UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE LORENA

IGOR HIDEKI CABIANCA YAMAMOTO

Arquitetura de dados híbrida no contexto de IoT e Big Data: um estudo para provisionamento de avicultura de precisão

Lorena

2021

IGOR HIDEKI CABIANCA YAMAMOTO

Arquitetura de dados híbrida no contexto de IoT e Big Data: um estudo para provisionamento de avicultura de precisão

Trabalho de graduação apresentado à Escola de Engenharia de Lorena da Universidade de São Paulo como requisito parcial para obtenção do título de engenheiro físico.

Orientador: Prof. Dr. Carlos Yujiro Shigue

Lorena

2021

Ficha catalográfica elaborada pelo Sistema Automatizado
da Escola de Engenharia de Lorena,
com os dados fornecidos pelo(a) autor(a)

Yamamoto, Igor Hideki Cabianca
Arquitetura de dados híbrida no contexto de iot e big data: um estudo para provisionamento de avicultura de precisão / Igor Hideki Cabianca Yamamoto; orientador Prof. Dr. Carlos Yujiro Shigue. - Lorena, 2021.
71 p.

Monografia apresentada como requisito parcial para a conclusão de Graduação do Curso de Engenharia Física - Escola de Engenharia de Lorena da Universidade de São Paulo. 2021

1. Internet das coisas. 2. Big data. 3. Avicultura. I. Título. II. Shigue, Prof. Dr. Carlos Yujiro, orient.

À minha irmã, Aika Yamamoto.
Te amo eternamente.

Agradecimentos

Meus agradecimentos especiais à Universidade de São Paulo e à Escola de Engenharia de Lorena, que, por tanto tempo, proporcionaram a estrutura de ensino e de pesquisa mais que necessária para a minha formação, aos grandes professores que tive contato, que sempre buscaram por consolidar todo o conhecimento possível, e aos meus amigos, que em muitos momentos me ensinam coisas que fogem da minha capacidade de compreensão e que eu serei eternamente grato.

"Não importa qual caminho trilhe, não se ilhe,
sonho que se sonha junto é o maior louvor.
Amem!"
Síntese

RESUMO

YAMAMOTO, IGOR H. C. **Arquitetura de dados híbrida no contexto de IoT e Big Data: um estudo para provisionamento de avicultura de precisão**. 2021. Número de folhas 71 f. Monografia (Trabalho de graduação de Engenharia Física) - Escola de Engenharia de Lorena, Universidade de São Paulo, Lorena, 2021.

O presente trabalho tem como objetivo integrar softwares *Open-Source* e o provedor de serviços de nuvem AWS (*Amazon Web Services*) para provisionar uma arquitetura híbrida de dados orientada à IoT (*Internet of Things*). Neste estudo de caso, a arquitetura é constituida de três camadas: *gateway* (coleta de dados de sensores e limpeza), regional (comunicação com *gateway* e esteira de dados) e nuvem (armazenamento de dados estruturados e não estruturados). A comunicação com os sensores é feita via protocolo MQTT, com servidores hospedados em cada nódo de *gateway*, e então publicados a tópicos de um *cluster* Kafka por intermédio de um módulo Python. Na camada regional, as mensagens são tratadas em uma esteira de dados NiFi e então enviadas aos serviços da AWS S3 (Batch, *data lake*), DynamoDB (Stream, não relacional) e RDS (Stream, dados estruturados). As aplicações *Open-Source* foram virtualizadas em containers Docker e o código de integração foi publicado no GitHub. A integração dos serviços foi testada e usada em uma prova de conceito através de um simulador de dados sintéticos orientado ao cenário de avicultura de precisão, constituido de duas estações de monitoramento e supervisionando métricas de temperatura, amônia, luminosidade e umidade relativa. Como resultado, foi obtido um relatório com as métricas em tempo real dos sensores em cada gateway.

Palavras-chave: Internet das coisas, *Big Data*, avicultura

ABSTRACT

YAMAMOTO, IGOR H. C. **Hybrid data architecture oriented to IoT and Big-Data: a study on smart poultry provision.**. 2021. 71 p. Monograph (Undergraduate Thesis in Engineering Physics). Engineering School of Lorena, University of São Paulo, Lorena, 2021.

The current work aims to integrate Open-Source software and AWS services (Amazon Web Services) to provision a hybrid data architecture focused on IoT (Internet of Things). In this study case, the architecture is composed of three layers: gateway (data collection and cleaning), regional (communication with gateways and data pipeline), and cloud (structured and non-structured data storage). The communication with the sensors is performed via MQTT, with *brokers* hosted in each gateway, and then published to topics of a Kafka cluster by a Python module. In the regional layer, messages are processed with NiFi and then sent to AWS S3 (Batch, data lake), DynamoDB (Stream, non-relational) and RDS (Stream, structured). Open-Source applications were virtualized using Docker containers and the integration code was published on GitHub. The integration was tested and used in a proof of concept based on smart poultry, modelled with two monitored stations and gathering metrics of temperature, ammonia, luminosity and relative humidity. As a result, it was possible to obtain a report with real-time metrics of the sensors in each gateway.

Keywords: Internet of Things, *Big Data*, poultry

Lista de abreviaturas e siglas

AWS	— *Amazon Web Services*
CSV	— *Comma Separated Values*
DW	— *Data Warehouse*
ETL	— *Extract, Transform, Load*
GCP	— *Google Cloud Platform*
IA	— *Inteligência Artificial*
IoT	— *Internet of Things*
MQTT	— *Message Queue Telemetry Transport Protocol*
RDBMS	— *Relational Database Management System*
RDS	— *Relational Database Service*
S3	— *Simple Storage Service*
SQL	— *Structured Query Language*
TTL	— *Time To Live*

Lista de ilustrações

Lista de tabelas

Sumário

1 INTRODUÇÃO

O cenário global de informações atualmente representa uma rede extremamente complexa, interconectada pela Internet e com várias fontes que geram uma volumetria de dados absurda. Com o advento da evolução tecnológica e da eletrônica, esta rede passou de ser definida apenas por máquinas, operadas por seres humanos, para ser complementada também por "coisas" que se encontram conectadas à Internet. Essas "coisas" são sensores, atuadores ou qualquer dispositivo que possa receber e transmitir dados. Com isso, a Internet deixa de ser uma rede global orientada apenas às pessoas, e passa então a ser uma "Internet das coisas" (*Internet of Things*, IoT).

Se tratando de aplicações, este paradigma traz novas possibilidades disruptivas de tecnologia e negócio. Um modelo de exemplo são as transformações "*smart*", isto é, a habilidade de se trazer a capacidade computacional e a conexão com a Internet para "dar vida" a utensílios domésticos, como lâmpadas (*smart LEDs*) e televisores (*smart TVs*). Porém, vai muito além de simplesmente deixar as coisas "inteligentes": com IoT, é possível realizar o monitoramento em escala de indústrias, de estações de produção agrícola, alimentar algoritmos de previsão, integrar com outras fontes para geração de valor, dentre muitas outras atividades. De fato, este paradigma já está presente em setores como hospitalar, automotivo, industrial, agrícola, manufatura, logística e doméstico, e promete, ao longo dos próximos anos, estar presente em todas as esferas do mercado de tecnologia.

Com isso, vem à tona novos desafios, sendo um deles a de escalabilidade. Como realizar uma implementação em escala de dispositivos de sensoriamento? Como fazer a manutenção e a calibração dos sensores? Como garantir a integração de ponta a ponta sem problemas que sejam críticos? Ainda que todas essas perguntas já tenham respostas, modelos ou metodologias, um problema recorrente é em questão à infraestrutura de dados, ou seja, quais recursos serão usados para provisionar toda a esteira de tratamento e armazenamento de dados.

Esta imensa teia de informações e interações entre usuários e dispositivos introduz uma complexidade extrema, e está associada, no universo de dados, com uma volumetria, velocidade e variabilidade alta, e que pode trazer diferentes valores dependendo de como é tratado. Esses quatro fatores usualmente definem um cenário que passou a ser comum entre os produtos digitais que é hoje denominado como "*Big Data*", isto é, casos que possuem como fator inerente a necessidade de lidar com uma diversidade de informações e mesmo assim buscar gerar valor através de produtos ou serviços.

Do lado da tecnologia, temos atualmente uma ampla gama de ferramentas e

produtos que provisionam soluções em diversos casos. Por exemplo, é possível hospedar toda a infraestrutura tecnológica em um provedor de serviços em nuvem, como a *Amazon Web Services* (AWS) ou a *Microsoft Azure*, as quais oferecem soluções proprietárias e nativas ao ambiente, mas que também possibilitam fazer o uso de *softwares* de código aberto (*open-source*). De fato, as possibilidades de arquitetura atualmente são diversas, e em geral elas oferecem escalabilidade, resiliência, baixo custo e segurança da informação.

Neste estudo, realizamos uma revisão sobre a literatura no assunto de arquitetura de dados para o contexto de IoT. Com base nisso, desenhamos uma proposta de arquitetura híbrida e uma prova de conceito baseado em ferramentas *open-source* integradas com serviços de armazenamento da provedora AWS, orientada ao assunto de avicultura de precisão. Os testes foram feitos através de um simulador de dados sintéticos de IoT, e os códigos da integração foram virtualizados em *containers* Docker e distribuidos em um repositório do GitHub. Como resultado, foi possível extrair as métricas dos sensores por meio de um *dashboard* em tempo real.

2 OBJETIVOS

Este estudo tem como objetivo revisar a literatura acerca de arquiteturas de Big Data dedicadas a aplicações IoT, com foco em avicultura de precisão. A partir desta revisão, é proposto um modelo de arquitetura de dados híbrida orientada para aplicações IoT. A validação, testes das integrações e análises de resiliência das ferramentas são feitas a partir de um simulador de dados sintéticos. Por fim, busca-se disponibilizar as métricas e os dados gerados pelos sensores em um dashboard em tempo real.

3 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão introduzidas as técnologias utilizadas neste trabalho, bem como o tópico de avicultura de precisão e sua relação com IoT.

3.1 *Big-Data* e computação em nuvem

Ao se pensar em dispositivos que estão integrados à internet, em especial frente aos grandes avanços da eletrônica, as possibilidades de emergência de tecnologias disruptivas são impulsionadas como nunca antes foi visto. A partir disso, ideias de intregações multiparadigmáticas, como Indústria 4.0, aprendizado de máquina, inteligência de negócio e novas experiências de usuário deixam de ser promessas futurísticas para então se tornar tendências da tecnologia.

Porém, um fenômeno que acontece ao se integrar muitos atuadores que possuem algum tipo geração, interação ou análise de informações é a presença de um volume de dados muito alto, acompanhado de um grande fluxo (valocidade), variabilidade e de diferentes tipos de valores. Essas quatro características são conhecidas como os *4V*'s que compõem o que é entendido hoje como *Big Data*, ou seja, problemas ou aplicações que necessitam de uma infraestrutura de computação flexível, robusta e simples para comportar todas as tarefas corporativas de forma resiliente, segura e escalável (MALEK et al., 2017). De fato, com uma estimativa de mais de 1 trilhão de dispositivos integrados à Internet até 2030, garantir valor de negócio, bem como assegurar estabilidade e promover capacidades analíticas sobre os dados deixa de ser uma tarefa simples e passa a exigir infraestruturas que ofereçam características como processamento distribuído, comunicação de eventos em tempo real, resiliência para suportar fluxos altos e variáveis de dados (*high-throughput*) e capacidade de rápida escrita e leitura em memória para bancos de dados (MARJANI et al., 2017).

Pensar em uma demanda tão alta por flexibilidade de recursos, combinado com cenários que exigem escalabilidade rápida e criação de produtos impulsionado por execuções ágeis, faz entender que uma infraestrutura computacional para prover uma solução de IoT não é conveniênte em modelos de computação tradicionais, como por exemplo em estações completamente físicas (conhecido também como *on-premise* (PAHL; XIONG; WALSHE, 2013)). Um paradigma atual que mitiga muitos desses problemas é o de *Cloud computing* (computação em nuvem). Nele, o acesso a infraestruturas, plataformas e *softwares* são feitos de acordo com a necessidade, onde a alocação ocorre via requisições ou sob demanda, e o pagamento é respectivo ao uso (filosofia *pay-as-you-go*) (AUMONT et al., 2018). Muitas

empresas atualmente oferecem tais serviços, e um dos principais deles são a *Amazon Web Services* (AWS), *Microsoft Azure*, e *Google Cloud* (GCP), *IBM Cloud*, *Oracle Cloud*, *Digital Ocean*, dentre muitos outros (JADEJA; MODI, 2012).

Ainda que seja possível realizar a migração de uma infraestrutura inteiramente *on-premise* para à nuvem, alguns fatores exigem atenção e cuidado no momento de decisão, como custos, disponibilidade e segurança (JADEJA; MODI, 2012). Desta forma, muitas organizações acabam optando por implementar uma arquitetura híbrida, ou seja, distribuir as suas diversas tarefas entre recursos físicos e em nuvem. Nesta abordagem, é possível otimizar e flexibilizar o que já se tem disponível, bem como sofisticar e diversificar a arquitetura com a possibilidade de, por exemplo, contratar serviços da nuvem apenas para armazenamento ou qualquer outra tarefa específica (ODUN-AYO et al., 2018).

Neste trabalho, é utilizado a abordagem híbrida para alocar tarefas de processamento em máquinas físicas e realizar o armazenamento na nuvem da AWS. A seção de Metodologia Sec.(4) apresenta o escopo da proposta, bem como o detalhamento de cada uma das tecnologias, *softwares* e serviços da nuvem utilizadas.

3.1.1 *Edge* e *Fog computing*

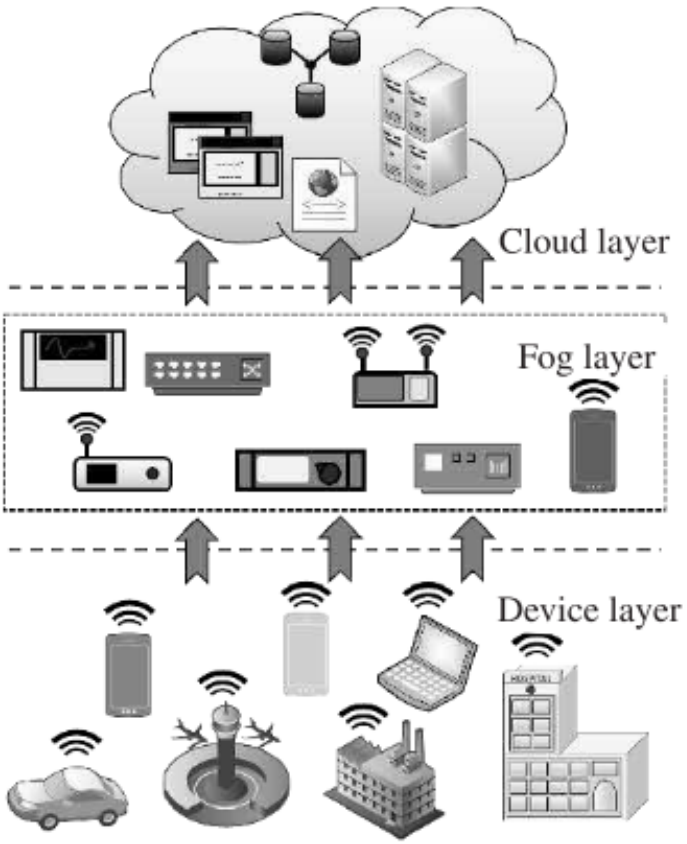
Ao se tratar de integrações de dispositivos à nuvem, é pertinente mencionar algumas limitações comuns. Como já foi mencionado, uma rede IoT tem como uma das principais características a conectividade intermitente, ou seja, é comum que sensores tenham oscilações de acesso à Internet, o que impacta o uso e o valor dos dados gerados (AUMONT et al., 2018). Além disso, muitos dispositivos não são apenas sensores, mas como também atuadores, isto é, realizam uma determinada ação programada baseado em um sinal de entrada. Em aplicações de veículos autônomos, medicina inteligente ou agricultura de precisão, a falta de conexão de um atuador com a Internet pode ser fatal e causar consequências graves.

Com isso, surge a necessidade de trazer o poder de processamento computacional, usualmente alocado em *clusters* na nuvem, mais próximo aos sensores (também conceituado como "*Edge*", ou borda). Qualquer camada de processamento que esteja conectada a ele e que preceda contato com a nuvem é denominado de *Fog-computing*, isto é, "computação em névoa", assim como demonstra a figura Fig.(1). Qualquer dispositivo que consiga suportar este poder de processamento é denominado "nódo". Alguns dos principais problemas que são atendidos mediante este modelo são (AUMONT et al., 2018):

- **Mobilidade e reconhecimento geográfico**: suporte a mobilização livre, mediante reconhecimento dos pontos geográficos que se encontram tanto os sensores quanto a sua própria localização;

- **Conectividade de baixa latência**: rápida comunicação com *data-centers*, estações físicas ou serviços de núvens, assim como com os seus sensores;
- **Heterogeneidade e interoperabilidade**: suporte a processamentos que servem à diversas finalidades, onde todos os nódos devem ser inter-comunicáveis.

Figura 1 – Arquitetura de *fog computing*



Fonte: AUMONT et al. (2018)

Além os pontos citados, a camada de *fog* é utilizada usualmente para a realização de algum nível de pré-tratamento de dados, como validações de esquema e segurança de fonte (GUARDO et al., 2018). Neles ainda é comum existir abstrações de protocolos e tecnologias de comunicação, os quais, no contexto de IoT, podem ser os mais variados, como WiFi, LTE, Bluetooth, Zigbee, MQTT, XMPP, dentre outros (KALLA; PROMBAGE; LIYANAGE, 2020).

Tratando-se dos nódos de *fog*, existem alguns tipos de dispositivos que possibilitam a existência da camada intermediária de processamento. Alguns exemplos são *routers*, *switches*, porém, o mais comum são os *gateways*, que desempenham a função de intermediário entre os sensores e a nuvem (AUMONT et al., 2018). Porém, no trabalho de CIRANI et al. (2015), o conceito de *gateway* é projetado além de um simples intermediário de comunicação, e conceitua o que denominaram de *IoT Hub*, isto é, um nódo de *fog*

que também herda funcionalidades procedurais e de processamento lógico sobre os sinais gerados pelos sensores, cujo protótipo foi implementado através de um módulo de *Raspberry Pi (RPi) Model B*.

No presente projeto, os nódos foram desenhados para suportar as mesmas funcionalidades lógicas que no trabalho de CIRANI et al. (2015). Desta forma, a implementação dos *softwares* foram via virtualização em imagens Docker, com suporte à arquiteturas *arm32* e *arm64*, isto é, compatíveis com módulos de Raspberry Pi. Uma contextualização de todos os recursos será apresentado a seguir, e os detalhes de integração serão discutidos na seção de Metodologia Sec.(4).

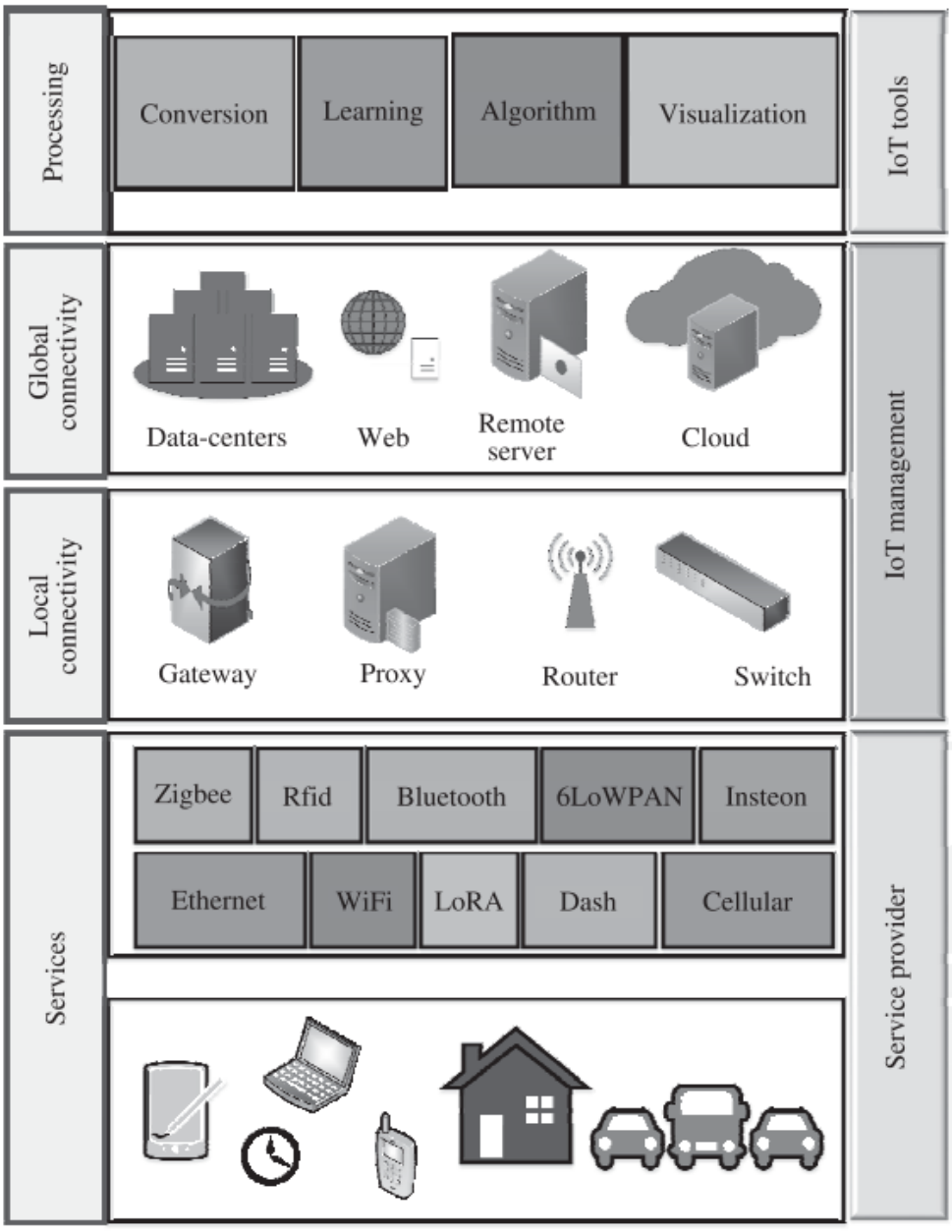
3.2 IoT: a integração dos objetos à internet

Internet of Things, ou IoT, é o termo atribuido à grande rede de dispositivos físicos que se encontram interconectados através da *Internet* (LIYANAGE et al., 2020). Tais dispositivos contém tecnologias de sistemas embarcados e são usados para comunicação, sensoriamento e interação com seus ambientes externos. AUMONT et al. (2018) afirma que sistemas IoT são usualmente caracterizados por possuirem: i) arquiteturas associadas eficientes e escaláveis; ii) quantidade massiva de dispositivos e nódos interconectados; iii) conectividade intermitente ou instáveis. A aplicação deste paradigma tecnológico se encontra presente em diversos contextos, como automação residencial, gestão e monitoramento de tráfego, sensoriamento climático, agricultura de precisão e *Big-Data* e *Analytics*, tornando-se assim um facilitador na criação de tecnologias de multi-domínio (AUMONT et al., 2018; Shanzhi Chen et al., 2014; TALAVERA et al., 2017; LEE; LEE, 2015).

A disseminação de aplicações IoT pressupõe a orquestração de diversas tecnologias interdependentes que estão, grosso modo, caracterizadas em quatro níveis assim como mostra a figura Fig.(2): i) serviços e dispositivos, que, através de protocolos de conectividade como WiFi, Bluetooth, RFID ou LoRa, geram ou recebem dados; ii) conectividade local e *gateways*, responsáveis por prover os dispositivos locais com conexão à Internet e/ou capacidade de processamento lógico (chamado também de *fog computing*); iii) conectividade global, a qual dispõe de recursos de rede, infraestrutura e *data-centers* para prover conectividade e segurança dos dados gerados pelos dispositivos; iv) processamento e aplicações, onde os dados coletados pelos dispositivos são usados para visualização, tratamento, análise, aprendizado de algoritmos de *Machine-Learning* ou em qualquer tipo de produto ou geração de valor (AUMONT et al., 2018).

