consumo de memória não se mostrou relevante nesta etapa, estando sempre no consumo médio de 35Mi de memória durante todos os incrementos na carga de trabalho.

Figura 23 - Consumo de recursos do Worker com 150 tarefas por segundo



Fonte: Autoria própria (2025)

Figura 24 - Histórico do fluxo de mensagens com topo em 150 tarefas por segundo



Fonte: Autoria própria (2025)

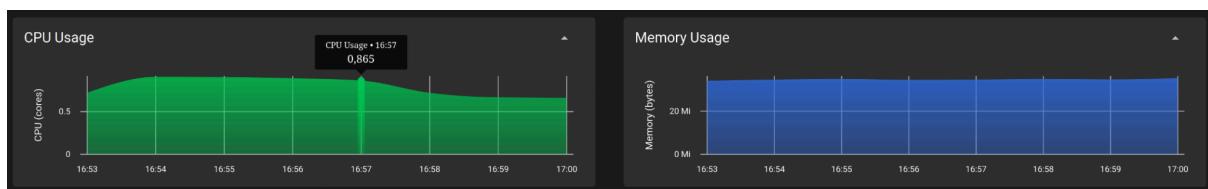
O desempenho do *Worker* se mostrou estável até a carga de 200 tarefas por segundo, quando esse nível foi incrementado para 250 tarefas por segundo o *Worker* começou a apresentar instabilidade no processamento das mensagens. A Figura 25 mostra picos irregulares na linha verde após às 16:52, o consumo de CPU já se mantinha acima de 70m, chegando a alcançar 90m em alguns momentos conforme mostrado na Figura 26.

Figura 25 - Fluxo de mensagens com 250 tarefas por segundo



Fonte: Autoria própria (2025)

Figura 26 - Consumo de recursos do Worker com 250 tarefas por segundo

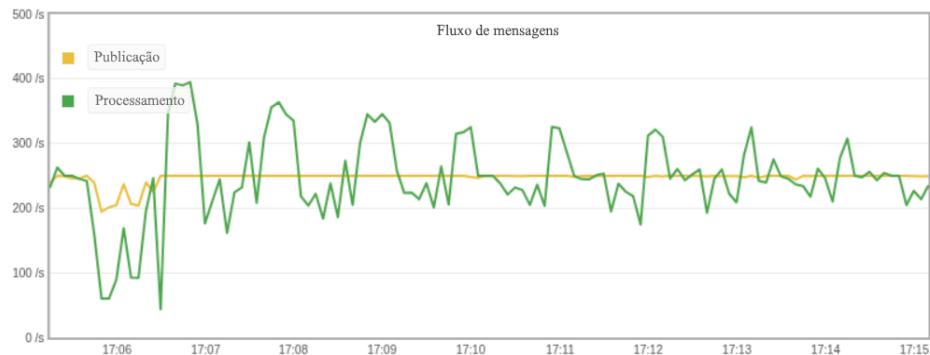


Fonte: Autoria própria (2025)

Nos momentos seguintes, o *Worker* instabilidades com a mesma carga de trabalho. Com a Figura 29 confirmamos que o consumo de CPU alcançou em alguns momentos o limite de 1000m que foi configurado ao *Pod*. Apesar do processamento de tarefas estar

levemente instável, conforme mostrado por picos e quedas na linha verde na Figura 27, o *Worker* ainda é capaz de suportar essa carga de trabalho com algumas ressalvas.

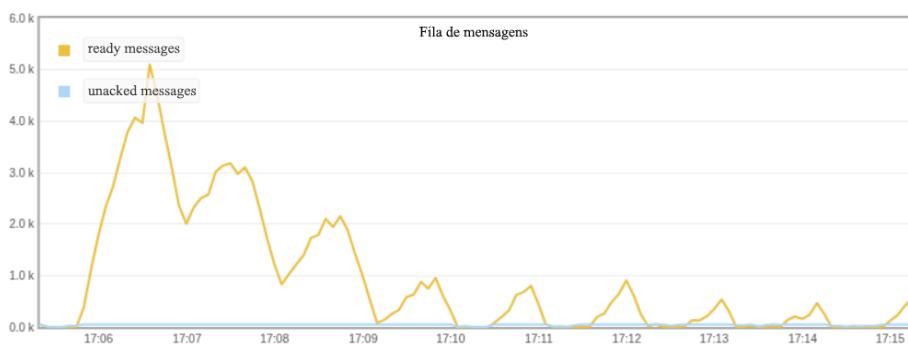
Figura 27 - Fluxo de mensagens levemente instável com 250 tarefas por segundo



Fonte: Autoria própria (2025)

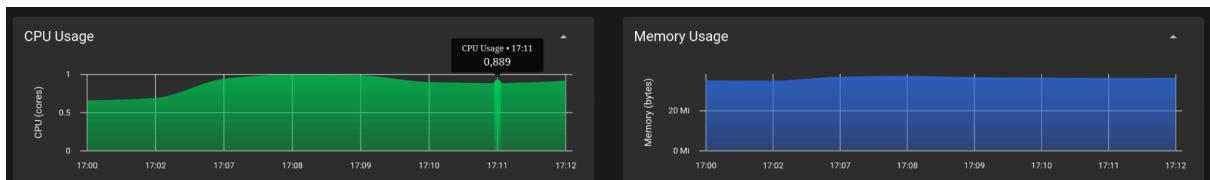
Até o presente momento foi visto apenas o gráfico do fluxo de mensagens, e o *Worker* processava quase que instantaneamente todas elas sem deixar que ficasse acumuladas, porém graças às instabilidades causadas pela carga de trabalho de 250 tarefas por segundo o *Worker* deixou as mensagens se acumularem ao longo de alguns minutos. Isso significa que a publicação de mensagens por segundo ultrapassou o número de mensagens processadas por segundo, na Figura 28 temos a introdução da fila de mensagens, lá ficam as mensagens que estão aguardando para serem consumidas pelo *Worker*, nela é possível ver duas linhas, a amarela que representa o número de mensagens na fila e a linha azul, que está quase imperceptível no gráfico, representa o número de mensagens que foram entregues ao *Worker*, mas que ainda não tiveram o processamento confirmadas.

Figura 28 - Fila de mensagens



Fonte: Autoria própria (2025)

Figura 29 - Consumo de recursos do Worker atingindo o limite



Fonte: Autoria própria (2025)

Essas mensagens não confirmadas são as mensagens que estão dentro das 50 *threads* do *Worker*, é exatamente por isso que o valor delas no gráfico é constante em 50 quando há mensagens acumuladas pois o *Worker* processa no máximo 50 delas ao mesmo tempo, conforme mostrado na Figura 10. Chamaremos a partir de agora as mensagens acumuladas de *ready* (linha amarela da fila de mensagens) e as mensagens não confirmadas de *unacked* (linha azul da fila de mensagens).

Nessa carga de trabalho de 250 tarefas por segundo o *Worker* atingiu o limite da sua performance, sendo capaz de suportar a carga apesar de não ter mais a estabilidade vista nas cargas anteriores. Ao incrementar a carga de trabalho para 300 tarefas o esperado é que a fila de mensagens cresça, pois o *Worker* não terá mais recursos de CPU para processar as mensagens excedentes.

