Programa de Pós-Graduação em

Computação Aplicada

Doutorado Acadêmico

Márcio Miguel Gomes

STEAM: Um modelo para processamento de eventos e enriquecimento de fluxos de dados IoT na borda da rede

São Leopoldo, 2022

Márcio Miguel Gomes

STEAM: UM MODELO PARA PROCESSAMENTO DE EVENTOS E ENRIQUECIMENTO DE FLUXOS DE DADOS IOT NA BORDA DA REDE

Tese apresentada como requisito parcial para a obtenção do título de Doutor pelo Programa de Pós-Graduação em Computação Aplicada da Universidade do Vale do Rio dos Sinos — UNISINOS

Orientador:
Prof. Dr. Rodrigo da Rosa Righi

Coorientador:
Prof. Dr. Cristiano André da Costa

São Leopoldo
2022

G633s Gomes, Márcio Miguel.
STEAM : um modelo para processamento de eventos e enriquecimento de fluxos de dados IoT na borda da rede / por Márcio Miguel Gomes. – 2022.
116 f. : il. ; 30 cm.

Tese (doutorado) — Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, São Leopoldo, RS, 2022.
Orientador: Dr. Rodrigo da Rosa Righi.
Coorientador: Dr. Cristiano André da Costa.

1. Internet das coisas. 2. Computação de borda. 3. Análise de dados. 4. Enriquecimento de dados. 5. Processamento de eventos complexos. 6. Framework. I. Título.

CDU: 004.738.5

Catalogação na Publicação (CIP):
Bibliotecário Alessandro Dietrich - CRB 10/2338

UNISINOS

UNIVERSIDADE DO VALE DO RIO DOS SINOS
Ciências Exatas e Tecnológicas
Programa de Pós-Graduação em Computação Aplicada – PPGCA

Márcio Miguel Gomes

STEAM: Um modelo para processamento de eventos e enriquecimento de fluxos de dados IoT na borda da rede

Tese apresentada como requisito parcial para a obtenção do título de Doutor pelo Programa de Pós-Graduação em Computação Aplicada da Universidade do Vale do Rio dos Sinos - UNISINOS

Aprovado em _23 / 02 / 2022_

BANCA EXAMINADORA

Prof. Dr. Mário Antônio Ribeiro Dantas - UFJF

Prof. Dr. Dalvan Griebler - PUCRS

Prof. Dr. Rafael Kunst - UNISINOS

Prof. Dr. Cristiano André da Costa - UNISINOS

Prof. Dr. Rodrigo da Rosa Righi (Orientador)

Visto e permitida a impressão

São Leopoldo, _28_ de _fevereiro_ de _2022_

Prof. Dr. Rafael Kunst
Coordenador PPG em Computação Aplicada

RESUMO

CONTEXTO: A Internet das Coisas é um ambiente em franca expansão, em que objetos, animais ou pessoas estão equipados com os mais variados sensores e possuem a capacidade de transferir automaticamente seus dados através de uma rede. Por sua natureza limitada, os sensores e dispositivos de borda normalmente apenas repassam os dados coletados para serem processados por sistemas centralizados na nuvem, e em muitos casos, aguardam o retorno de um resultado. Esta transferência do processamento local para remoto resulta em questões críticas como perda de conexão, alto tempo de resposta, sobrecarga no sistema computacional, além de necessitar uma estrutura robusta e escalável para comunicação dos dados e processamento centralizado. **OBJETIVO**: Assim, foram identificados dois grandes desafios. Primeiro, idealizar um modelo capaz de trazer o processamento de dados da nuvem para a borda de rede. Segundo, implementar uma solução que atenda as restrições e a heterogeneidade do ambiente IoT, tanto do ponto de vista de *hardware* quanto de *software*. A contribuição científica consiste na proposta de um modelo contendo várias camadas, desde a coleta dos dados, processamento, avaliação e publicação de resultados, além da implementação de um conjunto de classes e funções que auxiliam no desenvolvimento de aplicativos IoT executados por dispositivos com poucos recursos computacionais na borda da rede. Os principais resultados práticos são o uso facultativo da nuvem, processamento próximo ao tempo real e simplicidade no desenvolvimento das aplicações. **METODOLOGIA**: A metodologia de avaliação consiste em propor um modelo e implementar um *framework* chamado STEAM. A validação do modelo se dá pela implementação de aplicações construídas com o *framework* STEAM, juntamente com a avaliação de métricas de desempenho e de uso de recursos computacionais como CPU, memória e rede. **RESULTADOS**: Os experimentos realizados em uma indústria de semicondutores através da implementação de 2 aplicações e 4 cenários de teste, demonstraram a viabilidade tanto do modelo quanto do *framework* STEAM. Como um dos objetivos era construir aplicações leves explorando os princípios da computação de borda, foram obtidos em média menos de 1,0% de carga de CPU e menos de 436kb de consumo de memória em uma Raspberry Pi 3 modelo B+. Além disso, foram alcançados tempos de resposta rápidos, processando até 239 pacotes de dados por segundo, e redução do tamanho dos dados de saída para 14% do tamanho dos dados brutos de entrada, tanto ao notificar eventos quanto na integração com um aplicativo de painel de controle remoto. **CONCLUSÃO**: A proposta se mostrou viável com resultados promissores, apresentando o *framework* STEAM como uma alternativa leve, rápida e precisa para desenvolvimento de aplicações IoT com processamento de dados na borda da rede, eliminando a dependência de processamento na nuvem.