Com a evolução de tecnologias de informação e comunicação, bem como do design de sistemas IoT e arquiteturas, este paradigma promete transformar e trazer novas possibilidades para os principais setores da economia, como produção industrial, transporte, medicina e indústria agrária (TOKAREVA; VISHNEVSKIY, 2018). Além disso, novos

Figura 2 – Níveis do paradigma IoT.



Fonte: AUMONT et al. (2018)

modelos de negócio e de valor são possíveis, dada a possibilidade de integrar grandes tendências tecnológicas, como Indústria 4.0, *Big-Data*, Inteligência artificial e *blockchain* (GUEVARA; SILVA, 2019; HAJIHEYDARI; TALAFIDARYANI; KHABIRI, 2019).

No Brasil, o cenário ainda é de instauração e adaptação, sendo, em sua grande parte, um parque para implementação de técnologias cujas patentes são majoritariamente de origem norte-americana ou chinesa (ALMEIDA et al., 2015). Ainda assim, promissor: com uma expectativa de geração de 50 a 200 bilhões de dólares até 2025 apenas em território brasileiro, está posicionada, segundo BNDES, como uma das maiores tendências tecnológicas do setor de tecnologia da informação (SILVA; JESUS, 2020). O desenvolvi-

mento no Brasil é atualmente orientado de acordo com o Plano Nacional de Internet das Coisas, decretada pelo Governo Federal em 25 de Junho de 2019, a qual planeja fazer o uso da tecnologia para promover melhorias sociais, incrementar produtividade em setores industriais e fomentar atividade econômica.

3.3 Softwares e ferramentas

Aqui serão abordadas algumas das ferramentas cruciais para o desenvolvimento deste projeto. Em suma, serão apresentadas as ferramentas de distribuição de eventos em tempo real (*stream*), chamados também de "mensageria", os *softwares* de processamento de dados e as técnicas de armazenamento de dados.

3.3.1 *Streaming* de dados e mensageria

O termo *streaming* de dados se refere, em suma, à habilidade de se extrair, tratar e armazenar dados em tempo real. AKIDAU; CHERNYAK; LAX (2018) definem que um sistema *stream* é orientado à aplicações onde os cojuntos de dados tendem, teoricamente, ao infinito, cujo tratamento, transmissão e armazenamento devem ser feitos de elemento a elemento. Em comparação ao tratamento de dados agregados (conjuntos finitos, com operações orientadas aos *batchs* de dados), o paradigma de *streaming* implica em novos desafios (BAHRI et al., 2021), como:

- **Volumetria e uso de memória**: a quantidade de eventos e registros que entram em um *stream* possui margem a variabilidade extremamente alta, de forma que toda a infraestrutura deve ser adaptável e escalável. Desta forma, assim como o conjunto de dados tende ao infinito, a disponibilidade de recursos de memória alocáveis deve ser compatível com a quantidade de registros sendo processado de forma distribuída;
- **Tempo de execução e processamento**: dependendo da aplicação, o tempo de execução deve ser o mínimo possível para atender às requisições de negócio;
- **Reestruturação de paradigma**: muitos modelos de inteligência artificial e de inteligência de mercado foram construidos, historicamente, visando o modelo tradicional de processamento (*batch*, ou seja, dados agregados). Para modelos em *stream*, o processamento deve ser feito em tempo real, e portanto pode implicar em mudanças de modelos e algoritmos disseminados em organizações e comunidades.

Uma aplicação de *streaming* tem, como um dos principais componentes, um *software* que realiza a distribuição de eventos em tempo real. Neste contexto, tais eventos são também denominados "mensagens" (*message*), onde o gerenciamento dessas mensagens

é feito via um *message broker*. Em geral, nos *brokers*, as mensagens podem ser enviadas (publicadas) e resgatadas (consumidas), e esta interação ocorre mediante um "tópico" (*topic*). Qualquer mensagem que é publicada a um tópico é enfileirada, de forma que qualquer consumo é cronológico sobre a ordem de publicação.

Uma das principais vantagens de se utilizar um *software* de mensageria é a possibilidade de consolidar em uma ferramenta única todo o gerenciamento de integração e comunicação entre sistemas. É possível suportar, para um mesmo tópico, diversas fontes que realizam publicações, o que pode ser consumido por uma ou mais aplicações. Dentre alguns exemplos de *softwares* de mensageria, temos o RabbitMQ, Apache Kafka, Mosquitto MQTT e Apache Pulsar.

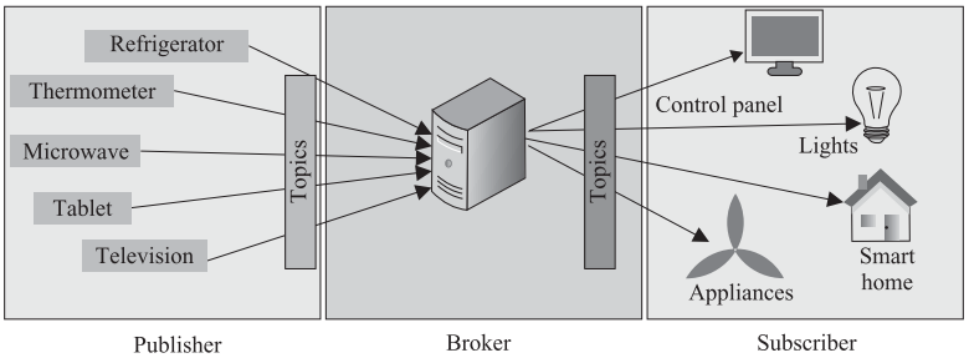
Neste projeto, implementamos duas camadas de comunicação por mensageria: uma entre os sensores e os nódos de *gateway*, no qual a comunicação é feita via Mosquitto MQTT, e outra entre os nódos de *gateway* e a camada de tratamento de dados, no qual a comunicação é feita via Apache Kafka. Elas serão abordadas a seguir, e o detalhamento da integração é feito na seção de Metodologia Sec.(4).

Em IoT, é mais desejável estabelecer a interconexão entre diversos dispositívos através de redes que se comuniquem de forma simples, rápida e segura. As normas implementadas para trocas de mensagens e mecanismos de autenticação via rede são chamadas de "protocolos", e dentro do contexto de tecnologia de informação existem diversos, como HTTP (*Hypertext Transfer Protocol*), FTP (*File Transfer Protocol*), SSH (*Secure Shell*) e TCP/IP (*Transmission Control Protocol/Internet Protocol*), onde cada um possui uma série de tipos de mensagens possíveis que podem ser trocadas, como POST, GET, PUT e DELETE no contexto de HTTP (GODFREY, 2009).

Um protocolo que atende às necessidades de segurança e performance no contexto de IoT é o "*Message Queue Telemetry Transport*"(MQTT), o qual é baseado no modelo de publicação-consumo. Neste protocolo, os tópicos seguem uma estrutura hierárquica, onde as mensagens (agnósticas de esquema de conteúdo), ao serem publicadas, são distribuidas dentre todos os "*clients*"(usuários) conectados (AUMONT et al., 2018). A figura Fig.(3) mostra exemplos de publicações e consumos que podem ser intermediados via um servidor MQTT. Neste projeto, a camada de comunicação via protocolo MQTT é realiada através do *software* Mosquitto MQTT (LIGHT, 2017).

Diferente do Mosquitto MQTT, o *software* Apache Kafka, um *message broker* desenvolvido pela empresa LinkedIn e lançado em forma de código aberto, é orientado à distribuição resiliente e de alta volumetria de dados, e possui suporte à paralelização de *brokers*, persistência de dados, *throughput* alto e suporte a diversos tipos de *clients* (GARG, 2013). Por ser distribuido, as mensagens podem ser armazenadas em um ou mais *brokers*, onde a coordenação de mensagens é feito através de uma instância de Apache

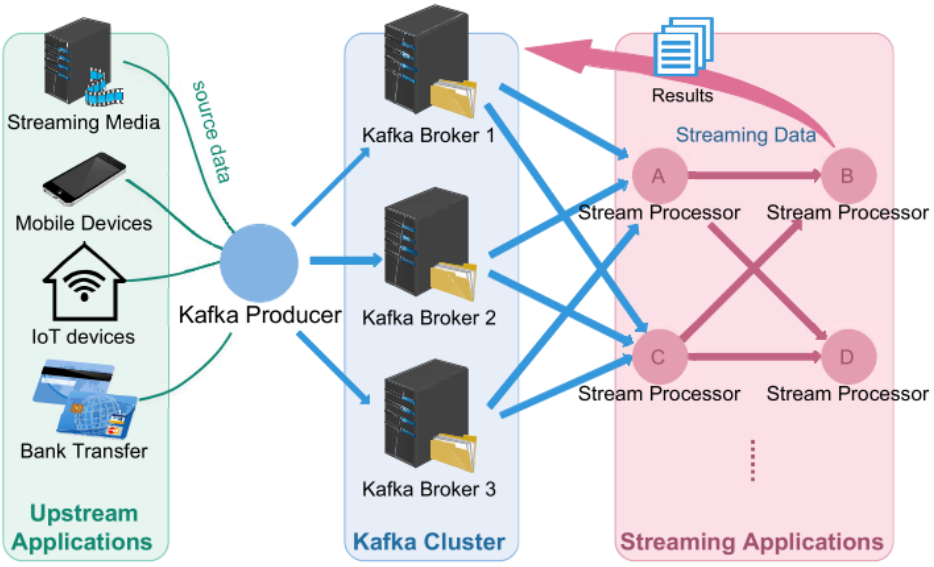
Figura 3 – Exemplo de publicação e consumo via MQTT



Fonte: AUMONT et al. (2018)

Zookeeper (BURNS, 2018). A figura Fig.(4) mostra exemplos de publicações e consumos que podem ser intermediados via um servidor Apache Kafka.

Figura 4 – Exemplo de publicação e consumo em *brokers* Kafka

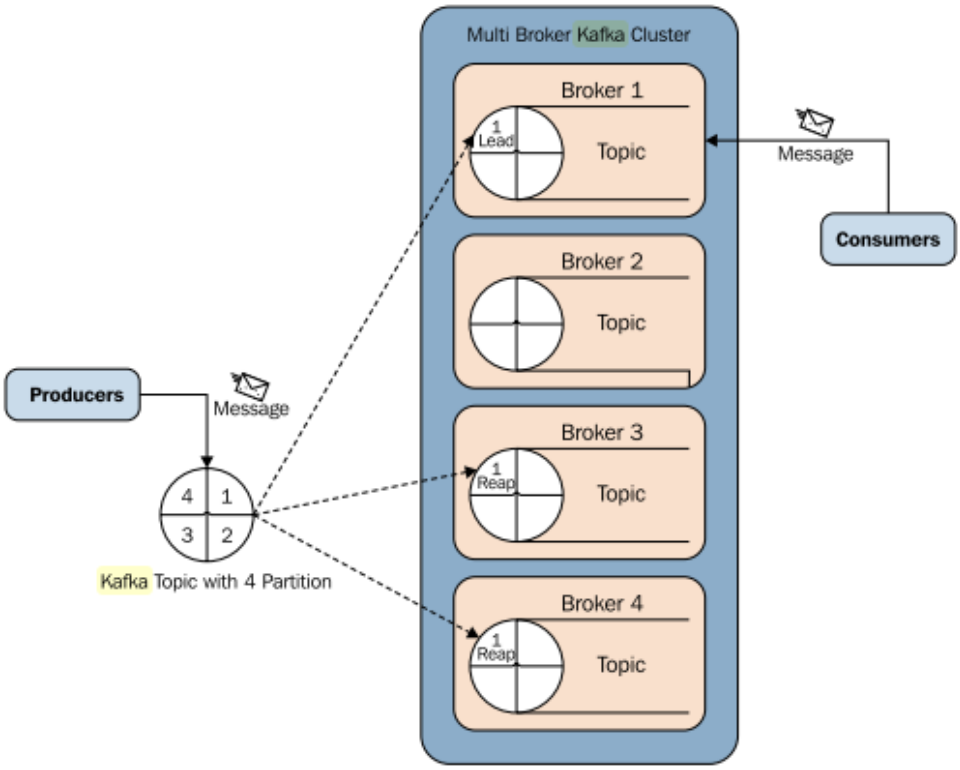


Fonte: WU; SHANG; WOLTER (2020)

A quantidade de *brokers* que as mensagens são replicadas é chamada de "fator de replicação", e é usada para provisionar resiliência e redundância de mensagens em aplicações que exigem extrema segurança na entrega de mensagens. Ainda, para cada *broker*, as mensagens são inseridas em partições específicas, de forma que, no momento de criação dos tópicos, cada partição do tópico é mapeado para a partição respectiva dos *brokers* no qual a mensagem será replicada (GARG, 2013). Por exemplo, em um caso de um *cluster* de 4 *brokers*, um tópico com fator de replicação e particionamento iguais à 2 e 10, respectivamente, pode ter a primeira partição do tópico atribuidas a partição 1 dos *brokers* 1 e 3, da segunda partição do tópico atribuidas a partição 2 dos *brokers* 2 e 4, e assim por diante. A figura Fig.(5) mostra demonstra o comportamento do fator de

replicação e do particionamento.

Figura 5 – Fator de replicação e particionamento



Fonte: GARG (2013)

3.3.2 Esteira de processamento de dados

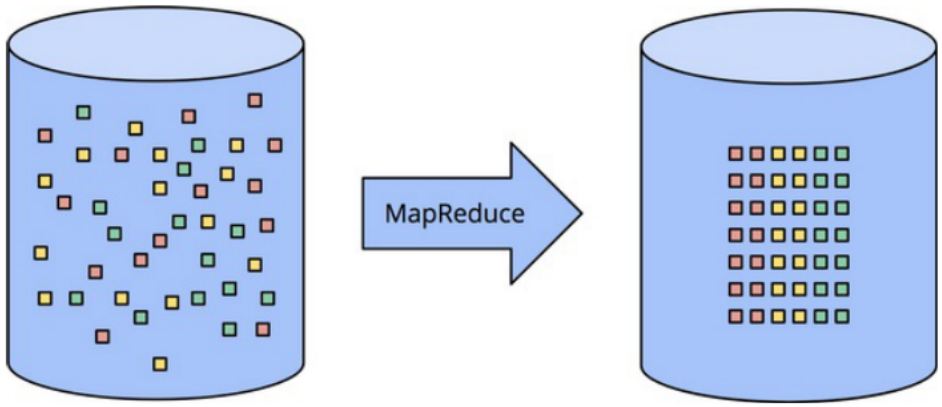
O modelo padrão de tratamento de dados é concebido conceitualmente com a ideia de processamentos em *batch*: conjuntos de dados com volumetria pré-definida (*bounded*), alta entropia e que, ao passar por um processo de transformação, são estruturados em conjuntos que expressam maior valor de negócio, o que é representado na figura Fig.(6a) (AKIDAU; CHERNYAK; LAX, 2018). Porém, é comum que os dados de entrada não sejam fixos, mas sim incrementais (*unbounded*), o que implica em processos e esteiras de tratamentos que operem sobre *batchs* incrementais (conhecidos também como *mini-batch*, representado na figura Fig.(6b)) ou em *streams* (figura Fig.(6c)).

As transformações feitas dependem diretamente do conteúdo e de como os dados devem ser tratados, e podem ser as mais variadas possíveis. Usualmente elas são transformações do tipo orientado à elemento (*element-wise*, correspondência de registros de $1:1$), de agrupamento (*grouping*, correspondência de de registros $m:n$) ou de composição (*composite*, correspondência de campos $m:n$) (AKIDAU; CHERNYAK; LAX, 2018).

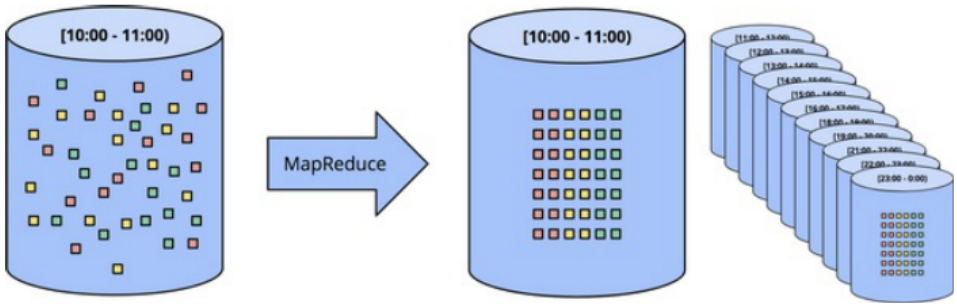
Neste projeto, as camadas de transformação são implementadas em *stream* e se encontram em dois pontos. O primeiro, o qual faz uma pré-validação dos dados gerados

Figura 6 – Paradigmas de processamento de dados.

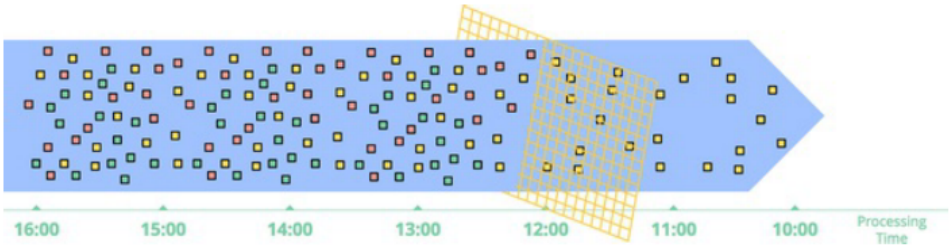
(a) Transformação de *batch* com entrada fixa (*bounded*)



(b) Transformação de *batch* com entrada incremental (*unbounded*)



(c) Transformação de *streams* (*unbounded*)



Fonte: AKIDAU; CHERNYAK; LAX (2018)

pelos sensores, é feito via um *script* em Python que recebe os dados em um servidos MQTT e envia-los aos tópicos Kafka. O segundo é uma esteira de dados Nifi, que será discutido em maior detalhes a seguir. Toda a integração é detalhada na seção de Metodologia Sec.(4).

O Apache Nifi, originalmente lançado como NIagra FIles, é uma solução orientada à manutenção e desenvolvimento de fluxos e esteiras de dados com uma tecnologia de processamento em fluxo ("*data in motion*") (ISAH; ZULKERNINE, 2018). Alguns dos princípios que guiam o desenvolvimento do projeto são: i) garantia de entrega dos dados consumidos e processados, isto é, o sistema é resiliênte à possíveis quebras; ii) priorização de enfileiramento de eventos, assim como sistemas de mensageria; iii) armazenamento em memória dos eventos para disponibilização rápida (*data buffering*) (CHATTI, 2019).

Cada registro que entra no sistema é considerado um "arquivo", o qual possui um

conteúdo e atributos (meta-dados). Esses arquivos são chamados de "*flow-files*", ou seja, arquivos que se encontram em fluxos. A cofiguração dos fluxos é feita via uma interface visual, onde cada processo (*processor*) é acionado em cadeia, e seguem caminhos específicos dependento do status de execução (sucesso, falha, retentativa, ou saídas específicas). Os processos podem ser agrupados no que se é chamado de "*process group*", e esses agrupamentos podem ser exportados como *templates* para modularização de fluxos. As figuras Fig.(16), Fig.(17) e Fig.(18) demonstram algumas das esteiras desenvolvidas para este trabalho.

3.3.3 Armazenamento de dados

Atualmente existem diversas maneiras de se armazenar grandes volumes de dados, e a escolha mais adaptada da arquitetura e de seus componentes depende de diversos fatores como consumo pelos usuários, escalabilidade, performance e integrabilidade, possíveis riscos, adaptabilidade, dentre muitos outros (MALASKA; SEIDMAN, 2018). Porém, é cada vez mais recorrente uma arquitetura de dados organizacional refletir camadas de *data-lake*, dados estruturados (bancos de dados relacionais) e dados não estruturados (bancos de dados NoSQL).

Cada uma dessas camadas serão discutidas a seguir, abordando os conceitos que serão usados na modelagem da integração dos serviços. Todas elas foram implementadas em serviços da AWS, como o S3, DynamoDB e RDS.

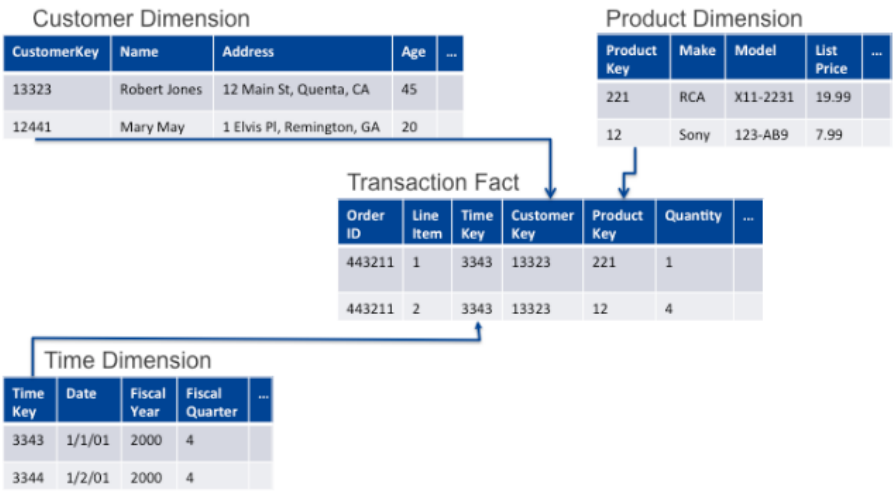
A alta volumetria, variabilidade e velocidade associado ao universo de Big Data, combinado com a agilidade de projetos de *analytics* e a demanda por gerar valor a partir dos dados à disposição acabam por impor novas necessidades de infraestruturas.

Neste contexto, emerge a ideia de *data-lake*: um macro-ambiente no qual todos os dados, independente de sua origem ou de sua forma, são armazenados e acessíveis a uma ampla gama de usuários. O objetivo é oferecer ao negócio a habilidade de se "servirem" ("*self-service*") de tais recursos de forma que não demandem ajudas de áreas de tecnologia e que consumam os dados de forma democratizada (GORELIK, 2019).

Um caso clássico de uso de *data-lake* é o processo de limpeza e estruturação de dados. É usual realizar uma série de operações e transformações em cima dos conjuntos que se dispõe a fim de se agregar valores de negócio, e usualmente este processo se chama *"Extract, Transform, Load"* (ETL) (VASSILIADIS; SIMITSIS; SKIADOPOULOS, 2002). Por finalidade de rastreabilidade e organização, é importante manter cada etapa de tratamento de armazenados em diferentes níveis.

Existem diversas ferramentas associadas a um ambiente de *data-lake*. A mais famosa é o Apache Hadoop, uma plataforma de armazenamento massiva, paralelizado e escalável. Muitas tecnologias e serviços oferecidos por provedores de nuvem possuem como base o

Figura 7 – Diagrama de relacionamento de um modelo *star schema*.



Fonte: PETROV (2019)

Apache Hadoop, como o *AWS Simple Storage System* (AWS S3) e o *Azure Blob Storage* (NARGESIAN et al., 2019).

Bancos de dados relacionais (*Relational Database Management System*, RDBMS) são softwares de manutenção de dados que já se encontram estruturados e que usualmente servem à alguma finalidade analítica ou de negócio. Os dados são estruturados em tabelas, e levam o termo "relacional" dado que tais entidades podem assumir relações entre sí através de chaves primárias (*primary key*) e chaves estrangeiras (*foreign key*) (PETROV, 2019).

As tabelas devem ser criadas e pré-adaptadas para hospedar os dados que chegam, de forma que existem inúmeras maneiras de se modelar tais relacionamentos e entidades. Um paradigma de modelagem muito usual é a modelagem dimensional, onde os dados se encontram estruturados de tal forma a reduzir redundâncias (isto é, o mesmo valor escrito em diversas linhas, colunas, tabelas e etc.) através do que se é chamado de "dimensões", ou seja, tabelas que são cruzadas para se extrair uma determinada informação sem a necessidade de se replicar o mesmo conteúdo em diversos lugares. Tabelas que possuem relações diretas com diversas dimensões seguem o que se é conceituado como "*star schema*" (esquema "estrela") (PETROV, 2019), assim como demonstra o diagrama da figura Fig.(7).

