UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA E ELETRÔNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Nícolas Vitor Yuki Obara Yamakoshi

Arquitetura em nuvem escalável para processamento de dados IoT

FLORIANÓPOLIS
2023

Nícolas Vitor Yuki Obara Yamakoshi

Arquitetura em nuvem escalável para processamento de dados IoT

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como requisito necessário para obtenção do grau de Bacharel em Engenharia Elétrica

Florianópolis, junho de 2023

Nicolas Vitor Yuki Obara Yamakoshi

Título: Arquitetura em nuvem escalável para processamento de dados IoT

Este Trabalho Conclusão de Curso foi julgado adequado para obtenção do Título de "Bacharel em Engenharia Elétrica" e aceito, em sua forma final, pelo Curso de Graduação em Engenharia Elétrica.

Florianópolis, 30 de junho de 2023.

Prof. Miguel Moreto, Dr.
Coordenador do Curso de Graduação em Engenharia Elétrica

Banca Examinadora:

Prof. Richard Demo Souza, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Walter Pereira Carpes Junior, Ph.D.
Universidade Federal de Santa Catarina

Elço João dos Santos Junior
Instituto SENAI de Inovação

Agradecimentos

Primeiramente, gostaria de agradecer aos meus pais, que me proporcionaram ensinamentos inestimáveis e uma vida extremamente privilegiada e estruturada. Sem eles, nada disso teria sido possível. Também agradeço minha família, que sempre proveu uma rede de apoio enorme a mim e a meus pais – em especial, aos meus padrinhos, por estarem sempre lá por mim e por serem meus segundos pais.

Gostaria de agradecer minha namorada Paula, que tem sido uma parceira incrível desde o início do nosso relacionamento e que me ajudou inúmeras vezes, de maneira direta ou indireta, durante todo esse tempo que estamos juntos.

Ademais, agradeço a todos meus professores do Departamento de Engenharia Elétrica e do Centro Tecnológico, que me proporcionaram um ensino de qualidade e gratuito durante estes seis anos em que estive na Universidade. Em especial, agradeço aos professores Walter Pereira Carpes Jr., Jhoe Batistela e André Luís Kirsten, por terem me acolhido extremamente bem durante meu tempo como IC e petiano e pelos ensinamentos, que levarei para toda a vida. Por fim, agradeço especialmente ao prof. Richard Demo Souza, pela orientação deste trabalho, por toda paciência durante este processo e por fazer com que este TCC tenha sido um processo agradável.

Além disso, gostaria de agradecer aos diversos mentores que tive no meu percurso profissional até então, que me ensinaram sobre engenharia de software na prática: Antônio Duarte, Andrio Frizon e Henrique Furtado, da Olby; Wilfried Kirschenmann, da Aneo (França); e Loïc Raucy e Ronan Pelliard, da Datadog (França).

Também gostaria de agradecer ao grupo PETEEL, que me permitiu me desenvolver como um profissional, me abriu inúmeras portas e me apresentou diversas pessoas incríveis.

Agradeço também aos meus colegas de departamento, com quem tive o privilégio de dividir os ônus e bônus da graduação: André Gil de Oliveira, Arthur de Deus, Danilo Aurich, João Victor Faria, Jonas Pacheco, Milena Neves, Ricardo Marques, Rodrigo Martinez e Vinicius Hirassaki.

Finalmente, mas não menos importante, agradeço também a todos meus amigos, que estiveram sempre presentes durante este período formativo da minha vida e me deram todo o suporte para concluí-lo.

Resumo

A indústria de IoT está em expansão acelerada e com isso, infraestrutura de computação em nuvem são necessárias para receber e processar estes dados em larga escala. Deste modo, este trabalho visa desenvolver uma arquitetura escalável em nuvem de um sistema para receber, processar e armazenar dados vindos de sistemas IoT. A metodologia aplicada baseou-se na concepção e desenvolvimento desta arquitetura usando componentes básicos, tendo como plano de fundo uma aplicação de IoT industrial para monitoramento de temperatura, pressão e taxa de produção de unidades defeituosas em máquinas fabris. Além disso, foram realizados testes de carga para estabelecer os limites técnicos do projeto e validar os requerimentos funcionais e não-funcionais. Dos resultados obtidos, foi possível desenhar a arquitetura, implementá-la por completo e implantá-la na nuvem. Ademais, seus limites foram estabelecidos por meio de testes de carga, tanto em um ambiente local quanto na nuvem.

Palavras-chave: Internet das Coisas. Sistemas distribuídos. Micro-serviços. Computação em Nuvem. Testes de carga.

Abstract

The IoT industry is expanding rapidly and with this, cloud computing infrastructures are needed to receive and process this data on a large scale. Thus, this work aims to develop a scalable cloud architecture of a system to receive, process and store data coming from IoT systems. The applied methodology was based on the design and development of this architecture using basic components, with the background of an industrial IoT application for monitoring temperature, pressure and production rate of defective units in manufacturing machines. In addition, load testing was performed to establish the technical limits of the design and validate the functional and non-functional requirements. From the results obtained, it was possible to design the architecture, implement it in full, and deploy it in the cloud. Furthermore, its limits were established through load testing, both in a local environment and in the cloud.

Keywords: Internet of Things. Distributed systems. Microservices. Cloud computing. Load tests.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

IoT - Sigla para *Internet of Things*.

GCP - Sigla para *Google Cloud Platform*.

AWS - Sigla para *Amazon Web Services*.

AMQP - Sigla para *Advanced Message Queueing Protool*.

MOM - Sigla para *Message-Oriented Middleware*.

MQTT - Sigla para *Message Queueing Telemetry Transport*.

M2M - Sigla para *Machine-to-Machine*.

HTTP - Sigla para *HyperText Transfer Protocol*.

EC2 - Sigla para *Elastic Cloud Compute*.

SQL - Sigla para *Structured Query Language*.

DBMS - Sigla para *Database Management Systems*.

IaC - Sigla para *Infrastructure-as-Code*.

HPA - Sigla para *Horizontal Pod Autoscaler*.

GKE - Sigla para *Google Kubernetes Engine*.

Sumário

1 Introdução

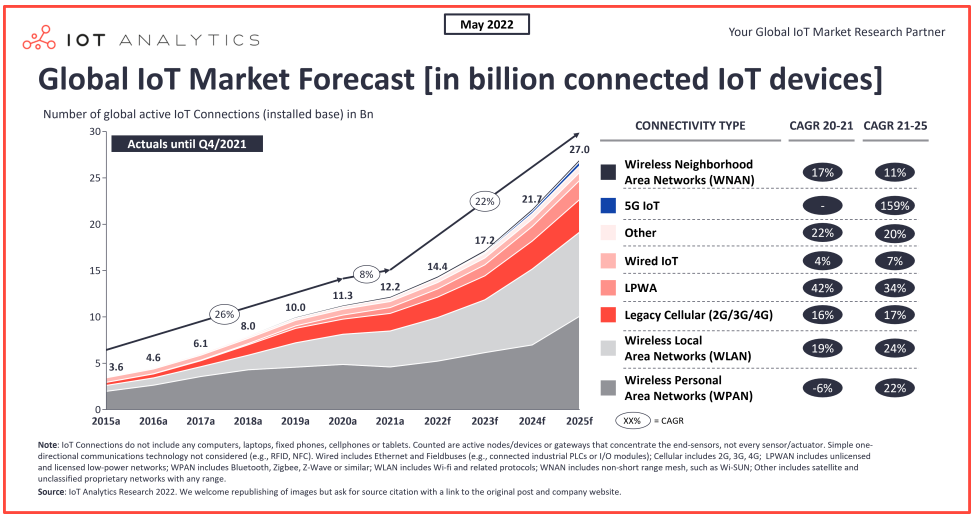
Com a evolução das tecnologias *Internet of Things* (IoT) – ou "*Internet* das coisas", em português –, o campo das telecomunicações mescla-se cada vez mais com o campo da computação e da engenharia de *software*. Desta amálgama, destaca-se a arquitetura de nuvem, componente essencial em sistemas IoT nas etapas pós-sensoriamento. Este trabalho, portanto, se propõe a desenvolver e avaliar uma arquitetura em nuvem para suportar um sistema IoT.

Neste breve capítulo, serão apresentados: o contexto deste trabalho e os seus objetivos, tanto gerais quanto específicos.

1.1 Contexto

A indústria de IoT está em uma expansão acelerada: de acordo com o site *IoT Analytics* [Hasan 2022], em 2022 houve um crescimento de 18% no número de dispositivos IoT conectados no mundo, atingindo a marca de 14,4 bilhões unidades.

Figura 1 – Gráfico sobre o número de dispositivos IoT conectados.



Fonte: Retirado de [Hasan 2022].

Com essa escala, o mercado necessita de soluções de arquitetura em nuvem a altura para dar suporte a essa população de dispositivos e suportar a quantidade massiva de dados produzidos por eles. O suporte necessário, no caso de IoT, é o recebimento de inúmeras requisições Web por segundo, para armazenar e/ou tratar os dados vindos de múltiplos *gateways* IoT.

Porém, neste contexto, o desafio é manter um nível de serviço confiável: receber, processar e armazenar 100% dos dados enviados pelos *gateways*, mesmo durante os

picos onde 100% dos dispositivos estejam enviando requisições. Um dos aspectos mais importantes a levar em conta é, portanto, a *escalabilidade*: a capacidade do próprio sistema de requisitar mais recursos ao provedor (e.g. aumentar a memória, adicionar mais *cores* de processamento, etc.) de acordo com a demanda atual. Assim, uma falha neste sistema ou em sua escalabilidade, dependendo da aplicação em questão, poderia resultar em perda de dados sensíveis, leituras estatísticas erradas, entre outros problemas.

Por isso, dados estes desafios, todos os maiores provedores de serviços em nuvem do mercado de computação em nuvem, como *Amazon Web Services* (AWS), *Google Cloud Platform* (GCP), *Microsoft Azure*, etc., já oferecem uma gama de serviços prontos especializados em IoT.

Estes serviços podem ser muito úteis para casos de uso menores - seja por ser um caso de uso clássico ou por falta de mão-de-obra especializada. Porém, tais serviços, por serem tanto prontos quanto administrados pelos provedores, possuem algumas desvantagens, como:

- **Falta de customização**: não poder modificar o sistema de acordo com necessidades mais específicas, como adicionar redundâncias, filtrar dados como quiser, entre outros;
- **Custos mais elevados**: por ser um serviço gerenciado, existe o custo adicional da gerência feita pelo próprio provedor;
- **Risco de *vendor lock-in***: visto que usando uma própria infraestrutura que depende apenas de serviços básicos, como banco de dados, filas, etc., é muito mais fácil de trocar de provedor, quando comparado a depender de uma solução complexa e menos oferecida no mercado.

A fim de encontrar uma solução satisfatória neste contexto, este trabalho busca aprofundar o conhecimento dos leitores em relação às arquiteturas em nuvem escaláveis e desenvolver uma arquitetura completa, que utilize apenas componentes básicos e que seja escalável. Por fim, este trabalho também se propõe a avaliar a escalabilidade do sistema, realizando testes de carga por simulações de requisições vindas de um *gateway* IoT. O presente trabalho, portanto, não se concentra na parte de sensoriamento e comunicação local de IoT, mas sim no que ocorre após os dados serem transmitidos dos *gateways* para a nuvem.

1.2 Objetivo Geral

Este trabalho visa desenvolver uma arquitetura *escalável* (i.e. cuja disponibilidade se adapta automaticamente ao volume da demanda) de um sistema de processamento de dados, especificamente para o contexto de recepção de um volume massivo de dados,

vindos de sistemas de IoT. Em suma, tal processamento envolve: enriquecer e/ou extrair os *payloads*, armazená-los em uma base de dados e prover uma maneira do usuário consultar/interagir com eles.

1.3 Objetivos Específicos

Para atingir o objetivo geral, alguns objetivos específicos intermediários devem ser traçados:

- Estudo sobre diferentes métodos de escalabilidade de arquiteturas em nuvem e aprofundamento dos conhecimentos em *design* de sistemas de informação;
- Estudo das principais problemáticas, casos de uso e protocolos no contexto de processamentos de dados em IoT;
- Desenvolvimento da arquitetura de processamento de dados em IoT;
- Condução de testes de carga e simulações para avaliar os limites do sistema desenvolvido.

2 Revisão de literatura

Neste capítulo, alguns conceitos essenciais serão brevemente revisados, para melhor embasar o presente texto e melhor compreensão do leitor.

2.1 Internet of Things

IoT é um termo usado para descrever situações nas quais uma gama de dispositivos e sensores têm suas capacidades ampliadas por processamento computacional e conectividade à Internet [Rose, Eldridge e Chapin 2015]. O termo é comumente associado a diversas aplicações, como: automação industrial e residencial; sensoreamento em agricultura; rastreamento de cargas; entre outros.

Nesta seção, será introduzida a arquitetura básica de uma rede IoT em quatro camadas. Em seguida, iniciaremos uma breve comparação sobre os protocolos de comunicação utilizados, questão que será central no tangente à IoT neste trabalho.

2.1.1 Arquitetura básica

A arquitetura de uma rede IoT pode ser organizada em cinco camadas, cada uma com sua organização funcional, seus componentes físicos e formatos de dados [Vashi et al. 2017]. A Fig. 2 as ilustra.

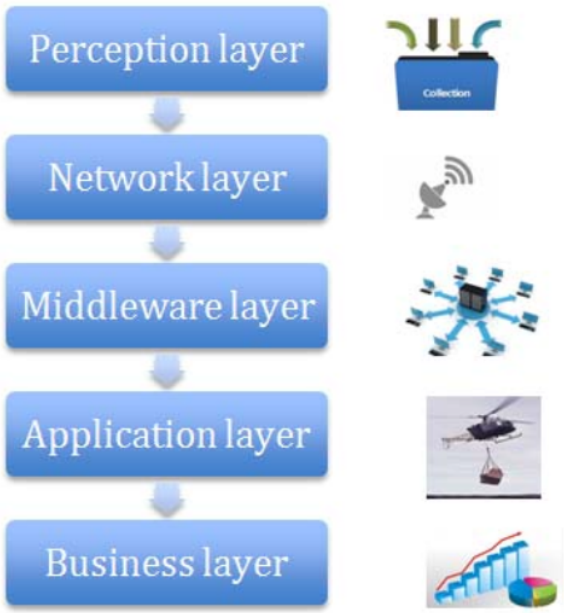
Dentre as cinco camadas, a *Application layer* (camada de aplicação) é a de maior interesse neste trabalho, visto que nela se encontra toda a lógica de negócios das aplicações IoT. Portanto, é justamente nela que uma arquitetura de processamento de dados estará localizada.