Palavras-chave: Internet das Coisas. Computação de Borda. Análise de Dados. Enriquecimento de Dados. Processamento de Eventos Complexos. Framework.

ABSTRACT

CONTEXT: The Internet of Things is a fast expanding environment in which objects, animals, or people are equipped with the most diverse sensors and can automatically transfer their data through a network. Due to their limited nature, sensors and edge devices usually only relay the collected data to be processed by centralized systems in the cloud and, in many cases, wait for a response. This transfer from local to remote processing results in critical issues such as loss of connection, high response time, computer system overhead, in addition to requiring a robust and scalable structure for data communication and centralized processing. **OBJECTIVE**: Thus, we identified two challenges. First, devise a model capable of bringing data processing from the cloud to the network edge. Second, implement a solution that meets the constraints and heterogeneity of the IoT environment, both from a hardware and software perspective. The scientific contribution consists in the proposal of a model containing several layers, from data collection, processing, evaluation, and publication of results, in addition to the implementation of a set of classes and functions that facilitate the development of IoT applications executed by devices with few computational resources at the edge of the network. The main practical results are the optional use of the cloud, near real-time processing and simplicity in application development. **METHODOLOGY**: The methodology consists of proposing a model and implementing a *framework* called STEAM. The validation of the model takes place through the implementation of applications built with the STEAM *framework*, besides the evaluation of performance metrics and computational resources usages such as CPU, memory, and network. **RESULTS**: The experiments carried out in a semiconductor industry through the implementation of 2 applications and 4 test scenarios demonstrated the viability of both the model and the *framework* STEAM. Since one of the goals was to build lightweight applications in edge computing, we achieved an average of less than 1.0% CPU load and less than 436kb of memory consumption on a Raspberry Pi 3 model B+. In addition, we reached fast response times, processing up to 239 data packets per second, reducing the size of the output data to 14% the size of the raw input data when notifying events, and integrating with a remote control panel application. **CONCLUSION**: The proposal proved to be viable with promising results, presenting the *framework* STEAM as a lightweight, fast and accurate alternative for the development of IoT applications with data processing at the edge of the network, eliminating the processing dependency in the cloud.

Keywords: Internet of Things. Edge Computing. Data Analysis. Data Enrichment. Complex Event Processing. Framework.