Bancos de dados relacionais são usualmente implementados, no contexto de aplicações organizacionais, em *Data Warehouses* (DW), isto é, ambiente de armazenamento estruturados de dados que são consumidos por equipes analíticas para a criação de relatórios e *dashboards*. Esses consumo ocorre através de *queries*, isto é, comandos escrito em linguagem SQL (*Structured Query Language*) que executam uma dada chamada em cima das tabelas mencionadas e retornam os dados correspondentes. Atualmente, com a evolução de diversas tecnologias de *data-lake*, ambientes DW deixaram de focar no

armazenamento global de informações e passaram a provisionar soluções analíticas que requerem dados extremamente organizados e estruturados, alimentados por múltiplas fontes e que reproduzem acurácia histórica (GORELIK, 2019).

Em contrapartida às bases RDBMS, existem bancos de dados que são orientados ao armazenamento de dados não relacionais, isto é, que não possuem o mesmo uso que dados estruturados. Esses bancos são amplamente conhecidos como NoSQL, e realizam o armazenamento de diversos tipos de dados, estruturados em diversas formas, como estrutura de documento (MongoDB e DynamoDB) ou de esquema chave-valor (Redis) (Jing Han et al., 2011). Algumas das vantagens de usar bases não relacionais são baixo custo, operações rápidas de leitura e escritas e flexibilidade de conteúdo (CATTELL, 2011).

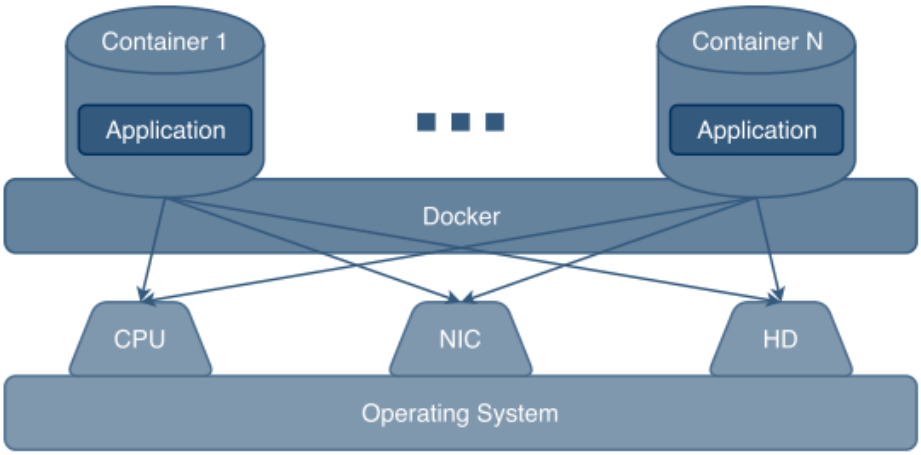
3.3.4 *Containers*, aplicações virtualizadas e Docker

O conceito de virtualização é comum no contexto de desenvolvimento, e usualmente está associado com práticas que permitem e promovem o isolamento de um *software* abstraindo dependências que possam ou não estar configuradas adequadamente na máquina hospedeira (*host*). No universo de aplicações, é comum o uso da ferramenta Docker para realizar virtualizações, onde cada instância de um *software* virtualizado (chamado também de serviço Docker) é contido em ambientes chamados "*containers*" (KROPP; TORRE, 2020). Cada unidade executável de um *container* é chamada de "imagem", a qual possui como componentes a própria aplicação, bibliotecas de dependência, variáveis de ambiente e arquivos de configuração.

Por mais similares que sejam a máquinas virtuais, os *containers* Docker abstraem os recursos da máquina hospedeira através do módulo `runC`, de forma que tais recursos são compartilhados diretamente com as instâncias das imagens em execução. Desta forma, qualquer máquina que possua o Docker instalado é capaz de rodar os containers de forma abstrata e exatamente igual ao ambiente em que foi desenvolvido, garantindo flexibilidade, interoperabilidade dentre diferentes máquinas e ambientes (desenvolvimento local ou nuvem) e escalabilidade (KROPP; TORRE, 2020). No contexto deste projeto, BELLAVISTA; ZANNI (2017) desenvolveram uma prova de conceito de implementação *gateways* IoT via conteinerização da camada de *middleware* que, como resultado, promoveu escalabilidade de desenvolvimento e orquestração dinâmica de atualizaçõs e configurações.

A unidade básica de imagens Docker parte do que se chama "*Dockerfile*", isto é, um arquivo que contém as instruções de criação do *container*. Com este arquivo, é possível compilá-lo (*build*) em uma imagem executável. O esquema de compilação é baseado em camadas, ou seja, cada comando contido no arquivo é uma camada que será executada em ordem, e a atualização de cada comando implica apenas na alteração da camada respectiva. É comum reaproveitar *containers* e imagens oficiais já desenvolvidas de projetos conhecidos

Figura 8 – Diagrama de *containers* Docker e consumo de recursos do *host*.



Fonte: AKIDAU; CHERNYAK; LAX (2018)

ao invés de desenvolve-los, o que, por padrão, é importado diretamente do repositório oficial (*registry*) do Docker, chamado de Docker Hub.

Aplicações usualmente exigem não apenas um, mas diversos *softwares* em execução simultânea, integrados por uma rede interna e com compartilhamento de volumes (dados) (DUSIA; YANG; TAUFER, 2015). Desta forma, é usual realizar estas integrações via o *software* Docker Compose, uma ferramenta orientada a execução simultânea de diversos *containers* Docker integrados e que se encontram definidos em um arquivo YAML (JANGLA, 2018).

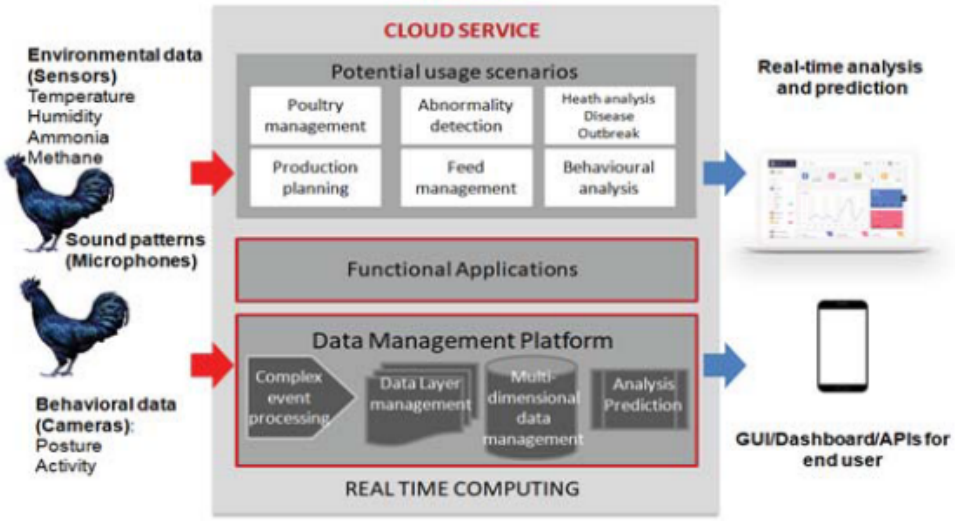
Neste projeto, usamos o Docker Compose para definir os *softwares* que devem ser executados e integrados em cada uma das camadas, sendo cada serviço apontado para arede de conexão Docker da camada. Em específico, criamos a imagem do serviço Python a partir de um Dockerfile para que sejam instaladas as bibliotecas de dependência. Os detalhes desta integração serão explicados posteriormente.

3.4 Avicultura de precisão

O uso de IoT para monitoramento de estações de avicultura já foi abordado em diversos estudos na literatura. Um dos principais desafios é realizar o monitoramento de ambiente, como temperatura e umidade, a fim de se manter as condições ótimas para produções de ovos e da avicultura. Ainda, com o uso de uma infraestrutura de monitoramento, é possível utilizar de algoritmos e métodos de aprendizado de máquina para realizar a gestão eficiente de produção e alimentação, detecção de anomalias e análises comportamentais, assim como mostra a figura Fig.(9).

A ONG "*Humane Farm Animal Care DBA Certified Humane*" disponibiliza em

Figura 9 – Exemplo de esteira de dados para aplicações de Inteligência Artificial em estações de avicultura.



Fonte: SINGH et al. (2020)

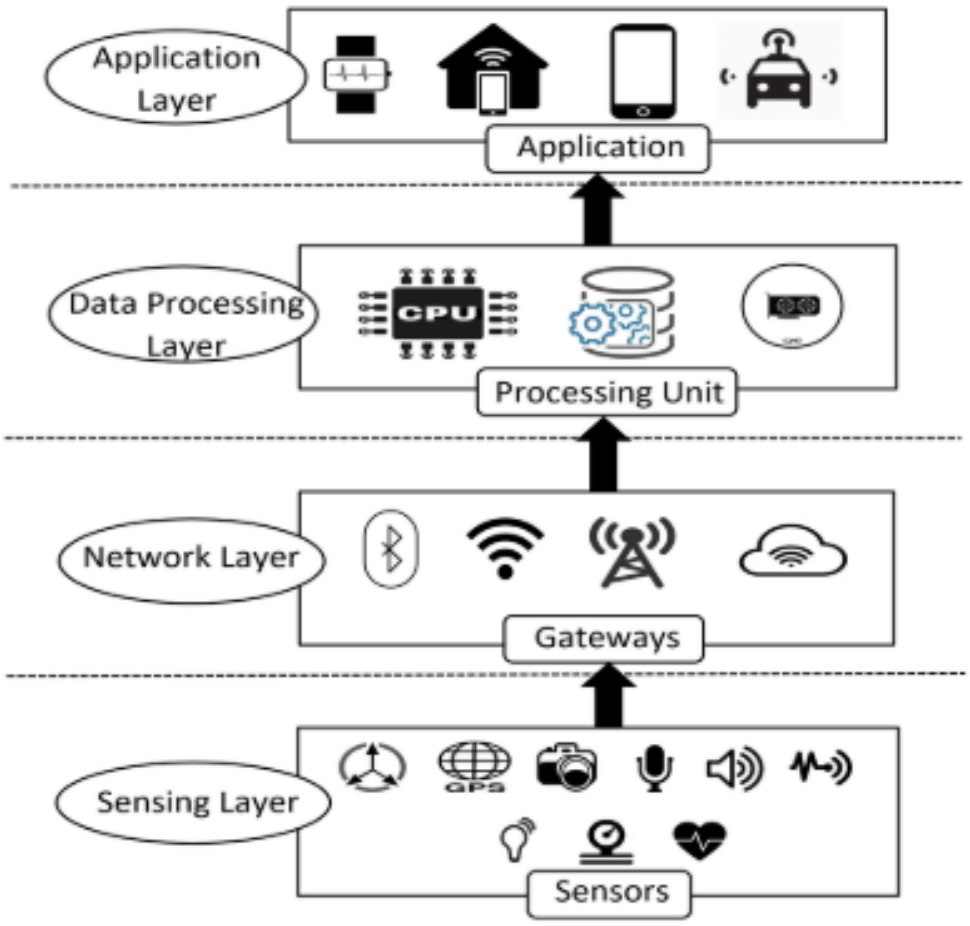
seu portal o acesso à diversas diretrizes e normas de cuidados de animais. Especificamente se tratando de produção de aviculturas, a organização elenca como fatores principais associados ao ambiente i) a temperatura, que deve ser em média de 41°C; ii) a luminosidade, que deve ser no mínimo equivalente à 20 lux; iii) concentração de amônia (não deve exceder 25 p.p.m.), monoxido (no máximo 10 p.p.m.) e dióxido de carbono (menor que 3000 p.p.m.); iv) umidade relativa, que deve ser preferencialmente de 50 à 75%, e ventilação de ar (HUMANE FARM ANIMAL CARE, 2014). Outros fatores, como infraestrutura das estações, alimentação e hidratação também são apontados pela organização, porém, para o presente projeto, usamos as métricas temperatura, umidade relativa, concentração de amônia e temperatura para desenvolver a prova de conceito da arquitetura de dados.

SINGH et al. (2020) propõem um modelo de arquitetura orientado a inteligência artificial (IA), com a finalidade de prever possíveis doenças com base em parâmetros sonoros, imagens e vídeos. Já DEBAUCHE et al. (2020) utilizam IA para examinar a qualidade do ar a partir de métricas de concentração de monóxido e dióxido de carbono, bem como a concentração de amônia.

À respeito do impacto desses fatores, eles foram escolhidos por apresentarem grande impacto no bem estar da avicultura. Por exemplo, é conhecido que a temperatura possui influência sobre a qualidade de ingestão de nutrientes e da dieta das aves (CHARLES; GROOM; BRAY, 1981). Ainda, aves que se encontram submetidas a concentrações de amônia acima de 25 p.p.m. contraem irritações nas membranas mucosas dos olhos e do sistema respiratório (WATHES; KRISTENSEN, 2000). Contração de doenças infecciosas por insuficiência respiratória também estão relacionadas a fatores como luminosidade e umidade relativa do ambiente (XIONG et al., 2017).

Ao se tratar de arquitetura de dados, LASHARI et al. (2018) propõe um modelo de monitoramento de estações de avicultura baseado em *fog computing*, assim como é apresentado na figura Fig.(10). O nódo intermediador é um *gateway*, hospedado em um Raspberry Pi, que desempenha o papel de intermediar os dados recebidos por sensores e enviá-los à uma unidade de processamento (esteira de dados). Já RAJ; JAYANTHI (2018) introduz uma camada de processamento entre a nuvem e os nódos de *gateway*, a qual é hospedada em um servidor *on-premise* e é destinada a processar os dados recebidos pelos *gateways*.

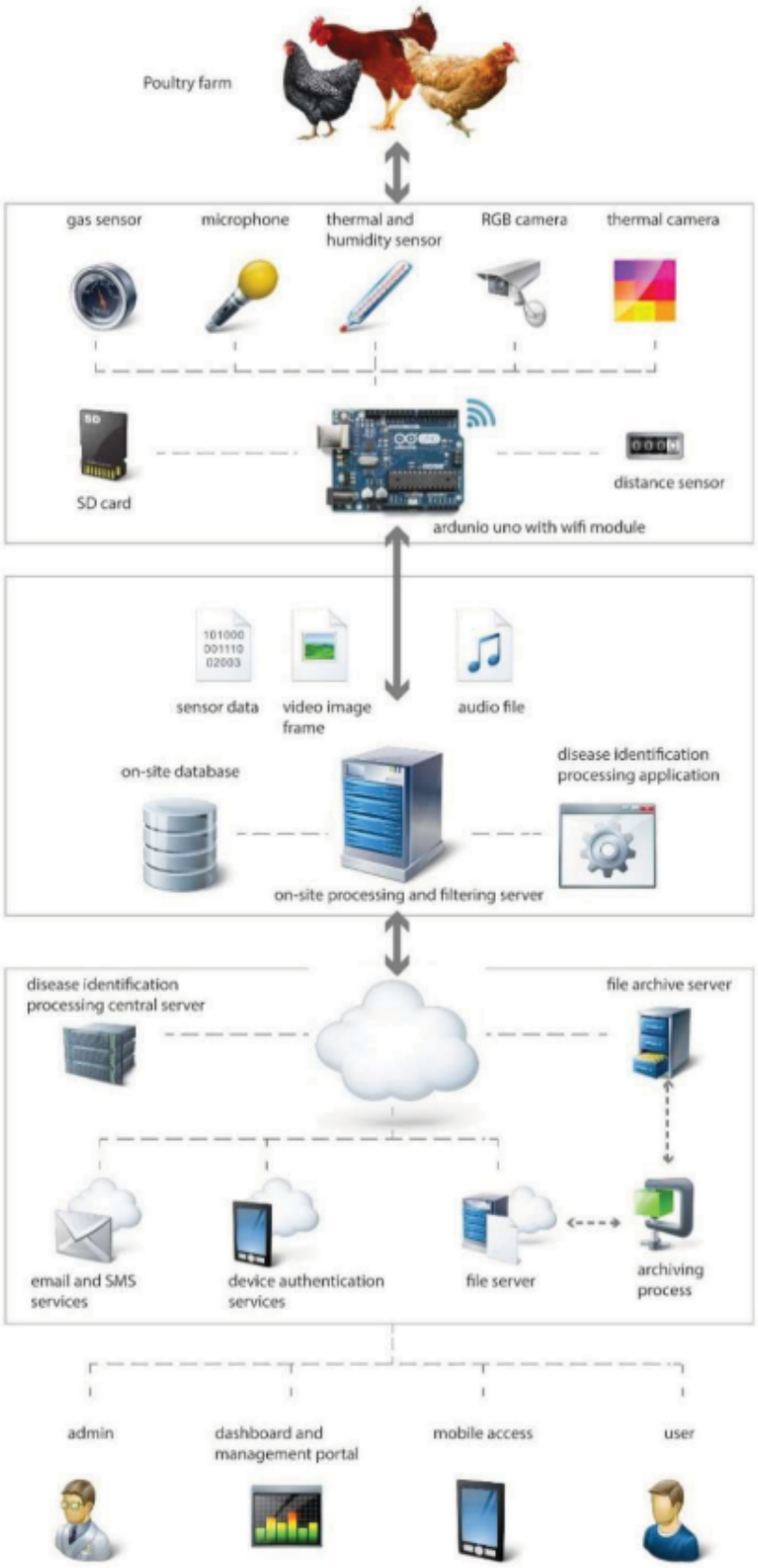
Figura 10 – Arquitetura de dados proposta por LASHARI et al. (2018).



Fonte: LASHARI et al. (2018)

Neste trabalho foi adotada a abordagem da camada *on-premise*, ou seja, integramos os dados recebidos pelos *gateways* à softwares de mensageria e tratamento em *stream*, hospedados em máquinas locais e dedicadas. Na seção de Metodologia Sec.(4) serão apresentados os componentes da arquitetura.

Figura 11 – Arquitetura de dados proposta por RAJ; JAYANTHI (2018).



Fonte: RAJ; JAYANTHI (2018)

4 METODOLOGIA

Nesta seção serão detalhados os passos seguidos para o desenvolvimento do estudo. Em suma, eles compreendem duas grandes fases, que são as etapas de desenvolvimento geral e específico. Na primeira, foram produzidos códigos de base que provisionam a integração de todos os serviços discutidos na seção de Fundamentação Teórica (Sec.(3)) com a finalidade de servir à aplicações IoT que contenham as camadas de arquitetura propostas neste estudo. Na segunda, os insumos gerados pela primeira etapa foram usados para modelar um cenário de avicultura de precisão, a qual foi caracterizada por uma modelagem de dados e de recursos (*brokers* e tópicos Kafka, nódos de *gateway*) para refletir especificidades do cenário em questão.

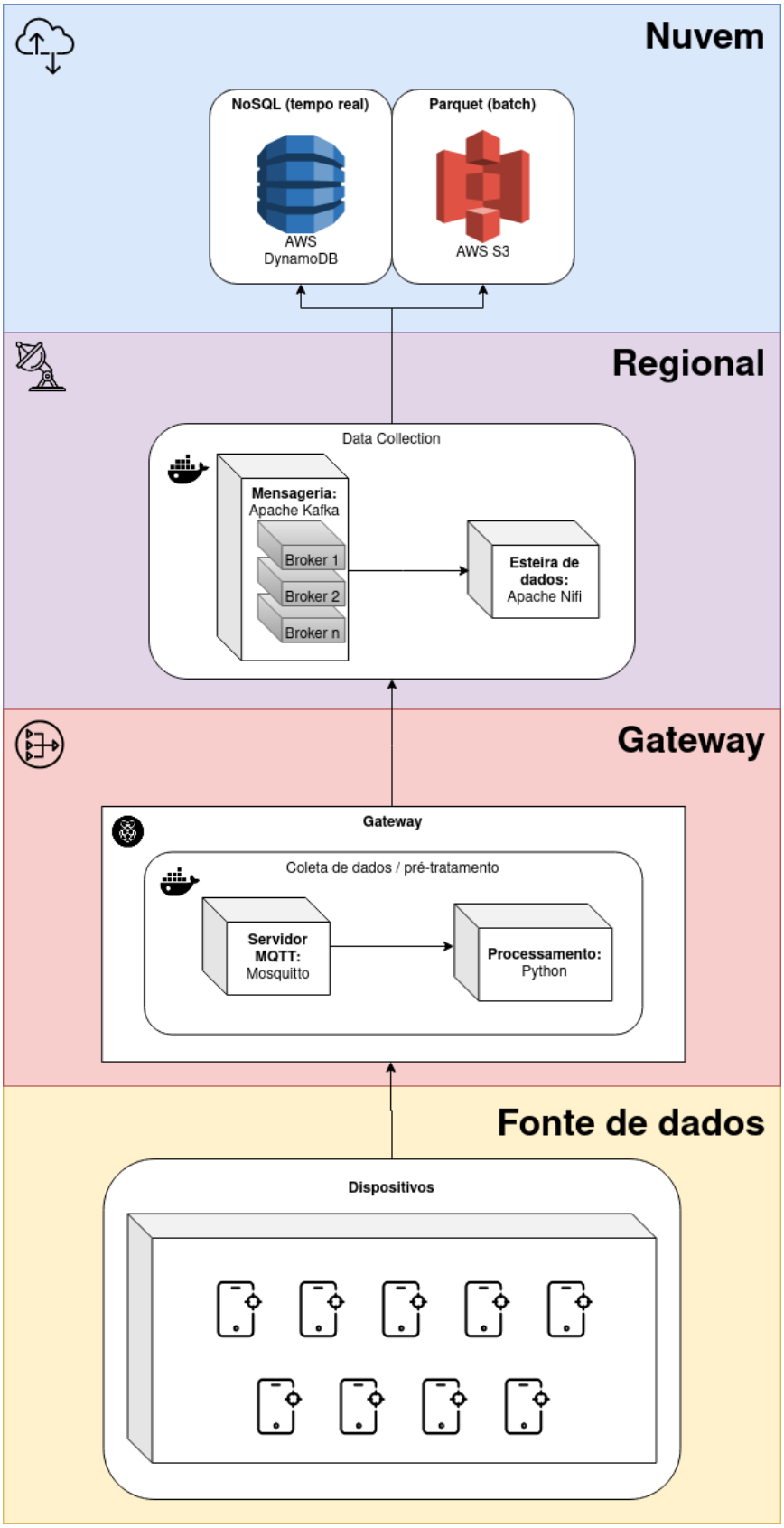
4.1 Etapa geral: Modelagem de integração de serviços

A arquitetura geral implementada neste estudo segue a representação da figura Fig.(12). Ela é consituida de três camadas principais:

- ***Gateway***: camada localizada em cada um dos dispositivos de sistema embarcado (e.g., Raspberry Pi), é encarregada por hospedar um servidor MQTT e um script Python que realiza o consumo deste servidor, bem como a limpeza e validação de estrutura de dados gerados pela fonte;
- **Regional**: camada responsável por receber em um *cluster* Kafka os dados validados por cada nódo de *gateway*, bem como realizar, via NiFi, tratamentos, análises, aglomerações e operações lógicas com base no conteúdo do dado. Por fim, envia os dados à nuvem;
- **Nuvem**: camada que faz uso de serviços e ferramentas proprietárias (AWS), a fim de prover armazenamento tanto em forma aglomerada (*Batch*) quanto em estrutura de documento (tempo-real, *Stream*)

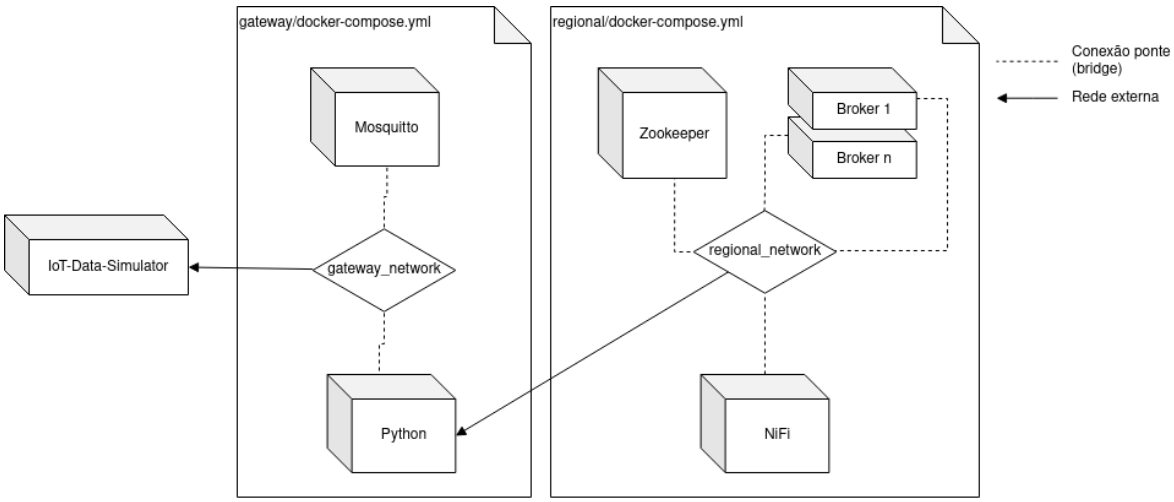
Para os níveis de *gateway* e regional, a integração de todos os softwares foi virtualizada via containers Docker, de forma que para cada camada foi escrito um arquivo `docker-compose.yml`. Nesses arquivos são definidas as redes de conectividade de cada um dos serviços, que seguem a estrutura lógica do diagrama apresentado na figura Fig.(13). Desta forma, todos os nódos de *gateway* devem ter o serviço de Python com acesso à rede externa da camada regional, assim como os dispositivos IoT devem ter acesso à rede externa do *gateway*.