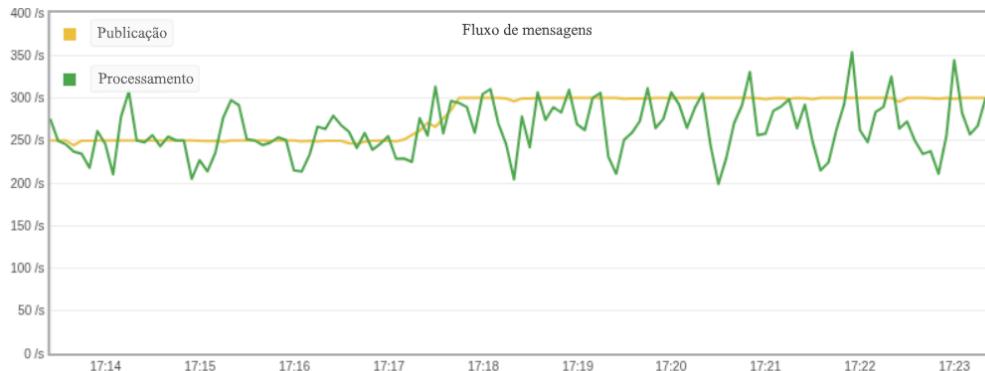
Conforme o esperado, as Figuras 30, 31 e 32 confirmaram a capacidade máxima do *Worker*. Na Figura 32 vemos o consumo de CPU constante em 1000m, ou seja, está usando o limite máximo de CPU reservado ao Pod. Na Figura 31 vemos que a linha verde de processamento não está mais acompanhando a linha amarela de publicação e na Figura 30 vemos a fila de mensagens crescendo constantemente. Em situações como essa, o HPA deveria aumentar o número de réplicas do *Worker*, mas neste experimento ele está desligado. Então o número de réplicas será incrementado para dois manualmente para reforçar o comportamento que será esperado no experimento da seção 6.3.4 com HPA.

Figura 30 - Fila de mensagens acumulando com carga de 300 tarefas por segundo



Fonte: Autoria própria (2025)

Figura 31 - Fluxo de mensagens com 300 tarefas por segundo



Fonte: Autoria própria (2025)

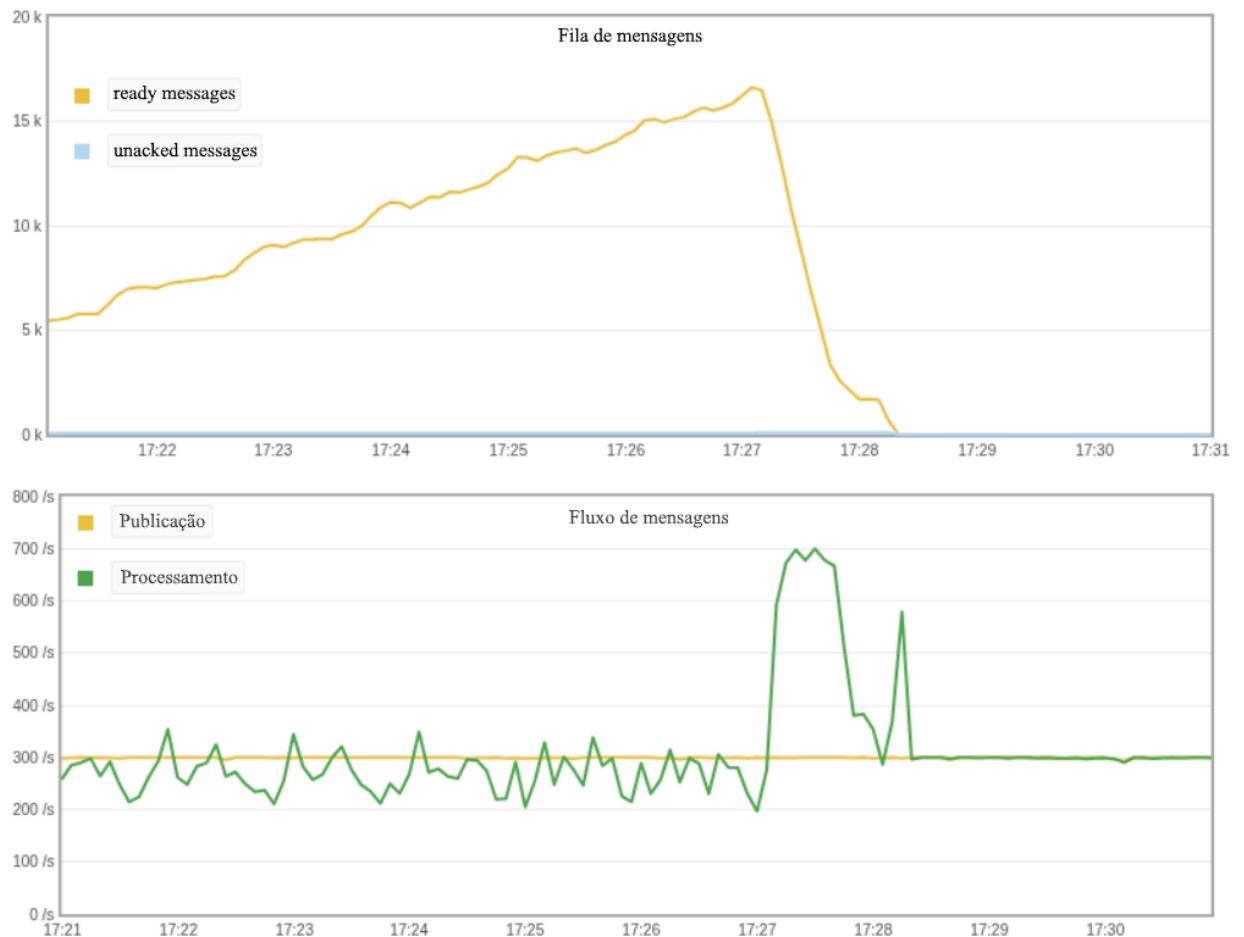
A partir de agora a fila e o fluxo de mensagens estarão empilhadas na mesma figura para melhorar a visualização dos acontecimentos e mais uma vez podemos confirmar o esperado, na Figura 33 vemos na fila de mensagens a linha amarela sofrer uma queda às 17:27, queda essa que coincide com o pico da linha verde no fluxo de mensagens também às 17:27. Foi nesse momento que o número de *Workers* aumentou para dois e logo em seguida refletiu no pico de processamento e na queda de mensagens acumuladas.

Figura 32 - Consumo de recursos do Worker com 300 tarefas por segundo



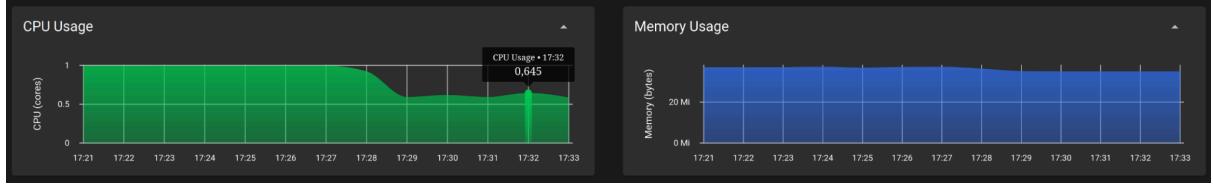
Fonte: Autoria própria (2025)

Figura 33 - Fila de mensagens e fluxo de mensagens com dois Workers



Fonte: Autoria própria (2025)

Figura 34 - Consumo de recursos do primeiro Worker

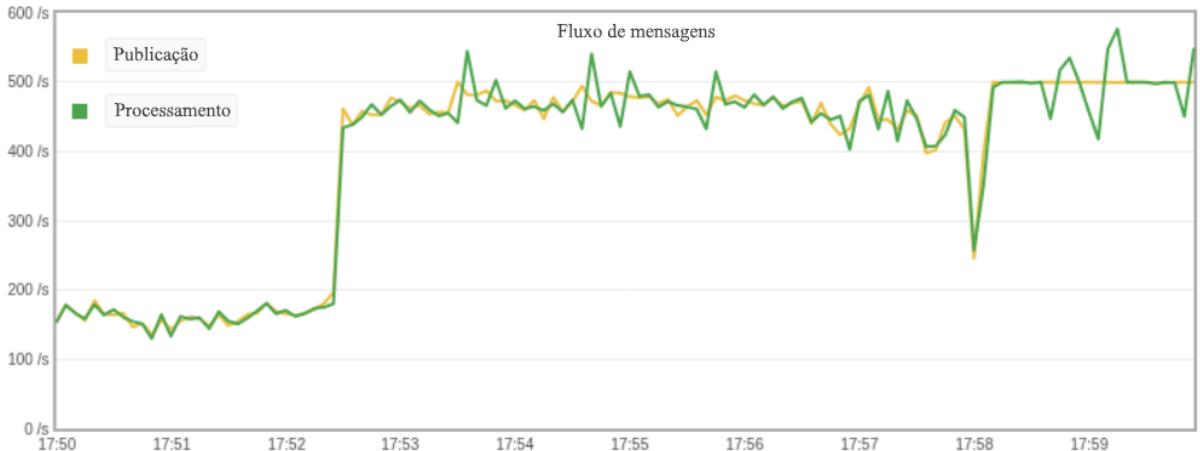


Fonte: Autoria própria (2025)

Vemos no fluxo de mensagens da Figura 33 que a linha verde voltou a se sobrepor à linha amarela, isso significa que os dois *Workers* agora são capazes de processar as 300 mensagens por segundo com facilidade. Na Figura 34 vemos o consumo de CPU do primeiro *Worker* cair drasticamente, cerca de 40% abaixo do uso anterior às 17:27.