LISTA DE FIGURAS

1 Monitoramento e identificação de situações de risco 22

2 Modelo conceitual da computação em névoa 26

3 Estrutura lógica de uma instância de evento 30

4 Fluxo de dados gerado por uma rede de sensores 32

5 Fluxo de eventos baseado em regras de monitoramento 33

6 Panorama do processamento de eventos 34

7 Arquitetura de referência para processamento de eventos 35

8 Cenário de uso de um processador de fluxo de eventos com módulo CEP . . 36

9 Objetivo principal dos trabalhos 51

10 Funcionalidades presentes nos trabalhos 52

11 Pacote de dados simples 60

12 Pacote de dados complexo 60

13 Comparativo entre a arquitetura tradicional IoT e a arquitetura STEAM . . . 64

14 Arquitetura STEAM 66

15 Dados recebidos em formato ASCII e convertidos para JSON 67

16 Análise dos dados e extração de informações relevantes 67

17 *Schema* JSON que define o formato padrão de saída 69

18 Enriquecimento e padronização dos dados 70

19 Pilha de protocolos IoT 71

20 Diagrama de classes do *framework* STEAM 74

21 Diagrama de sequência de uma aplicação STEAM 76

22 Infraestrutura da Aplicação 1 - STEAM vs Edge-Cloud 81

23 Infraestrutura da Aplicação 2 83

24 Metodologia e métricas do micro-benchmark para avaliação das aplicações STEAM 84

25 Exemplo de arquivo de log gerado pelo micro-benchmark 84

26 Aplicação básica desenvolvida com o *framework* STEAM 86

27 Código fonte da Aplicação 1 87

28 Aplicação configurada no *WSO2 Streaming Integrator* hospedado na Azure . 88

29 Execução da Aplicação 1 - STEAM vs Edge-Cloud 89

30 Tamanho médio dos pacotes de dados da Aplicação 1 - STEAM vs Edge-Cloud 90

31 Tempo de resposta da Aplicação 1 - STEAM vs Edge-Cloud 91

32 Código fonte da Aplicação 2 - Cenário 1 93

33 Código fonte da Aplicação 2 - Cenário 2 95

34 Captura de tela do painel de controle Node-RED para a Aplicação 2 - Cenário 1 96

35 Captura de tela do painel de controle Node-RED para a Aplicação 2 - Cenário 2 96

36 Uso de CPU e memória para a Aplicação 2 - Cenário 1 97

37 Uso de CPU e memória para a Aplicação 2 - Cenário 2 97

38 Distribuição de CPU e memória para a Aplicação 2 - Cenários 1 e 2 98

39 Tempo médio gasto por camada de processamento para a Aplicação 2 - Cenários 1 e 2 99

40 Distribuição do tempo total de execução para a Aplicação 2 - Cenários 1 e 2 100

41 Transformação dos dados durante o fluxo de processamento da Aplicação 2 - Cenário 1 101

42 Transformação dos dados durante o fluxo de processamento da Aplicação 2 - Cenário 2 101

43 Uso de CPU e memória para a Aplicação 2 - Cenário 3 - Painel de controle . 102

44 Uso de CPU e memória para a Aplicação 2 - Cenário 3 - Arquivo de log . . 103

45 Status da rede em um teste de caso real para a Aplicação 2 - Cenário 2 . . . 104

46 Status da rede em um teste de estresse para a Aplicação 2 - Cenário 3 104

LISTA DE TABELAS

1 *Corpus* da literatura 43

2 Trabalhos relacionados e suas principais características 50

3 Padronização das nomenclaturas utilizadas pelo módulo de enriquecimento . 68

4 Tempos de processamento e resposta para os experimentos da *Aplicação 1* . 91

5 Estrutura do *frame* de dados brutos do Cenário 1 da Aplicação 2 92

6 Estrutura do *frame* de dados brutos do Cenário 2 da Aplicação 2 94

7 Uso médio de CPU e memória para a Aplicação 2 - Cenários 1 e 2 98

8 Tempo médio gasto por camada de processamento para a Aplicação 2 - Cenários 1 e 2 99

9 Razão entre os tamanhos dos pacotes de dados medidos na saída e entrada . 102

10 Tempo médio gasto por camada de processamento para a Aplicação 2 - Cenário 3 104

11 Comparação qualitativa entre a literatura e o modelo STEAM 105

LISTA DE ALGORITMOS

1 Processo de execução de uma aplicação STEAM 77

LISTA DE SIGLAS

AMQP	Advanced Message Queuing Protocol
ANN	Artificial Neural Networks
ARIMA	Autoregressive Integrated Moving Average
ARPANET	Advanced Research Projects Agency Network
AVQ	Adaptive Vector Quantization
CEP	Complex Event Processing
CQL	Continuous Query Language
CoAP	Constrained Application Protocol
DDS	Data Distribution Service
DSP	Digital Signal Processor
EN	Edge Node
EPL	Event Processing Language
ESP	Event Stream Processing
EWMA	Exponentially Weighted Moving Average
FFT	Fast Fourier Transform
FPM	Frequent Pattern Mining
HTTP	Hypertext Transfer Protocol
HVAC	Heating, Ventilation and Air-Conditioning
IP	Internet Protocol
ISM	Industrial, Scientific and Medical
IoT	Internet of Things
JSON	JavaScript Object Notation
LR	Linear Regression
MLR	Multivariate Linear Regression
MQTT	Message Queue Telemetry Transport
MSE	Mean Squared Error
NILM	Non-Intrusive Load Monitoring
SAN	Sensing and Actuator Node
SQL	Structured Query Language
SoC	System on Chip
TBATS	Trigonometric seasonality, Box-Cox transformation, ARMA residuals, Trend and Seasonal
TCP	Transmission Control Protocol

WAMP	Websocket Application Messaging Protocol
WSN	Wireless Sensor Network
WWW	World Wide Web
XMPP	Extensible Messaging and Presence Protocol