Figura 12 – Representação da arquitetura geral.



Fonte: Autor

Figura 13 – Diagrama de conexões de serviços Docker.



Fonte: Autor

Para a geração dos dados sintéticos, foi utilizada a aplicação *IoT-data-simulator* (IBA-GROUP-IT, 2020), a qual suporta a configuração de diversos sensores, modelagem do esquema dos dados, conexão com aplicações de mensageria e execução de diversas sessões simultaneamente. Para cada sessão, é possível salvar um arquivo de extensão `.json` que detalha a configuração da sessão, o esquema de dados, sensores utilizados, tópicos publicados e a frequência de sinal.

Todo o código foi desenvolvido e publicado no *GitHub*, e se encontra atualmente no repositório `igor-yamamoto/tcc_proj` (YAMAMOTO, 2021). Os arquivos referentes às camadas de *gateway* e regional estão, respectivamente, nos diretórios `gateway/` e `regional/`. Os arquivos de configuração do servidor MQTT e a credencial de acesso, bem como o *template* de base da aplicação *IoT-data-simulator*, estão todos no diretório `gateway/templates/`. O tempate NiFi se encontra no diretório `regional/templates/`. O diretório `examples/` é destinado para armazenar usos e aplicações dos modelos desenvolvidos, e, para este projeto, foi usado para a etapa de desenvolvimento específico.

A tabela Tab.(1) apresenta detalhes de especificações de cada uma das tecnologias utilizadas nesta etapa.

4.1.1 *Gateway*

Em cada nódo de *gateway* estão hospedados os serviços Mosquitto e Python. O serviço de Python é orientado ao processamento das mensagens enviadas ao servidor MQTT, e portanto a sua execução foi configurada como dependente deste serviço. Ainda, em casos de queda do sistema, a política de reinicialização foi configurada para acontecer sempre que o serviço finalizar a execução de forma inesperada (`restart: always`).

As configurações dos *brokers* MQTT em cada nódo são definidas em arquivos

Tabela 1 – Especificações de softwares utilizados na implementação da etapa geral.

Escopo	Ferramenta	Imagem docker	Rede conectada
Fonte de dados	IoT-data-simulator	-	gateway_network (externo)
Gateway	Mosquitto	amd64/ eclipse-mosquitto	gateway_network
	Python	amd64/ python:3.9.7-alpine3.14	gateway_network, regional_network (externo)
Regional	Zookeeper	confluentinc/ cp-zookeeper:6.2.0	regional_network
	Kafka	confluentinc/ cp-kafka:6.2.0	regional_network
	Nifi	apache/ nifi:1.13.2	regional_network
Nuvem	AWS S3	-	-
	AWS DynamoDB	-	-

que se encontram no caminho `gateway/templates/mosquitto/mosquitto.conf`, e são mapeados diretamente aos arquivos de configuração das imagens Docker (`mosquitto/config/mosquitto.conf`). Neste arquivo também é feito a configuração de segurança, como permissões e credenciais de acesso (mapeando o arquivo `gateway/templates/mosquitto/mqtt_passwd`), configurado para este exemplo o usuário "*tcc_test*"e a senha "*12345*".

Para o Python, a subida do serviço é diferente em relação aos demais, pois nele constrói-se um container (*build* apontando ao caminho `gateway/builds/python/`) baseado na imagem `python:3.9.7-alpine3.14`, já instalando as dependências definidas no arquivo `code/requirements.txt` e iniciando o consumo e produção das mensagens (arquivo `code/scripts/mqtt_consumer.py`, método `run`). Para este serviço, as bibliotecas de dependência são:

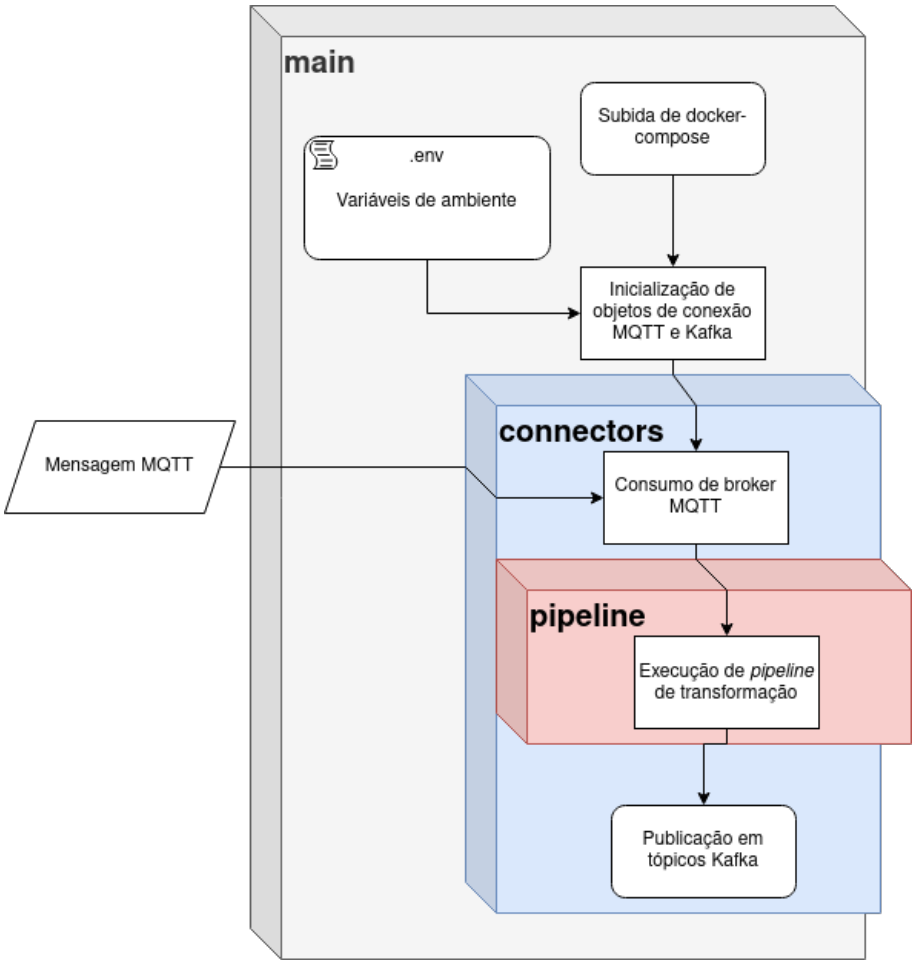
- `paho-mqtt`: biblioteca de conexão, consumo e produção de mensagens em *brokers* MQTT
- `python-decouple`: módulo de leitura e uso de variáveis de ambiente. Útil para desenvolvimento, testes e manutenção de hiperparâmetros e/ou credenciais

- `kafka-python`: biblioteca de conexão, consumo e produção de mensagens em *brokers* Kafka

Além do script de consumo, são copiados também para a imagem as classes de conexão com *brokers* (`/code/scripts/connectors.py`) e o *pipeline* de tratamento (`/code/scripts/pipeline.py`). Os módulos foram escritos para que, preferencialmente, qualquer configuração de variável fosse definida no arquivo de variáveis de ambiente `/code/scripts/.env` e qualquer lógica de tratamento dos dados consumidos fossem modeladas a partir do método `main` do arquivo de *pipeline*. Para esta etapa de desenvolvimento, o *brokers* de transformação escrito realiza apenas a validação do esquema de dados em *json*, mas é possível integrar qualquer lógica de tratamento de forma que ele tenha como resultado a mensagem em formato de texto (*string*).

A estrutura lógica do serviço Python segue o fluxograma da figura Fig.(14).

Figura 14 – Fluxo lógico do serviço Python.



Fonte: Autor

O desenvolvimento deste projeto foi baseado nas imagens Docker apresentadas na tabela Tab.(1). Para a camada de *gateway*, todas estão disponíveis no *DockerHub* (*registry* oficial do Docker), compiladas tanto em arquiteturas padrões (*amd64*, orientado

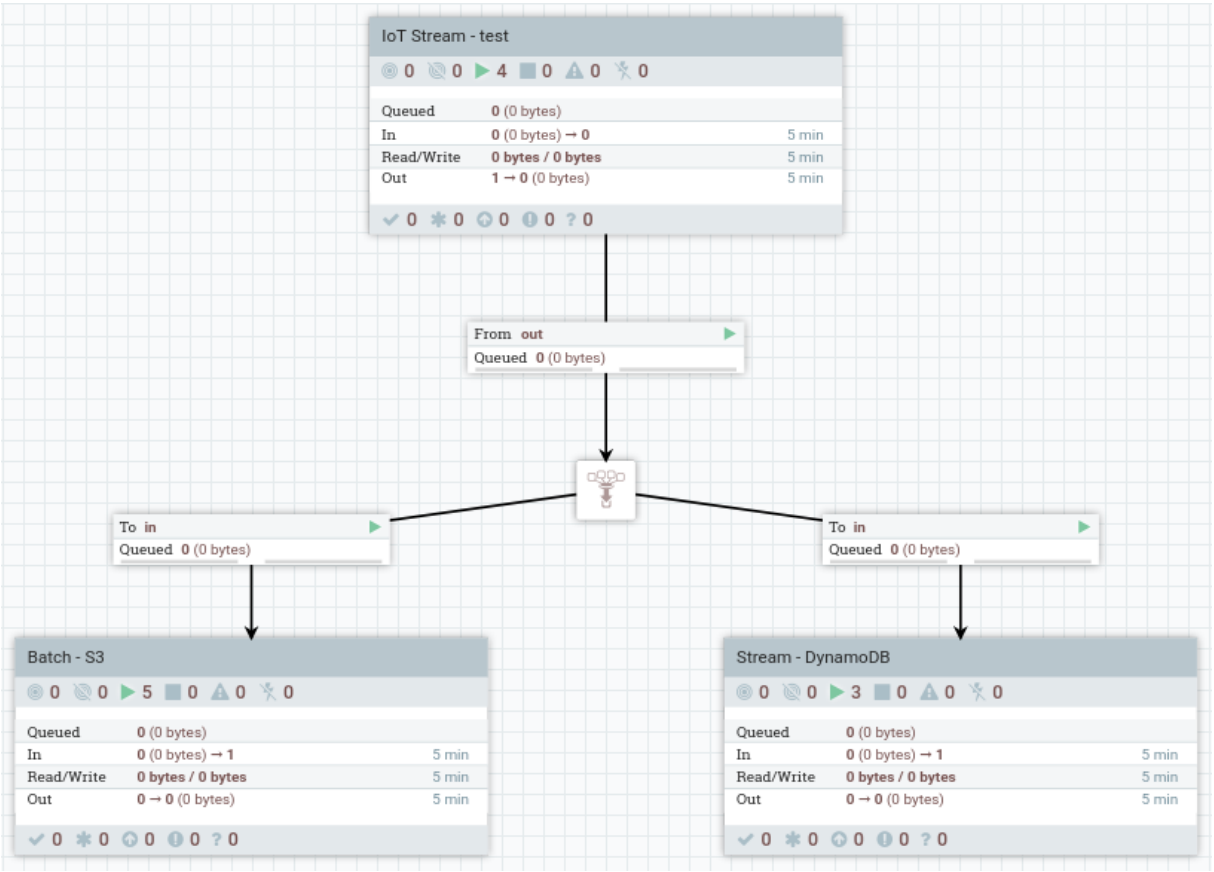
à processadores Intel), quanto arquiteturas *arm64* e *arm32* (arqueturas presentes em processadores de raspberry, como o Broadcom BCM2711 SoC).

4.1.2 Regional

A camada regional é constituida de um *cluster* Kafka e uma esteira NiFi. Esta camada é destinada para ser usada de forma única e centralizada, estando todos os *gateways* conectados à ela. Para tal, a implementação de qualquer especificidade deve ser modelado dentro do arquivo `regional/docker-compose.yml` (quantidade de *brokers* do *cluster*, quantidade de nódos do serviço Zookeeper, arquivo de configuração do NiFi, etc.).

Para a etapa geral, foi configurado apenas um broker Kafka orquestrado pelo Zookeeper. A porta mapeada para comunicação com o Zookeeper foi a porta padrão, i.e., `2181`. Scripts *bash* foram criado para auxiliar na criação de tópicos implementados neste projeto, e se encontram no diretório `regional/scripts/`.

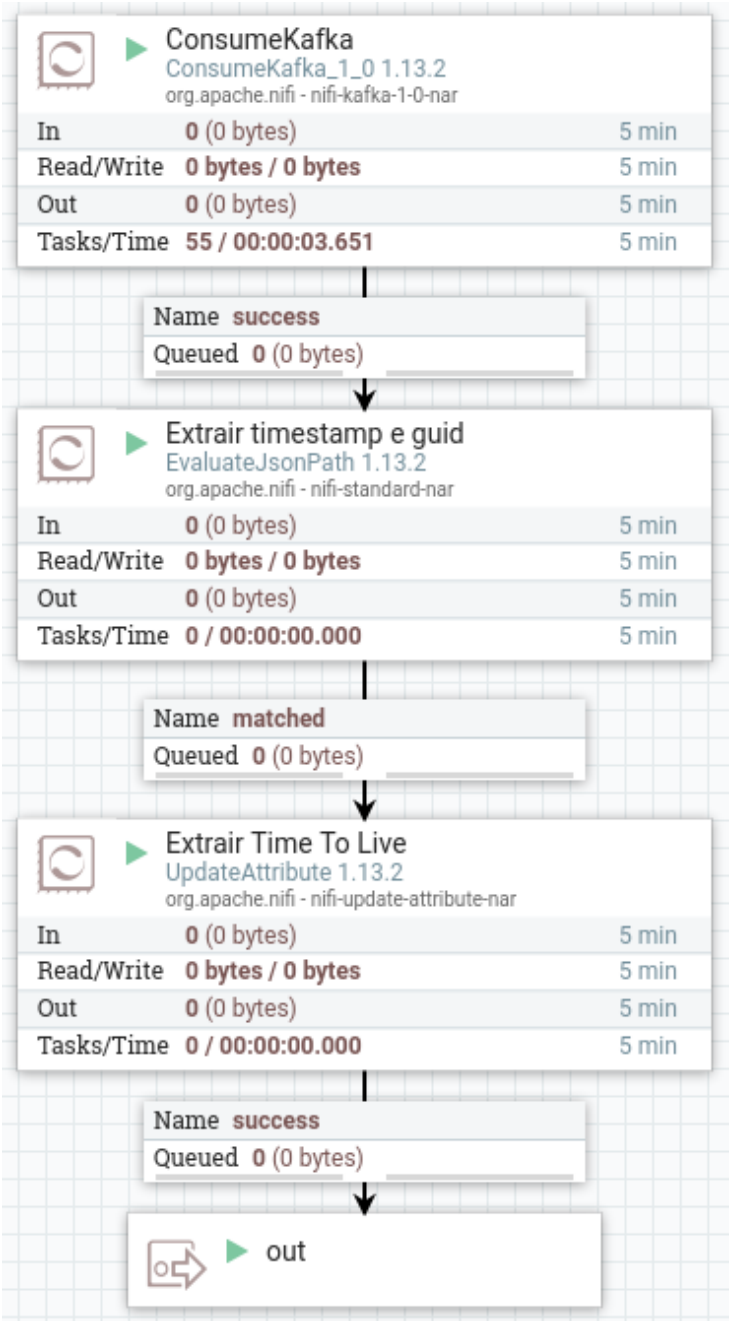
Figura 15 – Esteira de dados NiFi geral



Fonte: Autor

O *template* de todos os processos NiFi estão apresentados nas figuras Fig.(15), Fig.(16), Fig.(17) e Fig.(18). Ele possui três processos principais:

Figura 16 – Esteira de consumo Kafka

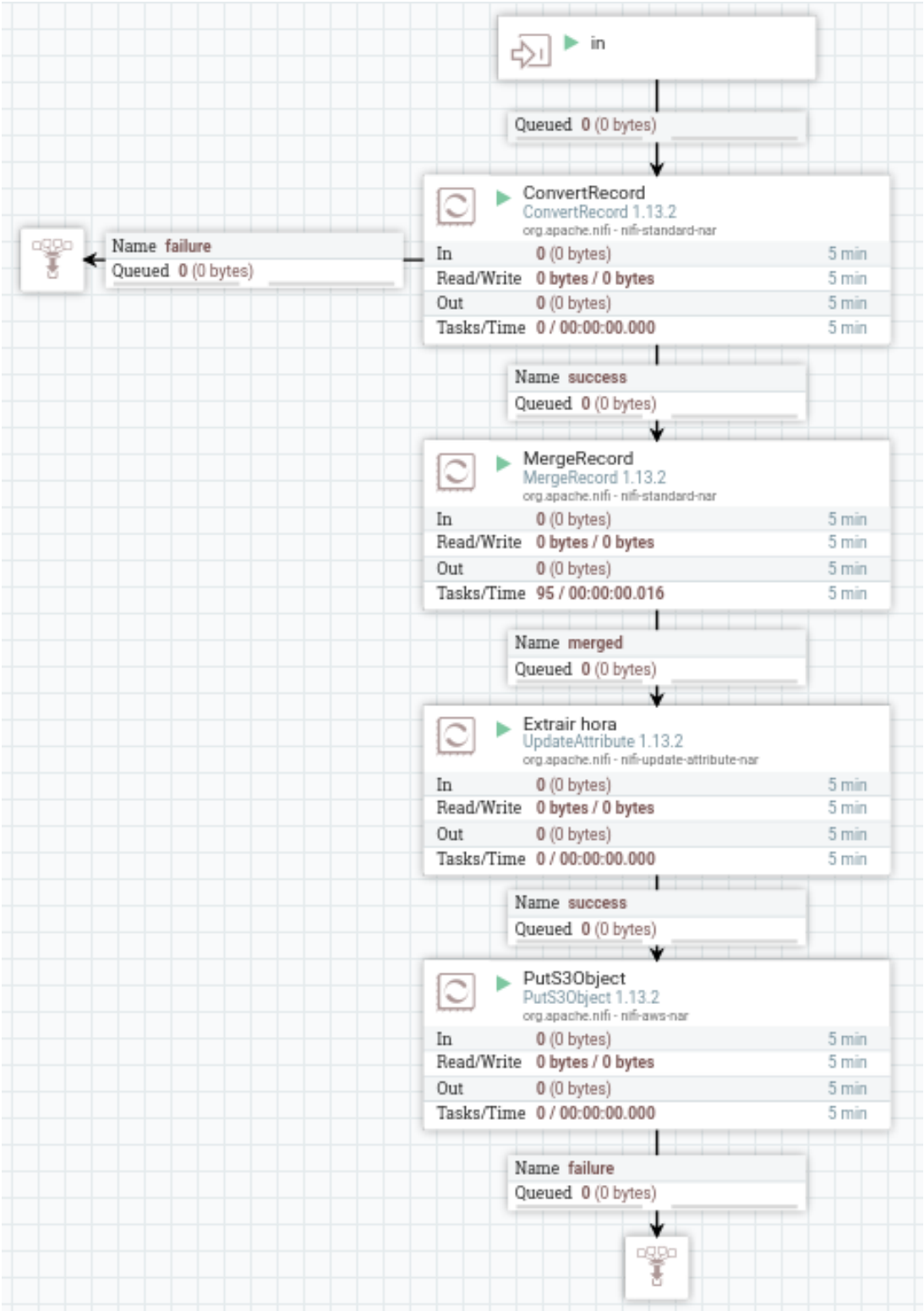


Fonte: Autor

- **Consumo de tópicos Kafka**: processo destinado a consumir todos os eventos registrados no tópico `test-topic` (figura Fig.(16)). A partir do conteúdo da mensagaem, é feito também uma extração dos valores de identificador único global (*GUID*) do dispositivo, bem como o contexto do evento (nesta etapa, configurado como "teste");

- **Geraração de arquivos aglomerados**: processo destinado ao serviço AWS S3 (figura Fig.(17)), no qual os dados coletados são transformados para estrutura posicional (formato `.csv`), aglomerados, transformados em arquivos de estrutura colunar (formato `.parquet`) e gravados no bucket `test-tcc-bucket` particionadas por data

Figura 17 – Esteira de ingestão S3

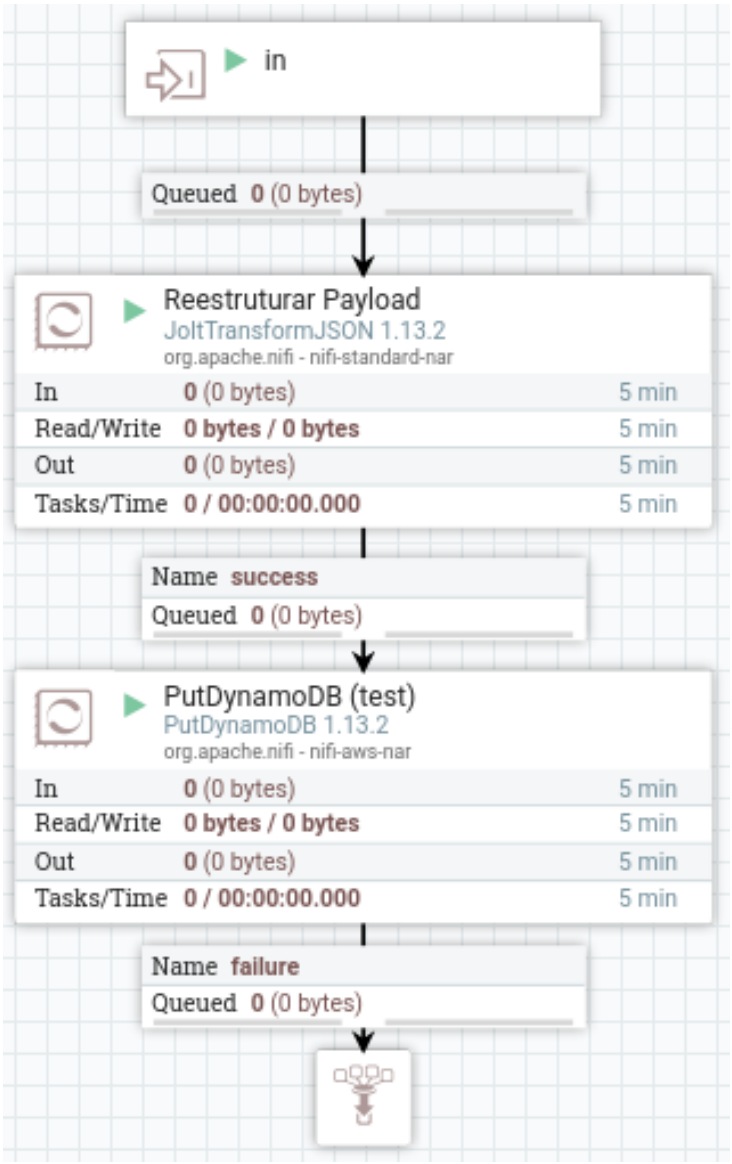


Fonte: Autor

e hora (caminho `test-tcc-bucket/test/date=${yyyy-MM-dd}/hour=${HH}`, onde `${yyyy-MM-dd}` e `${HH}` representam a data e a hora, respectivamente);

- **Gravação de eventos em tempo real**: processo destinado ao serviço AWS Dy-

Figura 18 – Esteira de ingestão DynamoDB



Fonte: Autor

namoDB (figura Fig.(18)), onde cada evento é tratado para armazenar o campo `timestamp_ttl`, o qual marca a data-hora de expiração do evento na base (configurado, para este projeto, como um dia a partir do momento de geração do evento) em formato *unix time* (quantidade de segundos a partir to tempo *epoch*, definido como 01/01/1970, 00:00:00), e então grava o evento na tabela DynamoDB `test-table`.