2.1.2 Protocolos de comunicação em IoT

Para a comunicação entre a camada de aplicação e a camada de rede, inúmeros protocolos podem ser usados. Dentre os mais conhecidos e comumente usados, estão [Henke 2022]:

- *Advanced Message Queuing Protocol* (AMQP): um *Message-Oriented Middleware* (MOM) (*middleware* orientado a mensagens) de código aberto e amplamente utilizado por agentes de mensagem, como *RabbitMQ*, *Apache ActiveMQ*, etc.;

Figura 2 – Camadas na arquitetura de IoT.



Fonte: [Vashi et al. 2017].

- *Message Queueing Telemetry Transport* (MQTT): protocolo criado especificamente para comunicação *Machine-to-Machine* (M2M) e IoT, e por sua leveza, possui muitas vantagens para aplicações remotas;
- *HyperText Transfer Protocol* (HTTP): por ser o protocolo *de facto* de servidores *web* na Internet, prevê uma facilidade de desenvolvimento e alta compatibilidade.

2.2 Engenharia de Software

De acordo com a definição de [IEEE Standard Glossary of Software Engineering Terminology 1990], Engenharia de *Software* é o desenvolvimento e manutenção de *software*, de maneira sistemática e disciplinada, como em qualquer outra engenharia.

Dada a alta complexidade e o tamanho dos *softwares* desenvolvidos atualmente, existem diversas responsabilidades a serem delegadas em um dado sistema, como: desenvolvimento da interface visual com os usuários (e.g. *website* ou aplicativo); desenvolvimento de um sistema central, no qual a lógica de negócios e os dados residem; manutenção do *software* em estado saudável, independentemente de fatores externos (e.g. nível de demanda, catástrofes, etc.); entre outros fatores. Deste modo, os *engenheiros de software* são normalmente classificados entre algumas categorias, como [Singh 2023]:

- Engenheiro *Front-End* (FE), que se responsabiliza pela interface com usuários/clientes do *software*;

- Engenheiro *Back-End* (BE), que se responsabiliza pela lógica por trás do *software* e a sua performance;
- Engenheiro DevOps, que se responsabiliza pela disponibilidade e confiabilidade dos sistemas e é focado em temas como *Cloud Computing* e *System Design*.

Nesta seção, serão introduzidos diversos tópicos da Engenharia de *Software*, relevantes ao tema de arquitetura escalável em nuvem para processamento de dados.

2.2.1 Design de Sistemas Distribuídos

O *Design* de Sistemas Distribuídos– mais conhecido como *Distributed Systems Design* (ou apenas *Systems Design*), em inglês – é um campo da Engenharia de *Software* que estuda e desenha a arquitetura de sistemas computacionais distribuídos [Rouse 2014]. Com o advento de sistemas de larga escala, que recebem milhões de usuários diariamente (e.g. buscador *Google*, redes sociais, etc.), o estudo do *Design* de Sistemas Distribuídos tornou-se imprescindível para muitos profissionais do mercado de Engenharia de *Software*.

De certo modo, o *Design* de Sistemas na Engenharia de *Software* tem como base a Engenharia de Sistemas convencional e portanto seus procedimentos são similares. Assim, uma abordagem metódica de *systems design* pode englobar as seguintes etapas [NASA 2017]:

1. Definição das expectativas dos atores: identificação dos atores e como eles pretendem interagir com o produto;
2. Definição dos requerimentos técnicos: transformação de expectativas em requerimentos técnicos;
3. Decomposição lógica: análise funcional para a criação de uma arquitetura do sistema e alocação de requerimentos gerais em componentes específicos;
4. Definição de solução de *design*: tradução das expectativas dos atores e da decomposição lógica em uma solução de *design*.

Neste trabalho, utilizaremos uma versão desta metodologia de *design*, porém mais direta e adaptada ao desenvolvimento de sistemas distribuídos de *software*.

Além de apenas satisfazer os pré-requisitos específicos, o objetivo do *Systems Design* é também conferir ao sistema os três seguintes pilares [Kleppmann 2021]:

- *Confiabilidade*: operação em serviço contínuo, mesmo que hajam falhas;
- *Escalabilidade*: habilidade de operar em cargas maiores (e variáveis);

- *Sustentabilidade*: facilidade de manutenção, operação e aperfeiçoamento.

2.2.1.1 Implementação de sistemas

Para a implementação do *Design* de Sistemas, o estado da arte recomenda a utilização de uma *linguagem declarativa* de Infrastructure-as-Code (infraestrutura como código) (IaC), i.e. uma linguagem onde o programador apenas declara o que quer que seja feito, sem a necessidade de especificar as instruções e sua ordem [Etheredge 2020]. Assim, uma vez decidida a arquitetura a ser implementada, o engenheiro necessita apenas descrevê-la corretamente à maquina, no formato de código. Além disso, por causa do uso de ferramentas como *Git* (ferramenta de gerenciamento de código fonte [Chacon e Straub 2014]), os estados da arquitetura são sempre salvos e há a possibilidade de reversão (*rollback*).

Dentre as opções de linguagens declarativas, destaca-se o *Terraform*.

2.2.1.2 Terraform

Terraform é uma ferramenta agnóstica de IaC, desenvolvida pela *Hashicorp* [Terraform 2023]. Por se tratar de IaC, ela possibilita definir infraestruturas complexas apenas com código, utilizando uma *linguagem declarativa* que não se restringe a nenhum provedor específico.

Como pontos positivos, podemos citar: todas as vantagens de IaC [Etheredge 2020]; replicabilidade da infraestrutura para múltiplas regiões e versões; agnosticismo, não necessitando de muito esforço para migrar de provedor; modularidade, possibilitando a definição de módulos personalizados e sua consequente reutilização; e finalmente, a existência de diversos módulos prontos para inúmeros componentes em diversos provedores (incluindo *Kubernetes*), facilitando o desenvolvimento [Terraform 2023].

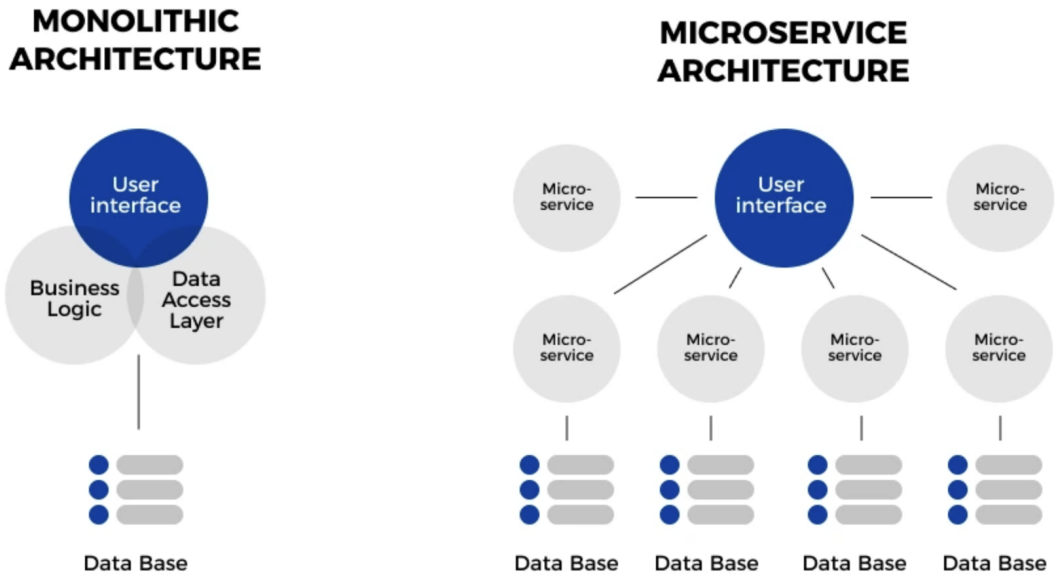
Terraform apresenta algumas desvantagens, como: a necessidade de armazenar e distribuir um arquivo contendo o estado atual da infraestrutura, pois a ferramenta não consegue extrair automaticamente o estado; a exigência de aprender uma nova linguagem; dificuldade de trabalhar com infraestrutura previamente existente; entre outros [Janczak 2018].

Apesar das desvantagens, o *Terraform* se tornou uma das melhores ferramentas agnósticas de IaC, suportando diversos componentes. Além disso, são poucas as ferramentas de IaC disponíveis no mercado. Os maiores provedores possuem suas próprias ferramentas, como o *AWS CloudFormation*. Portanto, as alternativas são escassas.

2.2.1.3 Arquitetura em micro-serviços

A *arquitetura em micro-serviços* é um padrão que define uma aplicação como um conjunto de pequenos serviços fracamente acoplados, onde cada um possui uma única responsabilidade [Davis 2022]. Para melhor ilustrar, o oposto de micro-serviços é a *arquitetura em monólito*, em que toda a lógica está concentrada em um único serviço, responsável por tudo. A Fig. 3 exemplifica esta diferença.

Figura 3 – Comparativo entre arquitetura em monólito e arquitetura em micro-serviços.



Fonte: [Davis 2022].

A principal vantagem dos micro-serviços em relação ao seu oposto é a possibilidade de melhor dividir o desenvolvimento, já que cada pequeno time de desenvolvimento pode ser responsável por um serviço e lançá-lo em produção independentemente dos outros. Outra vantagem é a possibilidade de escalar cada serviço independentemente, de acordo com a demanda heterogênea, e assim ter uma melhor economia de recursos. Portanto, em grande escala, os micro-serviços são muito utilizados.

No entanto, seu uso não é um consenso, especialmente para pequenos times de desenvolvimento e para pequenas empresas, visto que suas duas vantagens são irrisórias neste contexto [Fernandez 2022].

Dito isso, utilizaremos a arquitetura em micro-serviços no desenvolvimento deste trabalho, por melhor se adequar ao caso de uso (arquitetura para IoT em grande escala).

2.2.2 Computação em Nuvem

Mais conhecido como *Cloud Computing* em inglês, a Computação em Nuvem é uma tecnologia que possibilita provedores fornecerem diversos recursos computacionais como

um serviço, para satisfazer as necessidades dos seus usuários [Ray 2018]. Um exemplo clássico é o serviço *Elastic Compute Cloud* (EC2), da AWS: com ele, o usuário pode alugar poder de processamento computacional e pagar apenas pelas horas usadas [Amazon 2023].

Historicamente, o *Cloud Computing* veio como uma alternativa à prática da computação *on-premises*, i.e. aquisição e manutenção de servidores próprios por uma empresa [IBM 2017]. Normalmente, dados os elevados custos, tais empresas eram de grande porte. Assim, o *Cloud Computing* satisfez as necessidades de pequenas e médias empresas, ainda mais na atual era das *startups*. Hoje em dia, devido às suas vantagens, até mesmo grandes empresas realizam uma migração para esta modalidade de computação.

Dada a sua onipresença desta modalidade de computação, o *Systems Design* é associado à Computação em Nuvem em muitos contextos, visto que o *design* é geralmente implementado em nuvem.

2.2.2.1 Kubernetes

Muito utilizada em *cloud computing*, *Kubernetes* (K8s) é uma ferramenta de orquestração de contêineres, criada pelo Google em 2014. Por sua vez, contêineres são pequenas máquinas virtuais, que contém um *software* e todas as suas dependências [Docker 2023] – com o intuito de padronizar a execução, facilitar sua distribuição em múltiplos sistemas e reduzir a interferência de fatores externos, como sistema operacional do *host* (computador anfitrião).

Assim, o *Kubernetes* atua na coordenação destes contêineres, provendo uma série de funcionalidades para facilitar a logística, como: *autoscaling* (flexibilidade no número de instâncias de um componente de acordo com a demanda), *rollbacks* automatizados, auto-regeneração de contêineres defeituosos/não responsivos, entre outros [Kubernetes 2023]. Tudo isto pode ser administrado por uma aplicação com sintaxe simples e com uma boa integração com *Terraform*, portanto trazendo grandes vantagens à sua utilização. Não é à toa que a ferramenta está se tornando cada vez mais onipresente.

No entanto, a curva de aprendizado pode ser íngreme e algumas situações específicas necessitam de muito conhecimento do funcionamento interno do *Kubernetes* e um grande esforço de pesquisa. Além do mais, dada a complexidade, em muitos casos a sua utilização não se justifica.

2.2.3 Componentes de infraestrutura computacional

Dado o enfoque do presente trabalho, serão brevemente apresentados alguns dos componentes mais relevantes de uma infraestrutura computacional.

2.2.3.1 Banco de dados

Os bancos de dados são componentes responsáveis pelo armazenamento persistente de dados [Oracle 2023]. Estes dados podem estar organizados de maneira relacional (tipo *Structured Query Language* (SQL), similares a uma tabela *Excel*) ou não (tipo *NoSQL*). A Fig. 4 exibe um exemplo de uma tabela de dados relacionais em MySQL.

Figura 4 – Exemplo de uma tabela em MySQL.

```
mysql> select * from ConsumerDetails;
+----------+--------------+-----------------+---------------+
| Name     | Locality     | Total_amt_spend | Industry      |
+----------+--------------+-----------------+---------------+
| Raj      | Raj Nagar    |             750 | Manufacturing |
| Ajay     | Vijay Nagar  |             500 | Creative      |
| Sagar    | Shivam Nagar |             900 | News          |
| Akul     | Preet Vihar  |             350 | Teaching      |
| Rohan    | kakar Vihar  |            1150 | Tech          |
| Shantanu | Shanti Vihar |            2110 | Defense       |
| Natasha  | shakti nagar |            2200 | Aviation      |
| Kapil    | shakti nagar |             700 | Aviation      |
| Tanamy   | sikkim nagar |             900 | Defense       |
| Tarun    | nikepur      |            3000 | Manufacturing |
+----------+--------------+-----------------+---------------+
10 rows in set (0.00 sec)
```

Fonte: [Dewani 2020].