SUMÁRIO

1 INTRODUÇÃO **19**

1.1 Contextualização 20

1.2 Motivação 21

1.3 Questão de Pesquisa 22

1.4 Organização do Texto 24

2 FUNDAMENTAÇÃO TEÓRICA **25**

2.1 Computação em Névoa - *Fog Computing* 25

2.1.1 Nodos da Névoa 25

2.1.2 Atributos de um Nodo da Névoa 27

2.1.3 Computação em Neblina - *Mist Computing* 27

2.2 Eventos 28

2.3 Sistemas de Tempo Real 30

2.4 Fluxo de Dados e Fluxo de Eventos 32

2.5 Processamento de Eventos 33

2.6 Complex Event Processing (CEP) 35

2.7 Event Processing Language (EPL) 37

3 TRABALHOS RELACIONADOS **41**

3.1 Metodologia de Pesquisa 41

3.1.1 Estado da Arte 42

3.2 Discussão Sobre os Trabalhos Relacionados 50

3.3 Técnicas e Ferramentas Utilizadas na Borda 52

3.4 Lacunas e Oportunidades de Trabalho 55

3.4.1 Incerteza de Dados e Eventos 56

3.4.2 Componente Embarcado 57

3.4.3 Proatividade 57

4 MODELO STEAM **59**

4.1 Diretrizes do Projeto 59

4.1.1 Formatos de Dados e Comunicação com Sensores 59

4.1.2 Heterogeneidade e Limitações dos Dispositivos 61

4.1.3 Variabilidade no Processamento de Dados 61
4.1.4 Variedade de Protocolos de Comunicação IoT 62
4.2 Arquitetura STEAM 63
4.2.1 Arquitetura IoT versus STEAM 63
4.2.2 Modelo Detalhado 65
5 METODOLOGIA **73**
5.1 Implementação do *Framework* STEAM 73
5.1.1 Diagrama de Classes 73
5.1.2 Diagrama de Sequência 75
5.1.3 Algoritmo de uma Aplicação STEAM 76
5.1.4 Application Programming Interface - API 77
5.2 Métricas da Aplicação 1 - Edge-Cloud 80
5.3 Métricas da Aplicação 2 - Edge 82
6 AVALIAÇÃO E RESULTADOS **85**
6.1 Aplicação 1 - *Edge-Cloud* 85
6.1.1 Cenário de Teste da Aplicação 1 87
6.1.2 Realização do Experimento e Resultados da Aplicação 1 87
6.2 Aplicação 2 - *Edge* 92
6.2.1 Cenário de Teste 1 da Aplicação 2 92
6.2.2 Cenário de Teste 2 da Aplicação 2 93
6.2.3 Cenário de Teste 3 da Aplicação 2 94
6.2.4 Realização dos Experimentos e Resultados da Aplicação 2 95
6.3 Comparação dos Resultados 105
7 CONCLUSÃO **107**
7.1 Contribuições 107
7.2 Trabalhos Futuros 109
REFERÊNCIAS **111**

1 INTRODUÇÃO

Nas últimas décadas, os avanços tecnológicos na área da computação - tanto relacionados à miniaturização de dispositivos e aumento do poder computacional quanto à conectividade e soluções a nível de software - resultaram na disseminação de equipamentos “inteligentes” para as mais variadas finalidades [Kaur and Kumar, 2021]. O cenário de computação ubíqua visualizado por Mark Weiser no final do século passado está finalmente se tornando realidade [Weiser, 1991]. Equipamentos com processamento de dados local ou comunicação em rede já fazem parte do nosso dia a dia sem que nos demos conta disso.

O estereótipo de um computador com teclado, mouse e monitor está completamente ultrapassado. Não que essa configuração não seja mais válida, muito pelo contrário. Computadores tradicionais são muito úteis e desempenham um papel fundamental na nossa rotina de trabalho diária. Porém, atualmente convivemos com computadores que não apresentam essa característica física tradicional. Os equipamentos associados ao termo *smart* possuem algum tipo de processamento ou comunicação de dados [Meindl et al., 2021]. Eletrodomésticos normalmente passivos, como lavadoras de louça ou de roupa, condicionadores de ar, cafeteiras, televisores e até veículos, hoje possuem recursos de coleta de dados, processamento e comunicação em rede. Também existem os dispositivos vestíveis, comumente chamados de *wearables*, que monitoram, processam e transmitem diversos sinais vitais como batimento cardíaco, temperatura corporal, pressão sanguínea, contagem de passos, entre outros [Al Bassam et al., 2021].