Para a execução das tarefas mencionadas, foram configurados controladores de serviços, que são serviços compartilháveis entre múltiplos processos. No contexto deste projeto, foram configurados os controladores apresentados na figura Fig.(19), que são de conversão e leitura de esquema de dados (*CSVReader*, *CSVRecordSetWriter*, *JsonTreeReader* e *ParquetRecordSetWriter*, todos da versão 1.13.2), assim como um de configuração de credenciais de acesso à AWS (*AWSCredentialsProviderControllerService* 1.13.2).

Figura 19 – Controladores de serviços NiFi usados no projeto

	Name ▲	Type
	AWSCredentialsProviderControllerService	AWSCredentialsProviderControllerService 1.13.2
	CSVReader	CSVReader 1.13.2
	CSVRecordSetWriter	CSVRecordSetWriter 1.13.2
	JsonTreeReader	JsonTreeReader 1.13.2
	ParquetRecordSetWriter	ParquetRecordSetWriter 1.13.2

Fonte: Autor

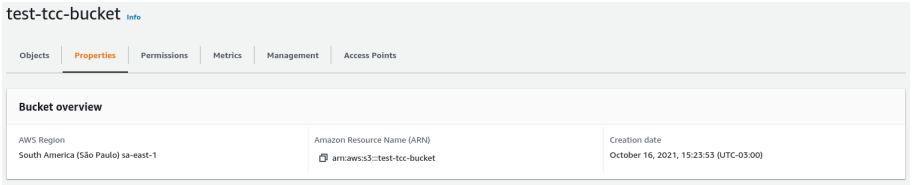
4.1.3 Nuvem

Os serviços utilizados da nuvem foram S3 e DynamoDB, do provedor AWS, assim como apresentado na figura Fig.(20). Eles são destinados para fazer o armazenamento de dados aglomerados (*data-lake*, *batch*) e em tempo real (NoSQL, *stream*), respectivamente. A configuração foi feita de forma padrão, ou seja, da maneira recomendada pela própria AWS, exceto para as tabelas DynamoDB, onde foi configurado o mapeamento de um atributo de expiração (*time to live*, TTL), assim como mostra a figura Fig.(20b).

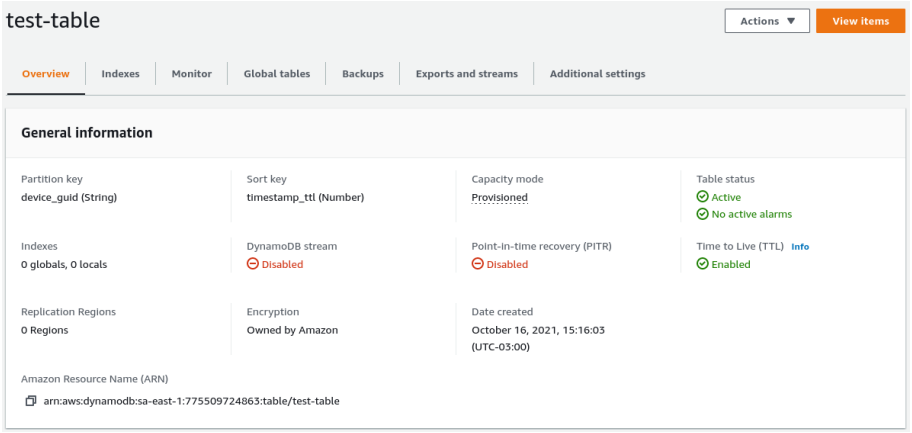
É válido mencionar também que, para a etapa específica, além dos serviços S3 e DynamoDB, foi usado o AWS RDS para hospedar um banco de dados Postgres e então possibilitar consumir os dados de forma analítica em um Dashboard. Maiores detalhes serão discutidos na próxima seção.

Figura 20 – Detalhes de serviços AWS S3 e AWS DynamoDB

(a) Detalhes do AWS S3



(b) Detalhes do AWS DynamoDB



Fonte: Autor

4.2 Etapa específica: Modelagem de experimento conduzido

Com os insumos gerados na primeira etapa, foi modelado um experimento orientado à avicultura de precisão. Nesta subseção, serão apresentados os detalhes desta modelagem, e as principais especificades de arquitetura. Em suma, o experimento consiste de duas estações de monitoramento, denominadas por A e B, na qual para cada uma é atribuido N sensores e um *gateway*. O *cluster* Kafka foi configurado, desta vez, com três *brokers*, sendo os tópicos divididos por tipo de métrica levantada pelos sensores (chamado aqui também de "*contexto*"). O diagrama da figura Fig.(21) apresenta a estrutura lógica de todos os softwares usados nesta arquitetura.

Todos os arquivos referente à esta etapa se encontram no diretório `examples/avicultura/`. Os arquivos de *template* do simulador, bem como configurações dos *gateways*, se encontram nas pastas `gateway-A/templates/` e `gateway-B/templates/` para os *gateways* A e B, respectivamente. Scripts *bash* de criação de tópicos Kafka se encontram em `regional/scripts/`, e o *template* da esteira NiFi se encontra em `regional/templates/`.

4.2.1 Dados sintético, modelagem e especificações

De acordo com o que foi apresentado na seção Fundamentação Teórica (Sec.(3)), alguns dos principais fatores que possuem influência sobre o bem estar de uma cultura de aves são temperatura, concentração de amônia, umidade e luminosidade (HUMANE FARM ANIMAL CARE, 2014). Existem diversos dispositivos que fazem o sensoriamento de tais condições, onde o conteúdo e a frequência de sinal é usualmente modelado por microcontroladores. Para este experimento conceitual, as características de cada tipo de sensor foi modelado de acordo com as informações da tabela Tab.(2). Nela também é apresentada a quantidade de cada um dos sensores presentes para cada estação de monitoramento.

Tabela 2 – Especificações de sensores principais modelados no experimento de caso.

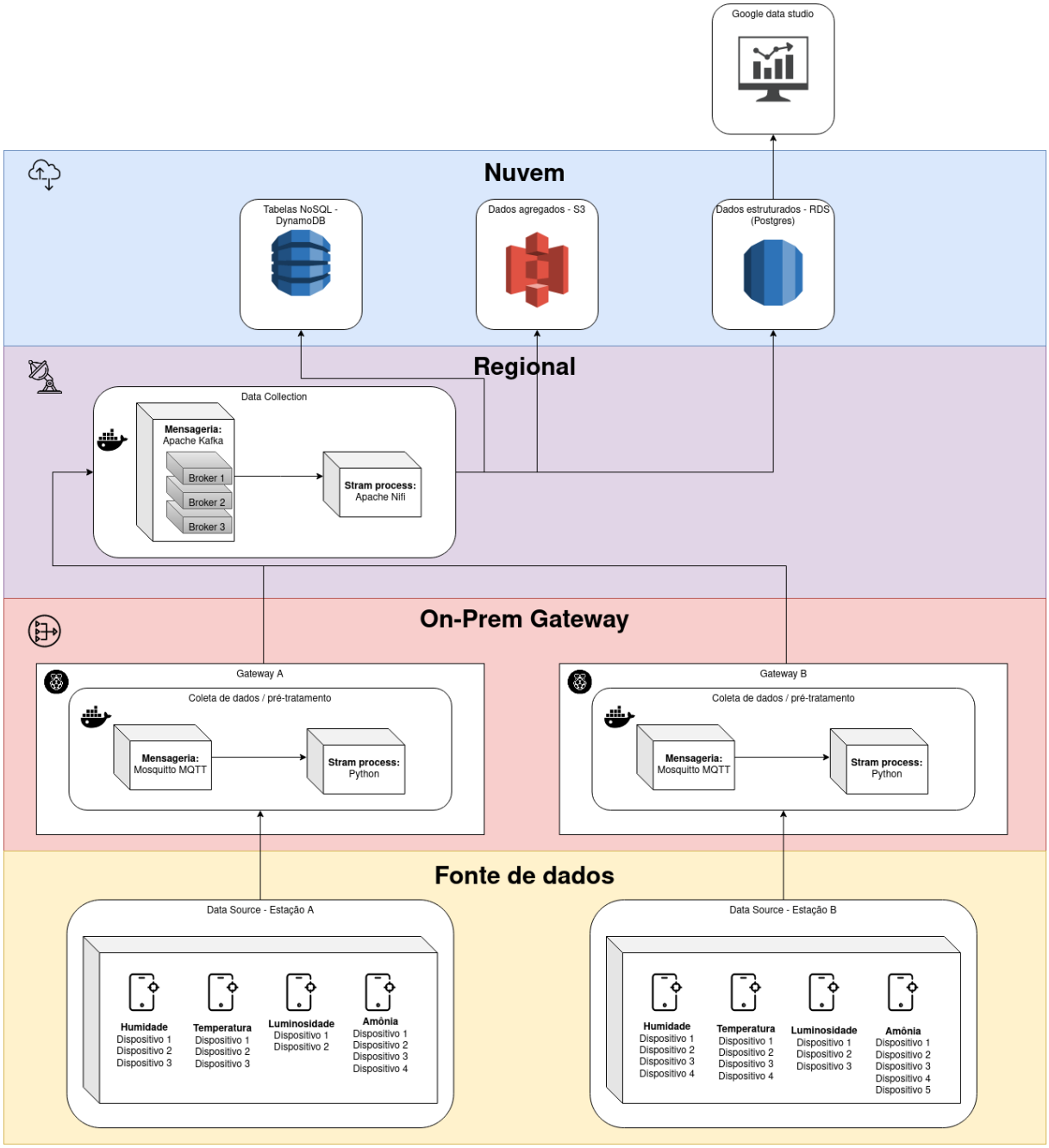
Tipo de sensor	Frquência de sinal	Faixa de valores	Qntd. A	Qntd. B
Temperatura	1 minuto	$37 \sim 40$ (°C)	3	4
Concentração de Amônia	30 segundos	$5 \sim 7$ (p.p.m.)	4	5
Umidade relativa	5 minutos	$60 \sim 65$ (%)	3	4
Luminosidade	5 minutos	$25 \sim 30$ (lux)	2	3

Fonte: HUMANE FARM ANIMAL CARE (2014)

O conteúdo do sinal emitido por eles é estruturado em um esquema de chave-valor (*json*), e seguem estrutura comum, que é constituido dos seguintes campos:

- **`device_guid`**: texto de identificador único do dispositivo

Figura 21 – Representação da arquitetura integrada para o caso estudo.



Fonte: Autor

- `guid`: texto de identificador único do sinal
- `timestamp`: número inteiro de marcador do momento de geração do sinal, em formato *unix time*
- `context`: texto de classificador do tipo de sensor
- **Valor de medida**: número inteiro de valor da medida do sensor. Nome do campo é dependente do tipo de medida (e.g., se temperatura, então campo de medida é `temperature`)

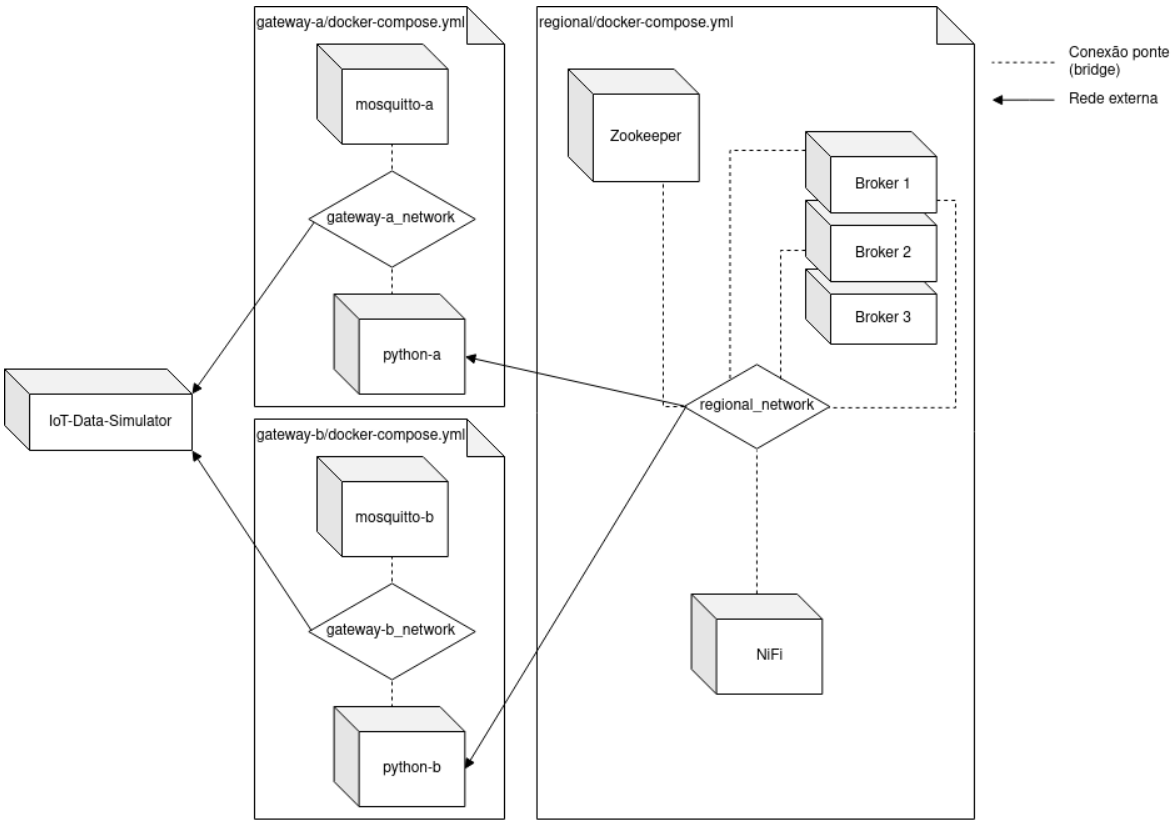
Código 4.1 – Exemplo de sinal gerado por um sensor de temperatura.

```
{
    "device_guid": "001",
    "guid": "1b878216-2b66-45d7-97d3-f859233ae85e",
    "timestamp": 1634484382,
    "context": "temperature",
    "temperature": 25
}
```

4.2.2 *Gateways* e pré-tratamento de dados

Para cada uma das estações de monitoramento foi configurado um *gateway*, denominados gateway-A e gateway-B. Em cada um deles foi criada uma rede de conexão interna, onde os serviços de Python se conectam à rede da camada regional via conexão externa, assim como demonstra a figura Fig.(22).

Figura 22 – Diagrama de redes dos serviços Docker para estudo de caso



Fonte: Autor

A esteira de tratamento do Python é semelhante à desenvolvida na primeira etapa, acrescentando apenas um novo processo que realiza a publicação no tópico Kafka `devices-dump` das mensagens que não ocorre êxito no mapeamento de contexto.

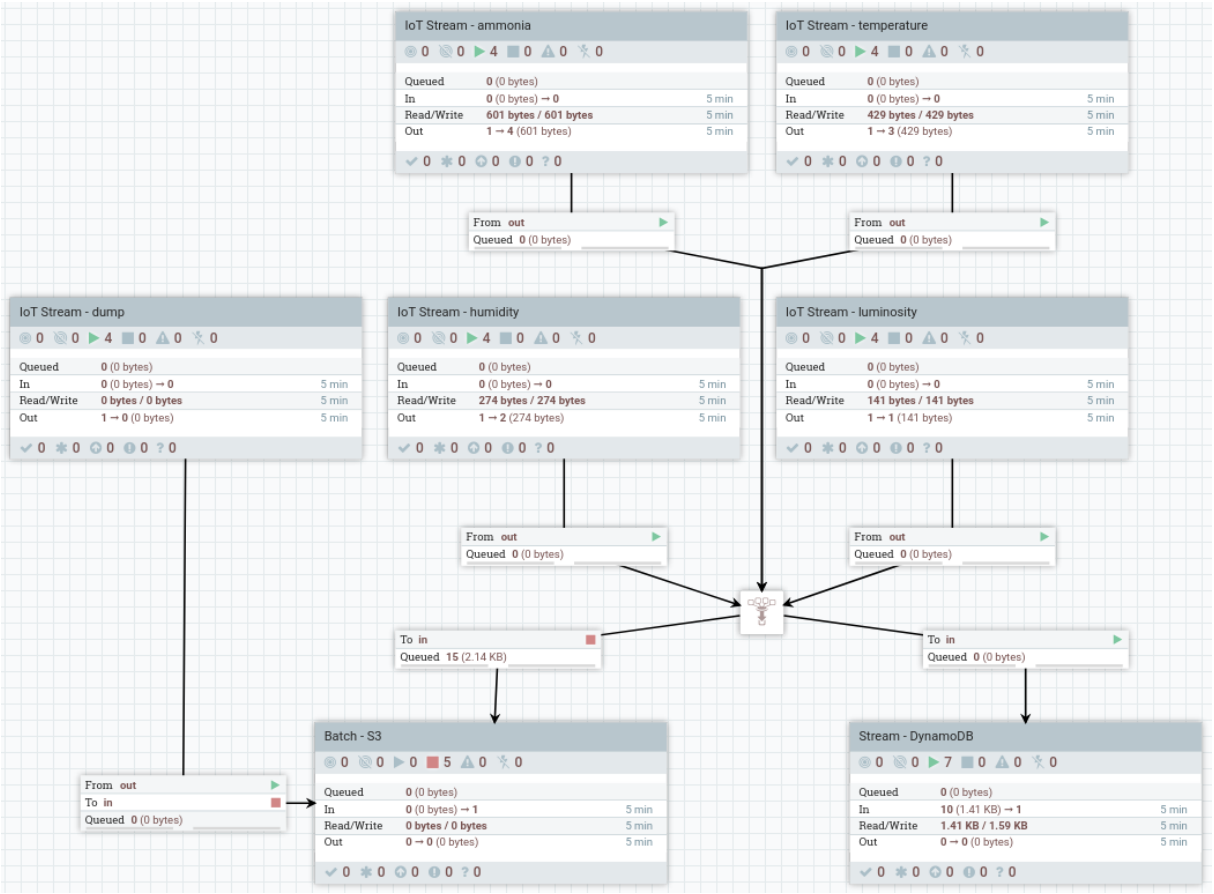
4.2.3 *Cluster* Kafka e esteira de dados Nifi

O Cluster Kafka foi configurado com três *brokers*, denominados de broker-a, broker-b e broker-c, com portas mapeadas à 29091, 29092 e 29093, respectivamente. Todos eles apontam a um único orquestrador Zookeeper, pela porta 2181.

Os tópicos no nível regional foram separados por contexto, a fim de se simplificar as operações lógicas na esteira NiFi. No total são 5 tópicos distintos: *devices-temperature*, *devices-ammonia*, *devices-luminosity*, *devices-humidity* e *devices-dump*. Todos eles foram criados com fator de replicação de 2 nódos e 10 partições cada, e podem ser iniciados diretamente através do script *bash* de criação `regional/scripts/kafka_create_topic.sh`.

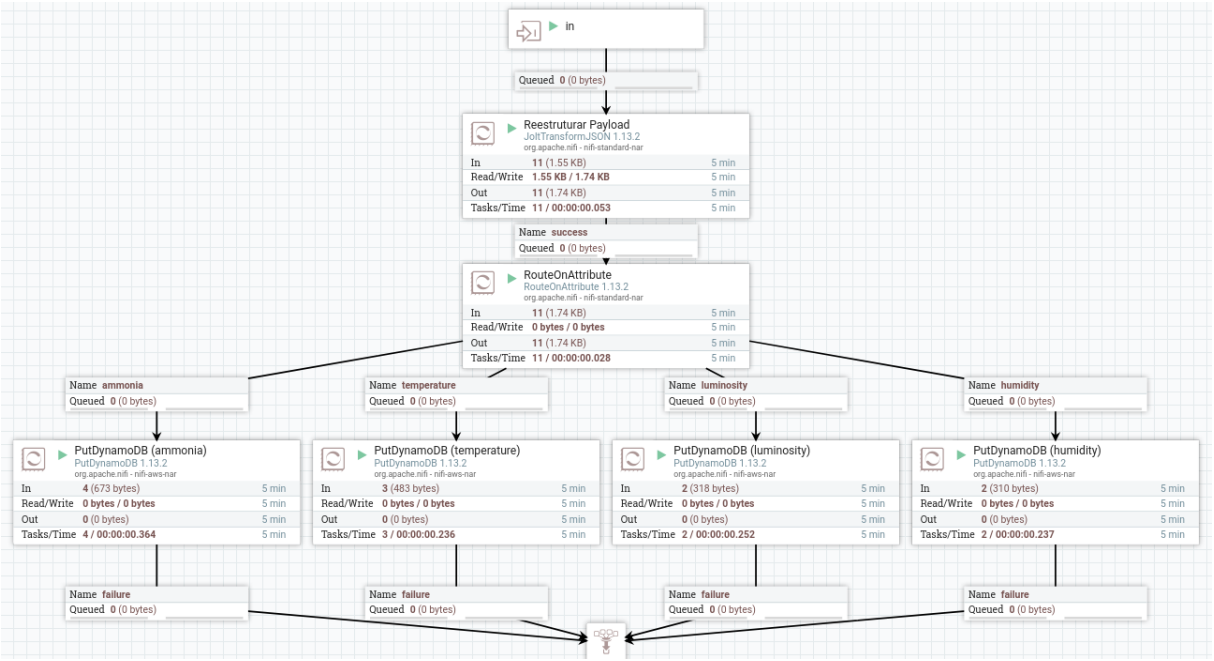
Exceto ao tópico *devices-dump*, todos as mensagens são salvas tanto em *batch* quando em *stream*. Para cada tópico Kafka, é configurado um processo como é apresentado na figura Fig.(16), e a esteira resultante é apresentada na figura Fig.(23). Uma mudança adicional foi feita sobre o processo de gravação em tabelas DynamoDB, onde foi criado para cada tabela um processo separado, assim como demonstra a figura Fig.(24).

Figura 23 – Esteira geral



Fonte: Autor

Figura 24 – Esteira para salvar em tabelas DynamoDB



Fonte: Autor

4.2.4 Armazenamento de dados em Stream e Batch

O armazenamento em *batch* e *stream* foi particionado em contexto, assim como os tópicos Kafka. Os arquivos *parquet* foram salvos no bucket `iot-stream-tcc`, todos particionados por data e hora. As tabelas do AWS DynamoDB foram configuradas da mesma maneira descrita na etapa geral, e levam os nomes *iot-stream-ammonia*, *iot-stream-humidity*, *iot-stream-luminosity* e *iot-stream-temperature*.

Além de tais serviços de armazenamento, foi usado também o banco de dados Postgres SQL para armazenamento de dados estruturados. Para cada contexto foi criada uma tabela, chamadas de tabelas fatos, onde todas elas consomem uma tabela uma única tabela, chamada de dimensão, a fim de se resgatar informações dos dispositivos da base. No total, as tabelas criadas foram `ft_ammonia`, `ft_humidity`, `ft_luminosity`, `ft_temperature` e `dm_device`, todas sobre o esquema `iotdb`. O diagrama da figura Fig.(25) resume o relacionamento entre as tabelas, bem como os campos, esquema e nulabilidade de cada uma. Esta base foi usada para alimentar um *Dashboard* do Google Data Studio, sumarizando as métricas recebidas dos dispositivos. Detalhes deste painel será discutidos em maiores detalhes na seção de Resultados Sec.(5).