Em seguida, a carga de trabalho foi incrementada para 500 tarefas por segundos, o esperado é que os *Workers* juntos tenham um comportamento semelhante ao de um único *Worker* com 250 tarefas por segundo. A partir das 17:53 no gráfico da Figura 35, vemos que o processamento de mensagens está instável, olhando mais atentamente vemos que a linha amarela está mais instável ainda, algo que não foi observado nas cargas anteriores. Isso indica que o *Scheduler* não está sendo capaz de publicar 500 tarefas por segundo, e mais uma vez o número de réplicas foi alterado manualmente, dessa vez nos *Schedulers*, e com dois deles vemos que a linha de publicação voltou a apresentar estabilidade após às 17:58 e que a linha de processamento também se comportou como esperado.

Figura 35 - Fluxo de mensagens com 500 tarefas por segundo



Fonte: Autoria própria (2025)

Por fim, a Tabela 2 reúne os dados coletados durante a primeira etapa do terceiro experimento, evidenciando o alto consumo de CPU e o uso baixo de memória, semelhante ao resultado obtido no *Scheduler*.

Tabela 2 - Consumo médio de recursos do Worker por carga de trabalho com endpoint local

Carga de tarefas por segundo	CPU Média (m)	Memória Média (MiB)
5 tarefas/s	21m	28Mi

Carga de tarefas por segundo	CPU Média (m)	Memória Média (MiB)
20 tarefas/s	73m	28Mi
50 tarefas/s	160m	28Mi
70 tarefas/s	240m	28Mi
100 tarefas/s	380m	30Mi
120 tarefas/s	450m	30Mi
150 tarefas/s	510m	30Mi
200 tarefas/s	640m	34Mi
250 tarefas/s	900m	34Mi

Fonte: Autoria própria (2025)

Contudo, esse cenário é um caso irreal, pois na prática a resposta da requisição HTTP não seria imediata, esse resultado se dá pelo uso do *endpoint* local e outros cenários serão testados nas próximas etapas.

6.3.3.2 ENDPOINT FIXO

A segunda etapa deste experimento será com o uso de um *endpoint* fixo externo ao *cluster*, isso resultará em um tempo de resposta mais condizente com a realidade de uma requisição HTTP pois o tempo de resposta não será quase imediato conforme o *endpoint* local da primeira etapa. O *endpoint* do serviço *web* usado será <https://example.com> e o experimento acontecerá seguindo os mesmos padrões aplicados na primeira etapa.

O fluxo de mensagens da Figura 37 vemos que o processamento das mensagens não acompanhou os incrementos na carga de publicações, isso ocasiona o crescimento da fila de mensagens. O *Worker* não conseguiu ultrapassar a carga de trabalho de 20 tarefas por segundo, a Figura 36 mostra que o consumo de recursos se manteve estável em 200m de CPU, apenas 40% do valor definido no *requests* do *Pod*.

Figura 36 - Consumo de recursos do worker com endpoint fixo

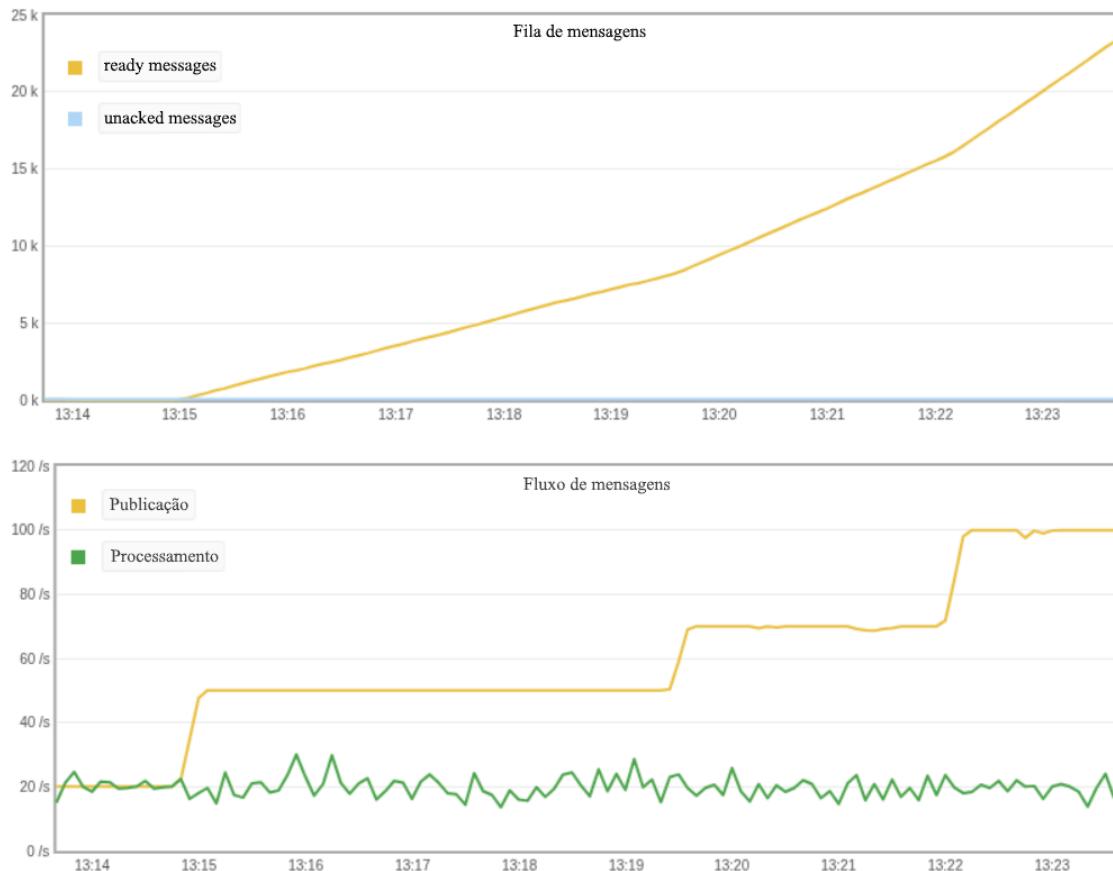


Fonte: Autoria própria (2025)

Esse comportamento aconteceu por conta do tempo de respostas das requisições HTTP, que variavam na casa de milissegundos até segundos, apontando que o desempenho do *Worker* está ligado ao tempo de resposta de cada resposta e provavelmente está sendo limitado por fatores externos, como latência da rede ou políticas de conexão do servidor *web* de destino. Suas *threads* passam a maior parte do tempo em estado de espera, aguardando a

resposta da rede, o que explica o baixo consumo de processamento mesmo com uma alta demanda de tarefas.

Figura 37 - Fila e fluxo de mensagens com endpoint fixo



Fonte: Autoria própria (2025)

Esse resultado de processamento obtido valida que a utilização de CPU é uma métrica ineficaz para este tipo de carga de trabalho, pois ela não reflete o acúmulo de tarefas na fila quando o tempo em espera é alto.