Esse cenário informatizado que se apresenta hoje em dia teve seus primórdios na década de 1970, com a popularização dos circuitos integrados, microcontroladores e microprocessadores. A microeletrônica aliada a controladores lógicos programáveis (CLP) estavam sendo usados em larga escala para processamento de dados e automatização na indústria. Naquela época, o objetivo da informatização era a automação de máquinas e processos pontuais, sem se preocupar com coleta, análise e compartilhamento de dados. Porém, a constante evolução dos dispositivos eletrônicos programáveis, tanto em relação à miniaturização quanto ao poder computacional e redução dos custos, permitiu que qualquer máquina ou objeto pudesse potencialmente receber um microcontrolador para processamento de dados.

Também na década de 1970 ocorreu o surgimento da ARPANET, uma rede de computadores desenvolvida a pedido do departamento de defesa dos Estados Unidos da América que interligava diversas universidades espalhadas por todo o território norte americano. A ARPANET foi fundamental para o desenvolvimento de tecnologias, padrões e protocolos de comunicação como o TCP/IP. Na sequência surgiu a Internet, que é uma rede de computadores com alcance mundial e disponível para acesso por qualquer pessoa, não apenas militares ou pesquisadores em universidades. A Internet se popularizou na década de 1990 através dos documentos de hi-

permídia contendo textos, áudios, vídeos e imagens, utilizando o protocolo de aplicação HTTP (Hypertext Transfer Protocol). Esse padrão ficou conhecido como WWW (World Wide Web), ou simplesmente Web. Nesse cenário, tanto a produção quanto o consumo de conteúdo são feitos por pessoas.

Na sequência do desenvolvimento tecnológico, Kevin Ashton [2009] imaginou o cenário vivenciado atualmente, descrito como "Internet das Coisas" (IoT), no qual qualquer objeto ou "coisa" pode ser monitorado e tem a capacidade de transmitir seus dados para um sistema computacional de forma autônoma, sem intervenção humana. Esse sistema é responsável por processar os dados gerando informações valiosas que não conseguiríamos perceber de forma simples. Com a disseminação da Internet das Coisas, o volume de dados, capturados, transmitidos e processados vem crescendo de forma substancial[1], fazendo com que seja necessário pensar em novas abordagens para processamento de dados, extração de informações e integração com demais sistemas [Sabireen and Neelanarayanan, 2021].

Na Internet das Coisas, os equipamentos são munidos de diversos sensores capazes de ler o ambiente em que estão inseridos, possuem microcontroladores ou microprocessadores para executarem cálculos, lógicas e algoritmos e se comunicam de forma autônoma uns com os outros ou com serviços hospedados em computadores convencionais ou até mesmo na nuvem. Essa capacidade de processamento, juntamente com a conectividade, tem como propósito transformar um conjunto de máquinas, dispositivos ou objetos aparentemente isolados, passivos e reativos em um espaço inteligente, no qual os dispositivos possam ser configurados e automatizados, trabalhando em conjunto para oferecerem uma experiência totalmente personalizada para os frequentadores do ambiente [Singh et al., 2021].

1.1 Contextualização

Ao contrário do ambiente gerado por equipamentos convencionais, uma característica desse novo cenário tecnológico "inteligente" é que há um volume muito elevado de dados sendo coletados, transmitidos e processados em tempo real. Os modelos de banco de dados relacionais e as técnicas tradicionais de consulta, processamento ou extração de dados não são adequadas para fluxos contínuos de dados. Além do mais, muitas informações que podem ser importantes para uma tomada de decisão não estão contidas nos dados brutos transmitidos pelos dispositivos, que precisam primeiramente ser minerados, filtrados, agregados e correlacionados uns com os outros antes de serem convertidos em dados úteis.

[1]Estudo conduzido pelo IDC (International Data Corporation) entitulado "*The Digital Universe of Opportunities: Rich Data and Increasing the Value of the Internet of Things*", publicado em abril de 2014, disponível em `http://www.emc.com/leadership/digital-universe/2014iview/index.htm`

Para realizar o processamento de fluxos de dados em tempo real, conhecidos pelo termo *streams*, David Luckham [2001] desenvolveu o conceito de Processamento de Eventos Complexos, do inglês "Complex Event Processing", ou simplesmente CEP. Ele propôs o uso de um conjunto de técnicas como sistemas baseados em regras, identificação de padrões, correlação de eventos, janela de tempo, entre outros, para propor um *framework* capaz de inferir um padrão emergente sobre um conjunto de diversas fontes de dados em tempo real. Com isso, é possível não apenas identificar um evento complexo em tempo real, mas também se antecipar a um acontecimento ou até identificar que um evento deveria ter acontecido, mas não aconteceu.