Os *Database Management Systems* (DBMS) (Sistemas de Gestão de Banco de Dados) são tecnologias que possibilitam a interação entre usuários e os bancos de dados, definindo assim todos os aspectos que caracterizam o banco de dado em si [Oracle 2023]. Por exemplo, o *MySQL* é um DBMS simples, que possui sua própria sintaxe, seu modo de armazenamento e suas características de desempenho.

2.2.3.2 Servidor web

Um servidor *web* (ou mais conhecido como *web server*) é um *software* capaz de responder às requisições *HTTP* vindas dos clientes a ele conectados, assim servindo as informações requisitadas [Yeager, MacGrath e McGrath 2000].

É preciso salientar que tais informações não são armazenadas no *web server* diretamente, mas sim em um banco de dados a ele conectado. Outro ponto importante é que normalmente o servidor *web* é responsável pela autenticação e autorização das requisições, podendo aceitá-las ou não dependendo do cliente e da informação requisitada.

Pelo uso do protocolo *HTTP*, existem alguns tipos importantes de requisições a serem destacados [MozDevNet 2023]:

- *GET* (OBTER): são requisições básicas de informação, como visualizar detalhes de um objeto ou obter o catálogo de itens de um site de varejista. Dependendo do que se quer obter, pode ser protegido ou não;

- *POST* (PUBLICAR): são requisições que alteram os dados, seja atualizando-os ou adicionando itens. Por questões de segurança, tais requisições são majoritariamente protegidas por autenticação;

- *DELETE* (APAGAR): como diz o nome, são requisições para apagar um objeto. Também são majoritariamente protegidos.

2.2.3.3 Agente de mensagens

Mais conhecidos como *message brokers*, os agentes de mensagem são componentes que administram o recebimento e envio de mensagens entre os outros componentes que estão conectados a eles [IBM 2023].

Neste contexto, mensagens podem ser qualquer tipo de conteúdo que possa ser expressado em *bytes*, como texto, números, imagens, etc.

Além da função principal de mediação e roteamento de mensagens, *message brokers* possuem recursos adicionais, como: armazenamento (efêmero ou permanente), agregação de mensagens, enfileiramento i.e. garantia de ordenamento correto na entrega, etc.

Algumas das tecnologias de agente de mensagens mais utilizadas na indústria são: *Apache Kafka*,*Apache ActiveMQ*, *RabbitMQ*, *Redis*, entre outros.

2.2.3.4 Banco de dados em memória

Os bancos de dados em memória funcionam de maneira similar aos bancos de dados convencionais, porém seu armazenamento é realizado em memória volátil (e.g. memória RAM) [Amazon 2023]. Isto garante mais velocidade em operações de escrita e leitura, porém não podem ser utilizados para armazenamento a longo prazo e possuem uma limitação de espaço maior.

Dados estes aspectos, os bancos de dados em memória são ideais como memória compartilhada em sistemas distribuídos, pela velocidade e efemeridade dos dados. Assim, caso seja necessário compartilhar dados em memória entre diferentes processos, podemos utilizar soluções como esta.

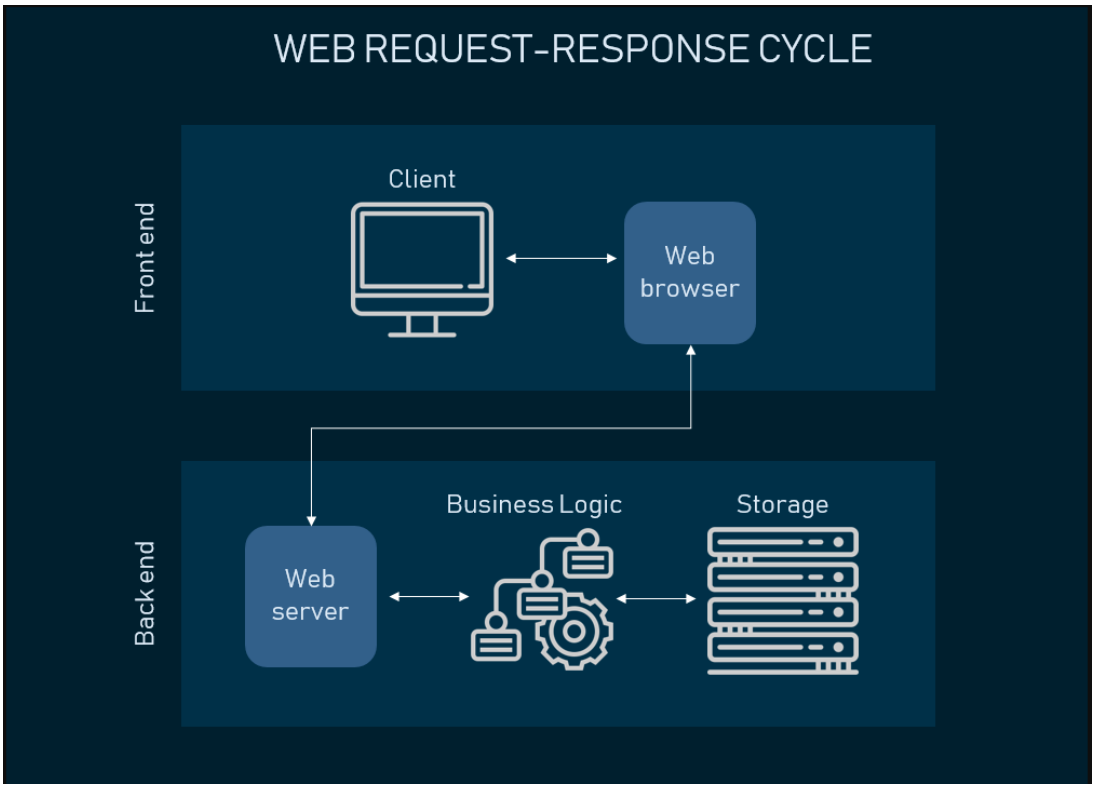
A principal tecnologia representante desta classe é o previamente citado *Redis*, que armazena pares chave-valor.

2.2.4 Desenvolvimento de sistemas back-end

No vasto campo da Engenharia de *Software*, o desenvolvimento de sistemas *back-end* se concentra principalmente na interface entre usuários/clientes e os dados. Em termos de responsabilidades, isso se traduz em: processamento de dados; validação de dados; na imposição da lógica de negócios; entre outras [Souto 2023].

A Fig. 5 mostra um ciclo comum de requisição-resposta na Web, ilustrando o papel do sistema *back-end*: o *client*, i.e. navegador *Web* operado pelo usuário, faz uma requisição ao *back-end*; o servidor *Web* recebe tal requisição e nela aplica a *business logic* (lógica de negócios) i.e. validação da requisição, processamento dos dados, geração da busca a ser feita no banco de dados, etc.; por fim, o banco de dados é consultado e os dados são atualizados e/ou buscados no *storage* (armazenamento), para gerar uma resposta à requisição original.

Figura 5 – Diagrama do ciclo requisição-resposta na Web.



Fonte: [Altexsoft 2019].

No entanto, pelo aumento da complexidade das aplicações e da gama de serviços que se utilizam de *software*, atualmente muitas aplicações fogem deste ciclo comum, adicionando múltiplas etapas de *business logic*, recebendo múltiplas fontes de dados, entre outros fatores.

Por fim, relacionando o tema com *Computação em Nuvem* e *Systems Design*, o engenheiro de *software back-end* se utiliza de servidores *web*, bancos de dados e outros componentes para construir este sistema.

2.2.4.1 Linguagens e ferramentas

Para implementação dos sistemas *back-end*, o engenheiro de *software* utiliza linguagens de programação *imperativa* i.e. linguagens onde o programador deve impor instruções específicas a serem executadas em uma certa ordem. Diversas linguagens são utilizadas para este fim, como: *Python*, *Java*, *Javascript*, *Golang*, etc. Cada uma delas tem suas vantagens e desvantagens sobre diversos aspectos, como desempenho e velocidade de desenvolvimento – e portanto, é necessária a reflexão na escolha de linguagem.

Além da programação em si, outros processos paralelos são essenciais para o bom desenvolvimento de um *software*. Um deles é o *versionamento de código*, i.e. manutenção do histórico de cada versão funcional do código [Chacon e Straub 2014]. Ele ajuda no caso de algum problema em uma nova versão ou na colaboração de múltiplos engenheiros em uma única parte do código. Na maioria dos casos, o versionamento é feito utilizando o *Git*, uma ferramenta amplamente utilizada na indústria: *GitHub* e *GitLab*, por exemplo, o utilizam para versionar e armazenar o código de milhões de projetos.

2.2.4.2 Python

Python é uma linguagem de programação de alto nível. Além de ser uma linguagem de *scripting*, ela também possibilita outros paradigmas de programação, como *Object-Oriented Programming* (programação orientada a objetos) e *programação funcional*.

Python possui muitos pontos positivos, como: alta velocidade de desenvolvimento, por conta da sua sintaxe simples; versatilidade de paradigmas de programação; ampla variedade de ferramentas existentes para qualquer aplicação, como processamento de imagens, programação científica, entre outros; e alta popularidade, visto que foi a quarta linguagem de programação mais utilizada por desenvolvedores profissionais em 2022 [StackOverflow] e portanto há amplo suporte e recursos sobre a linguagem na Internet.

Como pontos negativos, podemos brevemente citar: a baixa velocidade de execução; consumo elevado de memória; e a falta de tipagem forte i.e. não forçar o uso coerente de tipos nas variáveis, o que a torna suscetível a erros [AltexSoft 2021].

3 Metodologia

Neste capítulo, serão descritos todos os processos e métodos utilizados durante o desenvolvimento deste trabalho.

3.1 Definição de caso de uso e requerimentos

Como plano de fundo do objeto de estudo deste trabalho, foi necessária a escolha de um caso de uso de processamento de dados de IoT. A escolha seguiu alguns dos seguintes critérios, para melhor ilustrar a situação:

- Necessidade de armazenamento de dados históricos em tempo real;
- Carga variável ao longo do tempo;
- Necessidade de processamento em tempo real.

3.1.1 Requerimentos da lógica de negócios

Assim, o caso de uso escolhido como exemplo foi o *monitoramento de máquinas em um ambiente fabril*. Neste contexto, a partir dos pontos acima, arbitramos [1] que o sistema deve realizar as seguintes funções:

- Contabilizar unidades produzidas (funcionais e defeituosas) por máquina monitorada;
- Armazenar o identificador (ID) de cada unidade produzida e seu estado;
- Calcular dados estatísticos para cada máquina: média móvel de porcentagem de unidades defeituosas produzidas e média móvel de temperatura e pressão internas;
- Notificar o usuário responsável via e-mail em caso de: mais de 5% de unidades defeituosas em uma dada máquina na média móvel e/ou média móvel de pressão ou temperatura internas de uma máquina superarem seu valor nominal;
- Suportar uma carga variável de requisições, com picos de carga.

[1] Os requisitos abaixo foram escolhidos de modo a permitir que testes de carga do sistema, sem divergir daquilo que seria encontrado na prática.

3.1.2 Requerimentos técnicos

Já em termos técnicos, o sistema deve cumprir os seguintes pré-requisitos:

- Conectar-se a 100 sensores simultaneamente, recebendo em média 10 requisições por minuto por sensor;
- Processar e armazenar, no mínimo, 95% das requisições corretamente;
- Gerar notificações ao usuário com, no máximo, 1 minuto de atraso desde o ocorrido.

Por fim, para maximizar a portabilidade e evitar o *vendor lock-in* i.e. a dependência técnica de um provedor *cloud* específico por conta de inúmeros fatores, tentaremos limitar nosso uso de serviços *específicos* do provedor escolhido. Para isso, priorizaremos o uso de componentes auto-gerenciados, em vez de utilizar serviços gerenciados pelo provedor.

A partir deste contexto, pudemos melhor exemplificar casos de uso reais para arquiteturas em nuvem escaláveis para processamento de dados IoT.

É importante frisar que o presente trabalho pode ser facilmente adaptado para outros casos de uso e que a escolha específica deste é apenas para fins ilustrativos, de uma prova de conceito (*proof-of-concept*).

3.1.3 Métricas sensoreadas do caso de uso

Como parte do caso de uso especificado, aqui definiremos as métricas das máquinas fabris captadas pelos sensores e posteriormente enviados pelos *gateways* para nossa camada de aplicação. É necessário observar que, neste contexto, cada *payload* (pacote) representa a medição referente à uma unidade de produto produzida. A Tabela 1 especifica cada campo do pacote.

Tabela 1 – Dados do pacote.

Nome do campo	Descrição	Tipo do dado	Requerido
unit_id	Identificador da unidade produzida.	Número inteiro	Sim
created_at	Data e hora de produção da unidade.	*String* de data	Sim
is_defective	Indica se a unidade produzida é defeituosa.	Booleano	Não
machine_id	Identificador da máquina que produziu a unidade.	Número inteiro	Sim
machine_temperature	Temperatura interna da máquina no momento de produção da unidade.	Número real positivo	Não
machine_pressure	Pressão interna da máquina no momento de produção da unidade.	Número real positivo	Não

Fonte: elaborado pelo autor.

A nível do servidor de aplicação, cada pacote recebido tem sua estrutura verificada e caso não seja conforme a Tabela 1, o pacote é ignorado.

3.2 Escolha das tecnologias e ferramentas

Nesta seção serão abordadas algumas das tecnologias e ferramentas utilizadas, para melhor situar o leitor e justificar as decisões técnicas tomadas.

3.2.1 Python

Dado o exposto em 2.2.4.2, *Python* foi escolhida como linguagem para o desenvolvimento deste trabalho, tendo em vista a alta versatilidade — e portanto, a existência de múltiplos módulos compatíveis com os protocolos utilizados em IoT. Melhores resultados em desempenho podem ser alcançados com outras alternativas, mas a velocidade de desenvolvimento é uma prioridade.