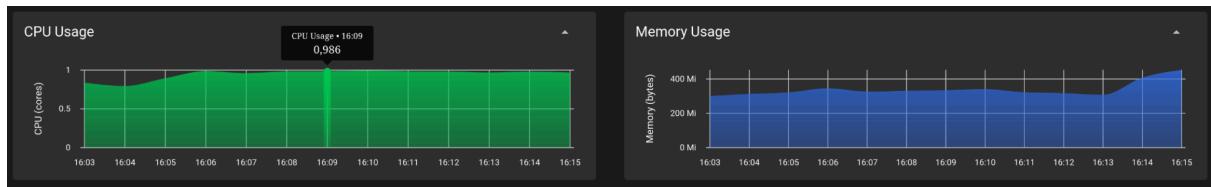
6.3.3.3 ENDPOINTS VARIADOS

Por fim, na terceira etapa do experimento continuará seguindo os mesmos padrões, mas agora usando uma lista de 100 *endpoints* variados que estão listados no repositório remoto do código fonte (SOUZA, 2025). O objetivo é simular um cenário ainda mais realista em relação à segunda etapa, distribuindo as requisições de diferentes servidores para validar o efeito de gargalo de um único *endpoint* fixo.

Como esperado, no fluxo de mensagens da Figura 39 confirmamos que a capacidade máxima de processamento do *Worker* aumentou ao usar *endpoints* variados, sendo capaz de manter o processamento em cerca de 60 tarefas por segundo, um resultado 3 vezes mais satisfatório ao obtido na segunda etapa. Porém em contrapartida o consumo de CPU atingiu o limite após às 16:06 mostrados na Figura 38, estando constante em 1000m, sendo este um

comportamento inesperado pois é 5 vezes mais CPU do que foi usado para processar um terço das tarefas dessas tarefas na segunda etapa.

Figura 38 - Consumo de recursos com endpoints variados



Fonte: Autoria própria (2025)

Figura 39 - Fluxo e fila de mensagens com endpoints variados



Fonte: Autoria própria (2025)

Este comportamento revela uma transição fundamental na natureza do gargalo do sistema. No cenário de *endpoint* fixo, o *Worker* era majoritariamente limitado por suas *threads*, passando a maior parte do tempo em estado de espera pela rede, resultando em baixo consumo de CPU. Ao distribuir as requisições entre 100 *endpoints* variados, o gargalo da rede foi significativamente mitigado, permitindo que as *threads* passassem menos tempo esperando e mais tempo ativamente gerenciando conexões, processando respostas e executando o código da aplicação. Essa mudança transformou o *Worker* em uma aplicação temporariamente limitada por CPU, explicando o aumento desproporcional no seu consumo. Esta descoberta é crucial, pois demonstra que o perfil de consumo de recursos do *Worker* não

é fixo, mas sim altamente dependente da performance dos serviços externos que ele monitora, reforçando ainda mais a conclusão de que a CPU é uma métrica volátil e inadequada para uma estratégia de autoescalonamento confiável neste contexto.

6.3.3.4 CONCLUSÃO DO TERCEIRO EXPERIMENTO

As etapas deste experimento foram repetidas mais de uma vez com diferentes velocidades de conexão com a internet. Para conexões mais lentas e instáveis, o desempenho do *Worker* se apresentou ineficiente com o tempo de resposta para cada requisição chegando a ultrapassar 30 segundos, aumentando o tempo ocioso de CPU e consequentemente reduzindo o consumo médio drasticamente e mesmo aumentando o número de réplicas manualmente a taxa de processamento não aumentava satisfatoriamente a ponto de reduzir o acúmulo de mensagens. Com conexões rápidas, o gargalo do tempo de resposta diminuiu e *Worker* teve o potencial máximo extraído, aumentando o uso de CPU. Logo, esta instabilidade nos resultados obtidos é um indicativo que o uso de CPU do *Worker* está relacionado a velocidade da conexão, fazendo essa métrica não ser confiável para definir o número de réplicas já que o processamento pode estar lento enquanto a fila de mensagens continua crescendo.

6.3.4 COMPORTAMENTO DO WORKER COM HPA

Este experimento tem o objetivo de configurar um HPA que seja capaz de controlar o número de réplicas do *Worker* para garantir que mensagens não fiquem acumuladas na fila.

Conforme os resultados obtidos no terceiro experimento, foi tirada a conclusão de que não é viável configurar um HPA com base na métrica de CPU, sendo instável e relacionada com a velocidade da conexão com a internet. O objetivo é não acumular mensagens na fila, sendo assim foi adotada uma nova estratégia, usar métricas personalizadas retiradas do próprio *RabbitMQ*. Isso foi possível com o uso do *Prometheus*, uma ferramenta que coleta métricas dentro de serviços *cluster* e traduz para que o HPA do *Kubernetes* consiga ler.

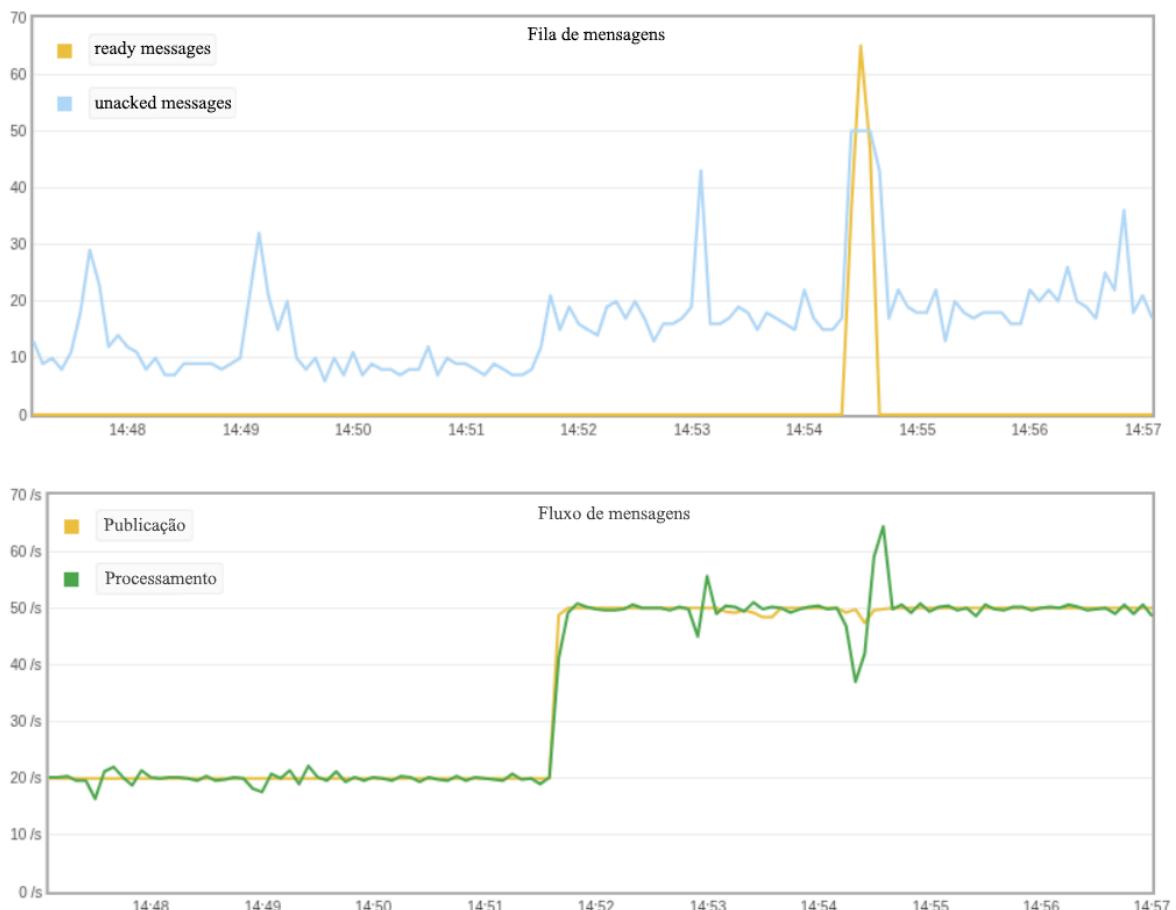
As métricas escolhidas representam o número de mensagens na fila (*ready messages*) e o número de mensagens que ainda não tiveram o processamento confirmado (*unacked messages*). Foi então configurado no HPA que 200 *ready messages* na fila e 35 *unacked messages* eram o limite aceitável para cada *Pod* de *Worker* ativo. O tempo de estabilização no campo *behavior* para *scaleUp* e *scaleDown* foram respectivamente configurados para 1 e 2 minutos, isso significa que o número de réplicas deve aumentar quando o número médio de *ready messages* por *Worker* ultrapassar 200 por mais de 1 minuto e deve reduzir as réplicas quando esse valor estiver abaixo de 200 por mais de 2 minutos.