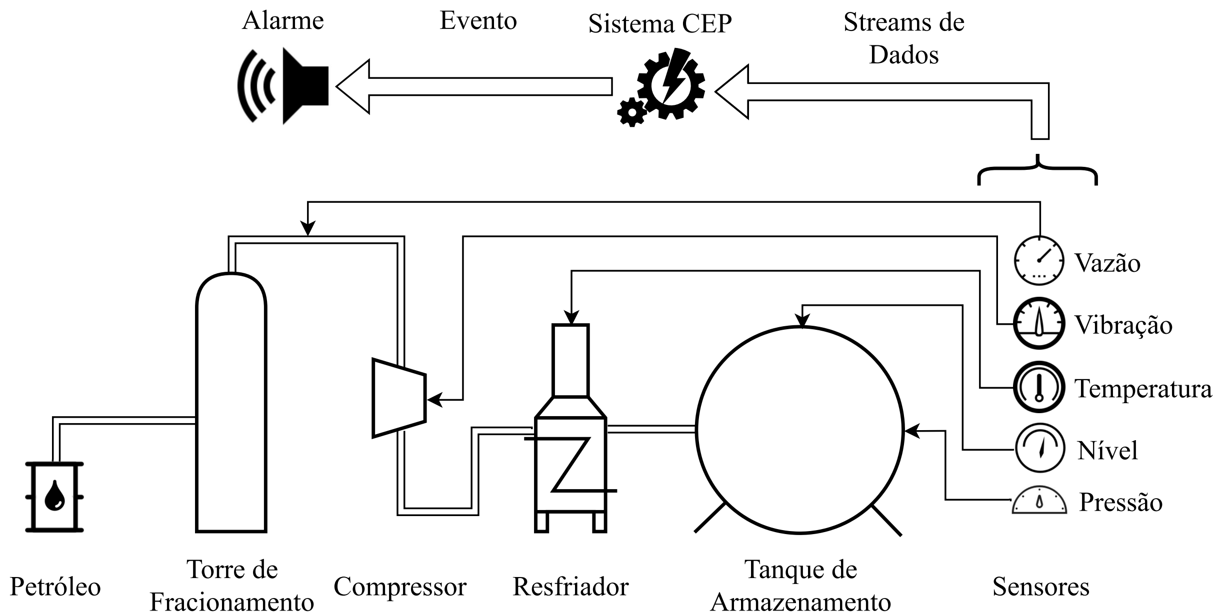
Uma característica muito marcante do CEP é a resposta rápida assim que um cenário toma forma. Do contrário, como consequência de um processamento lento, um alerta pode deixar de ser emitido dentro de um intervalo de tempo adequado. Por exemplo, sistemas CEP são usados amplamente por *traders* do mercado de ações para compra e venda de papéis. Para se posicionarem como compradores ou vendedores, eles analisam dezenas de indicadores em tempo real, como o valor momentâneo e histórico de uma ação, a quantidade de transações realizadas pelo mercado nos últimos minutos, o volume financeiro, média móvel, tendência, entre tantos outros [Abe, 2018]. Um sistema CEP utilizado por um *trader* como apoio para tomada de decisão deve responder de forma extremamente rápida, pois a compra ou venda de um papel realizada com atraso pode resultar em uma perda financeira expressiva.

1.2 Motivação

Na indústria, sistemas de monitoramento como o CEP podem ser utilizados para identificação de situações de risco, como por exemplo, vazamentos de produtos tóxicos. Um sistema de transporte e armazenamento de gás pode ser monitorado com sensores de pressão, temperatura, vazão, nível, vibração, etc, conforme Figura 1. Caso aconteça vazamento de gás e o sistema de alerta de vazamentos seja emitido vários minutos após a ocorrência, as pessoas mais próximas podem não ter chance de vestir uma máscara de gás ou escapar para um local seguro. Portanto, um sistema CEP deve ser capaz de detectar um evento complexo o mais próximo possível do tempo real. Além do tempo de resposta muito rápido, outra característica do CEP é a confiabilidade dos dados recebidos através dos *streams*. Visto que o sistema responde muito próximo do tempo real, os dados envolvidos precisam ser confiáveis, pois não cabe ao CEP identificar lacunas, filtrar ruídos ou eliminar dados fora da normalidade, chamados de *outliers*.