3.2.2 Kubernetes

No desenvolvimento do presente trabalho, utilizamos *Kubernetes* para implantar nossa arquitetura, principalmente pela sua função de *autoscaling*, pela integração com *Terraform* e pela arquitetura em micro-serviços que utilizaremos.

3.2.3 Terraform

Apesar das desvantagens expostas em 2.2.1.2, utilizaremos *Terraform* como ferramenta de IaC pela sua versatilidade com qualquer provedor e pela sua alta adoção.

3.2.4 Google Cloud Platform

GCP é a provedora de serviços em nuvem da gigante de tecnologia. Como qualquer outra provedora, ela oferece serviços relacionados a: computação em nuvem; banco de dados; inteligência artificial; até mesmo soluções em IoT.

Como principal ponto positivo, no contexto deste projeto, a GCP oferece US$ 300 para uso de recursos do nível gratuito. No restante, todas as grandes provedoras (AWS, *Microsoft Azure*, etc.) não apresentam diferenças técnicas significativas. Portanto, utilizamos a GCP para a implementação em nuvem deste projeto.

3.3 Ciclo de desenvolvimento

Durante todo o processo de desenvolvimento dos *softwares* pertinentes a este trabalho, foram utilizados conceitos presentes na metodologia *Agile* (ágil), como a prototipagem rápida e entrega incremental e contínua [Hat 2022]. Também foi empregado o uso de *Test-Driven Development* (desenvolvimento orientado a testes), onde os testes são escritos antes do desenvolvimento do código em si [Beck 2015].

Seguindo a cronologia do ciclo de desenvolvimento, começamos com o *systems design* – o processo de concepção da arquitetura, desde a análise dos requerimentos até sua implementação em nuvem. Isso se justifica pois a arquitetura define e distribui as responsabilidades dos outros componentes, principalmente dos micro-serviços e do método de conexão entre eles e os *gateways*.

Logo após os primeiros passos dados em relação ao *systems design*, iniciou-se o desenvolvimento dos *softwares* ligados à implementação da lógica de negócios, em formato de micro-serviços. Como dito anteriormente, tais funcionalidades foram testadas durante o próprio desenvolvimento, assim garantindo seu funcionamento já na implementação. Ademais, pela aspecto modular de um micro-serviço, alguns deles puderam ser desenvolvidos até mesmo em paralelo à própria arquitetura, assim agilizando o processo de desenvolvimento.

Por fim, foram realizados os testes de carga com as diversas arquiteturas, de forma a avaliar o cumprimento dos pré-requisitos técnicos e de lógica de negócios. O próprio ambiente de testes também teve que ser desenvolvido, visto que utiliza-se de *scripts* de programação para simular o papel dos *gateways* enviando dados. Assim, pudemos de fato emular o comportamento esperado das entradas e observar se o sistema realizava o comportamento esperado na geração das saídas.

Nas seções seguintes, abordaremos cada etapa descrita anteriormente em mais detalhes.

3.4 Design da arquitetura

Para iniciar o processo de *systems design*, foram analisados os requerimentos do sistema por duas perspectivas: lógica de negócios e técnica.

Pela primeira perspectiva, foi necessário avaliar diversas necessidades do contexto da lógica de negócios. Por exemplo, a necessidade de armazenamento a longo prazo indica a necessidade de uma base de dados. Já a necessidade de notificação de usuários geralmente indica o uso de um serviço específico para esta finalidade, como *Twilio*, *Firebase* ou *Sendgrid*.

Pela perspectiva técnica, o processo foi similar: para suportar a variação de carga, provavelmente é necessário adicionar escalabilidade ao sistema i.e., monitorar a taxa de uso dos recursos e demandar mais ao provedor *cloud*, caso necessário; para não haver perda de dados no caso de sobrecarga, pode ser necessário algum armazenamento de curto prazo, como uma *fila* ou sistema de *cache*.

É importante ressaltar que esta avaliação é feita de maneira sistemática, portanto ambas perspectivas foram avaliadas ao mesmo tempo no processo de concepção. A partir

desta análise inicial, obteve-se uma noção do que poderia ser utilizado e o que deveria ser descartado.

Após esta primeira etapa, pudemos analisar um pouco mais a fundo o aspecto de banco de dados e de modelagem de dados, visto que este é um componente muito importante: caso mal escolhido, pode causar perda de dados e ser um grande ponto de *bottleneck* (gargalo) da arquitetura. Dentre os critérios analisados, podemos destacar: razão entre leitura e escrita de dados; escalabilidade; necessidade de acessar dados em um contexto mais local (e.g. acessar apenas uma medição de uma máquina específica) ou mais global (e.g. acessar todas as medições de todas as máquinas); entre outros. Tudo isso define se usaremos um banco de dados do tipo *SQL* (e.g. *MySQL*, *PostgreSQL*) ou *NoSQL* (e.g. *MongoDB*, *Cassandra*) e qual das alternativas deste tipo usar, sempre levando em conta os *tradeoffs* oferecidos.

A mesma análise pôde ser feita em relação aos protocolos de comunicação entre a camada de rede e a camada de aplicação, já que podemos, por exemplo, conectar nossos *gateways* diretamente com o servidor de aplicação ou utilizar um método mais clássico: receber requisições HTTP do tipo *POST* com os dados.

A partir de todas estas análises, todos os aspectos foram examinados em conjunto e foi concebida uma arquitetura, que definiu os componentes e micro-serviços necessários.

Em seguida, foi iniciada a implementação da primeira arquitetura. Inicialmente, o desenvolvimento foi realizado em ambiente local, pela facilidade de prototipagem e pelo custo zero. Uma vez terminado o desenvolvimento e validado o funcionamento da arquitetura em ambiente local (incluindo o funcionamento dos micro-serviços, desenvolvidos em paralelo quando possível), prosseguimos para o *deployment* (implantação) no ambiente *cloud*, apenas adicionando algumas configurações extras ao IaC – pelo uso de *Kubernetes* e *Terraform*, não há diferença significativa no código entre a versão local e a versão *cloud*.

3.5 Desenvolvimento dos micro-serviços

Uma vez que as responsabilidades e o escopo de cada micro-serviço foi definido, foi iniciado o desenvolvimento dos micro-serviços que compõem a arquitetura. Tal implementação foi feita em paralelo com a implementação da arquitetura em si.

3.5.1 Modelagem de dados

Antes do desenvolvimento do micro-serviço de processamento, foi necessário definir quais dados gostaríamos de armazenar em nosso banco de dados. Para isso, utilizamos o contexto provido pelos requerimentos do sistema e pelos dados enviados pelos *gateways* (Tabela 1).

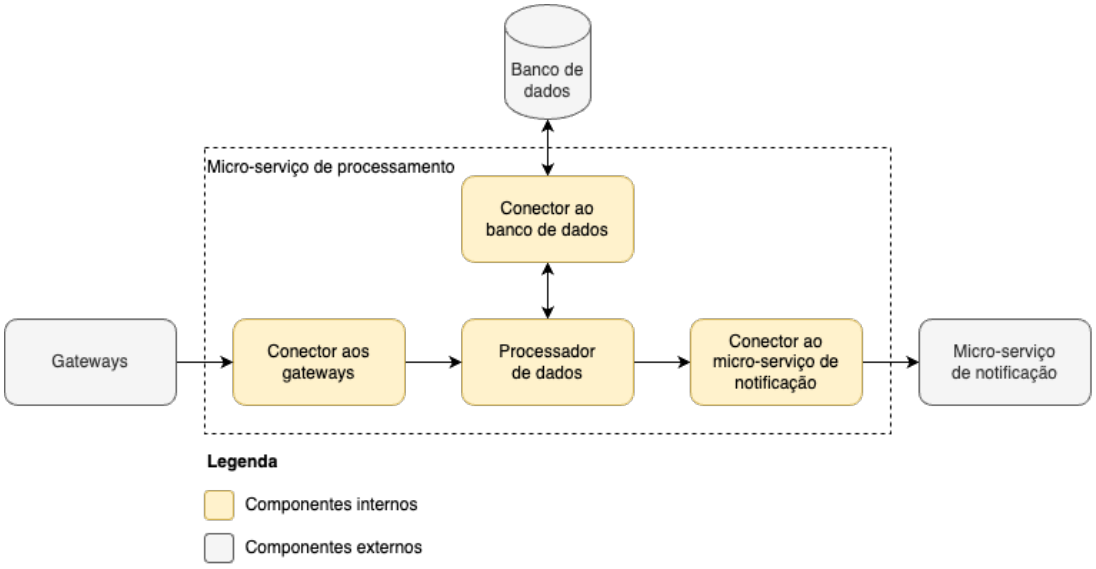
Também utilizamos os conceitos de normalização de base de dados, que em suma tentam reduzir a redundância entre os dados e melhorar sua integridade. Para isso, foi necessário dividir os dados providos no *payload* em três modelos diferentes, de modo a facilitar a análise e melhorar o desempenho do banco de dados. Tais modelos são: *machine* (máquina), *unit* (unidade produzida) e *measurement* (medida).

3.5.2 Micro-serviço de processamento

Uma vez definida a modelagem de dados, prosseguimos para o desenvolvimento do micro-serviço de processamento. Como diz o nome, este componente deve receber os dados dos *payloads* e processá-los i.e. buscar os dados relevantes no banco de dados, calcular se uma notificação é necessária e armazenar as novas informações no banco, seguindo a modelagem escolhida.

Iniciou-se o desenvolvimento do micro-serviço utilizando a linguagem *Python*. Para realizar suas tarefas, ele possui quatro componentes internamente, que serão descritos a seguir. A Fig. 6 exibe um diagrama interno deste micro-serviço.

Figura 6 – Diagrama simplificado interno do micro-serviço de processamento.



Fonte: elaborado pelo autor.

O primeiro componente é o *conector aos gateways*, que se responsabiliza por receber os dados vindos da camada de rede da nossa aplicação e passá-los ao *processador de dados* (discutido a seguir). Neste caso, exploramos o protocolo MQTT e portanto, este componente deve se inscrever no tópico relevante e receber os *payloads*. Além disso, este componente verifica a validade dos pacotes enviados (se possuem o tipo certo e se todos os campos requeridos num *payload* estão presentes).

O segundo componente é o *processador de dados* em si, que consiste em: uma função que receba os dados e coordene todos os passos necessários com os outros componentes

para efetivamente processar os dados requeridos.

O terceiro componente é o *conector ao banco de dados*, que se conecta a este elemento e abstrai as requisições necessárias para ler e escrever dados. Igualmente ao *conector de gatways*, ele deve ser modular e abstrair as especificidades do banco de dados escolhido.

Finalmente, o quarto componente é o *conector ao micro-serviço de notificação*, que comunica a este micro-serviço a necessidade de um alerta e os dados relevantes, como o identificador da máquina e seus parâmetros.

3.5.3 Micro-serviço de notificação

Este micro-serviço, muito menor que o anterior, possui apenas a função de se conectar ao serviço de envio de e-mail da *Sendgrid* e por meio dela enviar um e-mail formatado com as notificações. Isso tudo de acordo com a demanda vinda do micro-serviço de processamento.

A necessidade deste micro-serviço estar separado do micro-serviço de processamento se justifica pela dependência de comunicação síncrona com um componente externo: caso este esteja indisponível ou apresente alta latência, a função de processamento não ficará comprometida e o impacto desta latência é mitigado apenas a este micro-serviço.

A partir de testes, verificou-se a necessidade de integração com um banco de dados em memória: visto que enviar um e-mail para cada *payload* recebido em um momento de crise é insustentável, especialmente pois o número de e-mails enviados é limitado pelo serviço. Assim, utilizamos o *Redis* para definir um período mínimo entre e-mails e armazenamos todas as notificações deste período, enviado-as em uma única mensagem ao final dele.

3.6 Desenvolvimento e condução de testes de carga

Para desenvolver os testes de carga, é necessário que emulemos as entradas do nosso sistema da maneira mais fiel possível e observar as saídas esperadas. Podemos automatizar este processo, por meio de um *script*: um roteiro reprodutível a ser seguido pelo computador, que enviará vários *payloads* ao nosso sistema, com uma taxa variável para simular picos e estressá-lo.

Como ferramenta escolhida, foi utilizado *Locust*: uma ferramenta *open-source* (código aberto) em *Python* compatível com diversos protocolos, inclusive MQTT. Cabe ao desenvolvedor programar o comportamento de cada "usuário" (i.e. sensor neste contexto), definindo em qual tópico ele deve se conectar, quais mensagens deve enviar e qual a frequência de envio.

Em seguida, para executar um teste, é necessário definir: o número máximo de usuários concomitantes e a taxa de aceleração até atingir este máximo (e.g. adicionar dois usuários por segundo). A ferramenta então simula-os, gerando gráficos e relatórios sobre diversas métricas esperadas em um teste de carga, como número de requisições, latência, número de requisições falhas, etc.

Além disso, durante este processo de teste, foram observadas as métricas dos recursos consumidos pelo sistema e se houve a necessidade do sistema escalar de acordo com o número de *payloads*/usuários conectados. Isto para verificar se houve necessidade de escalabilidade.

Em relação ao ambiente dos testes, eles foram realizados primeiramente em um *cluster* local, utilizando a ferramenta *Minikube*, que permite a execução de um pequeno *cluster Kubernetes* em qualquer computador. Deste modo, foi possível observar o comportamento da arquitetura em menor escala e validar a metodologia de testes. Em seguida, os testes foram conduzidos na arquitetura em nuvem, na GCP. Além do teste em carga nominal, foram feitos experimentos com cargas elevadas (ordens de grandeza acima), para verificar o limite da aplicação.

4 Resultados

Neste capítulo serão apresentados os resultados obtidos ao decorrer deste trabalho, seguindo a metodologia exposta no capítulo anterior.