A mesma janela de tempo vale pra métrica *unacked messages*, que está aqui para tentar estabilizar o comportamento do HPA, pois uma fila de mensagens zerada não significa necessariamente que o número de *Workers* ativos excede a necessidade atual, por exemplo, se a métrica retornar abaixo de 50 *unacked messages* por segundo indica que os *Worker* ainda não está no desempenho máximo, mas se retornar o valor zerado significa que a quantidade de *Workers* excede a necessidade atual processamento, então o ideal é que os *Workers* estejam sempre próximos de atingir a capacidade máxima de desempenho. Além disso, o

número mínimo de réplicas foi definido para 1 com limite máximo de 10 réplicas e as tarefas estão configuradas com *endpoints* variados.

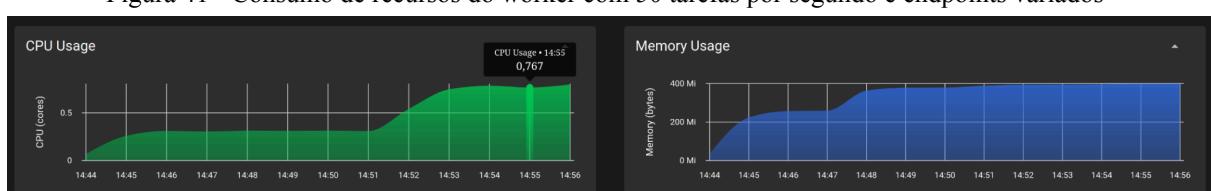
Seguindo o mesmo padrão dos experimentos anteriores, a carga de trabalho foi incrementada gradualmente com intervalos de tempo para estabilização do consumo. Com as Figuras 40 e 41 vemos o *Worker* processando 50 tarefas por segundo no fluxo de mensagens usando em média 750m de CPU e 400Mi de memória. De acordo com o experimento anterior sabemos que o limite do *Worker* com *endpoints* variados está em torno de 60 tarefas por segundo, ao incrementar mais uma vez a carga de trabalho para 70 tarefas por segundo é esperado que a fila de mensagens acumule e o HPA trabalhe para aumentar o número de réplicas.

Figura 40 - Fila e fluxo de mensagens com 50 tarefas por segundo e endpoints variados



Fonte: Autoria própria (2025)

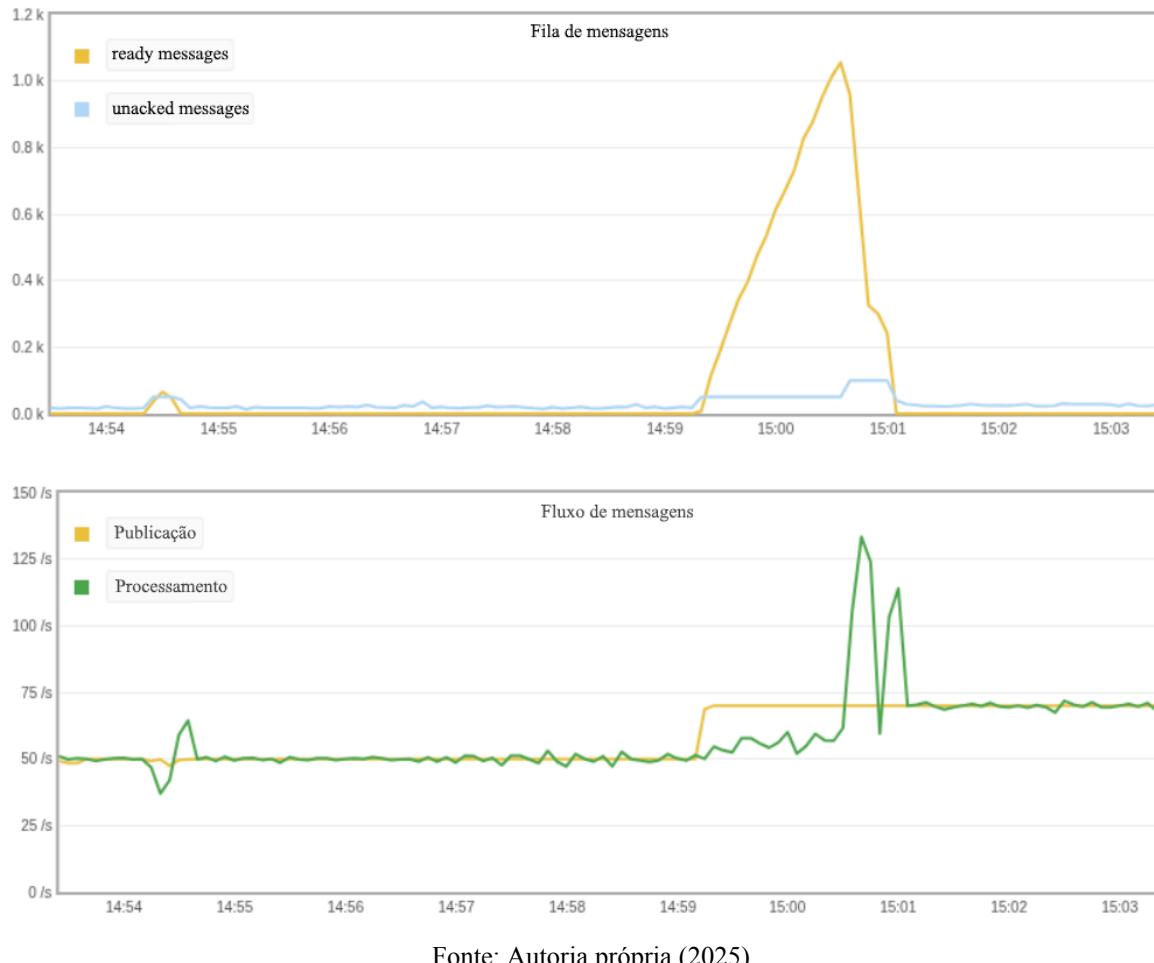
Figura 41 - Consumo de recursos do worker com 50 tarefas por segundo e endpoints variados



Fonte: Autoria própria (2025)

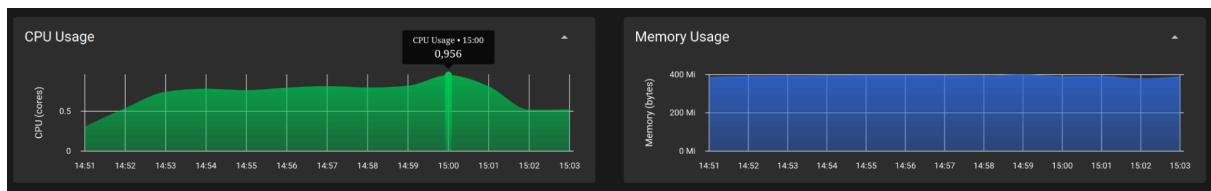
Às 14:59 no fluxo de mensagens da Figura 42 confirmamos que o *Worker* não é capaz de lidar sozinho com 70 tarefas por segundo, isso reflete na fila de mensagens crescendo rapidamente. Alguns segundos após as 15:00 vemos o aumento repentino no processamento de mensagens, também da Figura 42, isso aconteceu porque o HPA detectou que a fila de mensagens estava acima do valor de 200 mensagens por *Worker* durante mais de um minuto e aumentou o as réplicas para dois *Workers*, que juntos estabilizam a fila de mensagens a partir das 15:01. Na Figura 43 temos o gráfico do consumo de recursos do primeiro *Worker*, vemos que o consumo de CPU teve um pico de 956m às 15:00 e uma queda para 500m de uso quando o outro *Worker* foi iniciado.^[6]