Imagine uma situação em que o alarme de vazamento de gás em uma refinaria de petróleo dispara, pois um dos sensores sofreu interferência elétrica ou magnética e emitiu um dado impreciso. Nessa situação, o CEP pode identificar um vazamento de forma errada refletindo em consequências críticas, como evacuação de pessoas e parada de produção, além de colocar em

Figura 1: Monitoramento da produção de gás e identificação de situações de risco



dúvida a confiabilidade do sistema. Outra forma de ocorrer a geração de um dado incorreto é através de ataques ao sistema por pessoa mal-intencionada. Um funcionário insatisfeito pode querer sabotar a empresa e intencionalmente danificar um sensor, dando pancadas ou derramando algum líquido sobre ele. Em outra situação, um hacker pode invadir um servidor da empresa e injetar pacotes de dados aleatórios na rede. Essas ações poderiam gerar dados falsos, fazendo com que o CEP avaliasse um cenário a partir de informações incorretas, resultando no disparo ou não disparo de um evento, podendo gerar consequências inimagináveis.

1.3 Questão de Pesquisa

Analisando os cenários expostos anteriormente, identifica-se um dilema. Sistemas de monitoramento precisam ser muito rápidos na análise dos dados e identificação de eventos, mas também têm que ser perfeitamente confiáveis. Para que sejam rápidos, eles devem focar majoritariamente no processamento dos dados vindos dos *streams* para identificar um padrão e disparar um evento. Para serem confiáveis, eles precisam analisar cada dado que chega em tempo real e decidir se ele se encaixa no padrão de comportamento do *stream*, calculando um fator de confiabilidade ou incerteza que irá aceitar ou rejeitar o dado.

As técnicas de identificação de padrões, análise de similaridade e detecção de pontos fora da curva não são simples de serem executadas, consumindo memória e um tempo significativo para elaboração e treinamento de um modelo computacional e posterior processamento dos dados, além de necessitarem de atualizações de parâmetros frequentemente. Na etapa de elaboração

do modelo e treinamento, são utilizados dados históricos que representam a normalidade do sistema. Nessa etapa, são construídos modelos matemáticos, estatísticos ou até redes neurais para serem usados posteriormente. Quando os dados estiverem chegando pelo *stream* durante a operação normal do sistema, cada um deles é então comparado com o modelo construído anteriormente e é possível identificar se são válidos ou não.

Quando nos damos conta de que sistemas de monitoramento são centralizados, responsáveis por processar dados de milhares de *streams* em tempo real, o somatório dos tempos utilizados para processamento, análise, e tomada de decisão seria de grande influência no tempo de resposta total da aplicação. Além disso, o armazenamento dos modelos e a atualização de seus parâmetros ao longo do tempo necessitam de uso e gerenciamento de memória. Com base nessa situação, se coloca a seguinte questão de pesquisa:

Como seria um modelo para processamento de eventos e enriquecimento de fluxos de dados IoT na borda da rede?

Com base nos estudos realizados nesse trabalho, foram identificados os seguintes desafios intimamente ligados à resposta da questão de pesquisa:

- Coleta de Dados: Como fornecer uma interface de comunicação abrangente com a rede de sensores ou dispositivos IoT geradores de dados?

- Análise: Como proceder com a execução de técnicas de processamento de dados matemáticos, estatísticos, aprendizado de máquina, entre outros, capazes de extrair informações relevantes a partir de dados brutos?

- Tomada de Decisão: O que é necessário para avaliar um cenário dinâmico com múltiplas variáveis e identificar a ocorrência de um evento?

- Enriquecimento de Fluxos de Dados: Como mesclar o resultado da etapa de análise com os dados originais, fornecendo um *stream* enriquecido e relevante para as aplicações cliente?

- Publicação de Resultados: Quais formatos de dados e protocolos de comunicação devem ser oferecidos para permitir conectividade com as aplicações cliente?

1.4 Organização do Texto

Além deste capítulo atual que apresenta a introdução do trabalho com contextualização, motivação e questão de pesquisa, o capítulo 2 registra a fundamentação teórica, trazendo termos e conceitos essenciais para o entendimento da proposta. Na sequência, o capítulo 3 apresenta os trabalhos relacionados, discutindo os artigos que representam o estado da arte, técnicas utilizadas e desafios da área. O capítulo 4 apresenta em detalhes o modelo proposto e sua arquitetura. A metodologia de avaliação e a implementação do modelo são discutidas no capítulo 5, enquanto a realização dos experimentos e os resultados obtidos são apresentados no capítulo 6. Por fim, o capítulo 7 encerra a tese com as considerações finais, contribuições e proposta de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem por objetivo apresentar os termos e conceitos necessários para o entendimento do trabalho. Inicialmente são apresentados os conceitos de computação na borda da rede, mais especificamente em névoa (*fog computing*) e em neblina (*mist computing*). Depois é discutido o conceito de evento e como ele é representado em um sistema computacional. Na sequência são apresentados os fundamentos e técnicas dos sistemas processadores de eventos, suas estruturas e componentes.