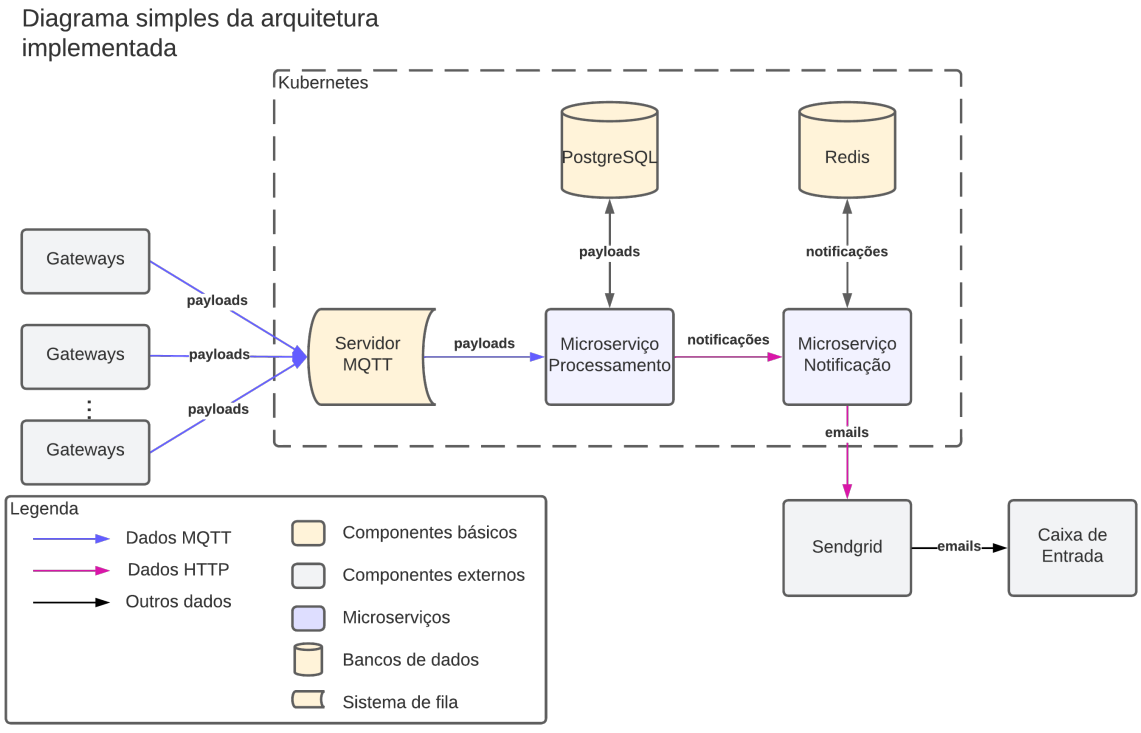
Os códigos-fonte da arquitetura, dos micro-serviços e dos testes podem ser acessados na íntegra no repositório público do projeto no *GitHub* [Yamakoshi 2023]. No arquivo *README.md* no repositório, também há breves instruções de como instalar e executar todos os *softwares* aqui descritos.

4.1 Design da arquitetura

4.1.1 Visão geral

Seguindo o exposto na Seção 3.4, a arquitetura da Fig. 7 foi concebida, seguindo alguns princípios. A seguir, serão descritos os componentes presentes nesta arquitetura e suas respectivas justificativas.

Figura 7 – Diagrama simplificado da arquitetura implementada.



Fonte: elaborado pelo autor.

Primeiramente, temos o servidor MQTT, responsável por receber os *payloads* diretamente dos *gateways* e enviá-los em direção ao micro-serviço de processamento. Neste contexto, este componente age como um *agente de mensagens* e uma fila, acumulando mensagens até que elas sejam consumidas por alguma instância do micro-serviço. Uma enorme vantagem é justamente o protocolo MQTT: ambos *gateways* e micro-serviços podem se comunicar diretamente o utilizando. Outro ponto positivo é a comunicação assíncrona, diferentemente do HTTP, possibilitando o enfileiramento de mensagens de modo simplificado. Em relação ao *broker* específico, escolheu-se o *Eclipse Mosquitto*, por ser leve e possuir amplo suporte da comunidade, facilitando o desenvolvimento com ele.

Em seguida, temos os dois micro-serviços, de processamento e de notificação, que serão melhor abordados na Seção 4.2. Mas em suma, ambos atuam em conjunto para receber os dados, processá-los, armazená-los e verificar se houve a superação de algum dos valores nominais. No caso positivo, agrupa-se os dados relevantes e é requisitado ao serviço de e-mail *Sendgrid* o envio de uma mensagem contendo as informações mais relevantes.

Em continuidade, destaca-se o uso do banco de dados relacional *PostgreSQL*, responsável por armazenar os *payloads* já processados pelo micro-serviço de processamento. A justificativa da escolha desta tecnologia se dá principalmente pela robustez deste tipo de banco de dados relacional. No entanto, vale destacar que nesta escala reduzida, a escolha de banco de dados pouco afeta o desempenho.

Posteriormente, há o uso de *Redis* como banco de dados em memória chave-valor. Como mencionado anteriormente, esta tecnologia é onipresente na indústria para esta finalidade e de fácil interação via bibliotecas em *Python*, portanto sua escolha foi trivial. Porém, justificando a necessidade de um componente como este, temos a necessidade de compartilhar dados em memória entre as diferentes *threads* das requisições ao micro-serviço de notificação.

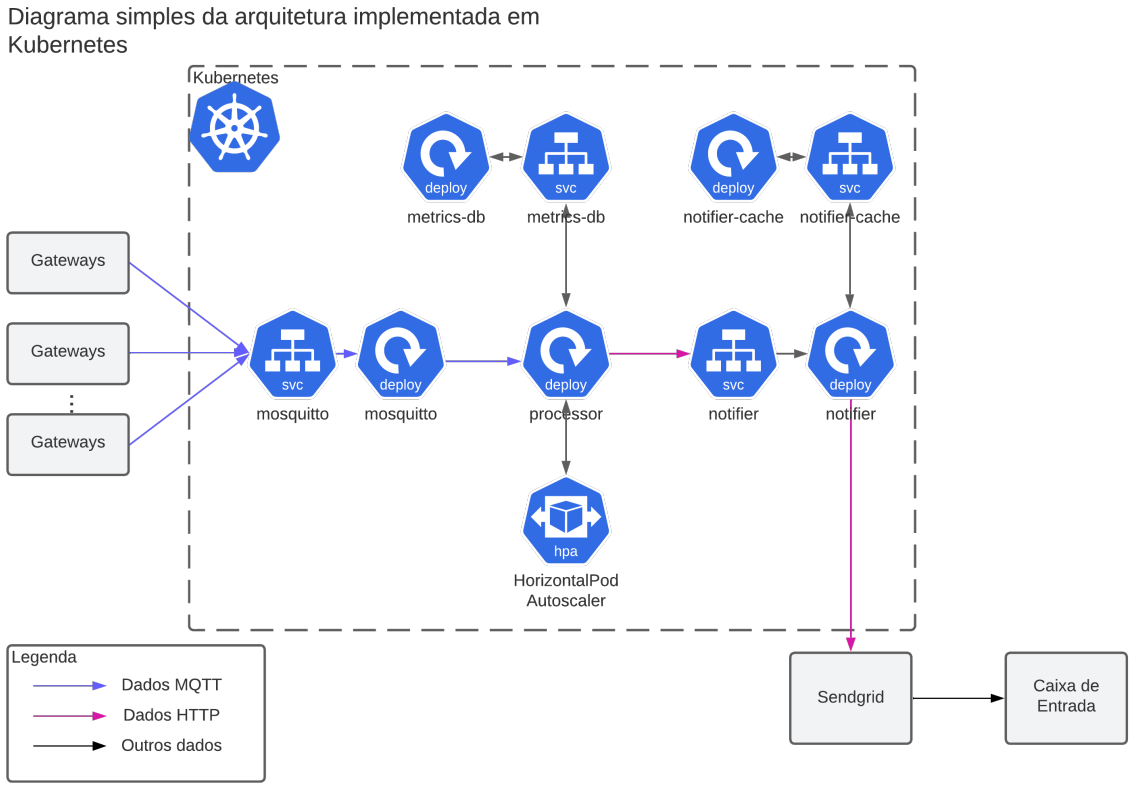
Por fim, foi utilizado o serviço *Sendgrid* para envio de e-mails automatizados. Dentre os possíveis serviços, este possuía uma cota considerável no nível gratuito, além de oferecer uma biblioteca para integração em *Python*.

4.1.2 Implementação em Kubernetes

Utilizando *Terraform*, o diagrama geral apresentado na Fig. 7 foi implementado em um *cluster Kubernetes*. A Fig. 8 ilustra um diagrama mais detalhado, utilizando os nomes definidos em código para cada componente. Vale observar que outros diversos componentes foram omitidos, por clareza.

Os detalhes de implementação do *Kubernetes* estão fora do escopo deste trabalho, mas em suma, cada componente na arquitetura da Fig. 7 foi substituído por um *Deployment* e possivelmente um *Service*: o primeiro é um grupo de contêineres, executando a aplicação;

Figura 8 – Diagrama da arquitetura implementada em Kubernetes.



Fonte: elaborado pelo autor.

enquanto o último é um componente de rede que expõe este grupo de contêineres à rede interna da arquitetura, assim possibilitando que outros componentes possam lhe acessar.

Ademais, destaca-se o *Horizontal Pod Autoscaler* (HPA) (escalador automático de *pods* horizontal), que ao se conectar com o micro-serviço de processamento, possibilita que o *cluster* gere mais réplicas deste componente. Este processo ocorre de maneira automática e baseado em métricas definidas pelo arquiteto, como por exemplo uso de CPU média dos contêineres ou uso médio de memória. Assim, caso uma das métricas ultrapasse o limite estipulado, novos contêineres do *Deployment* serão gerados até que o nível médio da métrica esteja abaixo do estipulado ou até que atinja-se o número máximo de réplicas. Na última versão da arquitetura, o micro-serviço de processamento possuía uma réplica inicialmente, podendo atingir até três, caso o limite de 40% de uso médio de memória seja atingido.

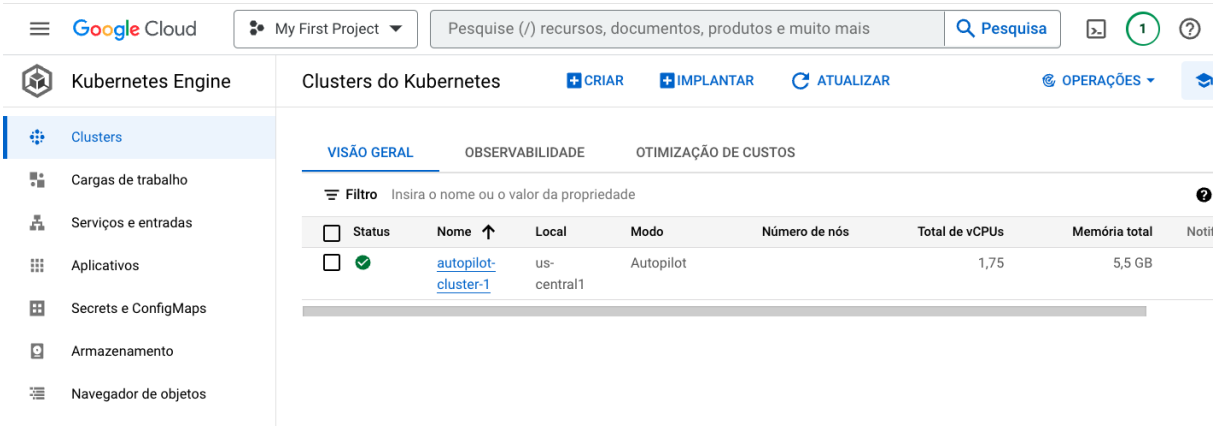
Em questão de funcionalidade, a arquitetura operou de maneira totalmente funcional quando implementada no ambiente local i.e. o computador pessoal do discente. Tantos testes manuais (publicando *payloads* com ferramentas externas) quanto testes de carga (abordados na Seção 4.3) geraram operações nos diversos componentes e resultaram em notificações via e-mail no endereço designado.

4.1.3 Implementação na nuvem

Para o *deployment* na nuvem, foi utilizado o *Google Kubernetes Engine* (GKE), um serviço para implantação facilitada de *clusters Kubernetes*. Foi necessário apenas criar um *cluster* via *console* e mudar uma simples linha no código IaC. Assim, ao realizar o *deployment*, o *Terraform* o fez nos servidores da GCP em vez de utilizar o ambiente local, exaltando a vantagem de utilizar IaC. Ao final do processo, uma cópia funcional da arquitetura foi instalada na nuvem, podendo ser acessada diretamente via IP estático em qualquer lugar.

A Fig. 9 exibe uma captura de tela da *dashboard* da GCP, indicando que o *cluster* foi implantado, com 1.75 vCPUs[1] e 5.5 GBs de memória. Estruturalmente, este *deployment* é igual à versão de testes em ambiente local e seria igual em outro serviço de *Kubernetes* de qualquer outro provedor. Portanto, verifica-se uma excelente portabilidade, não dependendo de especificidades de qualquer provedor.

Figura 9 – *Dashboard* da GCP, exibindo os recursos do *cluster* implantado na GCP.



Fonte: elaborado pelo autor.

Como um ponto positivo desta interface gráfica da GCP, diversas métricas de uso de recursos são expostas por meio de gráficos ao usuário, sem necessidade de configuração adicional. Deste modo, é possível observar e metrificar estes dados, que são importantes indicativos da saúde e do modo de operação das aplicações.

Finalmente, por questões econômicas, a infraestrutura na GCP foi deletada assim que os testes foram finalizados: por mais que a plataforma ofereça US$ 300 para novas contas, uma plataforma como esta pode facilmente consumir todos os créditos se o usuário se esquecer de desativá-la.

[1] vCPUs são equivalentes a CPUs, com a diferença de serem baseadas em *software* [Anderson 2022].

4.2 Desenvolvimento dos micro-serviços

Como mencionado anteriormente, os micro-serviços foram desenvolvidos para realizar o processamento de dados IoT. De modo mais detalhado, eles possuem as seguintes responsabilidades:

1. Receber *payloads* do servidor MQTT no formato JSON;
2. Verificar se o seu formato dos dados condiz com o esperado, examinando se todos os campos obrigatórios estão presentes e se seus valores condizem com o estipulado na Tabela 1 – *payloads* não conformes são ignorados;
3. Armazenar os dados processados em um banco de dados persistente;
4. Calcular a média móvel no último minuto para as seguintes métricas da máquina: pressão (atm), temperatura (°C) e taxa de unidades defeituosas produzidas (%);
5. Comparar as médias das métricas citadas com limites pré-estabelecidos;
6. Caso os limites tenham sido ultrapassados, enviar um relatório com os dados relevantes para um endereço de e-mail dado.