Figura 42 - Fila e fluxo de mensagens com 70 tarefas por segundo e endpoints variados



Fonte: Autoria própria (2025)

Figura 43 - Consumo de recursos do primeiro Worker com 70 tarefas por segundo e endpoints variados



Fonte: Autoria própria (2025)

Alguns minutos após o aumento de réplicas o fluxo de mensagens da Figura 44 se estabilizou e em seguida teve uma queda repentina às 15:04 e um aumento às 15:05. Isso aconteceu porque o HPA está configurado para reduzir o número de réplicas se o consumo de mensagens da fila ficar estável por mais de dois minutos, então ele reduziu o número de réplicas e um minuto depois aumentou novamente por causa da incapacidade de um único *Worker* lidar com 70 tarefas por segundo. Essa constante carga de trabalho é pequena para dois *Workers* e grande demais para um deles sozinho, por isso o HPA ficaria em um ciclo de *scaleUp* e *scaleDown*. Então, no fluxo de mensagens da Figura 45 às 15:07 a carga de trabalho foi incrementada para 100 tarefas por segundo e se manteve estável até pouco depois das 15:12, quando houve uma instabilidade na conexão com a rede de internet e a fila de mensagens cresceu brevemente, e o HPA aumentou para 3 réplicas do *Worker*, somente a partir das 15:15 o consumo se estabilizou e o HPA retornou para 2 réplicas.

Figura 44 - Fila e fluxo de mensagens com 70 tarefas por segundo e endpoints variados

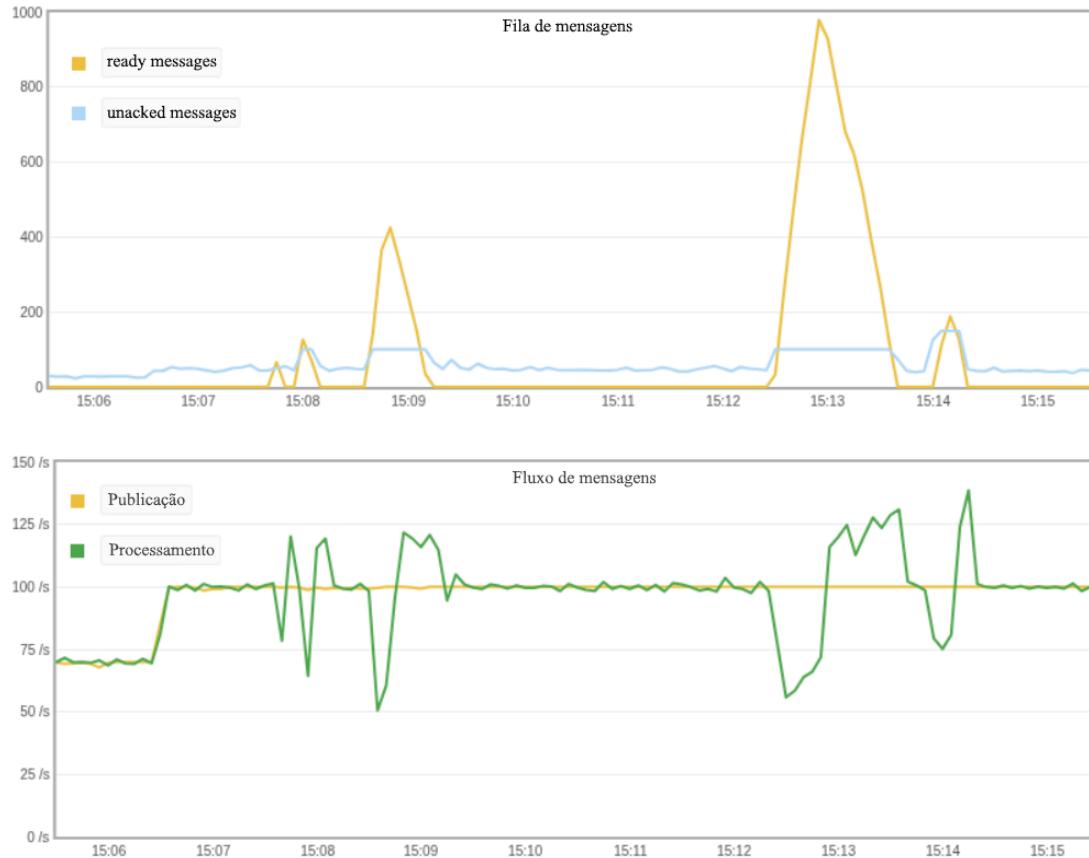


Fonte: Autoria própria (2025)

Na Figura 46 vemos o *log* de funcionamento do HPA com 4 mensagens, a primeira delas mostra o momento referente às 15:04, quando o número de réplicas foi reduzido para 1. A segunda mensagem nos informa que o número de réplicas aumentou para 2, porém há um detalhe nessa linha indicando que esse *log* aconteceu duas vezes, e uma delas foi a 15 minutos atrás em relação ao horário da captura do *log*, que se refere às 15:01 e a segunda

mensagem deste *log* se refere às 15:05. O terceiro *log* é do momento que aconteceu a instabilidade de rede e o número de réplicas aumentou para 3 às 15:12 e por fim, o último *log* refere-se a alguns segundos após às 15:15, quando o consumo se estabilizou e o HPA reduziu novamente para 2 réplicas.

Figura 45 - Fila e fluxo de mensagens com 100 tarefas por segundo e endpoints variados



Fonte: Autoria própria (2025)

Figura 46 - Log do HPA referente ao momento com 100 tarefas por segundo

Events:				
Type	Reason	Age	From	Message
Normal	SuccessfulRescale	13m	horizontal-pod-autoscaler	New size: 1; reason: All metrics below target
Normal	SuccessfulRescale	11m (x2 over 15m)	horizontal-pod-autoscaler	New size: 2; reason: external metric rabbitmq_queue_messages_ready(nil) above target
Normal	SuccessfulRescale	2m46s	horizontal-pod-autoscaler	New size: 3; reason: external metric rabbitmq_queue_messages_unacked(nil) above target
Normal	SuccessfulRescale	1s	horizontal-pod-autoscaler	New size: 2; reason: All metrics below target

Fonte: Autoria própria (2025)

Posteriormente, a carga de trabalho foi incrementada gradualmente até alcançar 400 tarefas por segundo com o objetivo de alcançar o limite de réplicas. No fluxo de mensagens da Figura 47 este incremento aconteceu entre às 15:19 e 15:20, vemos que a fila de mensagens começou a subir rapidamente, chegando a alcançar o pico de 20000 *ready messages*, um valor muito maior do que a média de 200 *ready messages* por *Worker* que foi estabelecido no HPA. Logo, o HPA respondeu rapidamente a esse aumento de mensagens, na Figura 49 temos o histórico de *logs* que mostram o aumento no número de réplicas, esses aumentos aconteceram 4 vezes até alcançar o máximo de 10 réplicas de *Worker*, vemos que o

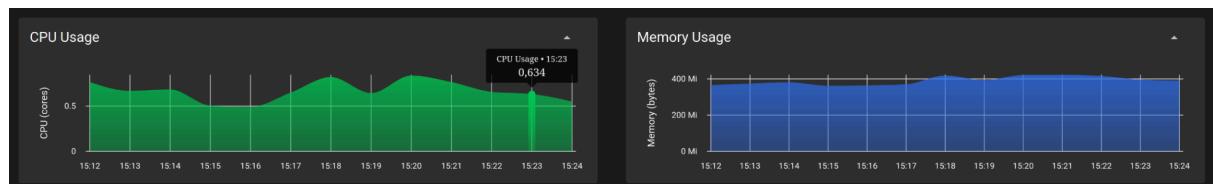
número de réplicas aumentou na seguinte ordem: 3 réplicas às 15:18, 5 réplicas às 15:20:30, 8 réplicas às 15:21:30 e 10 réplicas às 15:22:30, os incrementos não foram unitários e temos aqui um outro comportamento que foi configurado ao HPA, uma política do *scaleUp* onde o limite no incremento de réplicas foi definido para 50% da quantidade atual ou até no máximo 2 réplicas. Isso aconteceu por conta da grande quantidade de *ready messages* na fila e justifica o motivo das réplicas terem aumentado até o limite máximo em um período de tempo pouco menor que 5 minutos.