Figura 25 – Diagrama de tabelas no banco de dados Postgres, esquema `iotdb`

Fonte: Autor

5 RESULTADOS

Nesta seção serão apresentados os principais resultados obtidos do estudo realizado. Nele constam os resultados das integrações dos softwares, análise do *cluster* Kafka, bem como o consumo e a disponibilização dos dados na nuvem em formato de *dashboard*.

5.1 Integração de serviços

5.1.1 *Gateway*

Os testes dos *gateways* foram realizados através de comandos de execução sobre o container Docker do serviço Mosquitto. Para garantir, em um primeiro momento, que as mensagens estavam sendo recebidas pelo servidor, e portanto passíveis de serem consumidas por demais serviços, foi realizada a subida da imagem, o consumo de um tópico de teste chamado `test message` e então a publicação de mensagens. As linhas de código Cód.(5.1) e Cód.(5.2), escritas em linguagem *bash*, foram usadas para realizar o consumo e a produção de mensagens, respectivamente. A imagem Fig.(26) mostra o resultado da execução de tais comandos em um terminal *shell*, acompanhado logo abaixo dos logs de execução do comando.

Código 5.1 – Comando de consumo de tópicos mosquitto MQTT

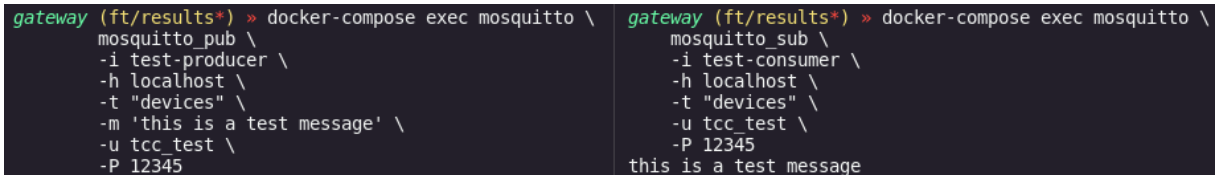
```
docker-compose exec mosquitto \
    mosquitto_sub \
    -i test-consumer \
    -h localhost \
    -t "devices" \
    -u tcc_test \
    -P 12345
```

Código 5.2 – Comando de produção em tópicos mosquitto MQTT

```
docker-compose exec mosquitto \
    mosquitto_pub \
    -i test-producer \
    -h localhost \
    -t "devices" \
    -m 'this is a test message' \
```

```
–u tcc_test \
–P 12345
```

Figura 26 – Execução de consumo e publicações de mensagens no tópico "devices". À esquerda a publicação de mensgaens e à direita o consumo, bem como o resultado de publicação.



Fonte: Autor

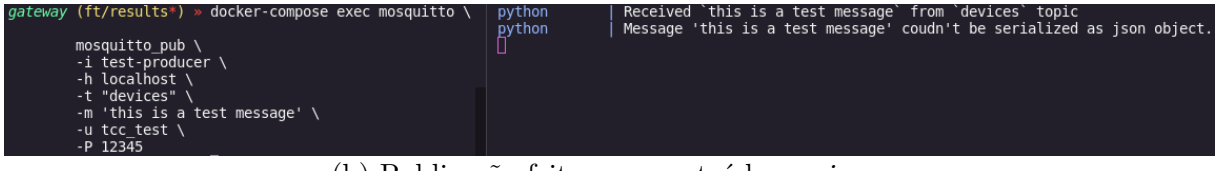
Código 5.3 – Logs de execução de publicação no tópico "devices"

```
1634691102: New client connected from 127.0.0.1:40850 as
            test–consumer (p2, c1, k60, u'tcc_test').
1634691108: New connection from 127.0.0.1:40852 on port 1883.
1634691108: New client connected from 127.0.0.1:40852 as
            test–producer (p2, c1, k60, u'tcc_test').
1634691108: Client test–producer disconnected.
```

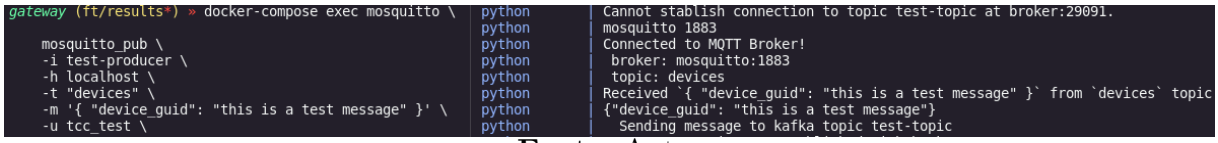
A recepção das mensagens pelo serviço Python foi testada da mesma maneira, isto é, através da subida do serviço, da execução do comando Cód.(5.2) e da verificação dos logs produzidos pelo serviço. Os resultados são apresentados na figura Fig.(27a). A mensagem *Message 'this is a test message' coudn't be serialized as json object.* se deve ao fato que o dado de entrada não é uma mensagem do tipo documento (*json*), o que não acontece quando é enviado a mensagem `{"device_guid": "this is a test message"}` (figura Fig.(27b)).

Figura 27 – Integração de servidor MQTT e Python. À esquerda as publicações e à direita os logs.

(a) Consumo do tópico "devices" pelo serviço Python.



(b) Publicação feita com conteúdo em *json*.



Fonte: Autor

É possível notar, a partir das figuras Fig.(27a) e Fig.(27b), que através da combinação de um servidor de protocolo MQTT e de um *script* Python, operações lógicas são possíveis de serem feitas sobre as mensagens recepcionadas dos sensores em tempo real. De fato, é possível, através do arquivo `requirements.txt`, definir qualquer pacote de Python que ele será instalado no momento de construção da mensagem. Cautela deve ser tomada na escolha de tais pacotes dado que existem fatores, como capacidade de processamento e armazenamento de dados, que podem impactar o desempenho em geral. Ainda, a qualidade de código possui gigante influência sobre a performance do serviço.

5.1.2 Regional

Da mesma forma que para os servidores MQTT, o *cluster* Kafka teve sua integração testada primeiramente através da produção de mensagens diretamente em seu servidor (isolado dos *gateways*) e então através de mensagens submetidas aos tópicos MQTT (integrado aos *gateways*.) Após a subida das imagens, o consumo e a produção de mensagens diretamente no servidor se deram pelos comandos Cód.(5.4) e Cód.(5.5), respectivamente, com o resultado evidenciado pela imagem Fig.(28).

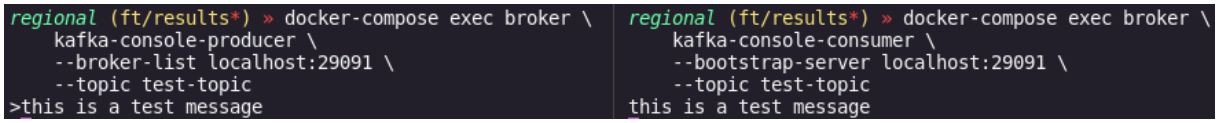
Código 5.4 – Comando de consumo de tópicos Kafka

```
docker-compose exec broker \
    kafka-console-consumer \
    --bootstrap-server localhost:29091 \
    --topic test-topic
```

Código 5.5 – Comando de produção em tópicos Kafka

```
docker-compose exec broker \
    kafka-console-producer \
    --broker-list localhost:29091 \
    --topic test-topic
```

Figura 28 – Execução de consumo e publicações de mensagens no tópico "test-topic", Kafka. À esquerda a publicação e à direita o consumo.



Fonte: Autor

Com o consumo apontando ao mesmo endereço, foi realizado um teste integrado ao *gateway* por meio do comando Cód.(5.2), aqui com o conteúdo já em *json*. O resultado foi semelhante, e está apresentado na figura Fig.(29). Ainda, utilizando de um processo

ConsumeKafka, foi possível consumir a mensagem através do NiFi, como demonstra a figura Fig.(30).

Figura 29 – Teste de integração do servidor MQTT ao *broker* Kafka.

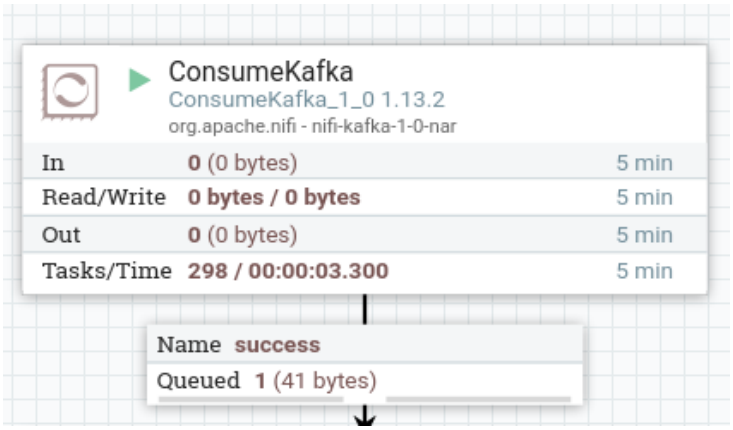
```
gateway (ft/results*) » docker-compose exec mosquitto \

    mosquitto_pub \
    -i test-producer \
    -h localhost \
    -t "devices" \
    -m '{ "device_guid": "this is a test message" }' \
    -u tcc_test \
    -P 12345
```

```
regional (ft/results*) » docker-compose exec broker \
    kafka-console-consumer \
    --bootstrap-server localhost:29091 \
    --topic test-topic
{"device_guid": "this is a test message"}
```

Fonte: Autor

Figura 30 – Consumo de mensagens publicadas em *broker* Kafka usando NiFi.

(a) Processo de consumo Kafka.



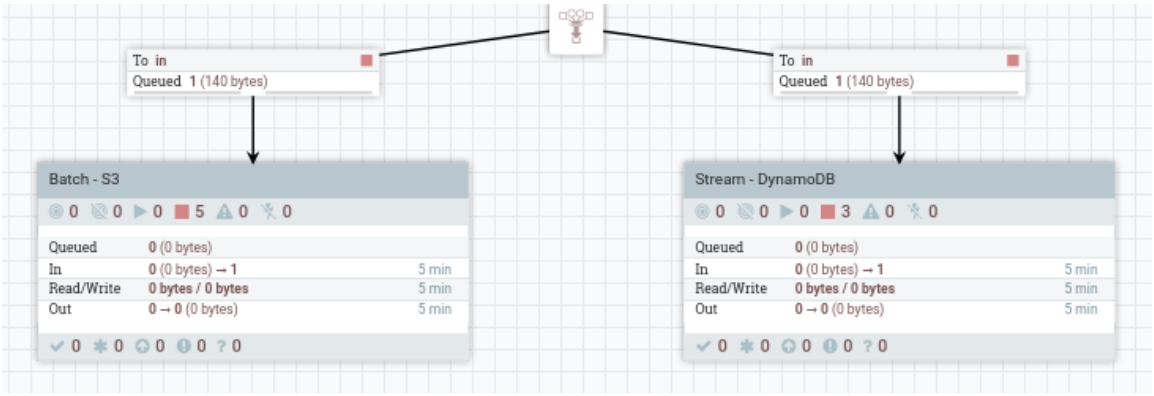
(b) Conteúdo da mensagem.

Fonte: Autor

5.1.3 Nuvem

Para testar a ingestão nos serviços AWS S3 e AWS DynamoDB, foi utilizado o modelo de conteúdo presente no código Cód.(4.1) atualizando o campo de *timestamp* para o momento da requisição. A figura Fig.(31) demonstra a entrada da mensagem nos processos de ingestão dos serviços. As figuras Fig.(32a), Fig.(32b) e Fig.(32c) demonstram o arquivo chegando no S3, ao passo que na figura Fig.(33a) e Fig.(33b) é apresentado a chegada dos documentos no DynamoDB.

Figura 31 – Chegada de mensagens nos processos de ingestão do S3 e DynamoDB.



Fonte: Autor

5.2 *Cluster* Kafka e replicação de mensagens

Assim como discutido na seção de Metodologia (Sec.(4)), a etapa específica teve na camada regional a implementação de um *cluster* kafka constituído de três *brokers*, onde todos apontam ao orquestrador Zookeeper pela porta 2181. Após subida dos serviços Docker, a verificação dos *brokers* ativos foi feito através do comando apresentado no código Cód.(5.6), a qual retornou o log de execução abaixo. É possível notar que os *ids* de *brokers* ativos são respectivos aos valores que foram definidos de *id* para cada um dos *brokers* no arquivo `docker-compose.yml`. Os detalhes de cada um deles pôde ser acessado diretamente pelo orquestrador Zookeeper através do comando `zookeeper-shell localhost:2181 get /brokers/ids/${id}` (`${id}` o *id* do *broker*), demonstrado na figura Fig.(34).

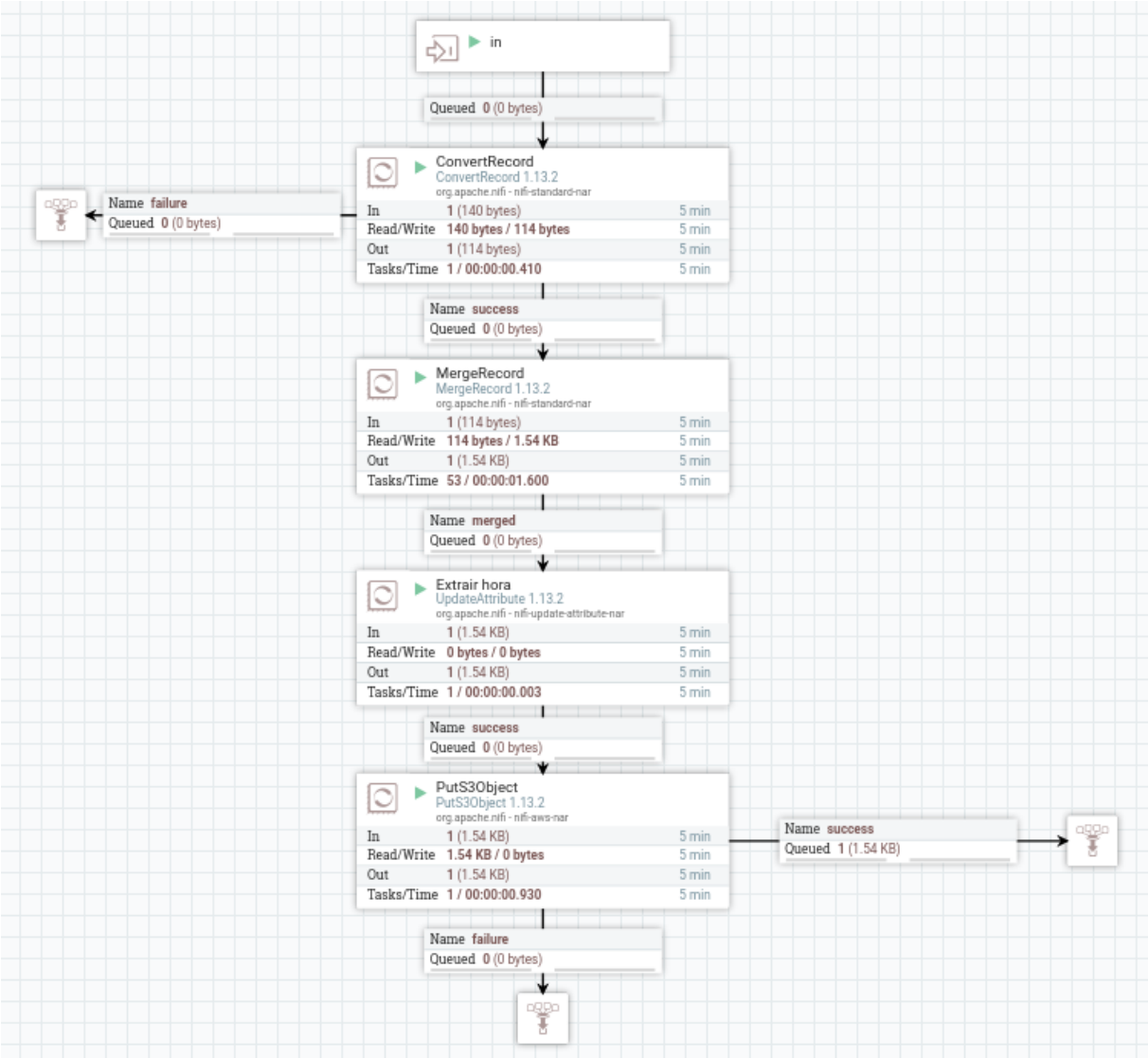
Código 5.6 – Comando de listagem de *ids* de *brokers* ativos no *cluster* Kafka.

```
$ docker-compose exec zookeeper \
    zookeeper-shell \
    localhost:2181 \
    ls /brokers/ids

Connecting to localhost:2181

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
    [1, 2, 3]
```

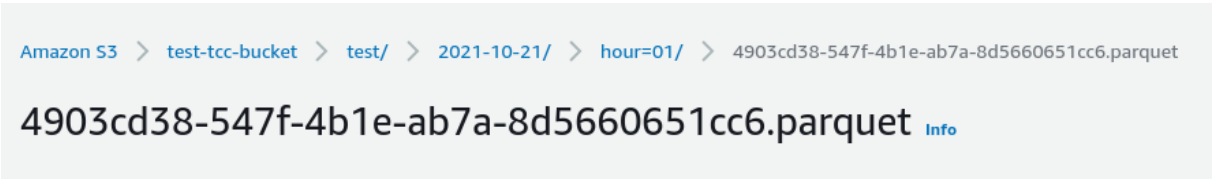
Com a subida do *cluster* Kafka, foi possível criar os tópicos da etapa específica com o fator de replicação e particionamento configurados, respectivamente, como 2 e 10. Com o comando `kafka-topics -describe -zookeeper zookeeper:2181 -topic`

Figura 32 – Ingestão de mensagens no S3.

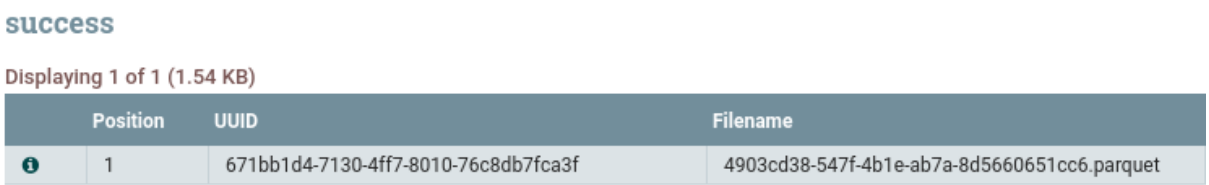
(a) Ingestão de dados no S3. Nota-se que o status da execução é de sucesso.



(b) Nome e caminho de arquivo no S3.



(c) Nome de arquivo no NiFi.

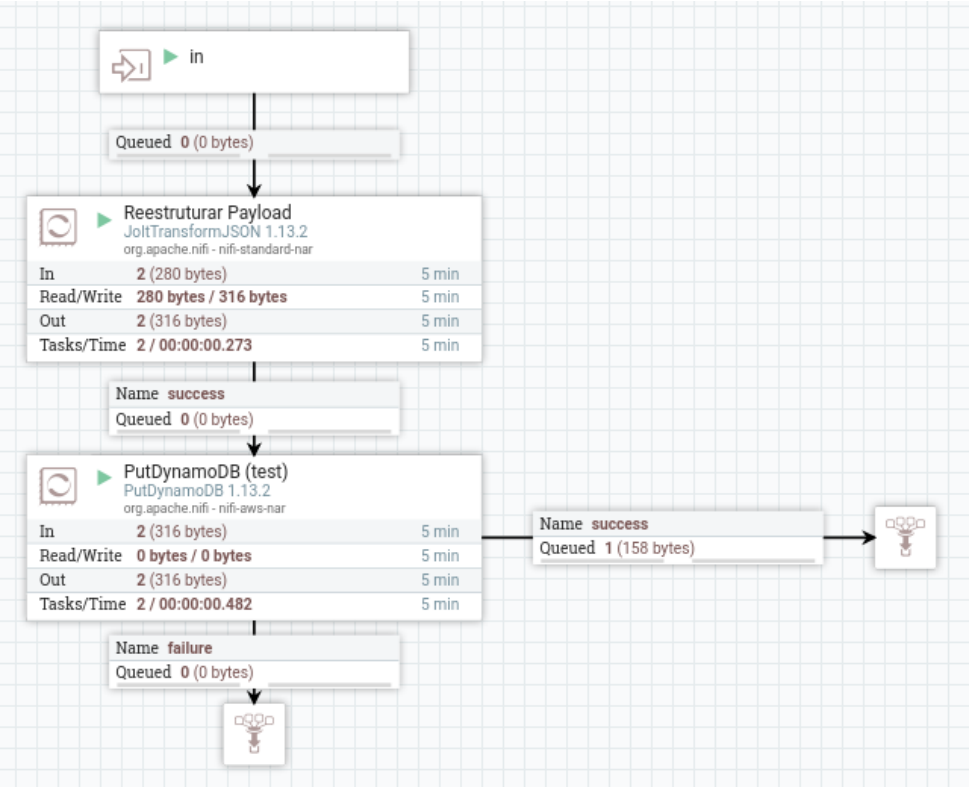


Fonte: Autor

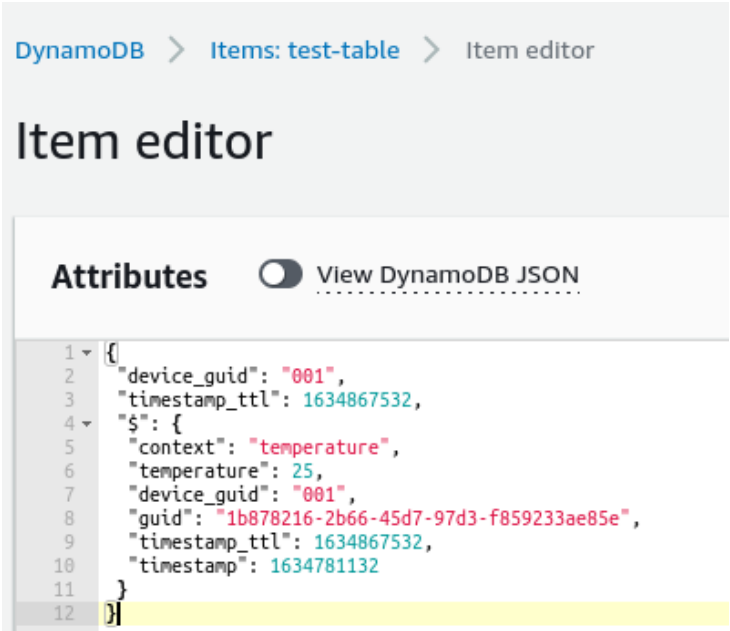
`${nome_topico}` (${nome_topico} sendo o nome do tópico) foi possível extrair as infor-

Figura 33 – Ingestão de mensagens no DynamoDB.

(a) Execução de ingestão de mensagem no DynamoDB com sucesso.



(b) Documento na tabela `test-table`.



Fonte: Autor

mações de particionamento, replicação e qual é o *broker* líder da partição, assim como mostra a figura Fig.(35) para o tópico `devices-temperature`.

Como padrão da imagem Docker utilizada neste projeto (*confluentinc/cp-kafka:6.2.0*), todas as mensagens registradas nos tópicos ficam sobre o diretório `/var/lib/kafka/data/`.