Vista a extensão desta lista de responsabilidades, tais funcionalidades foram divididas entre dois micro-serviços: de processamento e de notificações. Nesta seção, serão aprofundados os resultados mais relevantes obtidos durante esta etapa de desenvolvimento.

4.2.1 Modelagem de dados

Dada a Tabela 1, supomos que os dados foram transmitidos por meio de *payloads JSON*, um formato comumente utilizado para transmissão de dados estruturados. Assim, o Código Fonte 1 exibe um exemplo de *payload* enviado pelos *gateways* e recebido pelo micro-serviço de processamento.

Código Fonte 1 – Exemplo de *payload* enviado pelos gateways.

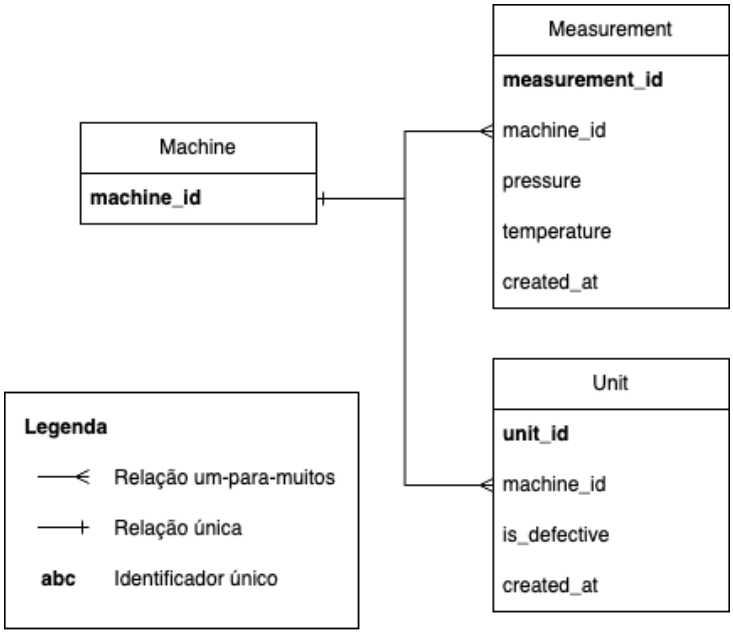
```
{
    "unit_id": 123,
    "created_at": "2022-05-18T11:39:22.519222",
    "is_defective": true,
    "machine_id": 123,
    "machine_temperature": 110.0,
    "machine_pressure": 1.2
}
```

Fonte: Elaborado pelo autor.

Como é possível observar, este *payload* contém os dados captados por um sensor em uma dada máquina após a fabricação de uma dada unidade. Deste modo, dados tanto da unidade quanto da máquina são enviados para a nossa arquitetura, que deve armazenar estes dados em um banco de dados.

Portanto, foi necessário modelar estes dados para o tipo de banco de dados escolhido, que foi o relacional (SQL). Neste processo, algumas boas práticas foram respeitadas para reduzir a redundância de informações, aumentar o desempenho e tornar os dados mais interpretáveis. Deste modo, foram criados três modelos distintos, cada um representando um tipo de objeto e contendo suas respectivas informações.

Figura 10 – Diagrama de entidades da modelagem de dados utilizada.



Fonte: elaborado pelo autor.

A Fig. 10 ilustra o diagrama de entidades desta modelagem. Podemos observar que os dados contidos no *payload* foram divididos entre os três modelos. Estes também se relacionam entre si por meio do campo *machine_ id*, de modo que uma máquina pode haver várias medidas (*measurements*) e várias unidades (*units*). Entretanto, o inverso não é válido – por isso a relação única está indicada. Vale ressaltar que, em um cenário mais realista, a entidade *Machine* possuiria mais campos, como sua localização, seu setor, entre outros.

Esta estrutura é criada no banco de dados *PostgreSQL* por meio de uma *migração* i.e. uma série de comandos que criam as tabelas correspondentes a esta modelagem. O Código Fonte 2 corresponde à migração realizada no banco de dados.

Código Fonte 2 – Migração executada no banco de dados PostgreSQL.

```
CREATE TABLE IF NOT EXISTS machines(
    machine_id INT NOT NULL,
    PRIMARY KEY(machine_id)
);

CREATE TABLE IF NOT EXISTS units(
    unit_id INT NOT NULL,
    machine_id INT NOT NULL,
    is_defective BOOLEAN,
    created_at  TIMESTAMP DEFAULT current_timestamp,
    PRIMARY KEY(unit_id),
    CONSTRAINT fk_machine
        FOREIGN KEY(machine_id)
            REFERENCES machines(machine_id)
            ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS measurements(
    measurement_id INT GENERATED ALWAYS AS IDENTITY,
    machine_id INT NOT NULL,
    pressure FLOAT,
    temperature FLOAT,
    created_at  TIMESTAMP DEFAULT current_timestamp,
    CONSTRAINT fk_machine
        FOREIGN KEY(machine_id)
            REFERENCES machines(machine_id)
            ON DELETE CASCADE
);
```

Fonte: Elaborado pelo autor.

4.2.2 Micro-serviço de processamento

O micro-serviço de processamento consiste em um *software* escrito em *Python*, que tem como objetivo processar os *payloads* enviados pelos *gateways*. Ele o faz integrando diversos outros componentes, agindo como um orquestrador (como mostra a Fig. 6).

4.2.2.1 Processamento

De maneira mais detalhada e abstraindo o que cada sub-componente do micro-serviço faz, o processamento ocorre em algumas etapas:

1. Recebimento e validação do *payload*;
2. Armazenamento do *payload* no banco de dados;

3. Consulta ao banco de dados das médias móveis das métricas observadas para a máquina em questão;
4. Verificação se os limites das métricas foram ultrapassados;
5. Enviar requisição HTTP ao micro-serviço de notificação, caso necessário baseado no item anterior.

O Código Fonte 3 ilustra o processo acima ao mostrar a função principal deste micro-serviço, que é chamada assim que o formato do *payload* é validado.

Código Fonte 3 – Função principal do micro-serviço de processamento.

```
def process(payload_dict):
    """
    Processes the payload in the dictionary format.
    """
    # Save to DB
    database_connector.save_payload_to_db(**payload_dict)

    # Calculates the start of moving average window
    average_window_start = payload_dict["created_at"] - timedelta(
        seconds=config["NOTIFICATION_TIME_WINDOW"]
    )

    # Get stats for machine_id
    averages = get_averages_for_machine(
        payload_dict["machine_id"], average_window_start
    )
    if not averages:
        return

    # Check for notifications
    notification_payload = notifier_connector.build_notification_payload(
        payload_dict["machine_id"], payload_dict["created_at"], averages
    )

    if len(notification_payload.get("warnings", [])) > 0:
        notifier_connector.send(notification_payload)
```

Fonte: Elaborado pelo autor.

4.2.2.2 Comunicação com o micro-serviço de notificação

Em relação à requisição HTTP enviada ao micro-serviço de notificação, são transmitidos apenas os dados mais relevantes dos que estavam disponíveis ao longo do processamento: identificador da máquina (*machine_id* em questão); data e hora em que o

processamento ocorreu; e uma lista das métricas relevantes, com seus valores atuais, seus respectivos limites e suas respectivas unidades. O Código Fonte 4 exemplifica um *payload* enviado ao micro-serviço de notificação.

Código Fonte 4 – Exemplo de *payload* enviado ao micro-serviço de notificação.

```
{
    "machine_id": 123,
    "created_at": "2022-05-18T11:40:22.519222",
    "warnings": [
        {
            "type": "pressure",
            "current_value": 1.0,
            "target_value": 0.9,
            "unit": "atm",
        },
        {
            "type": "temperature",
            "current_value": 79.4,
            "target_value": 75.0,
            "unit": "°C",
        },
        {
            "type": "defective",
            "current_value": 51.25,
            "target_value": 5.0,
            "unit": "%",
        },
    ],
}
```

Fonte: Elaborado pelo autor.

4.2.2.3 Processo de desenvolvimento

Durante o desenvolvimento do *software*, cada função e subcomponente foi testado individualmente, no que é comumente chamado de teste unitário. Assim, um teste automatizado é criado para cada caso que a função cobre, e.g. um conjunto de entradas. Depois, testamos se a função se comporta de acordo com o esperado para cada um dos casos e se o resultado é igual ao previsto. É muito importante frisar que estes testes, por serem automatizados, podem ser executados facilmente. Deste modo, podemos sempre verificar o bom funcionamento da função após qualquer mudança no código e assim reduzir a ocorrência de erros. Este processo também é importante pois em sistemas suficientemente complexos, a certeza de que os subcomponentes funcionam isoladamente facilita o processo de *debugging*.

Ao fim do processo de desenvolvimento, todos os códigos-fonte foram empacotados em uma imagem *Docker*, juntamente com seus pré-requisitos. Em seguida, esta imagem foi carregada para um repositório público de imagens (*Dockerhub*), de onde poderá ser obtida pelo *cluster Kubernetes*. Este, por sua vez, poderá executar o micro-serviço de processamento a partir desta imagem.

4.2.3 Micro-serviço de notificação

O micro-serviço de notificação, por sua vez, tem como objetivo receber requisições de notificação do outro micro-serviço e enviá-las ao serviço de envio automatizado de e-mails *Sendgrid*.

Como dito anteriormente, o micro-serviço de processamento transmite seus dados via HTTP, portanto o serviço em questão expõe um *endpoint* (ponto de acesso) único, agindo como um pequeno servidor. Este *endpoint*, */notifications/*, recebe apenas requisições do tipo *POST*, contendo os dados relevantes para a notificação (Código Fonte 4).

Ao receber uma requisição, bastaria formatar estes dados em um corpo *HTML* e utilizar a biblioteca para *Python* do *Sendgrid* para enviá-la via e-mail. No entanto, em um momento de crise de uma máquina, seu sensor enviaria em torno de 10 requisições por minuto, cada uma gerando uma notificação. Neste caso, em apenas 10 minutos, seriam enviados 100 e-mails, o que é além de impraticável, geraria custos significativos no próprio *Sendgrid*. Assim, foi implementado um sistema com memória, utilizando uma instância *Redis* como banco de dados em memória (efêmero).

4.2.3.1 Sistema com memória

Tal sistema consiste em interagir com o *Redis* em duas formas: armazenar *payloads* completos; e gerar um *timer* de um minuto, que indica se já houve tempo suficiente entre um envio de e-mail e outro. Assim, ao receber uma requisição de notificação, há duas possibilidades: caso seja a primeira notificação em um minuto, esta é enviada diretamente, junto com qualquer outra que esteja armazenada no *Redis*; caso contrário, o *payload* desta notificação é armazenado *Redis* e será enviado assim que uma nova requisição seja feita após o *timer* acabar. O Código-Fonte 5 mostra a função principal do *endpoint*, com esta lógica implementada em código.

Código Fonte 5 – Função principal do *endpoint* */notifications/*.

```
@app.route("/notifications", methods=["POST"])
def send_notifications():
    notification_payload = request.data

    # Notification payload validation
    if not _is_notification_payload_valid(notification_payload):
```

```
        return "", status.HTTP_400_BAD_REQUEST

    # Check if cooldown timer is over
    if redis_client.should_send_email():
        print("Sending email...")

        # Get stored payloads
        payloads = redis_client.retrieve_notification_payloads()

        # Add current one to batch
        payloads[notification_payload["machine_id"]] = notification_payload

        # Send them via email
        res = sendgrid_client.send_to_sendgrid(payloads)
        redis_client.set_cooldown()
        return (
            "",
            status.HTTP_204_NO_CONTENT if res else status.HTTP_503_SERVICE_UNAVAILABLE,
        )
    else:
        # Store payload
        res = redis_client.store_notification_payload(notification_payload)
        return (
            "",
            status.HTTP_200_OK if res else status.HTTP_503_SERVICE_UNAVAILABLE,
        )
```

Fonte: Elaborado pelo autor.

Essencialmente, esta lógica permite reduzir consideravelmente o número de mensagens, fixando a taxa de envio em 1 por minuto, independente do número de notificações. Entretanto, ela introduz uma pequena falha: mesmo que improvável, é possível que uma notificação seja armazenada e só seja enviada muito tempo depois, se nenhuma outra requisição for feita após o fim do *timer*. Uma maneira de solucionar este problema é acoplar um sistema de agendamento de tarefas, com o qual há certeza de que os demais e-mails sejam enviados ao fim do *timer*. No entanto, isto não é trivial e foge do escopo do trabalho.

Ao final do processo, o endereço de e-mail designado receberá uma mensagem como a exibida na Fig. 11, podendo conter os dados de múltiplas máquinas.

4.2.3.2 Processo de desenvolvimento

De maneira muito similar ao processo de desenvolvimento no micro-serviço de processamento (4.2.2.3), este micro-serviço foi desenvolvido orientado a testes: sempre realizando-os para validar suas funções.

Figura 11 – Exemplo de mensagem via e-mail com múltiplas notificações.

Fonte: elaborado pelo autor.

Igualmente, uma imagem *Docker* foi gerada e publicada, para que o *cluster Kubernetes* pudesse utilizá-la e executar o *software* aqui descrito.

4.3 Desenvolvimento e condução de testes de carga

Uma vez a arquitetura implantada, os micro-serviços desenvolvidos e o funcionamento completo do projeto verificado, iniciou-se o processo de testes de carga. Este tipo de teste consiste em gerar um grande fluxo de requisições contra o sistema e observar como ele reage: se continua operacional, quantos recursos adicionais ele consome, entre outros aspectos. Deste modo, após o desenvolvimento e validação do projeto, diversos testes de carga foram conduzidos na arquitetura do projeto, tanto em ambiente local quanto em nuvem.