Figura 47 - Fila e fluxo de mensagens com 400 tarefas por segundo e endpoints variados



Fonte: Autoria própria (2025)

Figura 48 - Consumo de recursos do primeiro worker ao decorrer do aumento de réplicas



Fonte: Autoria própria (2025)

Figura 49 - Log do HPA aumentando para 10 réplicas

```
Figura 19 - Log do HPA aumentando para 10 replicas
```

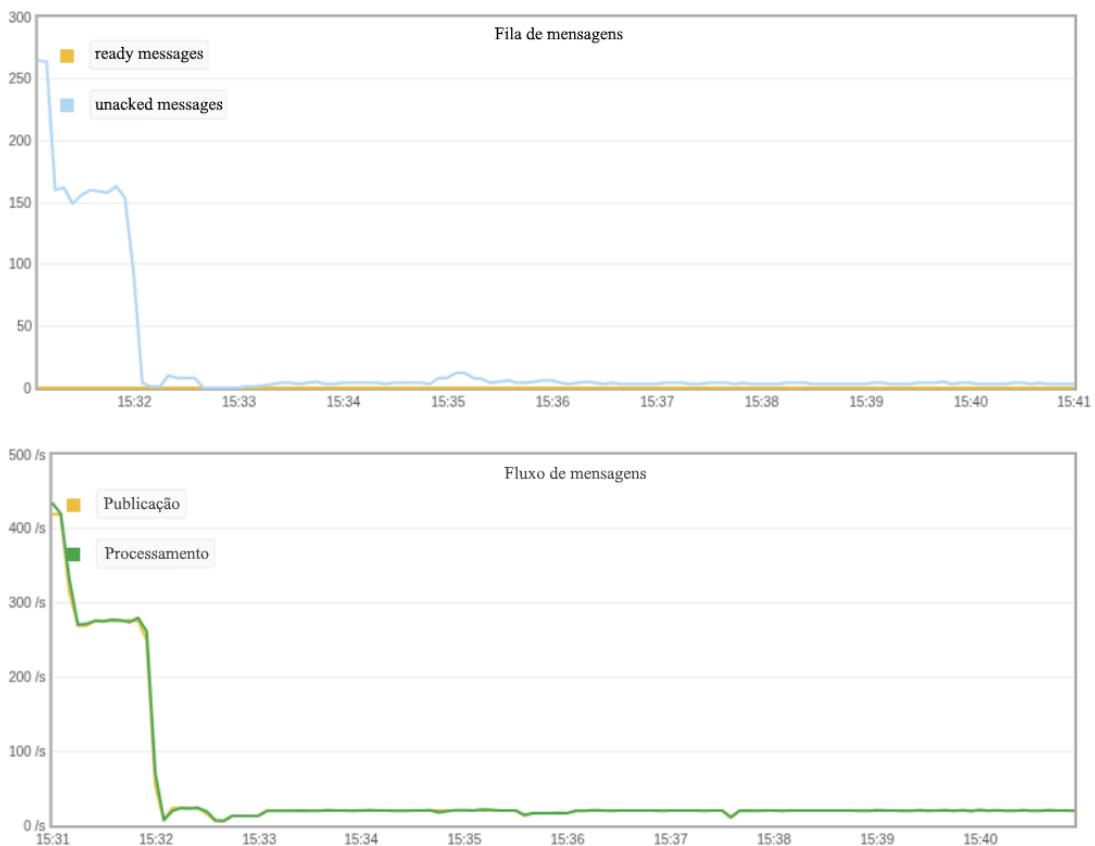
```
Normal SuccessfulRescale 4m56s horizontal-pod-autoscaler New size: 3; reason: external metric rabbitmq_queue_messages_ready(nil) above target
Normal SuccessfulRescale 2m41s horizontal-pod-autoscaler New size: 5; reason: external metric rabbitmq_queue_messages_ready(nil) above target
Normal SuccessfulRescale 101s horizontal-pod-autoscaler New size: 8; reason: external metric rabbitmq_queue_messages_ready(nil) above target
Normal SuccessfulRescale 40s horizontal-pod-autoscaler New size: 10; reason: external metric rabbitmq_queue_messages_ready(nil) above target
```

Fonte: Autoria própria (2025)

O processamento de mensagens se estabilizou após às 15:24 e a fila se manteve com 0 *ready messages* e com a média de 340 *unacked messages*. Mesmo com a fila de mensagens zerada o HPA não reduziu o número de réplicas, pois a segunda métrica, *unacked messages*, indicava que os 10 *Workers* estavam trabalhando numa faixa aceitável pois a média de 34 *unacked messages* por *Worker* está abaixo do critério definido no HPA. A Figura 48 mostra o histórico do consumo de recursos do primeiro *Worker* na janela de tempo onde aconteceram todas alterações na quantidade de réplicas observadas no *log* da Figura 49, o consumo médio de CPU se estabilizou em 500m após às 15:23, mas houveram picos de 900m às 15:18 e 15:20 que coincidem com os momentos de pequenos incrementos na carga de trabalho e antes do incremento no número de réplicas.

Por fim, para testar toda a capacidade do HPA a carga de trabalho foi reduzida para 20 tarefas por segundo com o objetivo de validar o funcionamento do *scaleDown*. Vemos no fluxo de mensagens da Figura 50 que a partir das 15:32 a nova carga de trabalho já estava reduzida e sabemos que um único *Worker* é o suficiente para lidar com essa demanda, porém minutos antes o HPA decidiu manter 10 réplicas para a carga de 400 tarefas por segundo, então o esperado é que ele reduza o número de réplicas.

Figura 50 - Fila de fluxo de mensagens reduzindo para 20 tarefas por segundo com endpoints variados



Fonte: Autoria própria (2025)

Com a Figura 51 temos o *log* do HPA que confirma que as réplicas foram reduzidas gradualmente, no *log* existem 3 mensagens, sendo a primeira delas informando que o número

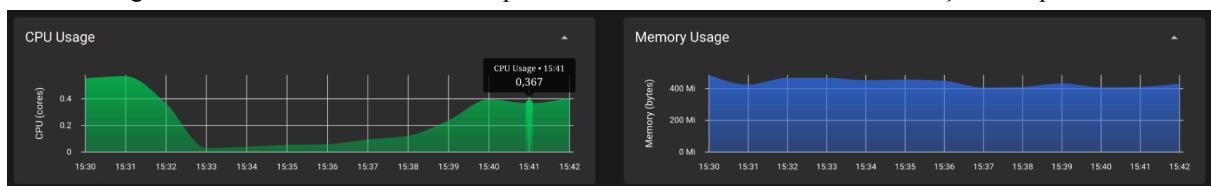
de réplicas caiu para 9, a segunda mensagem é uma junção de outras 5 mensagens e vemos que o número de réplicas caiu para 2 e por fim na última mensagem vemos que o HPA deixou apenas uma réplica de *Worker*. A carga de trabalho com 20 tarefas por segundos é muito baixa para ser processada por 10 *Workers*, na Figura 52 temos o gráfico de consumo de recursos do primeiro *Worker*, a partir das 15:33 vemos o consumo de CPU crescer gradualmente até alcançar a média de 360m após às 15:40. Esse crescimento no consumo coincide com a redução no número de réplicas, pois a carga de trabalho está voltando a ser processada pelo primeiro *Worker*.