Figura 34 – Extração de informações de *brokers* Kafka ativos no orquestrador Zookeeper.

```
sh-4.4$ zookeeper-shell localhost:2181 get /brokers/ids/1
Connecting to localhost:2181

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
{"features":{},"listener_security_protocol_map":{"LISTENER_DOCKER_INTERNAL":"PLAINTEXT","LIST
ENER_DOCKER_EXTERNAL":"PLAINTEXT"},"endpoints":["LISTENER_DOCKER_INTERNAL://broker-a:29091","
LISTENER_DOCKER_EXTERNAL://127.0.0.1:9091"],"jmx_port":-1,"port":29091,"host":"broker-a","ver
sion":5,"timestamp":"1634862108138"}
sh-4.4$ zookeeper-shell localhost:2181 get /brokers/ids/2
Connecting to localhost:2181

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
{"features":{},"listener_security_protocol_map":{"LISTENER_DOCKER_INTERNAL":"PLAINTEXT","LIST
ENER_DOCKER_EXTERNAL":"PLAINTEXT"},"endpoints":["LISTENER_DOCKER_INTERNAL://broker-b:29092","
LISTENER_DOCKER_EXTERNAL://127.0.0.1:9092"],"jmx_port":-1,"port":29092,"host":"broker-b","ver
sion":5,"timestamp":"1634862108945"}
sh-4.4$ zookeeper-shell localhost:2181 get /brokers/ids/3
Connecting to localhost:2181

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
{"features":{},"listener_security_protocol_map":{"LISTENER_DOCKER_INTERNAL":"PLAINTEXT","LIST
ENER_DOCKER_EXTERNAL":"PLAINTEXT"},"endpoints":["LISTENER_DOCKER_INTERNAL://broker-c:29093","
LISTENER_DOCKER_EXTERNAL://127.0.0.1:9093"],"jmx_port":-1,"port":29093,"host":"broker-c","ver
sion":5,"timestamp":"1634862108946"}
```

Fonte: Autor

Figura 35 – Detalhamento de tópico `devices-temperature` em *cluster* Kafka.

```
sh-4.4$ kafka-topics --describe --zookeeper zookeeper:2181 --topic devices-temperature
Topic: devices-temperature      TopicId: ovjflaTtSWeTMwN_aZOf-g PartitionCount: 10      ReplicationFactor: 2    Configs:
        Topic: devices-temperature      Partition: 0    Leader: 1       Replicas: 1,3   Isr: 1,3
        Topic: devices-temperature      Partition: 1    Leader: 2       Replicas: 2,1   Isr: 2,1
        Topic: devices-temperature      Partition: 2    Leader: 3       Replicas: 3,2   Isr: 3,2
        Topic: devices-temperature      Partition: 3    Leader: 1       Replicas: 1,2   Isr: 1,2
        Topic: devices-temperature      Partition: 4    Leader: 2       Replicas: 2,3   Isr: 2,3
        Topic: devices-temperature      Partition: 5    Leader: 3       Replicas: 3,1   Isr: 3,1
        Topic: devices-temperature      Partition: 6    Leader: 1       Replicas: 1,3   Isr: 1,3
        Topic: devices-temperature      Partition: 7    Leader: 2       Replicas: 2,1   Isr: 2,1
        Topic: devices-temperature      Partition: 8    Leader: 3       Replicas: 3,2   Isr: 3,2
        Topic: devices-temperature      Partition: 9    Leader: 1       Replicas: 1,2   Isr: 1,2
```

Fonte: Autor

Neste diretório, como cada tópico criado teve um parâmetro de particionamento igual a 10, foram criados 10 sub-diretórios de particionamento para cada tópico, o que acontece para cada *broker* ativo no *cluster*. A figura Fig.(36) mostra, para o *broker* `a`, os sub-diretórios criados como resultado do particionamento dos tópicos.

Figura 36 – Particionamento dos sub-diretórios, no *broker* `a`, para cada um dos tópicos criados.

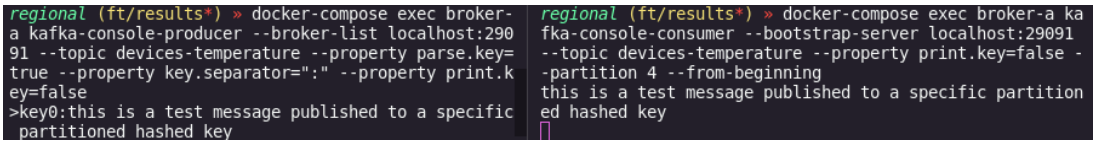
```
sh-4.4$ ls /var/lib/kafka/data/
cleaner-offset-checkpoint  devices-ammonia-7  devices-dump-3  devices-humidity-0  devices-humidity-8   devices-luminosity-5   devices-temperature-3  log-start-offset-checkpoint
devices-ammonia-1          devices-ammonia-8  devices-dump-5  devices-humidity-2  devices-humidity-9   devices-luminosity-6   devices-temperature-5  meta.properties
devices-ammonia-2          devices-ammonia-9  devices-dump-7  devices-humidity-3  devices-luminosity-0 devices-luminosity-7   devices-temperature-6  recovery-point-offset-checkpoint
devices-ammonia-3          devices-dump-1     devices-dump-8  devices-humidity-4  devices-luminosity-1 devices-temperature-0  devices-temperature-7  replication-offset-checkpoint
devices-ammonia-5          devices-dump-2     devices-dump-9  devices-humidity-6  devices-luminosity-4 devices-temperature-1  devices-temperature-9
```

Fonte: Autor

Para testar a replicação das mensagens, foi realizado um teste de publicação especificando uma chave (*key*) de particionamento. Esta chave, através de um algoritmo *round-robin*, elege qual particionamento do tópico as mensagens deverão ser publicadas. No exemplo da figura Fig.(37), a chave `key0` foi eleita para ser publicada na partição

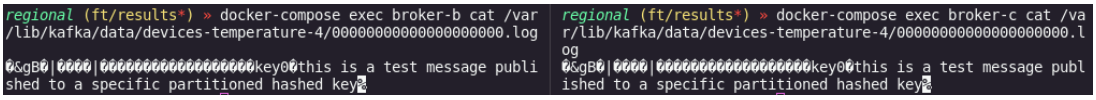
4 do tópico `devices-temperature`. Pela figura Fig.(35), esta partição está replicada nos *brokers* `b` e `c` (*id* 2 e 3, respectivamente), e portanto qualquer mensagen publicada com esta chave será salva nos arquivos `/var/lib/kafka/data/devices-temperature-4/00000000000000000000.log` dos *brokers* mencionados. A figura Fig.(38) demonstra justamente isso: a mensagem publicada na figura Fig.(37) foi replicada para os *brokers* `b` e `c` na partição 4 do tópico `devices-temperature`.

Figura 37 – Publicação de mensagens no tópico `devices-temperature` com chave de particionamento.



Fonte: Autor

Figura 38 – Conteúdo dos arquivos de *logs* nos *brokers* `b` (esquerda) e `c` (direita) para o tópico `devices-temperature`, partição 4.



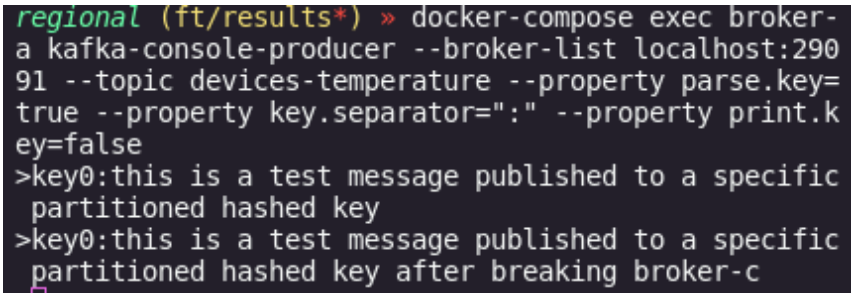
Fonte: Autor

Com a replicação de mensagens, o consumo delas pôde ser feito em cenários onde um dos *brokers* se torna inativo. Por exemplo, ao desligar o *broker* `c`, as mensagens publicadas ao tópico `devices-temperature` na partição 4 continuam passíveis de serem consumidas, pois elas continuam sendo escritas no *broker* `b`. A figura Fig.(39) demonstra este cenário, onde é possível verificar que as mensagens publicadas ao tópico continuam sendo gravadas no *broker* `b`. Com isso as mensagens publicadas ao tópico continuam disponíveis para consumo mesmo em cenários de quedas e pânes de nódos, garantindo maior resiliência à arquitetura.

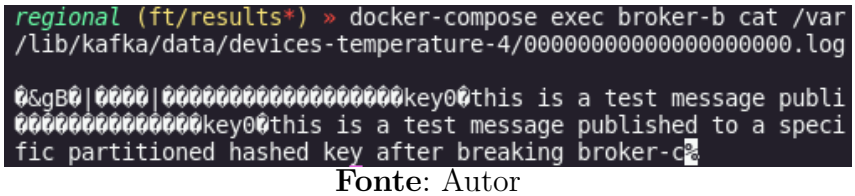
É válido destacar que a escolha apropriada da quantidade de *brokers*, do particionamento de cada tópico, de nódos de orquestramento (instâncias Zookeeper) e do fator de replicação dependem diretamente do cenário de uso, publicação e consumo dos tópicos. Quanto maior a quantidade de *clients* realizando a publicação e o consumo dos tópicos, preferencialmente maior deverá ser a quantidade de partições de cada tópico. Ainda, dependendo da criticidade e do valor de negócio de cada mensagem, a arquitetura deverá comportar mais *brokers* no *cluster* a fim de se garantir maior resiliência. Como este exemplo se trata de uma prova de conceito, o valor arbitrário de 10 partições por tópico e 3 *brokers* no *cluster* foi escolhido. Para adequar esta escolha a um caso aplicado, um levantamento de métricas dos fatores mencionados deve ser feito.

Figura 39 – Publicação e escrita de mensagens no cenário de queda do *broker* `c`.

(a) Publicação de mensagem.



(b) Mensagens publicadas no tópico `devices-temperature`, partição 4, após desligar o *broker* `c`.



Fonte: Autor

5.3 Consumo de dados da nuvem

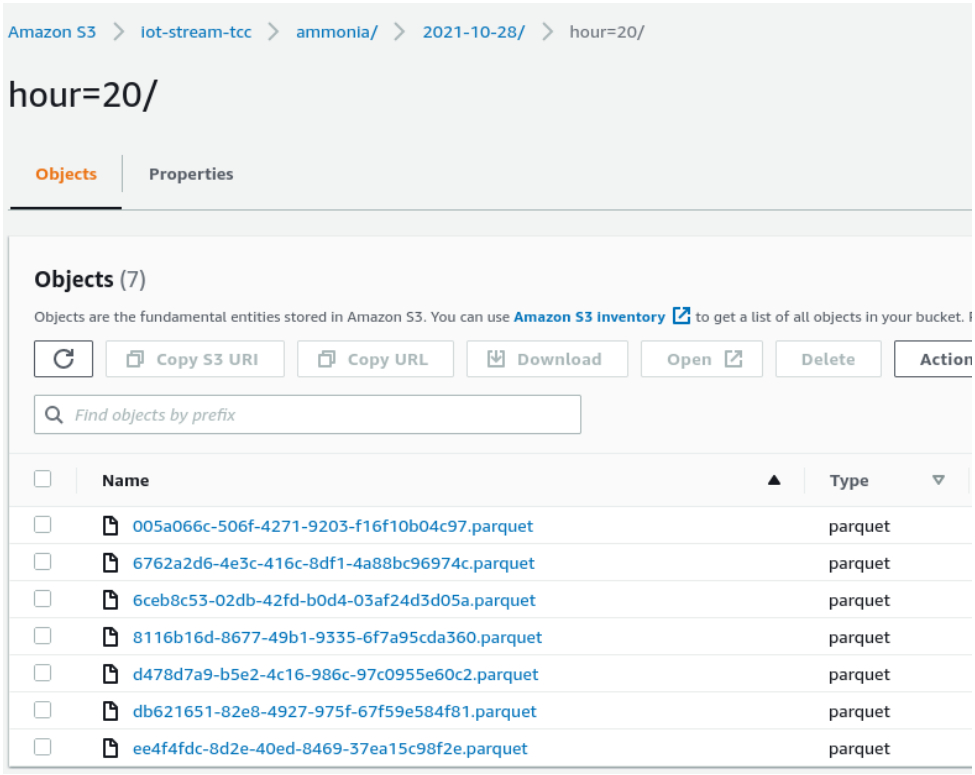
Na seção Metodologia Sec.(4), foram apresentados as especificações da arquitetura do cenário específico, bem como os dados gerados pela fonte que seriam consumidos e replicados à nuvem. A partir da modelagem detalhada na tabela Tab.(2), foi usado o software *IoT-data-simulator* para a geração dos dados e, conforme o detalhamento da arquitetura, dos serviços AWS S3, DynamoDB e RDS para o armazenamento em nuvem de tais dados para, respectivamente, batch, stream e dados estruturados.

Os dados em batch foram armazenados em um serviço AWS S3, particionados por data e hora de armazenamento de carga. Todos os arquivos foram convertidos em formato `.parquet` para otimização de volumetria de armazenamento. As figuras Fig.(40) demonstram um exemplo de carga executado no dia 28 de Outubro de 2021, às 20:00 horas. Cada arquivo possui em média de 3.5 KB, o que é respectivo à aproximadamente 5 minutos de agregação de registros conforme o tempo de disparo do simulador. Este volume pode ser ajustado dependendo da necessidade para, por exemplo, agregar mais ou menos dados, ou agregá-los em uma janela de tempo maior, de forma que a escolha mais adaptada deve ser feito com base no cenário de aplicação.

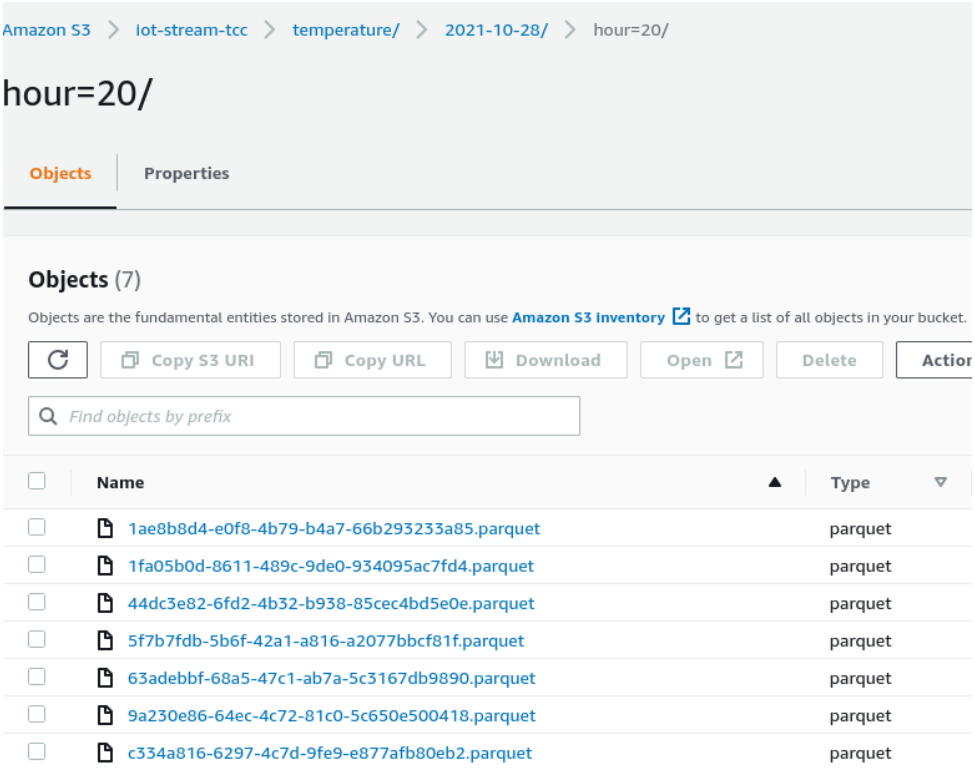
Os dados armazenados em forma de *stream* foram gravados em tabelas no DynamoDB separadas por tipo de evento recebido. Todos foram armazenados com tempo limite de expiração de um dia, de forma que durante este tempo eles se mantiveram disponíveis para consumo. As figuras Fig.(41) mostram o histórico de execução de requisições de escrita nas tabelas DynamoDB para a mesma carga executada no dia 28 de Outubro de 2021, às 20:00 horas.

Figura 40 – Armazenamento de *batch* de dados no S3, particionados por data e hora.

(a) Dados de amônia.



(b) Dados de temperatura.

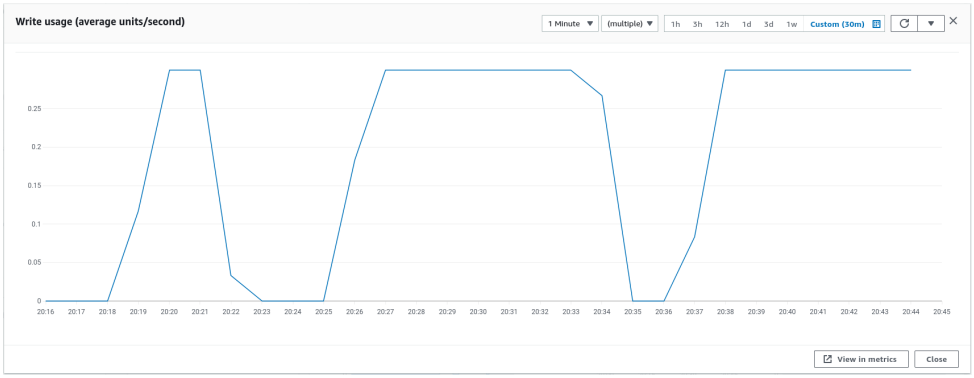


Fonte: Autor

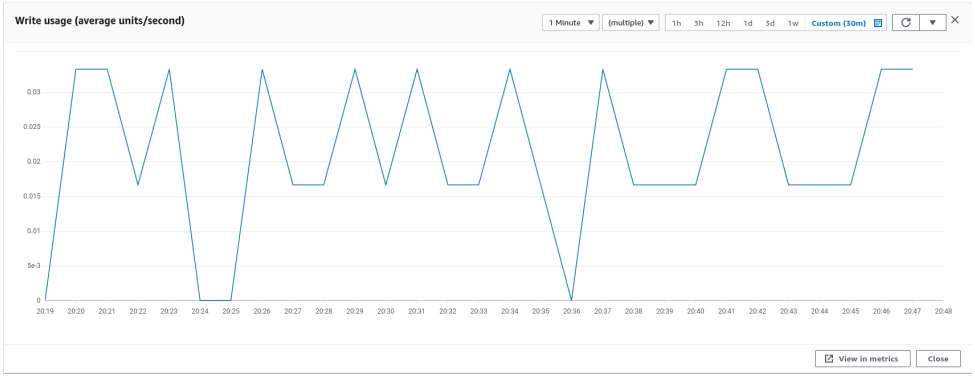
Através de um processo *PutDatabaseRecord*, no Nifi, os registros também foram ingeridos em um banco de dados Postgres, através do uso do serviço AWS RDS. Cada

Figura 41 – Histórico de requisições de escrita feito em cima das tabelas DynamoDB.

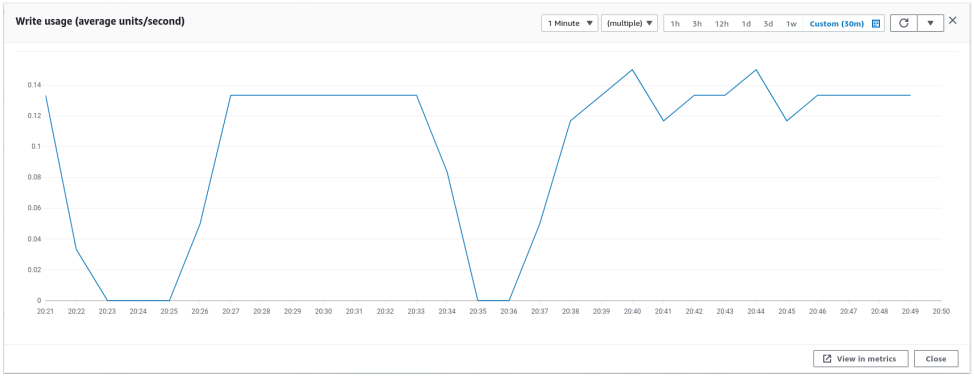
(a) `iot-stream-ammonia`.



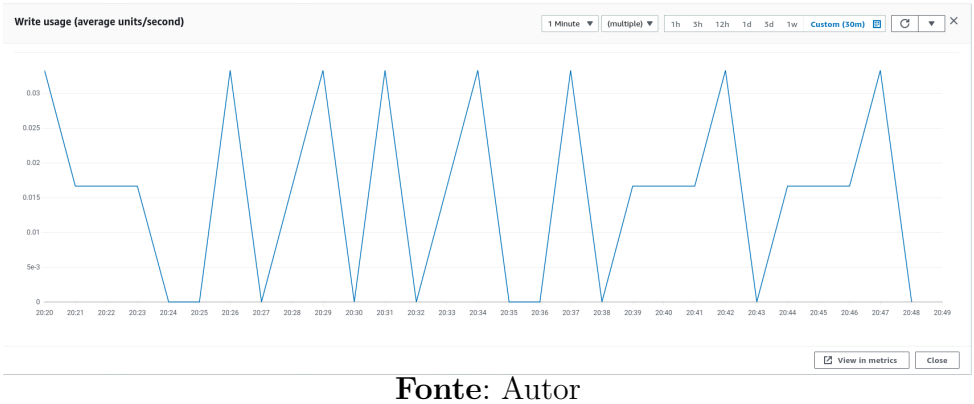
(b) `iot-stream-humidity`.



(c) `iot-stream-temperature`.



(d) `iot-stream-luminosity`.



Fonte: Autor

tipo de evento foi inserido nas suas tabelas respectivas, criando assim uma base contendo todos os eventos disparados pelos sensores simulados.

Os dados contidos no banco de dados Postgres foram, através de uma conexão JDBC, extraídos para o Google Data Studio, no qual foi possível criar uma tela de *Dashboard* para cada tipo de métrica, contendo, para cada estação (A e B) uma evolução temporal das medidas (métricas de cada dispositivo segregados por cores distintas), uma tabela com os últimos registros da base e um ponteiro de média. As telas de dashboard estão apresentadas nas figuras Fig.(42), Fig.(45), Fig.(43) e Fig.(44) para os eventos de amônia, umidade, temperatura e luminosidade, respectivamente.