4.3.1 Ambiente de testes

O ambiente de testes foi desenvolvido utilizando a ferramenta *Locust*, com a qual é possível apenas especificar o comportamento dos usuários gerados, por meio de tarefas que eles executarão.

O Código Fonte 6 exibe a única tarefa presente no *script* de teste, que consiste no envio de *payloads* que os *gateways* enviariam para a camada de aplicação. Os valores,

em geral, são aleatórios. Porém, com a manipulação das funções *get_random_defective()*, *get_random_temperature()* e *get_random_pressure()*, é possível definir qual porcentagem dos payloads gerará uma notificação. Deste modo, temos o controle fino da carga em cada etapa do processamento.

Código Fonte 6 – Tarefa principal executada pelos usuários emulados.

```
@task
def send_payload(self):
    """
    Send a random payload to MQTT server.
    """
    payload = {
        "unit_id": random.randint(1, config["NUMBER_OF_UNITS"]),
        "created_at": datetime.now().isoformat(),
        # Random between True and False
        "is_defective": get_random_defective(),
        "machine_id": random.randint(1, config["NUMBER_OF_MACHINES"]),
        "machine_temperature": get_random_temperature(),
        "machine_pressure": get_random_pressure(),
    }
    # Convert payload to bytes
    payload_bytes = json.dumps(payload).replace('"', "'").encode("utf-8")

    # Publish payload into MQTT_TOPIC
    self.client.publish(config["MQTT_TOPIC"], payload_bytes)

def get_random_defective():
    """
    Generate random boolean (only 5% of payloads should trigger warning)
    """
    return random.random() < (config["DEFECTIVE_LIMIT"] / 100.0)

def get_random_temperature():
    """
    Generate random temperature (only 10% of payloads should trigger warning)
    """
    return random.uniform(
        config["TEMPERATURE_LIMIT"] - 9.0, config["TEMPERATURE_LIMIT"] + 1.0
    )

def get_random_pressure():
    """
    Generate random presssure (only 10% of payloads should trigger warning)
    """
    return random.uniform(
        config["PRESSURE_LIMIT"] - 0.9, config["PRESSURE_LIMIT"] + 0.1
    )
```

Fonte: Elaborado pelo autor.

A partir deste comportamento definido em *Python*, o *Locust* se encarrega de gerar os usuários, de acordo com parâmetros fornecidos: número máximo de usuários em paralelo; e taxa de acréscimo de usuários por segundo. Além disso, a ferramenta também gera gráficos e relatórios das principais métricas: número de requisições, número de usuários, número de falhas, latência, entre outros.

4.3.2 Testes de carga em ambiente local

Inicialmente, os testes de carga foram conduzidos no ambiente local, em um pequeno *cluster Kubernetes* (*Minikube*), com 2 CPUs e 4 GB de memória.

4.3.2.1 Resultados em carga nominal

De acordo com a Subseção 3.1.2, a carga nominal do sistema é: 100 sensores, cada um enviando 10 requisições por minuto. Em termos de requisição por segundo (RPS), isso equivale a 16,67 RPS.

Portanto, o teste em carga nominal foi conduzido com: 100 usuários em paralelo, adicionando 1 usuário por segundo. Os resultados são exibidos na Tabela 2 e na Fig. 12.

Tabela 2 – Requisições executadas no teste em carga nominal em ambiente local.

Nome	Descrição	Número de requisições	Número de falhas	Latência média (ms)	RPS
publish:0:payloads	Publicação de *payloads*	5634	0	0	14.3
connect	Conexão inicial ao MQTT	100	0	0	0.3

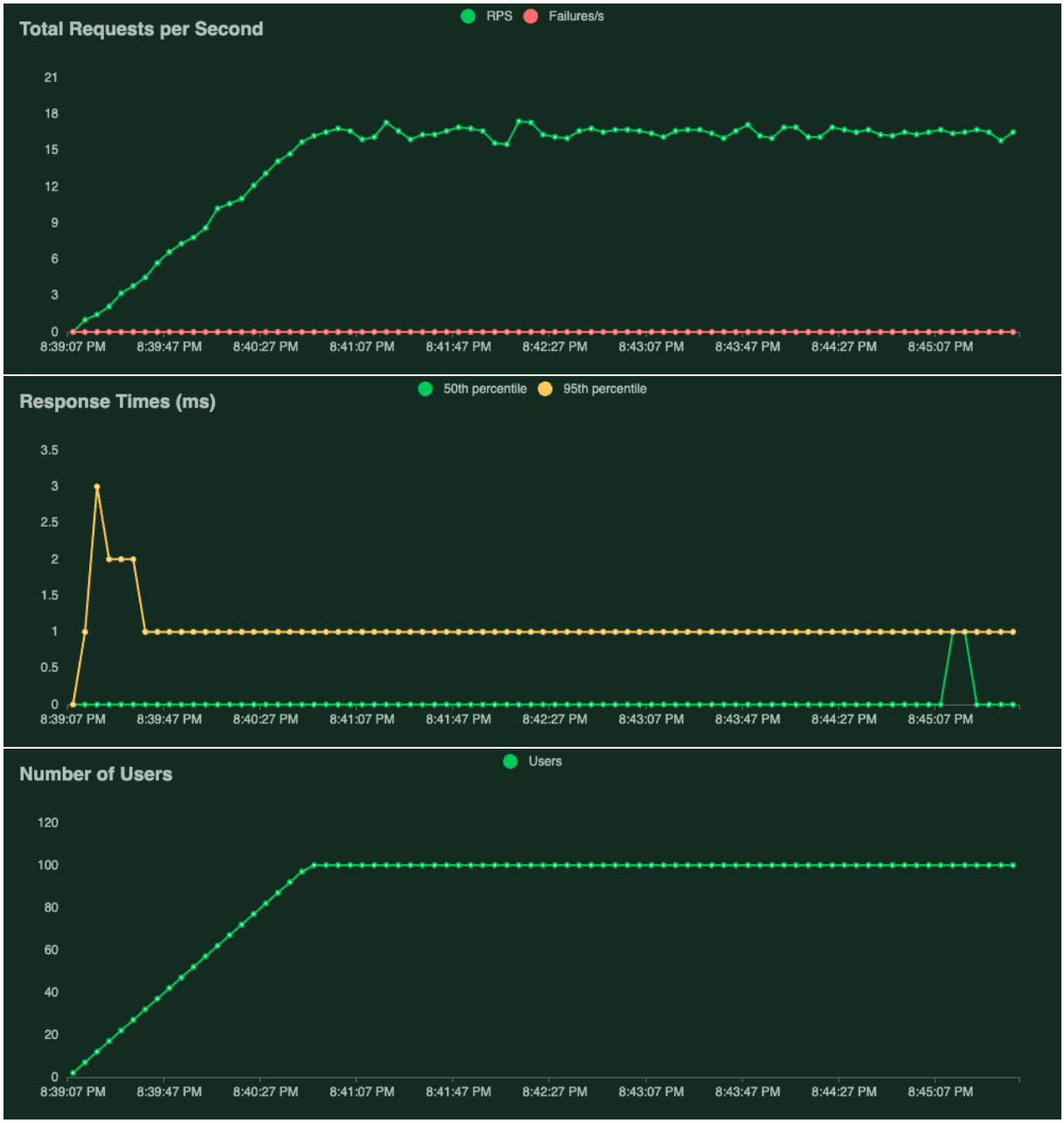
Fonte: elaborado pelo autor.

De maneira geral, o teste mostrou que não houve falhas em nenhuma chamada MQTT e de fato 100 usuários se conectaram ao *broker*, realizando em volta de 10 chamadas por minuto. Além disso, os valores apresentados na Tabela 2 são coerentes com o RPS calculado de 16,67.

Em questão de consumo de recursos computacionais, foram observadas as métricas de CPU e memória do micro-serviço de processamento, mas diferentemente do esperado, nenhuma variou o suficiente para engatilhar o *autoscaling* neste micro-serviço – definido em 40% de uso de memória. Assume-se então que por conta da baixa variação de consumo de recursos, não houve crescimento significativo na carga relacionada ao processamento. Um dos motivos é a baixa complexidade do processamento neste caso de uso específico, o que pode ser maior com *payloads* maiores ou casos de uso mais complexos. Outro motivo pode ser uma sobre-alocação de recursos neste componente e por consequência, uma sub-alocação em outros – isto será mais discutido na subseção seguinte.

Além destes resultados quantitativos, o serviço de e-mail também foi acionado inúmeras vezes (uma mensagem por minuto) assim que os testes se iniciaram. Assim foi

Figura 12 – Resultados do teste em carga nominal realizado em ambiente local.



Fonte: elaborado pelo autor.

comprovado que o processo inteiro estava em operação normal e correspondeu a todos os requerimentos de lógica de negócios e técnicos especificados na Seção 3.1. Portanto, o seu funcionamento foi totalmente validado em um *cluster* local.

4.3.3 Testes de carga na nuvem

Após os testes em ambiente local, iniciaram-se os testes de carga na nuvem, na GCP. Como exposto anteriormente, o *cluster Kubernetes* possuía como recurso 1.75 vCPUs e 5.5 GBs de memória. Em comparação ao ambiente local de testes, há uma pequena perda em termos de *CPU* mas há 1.5 GBs a mais de memória.

4.3.3.1 Resultados em carga nominal

Os testes de carga conduzidos em carga nominal no *cluster* da GCP foram muito similares aos descritos na seção anterior, tanto em termos qualitativos quanto quantitativos. Houve até mesmo a falta de necessidade de *autoscaling* no micro-serviço de processamento, devido ao baixo consumo de memória pelos mesmos motivos já descritos. Assim, não há muito interesse em analisá-los novamente.

4.3.3.2 Resultados em carga limítrofe

Por outro lado, também foi conduzido um teste em nuvem aumentado significativamente o número de requisições por segundo. Este aumento foi realizado da seguinte forma: aumentou-se o número de *payloads* por minuto por usuário de 10 para 100; o número máximo de usuários de 100 para 1000; e o número de usuários acrescidos por segundo de 1 para 10. Desta forma, teoricamente, haveria o RPS seria multiplicado por 100, indo de 16,67 para 1666,7.

A Fig. 13 e a Tabela 3 mostram os resultados quantitativos obtidos pelo relatório gerado no *Locust*.

Tabela 3 – Requisições executadas no teste em carga limítrofe em ambiente de nuvem.

Nome	Descrição	Número de requisições	Número de falhas	Latência média (ms)	RPS
publish:0:payloads	Publicação de *payloads*	123577	30	3	507.8
connect	Conexão inicial ao MQTT	337	0	0	1.4

Fonte: elaborado pelo autor.

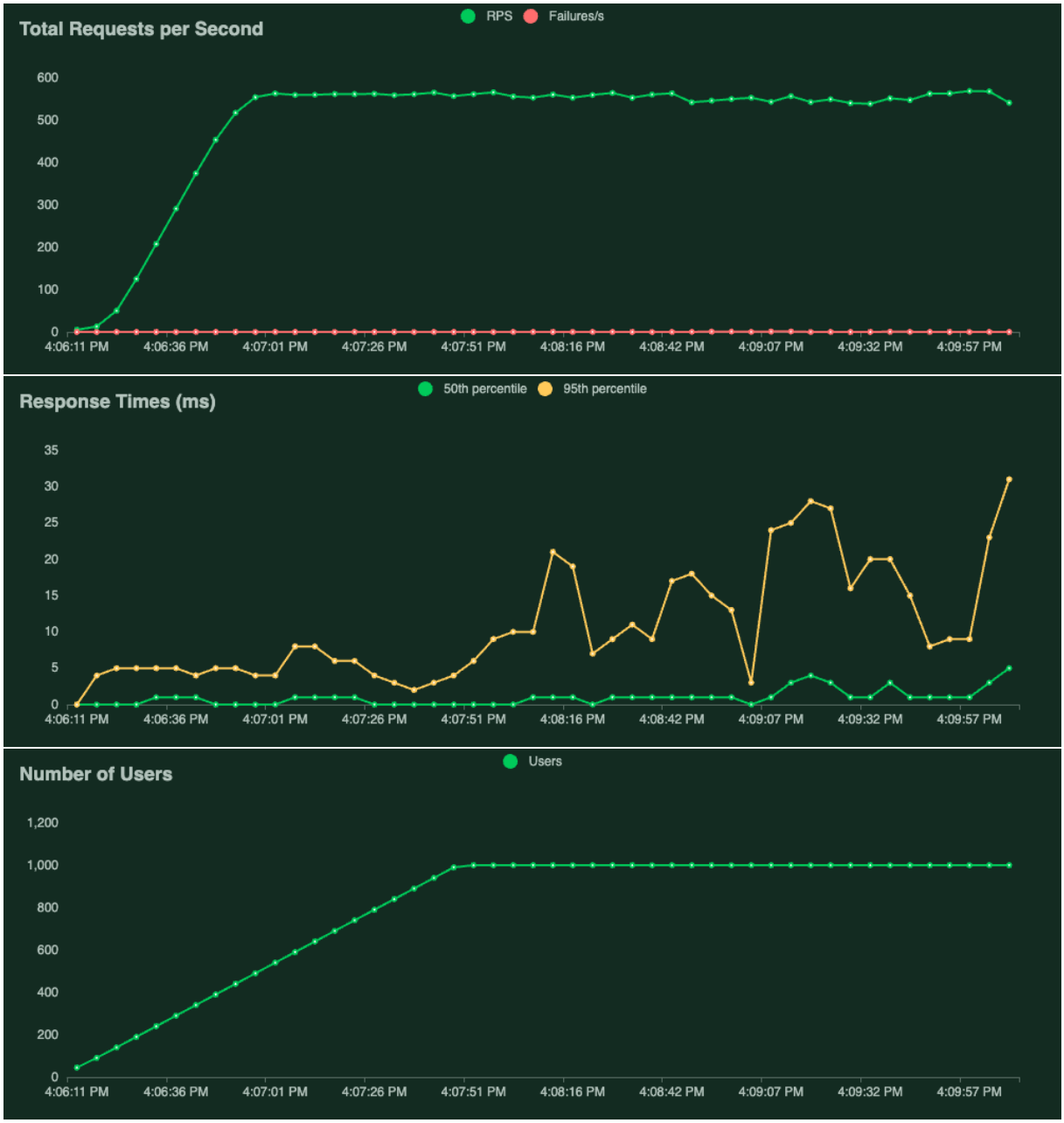
Diferentemente dos resultados anteriores, estes exibiram algumas diferenças drásticas: em vez de haver 1000 conexões, como esperado, houve apenas 307, o que nos indica que o resto dos usuários não conseguiu se conectar ao servidor MQTT; houve algumas requisições falhas, que são desprezíveis na quantidade de requisições realizadas, mas indicam que o *broker* estava atuando em seu limite; e o número de RPS diferiu bastante do esperado, atingindo apenas aproximadamente 500 RPS em vez dos 16667 RPS teóricos – o que faz sentido, visto que apenas um terço dos usuários se conectaram e evidencia que os usuários que conseguiram se conectar puderam enviar todos os seus *payloads*.