Figura 51 - Log do HPA reduzindo para uma réplica de Worker

```
Normal SuccessfulRescale 8m32s horizontal-pod-autoscaler New size: 9; reason: All metrics below target
Normal SuccessfulRescale 61s (x5 over 5m1s) horizontal-pod-autoscaler (combined from similar events): New size: 2; reason: All metrics below target
Normal SuccessfulRescale 1s (x2 over 35m) horizontal-pod-autoscaler New size: 1; reason: All metrics below target
```

Fonte: Autoria própria (2025)

Figura 52 - Consumo de recursos do primeiro Worker aumentando com a redução de réplicas



Fonte: Autoria própria (2025)

Por fim, o quarto e último experimento validou de forma conclusiva a eficácia da estratégia de autoescalonamento proposta ao *Worker*. A utilização das métricas *ready* e *unacked messages* em conjunto permitiu que o HPA fosse capaz de se adequar a diferentes cenários. Dessa forma, o experimento demonstrou que a abordagem com métricas de negócio customizadas cumpre o objetivo de garantir que as tarefas sejam processadas eficientemente, sem permitir que fiquem acumuladas na fila de mensagens por um tempo prolongado.

7 CONSIDERAÇÕES FINAIS

Este trabalho se propôs a investigar como projetar um sistema de verificação de serviços que fosse escalável, resiliente e distribuído utilizando *Kubernetes* e *RabbitMQ*. Para alcançar uma solução foi desenvolvido o *SentryWeb*, uma plataforma para agendamento e processamento de tarefas genéricas que funciona através dos microsserviços *Scheduler* e *Worker*, que juntos aos componentes do *Kubernetes* e o desacoplamento via mensageria do *RabbitMQ* permitiram a construção de um sistema que cumprem os requisitos propostos onde a resiliência foi alcançada através da arquitetura modular e a natureza distribuída do *Kubernetes* que podem reduzir pontos únicos de falha no sistema em um *cluster* físico e a escalabilidade através da configuração do HPA.

A principal contribuição técnica deste trabalho foram os experimentos para validação de uma estratégia eficiente de autoescalonamento capaz de funcionar em cenários distintos. Os resultados demonstraram de forma conclusiva que métricas tradicionais como utilização de CPU ou memória podem ser ineficazes, pois dependendo da essência do projeto o autoescalonamento pode exigir métricas personalizadas e precisas. Além disso, este trabalho

serve como um guia para a estruturação e dimensionamento de sistemas semelhantes, contando com a entrega de um protótipo funcional completo e toda sua infraestrutura como código.

É importante reconhecer as limitações presentes no trabalho. Todos os testes foram realizados no *Minikube*, que permite a configuração simplificada de um *cluster Kubernetes* virtual para um ambiente de desenvolvimento local. Fica em aberto para trabalhos futuros: a implementação de *Workers* especializados em outros tipos de tarefas, interface para gerenciamento das tarefas, configurações relevantes de *probes* do *Pod*, que servem como monitoramento da saúde dos *Pods*, sendo possível uma integração com a base atual do projeto, investigar a escalabilidade das dependências como *RabbitMQ* e *PostgreSQL*, por fim efetuar novos experimentos de validação em um ambiente real com um *cluster* físico e concluir a implementação de um *pipeline* automático de integração e entrega contínua (CI/CD) para agilizar o desenvolvimento de atualizações do projeto.

REFERÊNCIAS

ARGOCD. **Argo CD - Declarative GitOps CD for Kubernetes**. The Argo Project, 2025. Disponível em: <https://argo-cd.readthedocs.io/>. Acesso em: 27 jul. 2025.

BARBOSA, Fábio Luiz. Kubernetes em ambientes não persistentes. 2021. Trabalho de Conclusão de Curso (Especialização Arquitetura e Gestão de Infraestrutura de Tecnologia da Informação) - Universidade Tecnológica Federal do Paraná, Curitiba, 2021.

BURNS, Brendan et al. **Kubernetes: up and running: dive into the future of infrastructure**. " O'Reilly Media, Inc.", 2022.

CLOUDAMQP. **RabbitMQ for beginners - What is RabbitMQ? Part 1**. CloudAMQP, 2025. Disponível em:
<https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html>. Acesso em: 7 ago. 2025.

DE CARVALHO NETO, Pedro Moritz; CASTRO, Márcio; SIQUEIRA, Frank. Balanceamento de Carga Dinâmico em Ambientes Kubernetes com o Kubernetes Scheduling Extension (KSE). In: **Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)**. SBC, 2023. p. 169-180.

DOCKER. **Docker Documentation**. Docker, Inc., 2025. Disponível em:
<https://docs.docker.com/>. Acesso em: 27 jul. 2025.

FARIAS, Thalison Bruno Mendonca de; SILVA, Paulo HL. Smart task control: um sistema de agendamento e execução de tarefas. 2024.

GITHUB ACTIONS. **GitHub Actions Documentation**. GitHub, Inc., 2025. Disponível em:
<https://docs.github.com/en/actions>. Acesso em: 27 jul. 2025.

GRAFANA LABS. **Grafana Documentation**. Grafana Labs, 2025. Disponível em:
<https://grafana.com/docs/grafana/latest/>. Acesso em: 27 jul. 2025

HELM. Helm | The Kubernetes Package Manager. The Helm Authors, 2025. Disponível em: <https://helm.sh/docs/>. Acesso em: 27 jul. 2025.

JENSEN, Nikolas; MIERS, Charles C. Uma análise das vulnerabilidades de segurança do kubernetes. In: **Escola Regional de Redes de Computadores (ERRC)**. SBC, 2021. p. 67-72.

KUBERNETES. Documentação do Kubernetes. Disponível em: <<https://kubernetes.io/pt-br/docs/home/>>. Acesso em: 24 out. 2025.

MINIKUBE. Minikube Documentation. The Kubernetes Authors, 2025. Disponível em: <https://minikube.sigs.k8s.io/docs/>. Acesso em: 27 jul. 2025.

NETTO, Hylson et al. Replicação de máquinas de estado em containers no kubernetes: uma proposta de integração. **Anais do XXXIV Simpósio Brasileiro de Redes de Computadores-SBRC**, 2016.

NETTO, Hylson Vescovi et al. Coordenação de Containers no Kubernetes: Uma Abordagem Baseada em Serviço. In: **Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)**. SBC, 2017.

NGINX. NGINX Documentation. F5, Inc., 2025. Disponível em: <https://docs.nginx.com/>. Acesso em: 27 jul. 2025.

OPEN CONTAINER INITIATIVE. Open Container Initiative. The Linux Foundation, 2025. Disponível em: <https://opencontainers.org/>. Acesso em: 7 ago. 2025.

POLISELLO, Breno Otavio; GONÇALVES, Marcelo Augusto Silva. Implementação de um ambiente clusterizado de containers utilizando kubernetes e containerd. 2021. Trabalho de Conclusão de Curso (Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) – Faculdade de Tecnologia de São José do Rio Preto, São José do Rio Preto, 2021.

POSTGRESQL. PostgreSQL: Documentation. Disponível em: <<https://www.postgresql.org/docs/>>. Acesso em: 24 out. 2025.

PROMETHEUS. Prometheus: From metrics to insight. The Prometheus Authors, 2025. Disponível em: <https://prometheus.io/docs/introduction/overview/>. Acesso em: 27 jul. 2025.

PYTHON. Python 3.11 documentation. Python Software Foundation, 2025. Disponível em: <https://docs.python.org/3.11/>. Acesso em: 27 jul. 2025.

RABBITMQ. RabbitMQ Documentation. Disponível em: <<https://www.rabbitmq.com/docs>>. Acesso em: 24 out. 2025.

RODRIGUES, Pedro GR; GARCIA, Vinicius C. Implementando um sistema de conteinerização com Kubernetes usando GitOps.

SOUZA, Brenno Kevyn Maia de. **SentryWeb**: um sistema distribuído, escalável e resiliente para monitoramento de serviços. 2025. Repositório de código. GitHub, 2025. Disponível em: <https://github.com/BrennoKM/SentryWeb>. Acesso em: 7 ago. 2025.