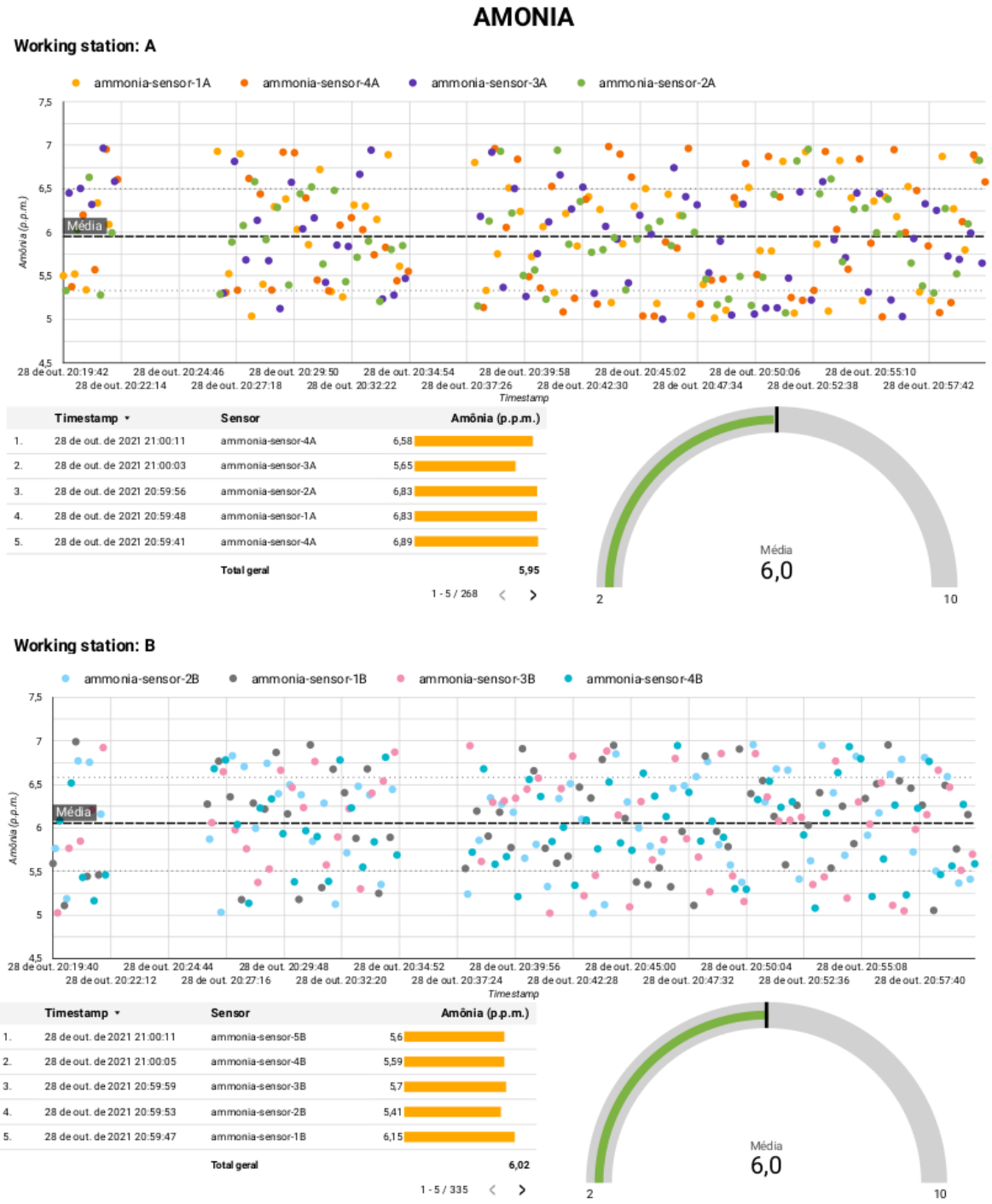
5.4 Discussões

É valido mencionar que o uso dos serviços discutidos nesta seção, bem como os resultados evidenciados, tomam como base um cenário hipotético e de relativamente baixa volumetria, variedade e velocidade de dados. Desta forma, a volumetria de dados disponíveis para consumo são relativamente baixos, e, ainda, não exprimem nenhum resultado com base em métricas reais, dado que os dados são gerados via um simulador.

Porém, assim como foi apontado na seção de Introdução Sec.(1), principalmente nos objetivos do projeto, a arquitetura proposta e testada de acordo com a metodologia discutida tinha como principal finalidade a de servir como uma prova de conceito, isto é, validar a possibilidade de integrar serviços que são já conhecidos amplamente por segurança, escalabilidade e resiliência, para interoperar conforme foram integrados a fim de se provisionar uma solução que fosse capaz de: i) oferecer um nível de processamento já na recepção dos dados dos sensores (*gateways*); ii) recepcionar todos os dados gerados por sensores em uma estação única, implementando uma esteira de dados conforme chegada de eventos (regional); iii) ser capaz de enviar todos os eventos, já validados pela esteira, à nuvem, tanto em tempo real quanto em forma agregada; iv) desenhar a integração de todos os serviços em imagens virtualizadas (Docker) para facilitar a manutenção e a reprodutibilidade; v) garantir que os registros enviados à nuvem pudessem ser, em casos específicos, usados para qualquer aplicação analítica ou de negócio.

De fato, todos os serviços foram implementados em arquivos `docker-compose.yml` respectivo à sua camada específica (regional ou *gateway*), e então integrados através de conexões internas ou externas. Como resultado, todos os eventos foram devidamente validados, tratados e então enviados para serviços de armazenamento situados na nuvem da AWS, de forma que pudessem ser resgatados tanto para finalidades analíticas (banco de dados Postgres e tabelas DynamoDB) quanto para reprocessamento ou de forma agregada (S3). As especificades de volumetria, frequência de requisições, esteiras de tratamento, análise dos dados, quantidade de *gateways* e de *brokers* no *cluster* Kafka devem ser

Figura 42 – Dashboard para os eventos de amônia.



Fonte: Autor

dimensionados de acordo com a aplicação.

É interessante que, com as métricas sendo acessíveis, é possível realizar ações e tomadas de ação com bases nos eventos registrados nas bases de dados. De fato, é possível fazer uso de atuadores para acionar sistemas de controle de temperatura e umidade relativa, bem como notificar quando a concentração de amônia está acima do limite estipulado. Esta implementação não é limitada a recursos robustos de processamento, como o regional,

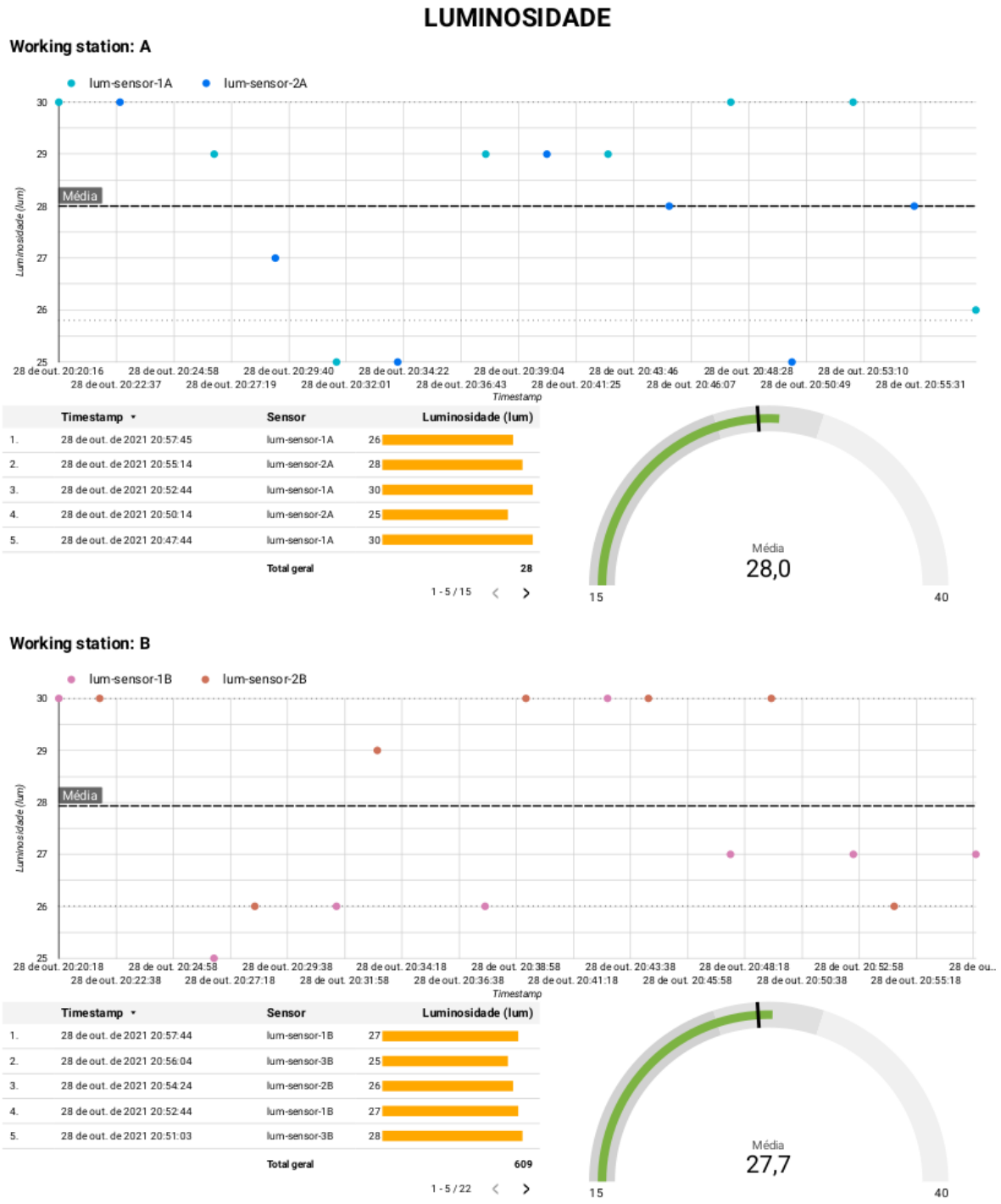
Figura 43 – Dashboard para os eventos de temperatura.

Fonte: Autor

mas também é possível de ser implementado em um nódo de gateway, trazendo, assim, a habilidade de se realizar *fog computing*.

Como possíveis desenvolvimentos e estudos futuros, é interessante trazer a arquitetura proposta para um caso real, visando evidenciar limitações que possam surgir referente a implementação da arquitetura ou se é necessário alterar algum dos componentes. Como

Figura 44 – Dashboard para os eventos de luminosidade.



Fonte: Autor

este projeto visou desenvolver um sistema agnóstico de caso de uso, é possível que surjam pontos específicos e que possam ser agregados ao projeto, dado que o mesmo se encontra de forma aberta no GitHub, para mitigar dificuldades e até mesmo ampliar as possibilidades de uso.

Figura 45 – Dashboard para os eventos de umidade.

Fonte: Autor

6 CONCLUSÃO

Neste estudo foi realizada a integração de uma arquitetura híbrida proposta para casos de sistemas IoT, onde, para cada camada desta proposta, foram integrados softwares *open-source* amplamente conhecidos para provisionar um cenário no qual muitos sensores em diversas estações de monitoramento geram dados de métricas que sejam validados, tratados e então enviados para a nuvem. Entende-se, neste estudo, que cada sítio onde se encontram os sensores configura uma "estação", na qual, para cada uma delas, se encontra um dispositivo de sistema embarcado, denominado de *gateway*, que hospeda um servidor de protocolo MQTT (Mosquitto) e um software para validação de eventos em tempo real (Python). Todos os *gateways* se comunicam à uma estação local e centralizada, chamada aqui de camada regional, onde os eventos são enviados a um *cluster* Kafka e então tratados por uma esteira de dados Nifi. Após tratamento, os dados são disponibilizados na nuvem através de bases específicas que servem finalidades diferentes.

Como estudo de caso, a arquitetura proposta foi usada para simular um cenário de uso de IoT para monitoramento de uma estação de avicultura. Para este caso, o desenvolvimento da integração foi orientado à duas estações, denominadas A e B, cada qual com uma quantidade específica de sensores, integrados a um *cluster* Kafka de três *brokers*, sendo os eventos tratados e enviados à nuvem por uma esteira de dados Nifi. Os dados foram gerados de forma sintética através do software *IoT-data-simulator*, onde, como resultado, foi possível obter um *dashboard* que resumia os valores de cada métrica emulada, bem como resgatar os dados agregados do serviço AWS S3, da base de dados NoSQL DynamoDB e do Postgres SQL.

Com os resultados obtidos, foi possível evidenciar que a presente proposta consegue provisionar cenários de queda de alguns dos serviços (por exemplo, broker Kafka) e o tratamento e validação dos eventos gerados em tempo real. Ainda, a conexão com a camada regional se provou possível após a implementação de duas estações de monitoramento, podendo ser extendido para demais nódos (estações) dependendo do caso de uso. É importante ressaltar que quaisquer especificades podem exigir novos dimensionamentos de recursos, como quantidade de nódos de *gateways*, *brokers* Kafka, agregação de dados na esteira Nifi, e estruturação e tratamentos específicos dos dados.

Como possíveis estudos futuros, sugerimos colocar os métodos e resultados propostos à prova em um cenário real. A arquitetura foi desenhada de forma agnóstica, necessitando ser testada e validada para cada caso de uso. Estudos nesta linha podem evidênciar eventuais melhorias de arquitetura ou de desempenho, o que pode contribuir para a evolução e melhorias futuras.

Referências

AKIDAU, T.; CHERNYAK, S.; LAX, R. *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*. Paperback. O'Reilly Media, 2018. 352 p. ISBN 978-1491983874. Disponível em: <https://lead.to/amazon/com/?op=bt&la=en&cu=usd&key=1491983876>.

ALMEIDA, G. et al. Internet of Things (Iot): Um Cenário Guiado Por Patentes Industriais. *GESTÃO.Org : Revista Eletrônica de Gestão Organizacional*, v. 13, n. Especial, p. 271–281, 2015. ISSN 1679-1827.

AUMONT, O. et al. *Introduction to IoT*. [s.n.], 2018. v. 43. 679–694 p. ISSN 17264189. ISBN 2709910403. Disponível em: <http://www.unil.ch/ssp/page34569.html>.

BAHRI, M. et al. Data stream analysis: Foundations, major tasks and tools. *WIREs Data Mining and Knowledge Discovery*, v. 11, n. 3, p. 1–17, may 2021. ISSN 1942-4787. Disponível em: <https://onlinelibrary.wiley.com/doi/10.1002/widm.1405>.

BELLAVISTA, P.; ZANNI, A. Feasibility of fog computing deployment based on docker containerization over RaspberryPi. *ACM International Conference Proceeding Series*, 2017.

BURNS, B. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2018. ISBN 1491983647.

CATTELL, R. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, v. 39, n. 4, p. 12–27, may 2011. ISSN 0163-5808. Disponível em: <https://dl.acm.org/doi/10.1145/1978915.1978919>.

CHARLES, D. R.; GROOM, C. M.; BRAY, T. S. The effects of temperature on broilers: Interactions between temperature and feeding regime. *British Poultry Science*, v. 22, n. 6, p. 475–481, jan 1981. ISSN 0007-1668. Disponível em: <http://www.tandfonline.com/doi/full/10.1080/00071688108447913>.

CHATTI, S. Using Spark , Kafka and NIFI for Future Generation of ETL in IT Industry. *Journal of Innovation in Information Technology*, v. 3, n. 2, p. 11–14, 2019.

CIRANI, S. et al. The IoT hub: a fog node for seamless management of heterogeneous connected smart objects. In: *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking - Workshops (SECON Workshops)*. IEEE, 2015. p. 1–6. ISBN 978-1-4673-7392-0. Disponível em: <http://ieeexplore.ieee.org/document/7328145/>.

DEBAUCHE, O. et al. Edge Computing and Artificial Intelligence for Real-time Poultry Monitoring. *Procedia Computer Science*, v. 175, n. 2019, p. 534–541, 2020. ISSN 18770509. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1877050920317762>.

DUSIA, A.; YANG, Y.; TAUFER, M. Network Quality of Service in Docker Containers. In: *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015. v. 2015-Octob, p. 527–528. ISBN 978-1-4673-6598-7. ISSN 15525244. Disponível em: <http://ieeexplore.ieee.org/document/7307643/>.

GARG, N. *Apache Kafka*. [S.l.]: Packt Publishing, 2013. ISBN 1782167935.

GODFREY, P. B. *Designing distributed systems for heterogeneity*. [s.n.], 2009. 178 p. ISBN 9781491983645. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-82.pdf>.

GORELIK, A. *The Enterprise Big Data Lake: Delivering the Promise of Big Data and Data Science*. O'Reilly Media, 2019. ISBN 9781491931523. Disponível em: <https://books.google.com.br/books?id=9yKJDwAAQBAJ>.

GUARDO, E. et al. A fog computing-based IoT framework for precision agriculture. *Journal of Internet Technology*, v. 19, n. 5, p. 1401–1411, 2018. ISSN 20794029.

GUEVARA, A. J. d. H.; SILVA, J. L. A. da. INTERNET OF THINGS (IOT) OPPORTUNITIES AND IMPACTS OF WELL-BEING ON CITIZENS AND SOCIETY. *Journal on Innovation and Sustainability RISUS*, v. 10, n. 3, p. 3–16, dec 2019. ISSN 2179-3565. Disponível em: <https://revistas.pucsp.br/risus/article/view/46504>.

HAJIHEYDARI, N.; TALAFIDARYANI, M.; KHABIRI, S. IoT Big Data Value Map. In: *Proceedings of the 2019 the 5th International Conference on e-Society, e-Learning and e-Technologies - ICSLT 2019*. New York, New York, USA: ACM Press, 2019. p. 98–103. ISBN 9781450362351. Disponível em: <http://dl.acm.org/citation.cfm?doid=3312714.3312728>.

HUMANE FARM ANIMAL CARE. *HFAC Standards for Chickens*. Middleburg, VA, 2014.

IBA-GROUP-IT. *IoT-data-simulator*. [S.l.]: GitHub, 2020. <https://github.com/IBA-Group-IT/IoT-data-simulator>.

ISAH, H.; ZULKERNINE, F. A Scalable and Robust Framework for Data Stream Ingestion. In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018. p. 2900–2905. ISBN 978-1-5386-5035-6. Disponível em: <https://ieeexplore.ieee.org/document/8622360/>.

JADEJA, Y.; MODI, K. Cloud computing - concepts, architecture and challenges. In: *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)*. IEEE, 2012. p. 877–880. ISBN 978-1-4673-0212-8. Disponível em: <http://ieeexplore.ieee.org/document/6203873/>.

JANGLA, K. *Accelerating Development Velocity Using Docker*. Berkeley, CA: Apress, 2018. ISBN 978-1-4842-3935-3. Disponível em: <http://link.springer.com/10.1007/978-1-4842-3936-0>.

Jing Han et al. Survey on NoSQL database. In: *2011 6th International Conference on Pervasive Computing and Applications*. IEEE, 2011. p. 363–366. ISBN 978-1-4577-0208-2. Disponível em: <http://ieeexplore.ieee.org/document/6106531/>.

KALLA, A.; PROMBAGE, P.; LIYANAGE, M. Introduction to IoT. In: *IoT Security*. Wiley, 2020. p. 1–25. Disponível em: <https://onlinelibrary.wiley.com/doi/10.1002/9781119527978.ch1>.

KROPP, A.; TORRE, R. Docker: containerize your application. In: *Computing in Communication Networks*. Elsevier, 2020. p. 231–244. ISBN 9780128204887. Disponível em: <https://doi.org/10.1016/B978-0-12-820488-7.00026-8https://linkinghub.elsevier.com/retrieve/pii/B9780128204887000268>.

LASHARI, M. H. et al. IoT Based Poultry Environment Monitoring System. In: *2018 IEEE International Conference on Internet of Things and Intelligence System (IOTAIS)*. IEEE, 2018. p. 1–5. ISBN 978-1-5386-7358-4. Disponível em: <https://ieeexplore.ieee.org/document/8600837/>.

LEE, I.; LEE, K. The Internet of Things (IoT): Applications, investments, and challenges for enterprises. *Business Horizons*, "Kelley School of Business, Indiana University", v. 58, n. 4, p. 431–440, jul 2015. ISSN 00076813. Disponível em: <http://dx.doi.org/10.1016/j.bushor.2015.03.008https://linkinghub.elsevier.com/retrieve/pii/S0007681315000373>.

LIGHT, R. A. Mosquitto: server and client implementation of the MQTT protocol. *The Journal of Open Source Software*, v. 2, n. 13, p. 265, may 2017. ISSN 2475-9066. Disponível em: <http://joss.theoj.org/papers/10.21105/joss.00265>.

LIYANAGE, M. et al. *IoT Security: Advances in Authentication*. Wiley, 2020. ISBN 9781119527923. Disponível em: <https://books.google.com.br/books?id=Bk6_DwAAQBAJ>.

MALASKA, T.; SEIDMAN, J. *Foundations for Architecting Data Solutions: Managing Successful Data Projects*. O'Reilly Media, 2018. ISBN 9781492038696. Disponível em: <https://books.google.com.br/books?id=IA1rDwAAQBAJ>.

MALEK, Y. N. et al. On the use of IoT and Big Data Technologies for Real-time Monitoring and Data Processing. *Procedia Computer Science*, Elsevier B.V., v. 113, p. 429–434, 2017. ISSN 18770509. Disponível em: <http://dx.doi.org/10.1016/j.procs.2017.08.281https://linkinghub.elsevier.com/retrieve/pii/S1877050917316903>.

MARJANI, M. et al. Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access*, v. 5, n. c, p. 5247–5261, 2017. ISSN 2169-3536. Disponível em: <http://ieeexplore.ieee.org/document/7888916/>.

NARGESIAN, F. et al. Data lake management. *Proceedings of the VLDB Endowment*, v. 12, n. 12, p. 1986–1989, aug 2019. ISSN 2150-8097. Disponível em: <https://dl.acm.org/doi/10.14778/3352063.3352116>.

ODUN-AYO, I. et al. Cloud Computing Architecture: A Critical Analysis. *Proceedings of the 2018 18th International Conference on Computational Science and Its Applications, ICCSA 2018*, IEEE, p. 1–7, 2018.

PAHL, C.; XIONG, H.; WALSHE, R. A Comparison of On-Premise to Cloud Migration Approaches. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. [s.n.], 2013. v. 8135 LNCS, p. 212–226. ISBN 9783642406508. Disponível em: <http://link.springer.com/10.1007/978-3-642-40651-5_18>.

PETROV, A. *Database Internals: A Deep Dive into How Distributed Data Systems Work*. O'Reilly Media, 2019. ISBN 9781492040316. Disponível em: <https://books.google.com.br/books?id=-l2vDwAAQBAJ>.

RAJ, A. A. G.; JAYANTHI, J. G. IoT-based real-time poultry monitoring and health status identification. In: *2018 11th International Symposium on Mechatronics and its Applications (ISMA)*. IEEE, 2018. p. 1–7. ISBN 978-1-5386-1078-7. Disponível em: <http://ieeexplore.ieee.org/document/8330139/>.

Shanzhi Chen et al. A Vision of IoT: Applications, Challenges, and Opportunities With China Perspective. *IEEE Internet of Things Journal*, v. 1, n. 4, p. 349–359, aug 2014. ISSN 2327-4662. Disponível em: <http://ieeexplore.ieee.org/document/6851114/>.

SILVA, I. L. d. O. da; JESUS, D. S. de. O IMPACTO DO AVANÇO DA INTERNET DAS COISAS NO BRASIL / EL IMPACTO DEL AVANCE DEL INTERNET DE LAS COSAS EN BRASIL. *Brazilian Journal of Development*, v. 6, n. 12, p. 101749–101758, 2020. ISSN 25258761. Disponível em: <https://www.brazilianjournals.com/index.php/BRJD/article/view/22108/17653>.

SINGH, M. et al. Artificial Intelligence and IoT based Monitoring of Poultry Health: A Review. In: *2020 IEEE International Conference on Communication, Networks and Satellite (Comnetsat)*. IEEE, 2020. p. 50–54. ISBN 978-0-7381-2517-6. Disponível em: <https://ieeexplore.ieee.org/document/9328930/>.

TALAVERA, J. M. et al. Review of IoT applications in agro-industrial and environmental fields. *Computers and Electronics in Agriculture*, v. 142, n. 118, p. 283–297, nov 2017. ISSN 01681699. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0168169917304155>.

TOKAREVA, M. S.; VISHNEVSKIY, K. O. The impact of the Internet of Things technologies on economy. *Business Informatics*, v. 3, n. 3, p. 62–78, 2018.

VASSILIADIS, P.; SIMITSIS, A.; SKIADOPOULOS, S. Conceptual modeling for ETL processes. In: *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP - DOLAP '02*. New York, New York, USA: ACM Press, 2002. p. 14–21. ISBN 1581135904. Disponível em: <http://portal.acm.org/citation.cfm?doid=583890.583893>.

WATHES, C. M.; KRISTENSEN, H. H. Ammonia and poultry welfare : a review. *World's Poultry Science Journal*, v. 56, n. September, p. 236–245, 2000.

WU, H.; SHANG, Z.; WOLTER, K. Learning to Reliably Deliver Streaming Data with Apache Kafka. In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020. p. 564–571. ISBN 978-1-7281-5809-9. Disponível em: <https://ieeexplore.ieee.org/document/9153457/>.

XIONG, Y. et al. Effects of relative humidity on animal health and welfare. *Journal of Integrative Agriculture*, CAAS. Publishing services by Elsevier B.V, v. 16, n. 8, p. 1653–1658, aug 2017. ISSN 20953119. Disponível em: <http://dx.doi.org/10.1016/S2095-3119(16)61532-0https://linkinghub.elsevier.com/retrieve/pii/S2095311916615320>.

YAMAMOTO, I. H. C. *tcc_proj*. [S.l.]: GitHub, 2021. <https://github.com/igor-yamamoto/tcc_proj>.