Em questão do *autoscaling* do micro-serviço de processamento, novamente ele não foi ativado, visto que o *cluster* alocou uma grande quantidade de recursos nele e assim, seu uso de memória não ultrapassou 3% durante todo o período.

A respeito do serviço de notificação, este continuou funcional, enviando novamente um e-mail por minuto contendo dezenas de notificações acumuladas.

Com tudo isso em mente, é possível afirmar que o fator limitante (ou *bottleneck*) da arquitetura neste caso foi o servidor MQTT, que não conseguiu abrir mais conexões simultâneas para suportar os demais usuários. Apesar disso, ele conseguiu manter uma boa

Figura 13 – Resultados do teste em carga limítrofe realizado em ambiente de nuvem.



Fonte: elaborado pelo autor.

qualidade de serviço para o usuários já conectados, enviando mais de 99,99% dos *payloads* corretamente. Além disso, 307 usuários equivale a mais do triplo da carga nominal.

Portanto, podemos afirmar que o limite da arquitetura em nuvem gira em torno de 300 usuários, com cada usuário realizando 100 requisições por minuto – aproximadamente 3 vezes e 10 vezes os valores nominais, respectivamente.

Caso fosse do interesse aumentar esta capacidade, indicaria-se o aumento de recursos alocados no servidor *Mosquitto Eclipse*, visto que este se mostrou o *bottleneck* da arquitetura. Para isso, é possível reduzir os recursos alocados no micro-serviço de processamento, visto que este foi sub-utilizado e sobre-alocado. Deste modo, basta ajustar

os limites configurados no código em *Terraform*, respeitando os limites do *cluster*. Não há a elasticidade do *autoscaling* e os recursos estarão alocados, mesmo em períodos de baixa conectividade (se eles existirem). Porém, o servidor estará sempre disponível, no caso de um pico repentino de conexões. Em contraste, com *autoscaling*, há uma latência até o *cluster Kubernetes* identificar o consumo elevado e de fato alocar mais recursos ao componente.

No entanto, caso seja desejado o *autoscaling* do servidor MQTT, isto pode não ser trivial com *Eclipse Mosquitto*, pois ele não suporta *scaling* horizontal i.e. geração de mais réplicas de modo distribuído [Light 2023] – uma das suas falhas, apesar da sua adoção para aplicações simples e fácil instalação. Assim, recomenda-se o uso de servidores mais adequados para este caso, como *HiveMQ*.

5 Considerações finais

5.1 Objetivos

Como dito no Capítulo 1, o presente trabalho possui tanto um objetivo geral quanto alguns objetivos específicos. Assim, nesta seção, abordamos cada um e verificamos se foram cumpridos.

5.1.1 Objetivo geral

Em suma, o objetivo geral deste trabalho é "desenvolver uma arquitetura escalável de um sistema de processamento de dados", respeitando os requisitos do sistema de acordo com o caso de uso definido e prover uma maneira do usuário interagir com os dados.

Dado o exposto no Capítulo 4, uma arquitetura foi concebida e desenvolvida em sua totalidade, utilizando apenas componentes básicos e micro-serviços desenvolvidos pelo autor. Além disso, todos os requisitos do caso de uso definido foram respeitados. Finalmente, o projeto interage com o usuário por meio de notificações, enviando-lhe os dados processados mais relevantes.

No entanto, quanto à escalabilidade do sistema, houve uma falha de *design*: o *bottleneck* identificado – o servidor MQTT – não foi o mesmo previsto na fase de concepção. Assim, o *autoscaling* foi aplicado em um componente sub-utilizado, enquanto a escolha do tipo de *broker* MQTT também não possibilitou o *scaling*. Deste modo, pode-se afirmar que, por mais que todos os requerimentos funcionais e não-funcionais do projeto tenham sido atingidos e que o *autoscaling* de um componente tenha sido implementado, não houve *escalabilidade* efetiva na arquitetura. Portanto, o objetivo geral do trabalho foi parcialmente atingido.

5.1.2 Objetivos específicos

A seguir, serão revisados os objetivos específicos e verificaremos se cada um foi atingido.

O primeiro objetivo específico é o estudo de escalabilidade em arquiteturas em nuvem e *design* de sistemas, o que foi feito durante o período inicial deste trabalho. O resultado pode ser verificado no Capítulo 2, que apresenta de modo breve os principais conceitos estudados. Portanto, esta meta foi completa.

O segundo objetivo está relacionado ao estudo de casos de uso e protocolos em IoT. Por mais que tenha sido o foco secundário deste trabalho, este estudo foi realizado durante

a revisão da literatura e a definição do caso de uso, para melhor contextualizar o projeto e torná-lo mais próximo da indústria. Deste modo, este objetivo específico também foi atingido.

O terceiro objetivo consiste no próprio desenvolvimento da arquitetura, para processar os dados IoT. A Seção 3.1 expõe todos os requerimentos, de lógica de negócios e técnicos do sistema e como demostrado durante os Capítulos 3 e 4, todos eles foram cumpridos e, portanto, este objetivo foi completo.

O último objetivo é a condução dos testes de carga para avaliar os limites do sistema desenvolvido. Como mostrado na Seção 4.3, diversos testes de carga foram feitos e o limite da aplicação foi descoberto. Portanto, este último objetivo específico também foi alcançado.

5.2 Conclusão

Com o crescimento da adoção da IoT e as constantes inovações na área, a escala das aplicações e o volume dos dados também cresceram em mesma magnitude. Assim, a arquitetura e a as aplicações que recebem e processam estes dados também devem evoluir, para se adaptar a este novo paradigma.

Desta forma, o desenvolvimento deste trabalho buscou discutir e aplicar os principais conceitos em escalabilidade, *design* de sistemas e engenharia de *software*, de modo geral. Como consequência, foi desenvolvida e implantada uma arquitetura completa para o processamento de dados em IoT, com alta portabilidade entre provedores – sendo assim uma alternativa aos serviços especializados em IoT de cada provedor. No entanto, devido a uma falha de *design*, não foi possível obter a escalabilidade almejada *de facto*. Mesmo assim, por mais que houve um foco em um caso de uso específico, o projeto pode ser adaptado e reutilizado para diversos outros.

Por fim, também foram conduzidos diversos testes de carga, tanto para discutir os principais conceitos desta etapa importante, quanto para validar diversos aspectos da arquitetura desenvolvida neste trabalho.

5.3 Trabalhos futuros

O domínio da engenharia de *software* voltada para *systems design* e IoT é vasto e há muitos trabalhos relevantes que podem ser desenvolvidos a partir deste.

Primeiramente, há diversas otimizações que poderiam ser feitas baseadas no *software* apresentado neste trabalho, como: a substituição de *Eclipse Mosquitto* por um *broker* escalável para atribuir mais *escalabilidade* ao sistema; a utilização de *caches* para diminuir

o número de consultas ao banco de dados, guardando dados frequentemente consultados em memória; a utilização de um agendador de tarefas para o envio periódico de notificações, em vez de depender de novos *payloads*; entre outras.

Outro exemplo é o estudo comparativo de alguns elementos escolhidos na arquitetura, como o protocolo de comunicação entre a camada de rede e a camada de aplicação. Assim, seria possível de fato validar a escolha, baseado em dados de performance.

Finalmente, outra abordagem possível é a econômica, visto que preço é grande parte do que define qual serviço ou provedor *cloud* usar para um dado caso de uso. Poderia então ser feito um comparativo entre diferentes provedores de serviços IoT e.g. AWS, GCP, *The Things Network*, entre outros.

Referências

ALTEXSOFT. *Web application architecture: How the web works*. AltexSoft, 2019. Disponível em: <https://www.altexsoft.com/blog/engineering/web-application-architecture-how-the-web-works/>. Citado na página 31.

ALTEXSOFT. *The good and the bad of python programming language*. AltexSoft, 2021. Disponível em: <https://www.altexsoft.com/blog/python-pros-and-cons/>. Citado na página 32.

AMAZON. *What is Amazon EC2? - amazon elastic compute cloud*. 2023. Disponível em: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>. Citado 2 vezes nas páginas 28 e 30.

ANDERSON, A. *Understanding computer processors: Cpus vs. vcpus and threads vs. Cores*. 2022. Disponível em: <https://www.makeuseof.com/cpu-vs-vcpu-threads-vs-cores/>. Citado na página 44.

BECK, K. *Test-driven development by example*. [S.l.]: Addison-Wesley, 2015. Citado na página 35.

CHACON, S.; STRAUB, B. *Pro Git, Second edition*. [S.l.]: Apress, 2014. Citado 2 vezes nas páginas 26 e 32.

DAVIS, A. *The Pros and cons of a monolithic application vs. microservices*. 2022. Disponível em: <https://www.openlegacy.com/blog/monolithic-application>. Citado na página 27.

DEWANI, R. *SQL techniques: Data Analysis Using SQL*. 2020. Disponível em: <https://www.analyticsvidhya.com/blog/2020/07/8-sql-techniques-data-analysis-analytics-data-science/>. Citado na página 29.

DOCKER. *What is a container?* 2023. Disponível em: <https://www.docker.com/resources/what-container/>. Citado na página 28.

ETHEREDGE, J. *Why infrastructure as code?* 2020. Disponível em: <https://www.simplethread.com/why-infrastructure-as-code/>. Citado na página 26.

FERNANDEZ, T. *When microservices are a bad idea*. 2022. Disponível em: <https://semaphoreci.com/blog/bad-microservices>. Citado na página 27.

HASAN, M. *State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally*. IoT Analytics, 2022. Disponível em: <https://iot-analytics.com/number-connected-iot-devices/>. Citado na página 19.

HAT, R. *What is agile methodology?* 2022. Disponível em: <https://www.redhat.com/en/devops/what-is-agile-methodology>. Citado na página 35.

HENKE, C. *A comprehensive guide to IOT protocols*. Emnify, 2022. Disponível em: <https://www.emnify.com/iot-glossary/guide-iot-protocols>. Citado na página 23.

IBM. *A Brief History of Cloud Computing*. 2017. Disponível em: <https://www.ibm.com/cloud/blog/cloud-computing-history>. Citado na página 28.

IBM. *What are message brokers?* 2023. Disponível em: <https://www.ibm.com/topics/message-brokers>. Citado na página 30.

IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, p. 1–84, 1990. Citado na página 24.

JANCZAK, J. *9 reasons why terraform is a pain, and 1 why you should still care*. 2018. Disponível em: <https://www.schibsted.pl/blog/9-reasons-why-terraform-is-a-pain-and-1-why-you-should-still-care/>. Citado na página 26.

KLEPPMANN, M. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. [S.l.]: O'Reilly, 2021. Citado na página 25.

KUBERNETES. *Overview*. Kubernetes, 2023. Disponível em: <https://kubernetes.io/docs/concepts/overview/>. Citado na página 28.

LIGHT, R. *Server support*. 2023. Disponível em: <https://github.com/mqtt/mqtt.org/wiki/server-support>. Citado na página 58.

MOZDEVNET. *HTTP request methods - http: MDN*. 2023. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>. Citado na página 29.

NASA. *NASA Systems Engineering Handbook: NASA/SP-2016 -6105 REV2*. [S.l.]: 12TH Media Services, 2017. Citado na página 25.

ORACLE. *What is a database?* 2023. Disponível em: <https://www.oracle.com/database/what-is-database/>. Citado na página 29.

RAY, P. P. An introduction to dew computing: Definition, concept and implications. *IEEE Access*, v. 6, p. 723–737, 2018. Citado na página 28.

ROSE, K.; ELDRIDGE, S.; CHAPIN, L. The internet of things: An overview. *The internet society (ISOC)*, Reston, VA, v. 80, p. 1–50, 2015. Citado na página 23.

ROUSE, M. *What is System Design? - Definition from Techopedia*. 2014. Disponível em: <https://www.techopedia.com/definition/29998/system-design>. Citado na página 25.

SINGH, S. *Types of software engineers: Roles amp; responsibilities*. 2023. Disponível em: <https://www.browserstack.com/guide/what-are-the-different-types-of-software-engineer-roles>. Citado na página 24.

SOUTO, M. *Front-end, back-end e full stack: O que são?* 2023. Disponível em: <https://www.alura.com.br/artigos/o-que-e-front-end-e-back-end>. Citado na página 31.

STACKOVERFLOW. *Stack overflow developer survey 2022*. Disponível em: <https://survey.stackoverflow.co/2022/>. Citado na página 32.

TERRAFORM. *What is infrastructure as code with terraform?: Terraform: HashiCorp developer*. 2023. Disponível em: <https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code>. Citado na página 26.

TERRAFORM. *What is terraform: Terraform: HashiCorp developer*. 2023. Disponível em: <https://developer.hashicorp.com/terraform/intro>. Citado na página 26.

VASHI, S. et al. Internet of things (iot): A vision, architectural elements, and security issues. In: *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*. [S.l.: s.n.], 2017. p. 492–496. Citado 2 vezes nas páginas 23 e 24.

YAMAKOSHI, N. V. Y. O. *IoT Processing Architecture*. 2023. Disponível em: <https://github.com/nyamak/iot-processing-architecture>. Citado na página 41.

YEAGER, N. J.; MACGRATH, R. E.; MCGRATH, R. E. *Web server technology: The Advanced Guide for World Wide Web Information Providers*. [S.l.]: Kaufmann, 2000. Citado na página 29.