2.1 Computação em Névoa - *Fog Computing*

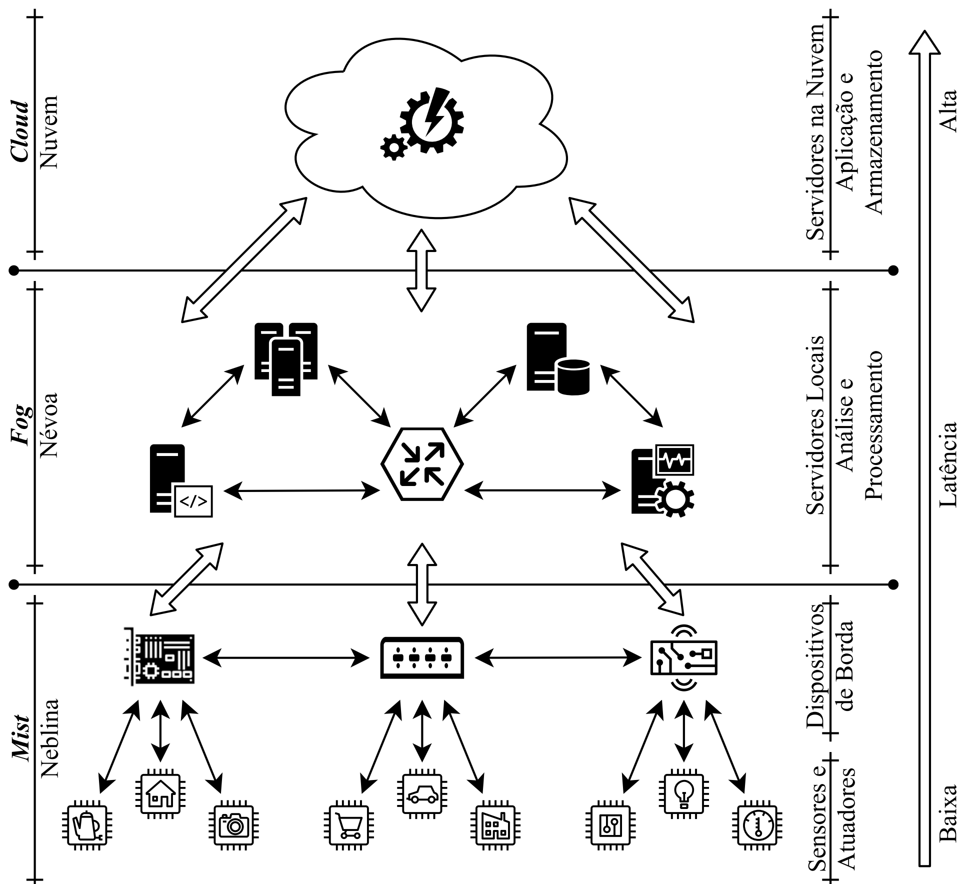
A computação em névoa, ou *fog computing*, é um modelo em camadas para permitir acesso ubíquo a recursos de computação escaláveis e compartilhadas. Segundo o NIST [Iorga et al., 2018], o modelo consiste em nós, tanto físicos quanto virtuais, que se situam entre os dispositivos finais e os serviços centralizados na nuvem. Os nós na névoa são sensíveis ao contexto e suportam um sistema comum de gerenciamento e comunicação de dados. A computação na névoa minimiza o tempo de resposta entre aplicativos e fornece recursos de computação local para os dispositivos finais, e quando necessário, provê conectividade de rede para serviços centralizados.

A Figura 2 ilustra a computação em névoa dentro de um amplo contexto de um ecossistema baseado em nuvem servindo dispositivos inteligentes. Aqui é possível identificar que a computação na névoa permite diminuir a latência entre os dispositivos finais e a nuvem, já que fornecem processamento e análise de dados de forma distribuída, mais próximos geograficamente dos consumidores finais.

2.1.1 Nodos da Névoa

Os nós ou nodos são os componentes principais da arquitetura de computação em névoa. Podem ser tanto componentes físicos, como *gateways*, *switches*, roteadores, servidores, ou componentes virtuais, como comutadores virtualizados ou máquinas virtuais. Um nó na névoa conhece sua distribuição geográfica e localização lógica dentro do contexto de seu *cluster*. Além disso, os nós da névoa fornecem serviços de comunicação entre a camada de borda da rede e os recursos de computação centralizados na nuvem, quando necessário [Iorga et al., 2018].

Figura 2: Modelo conceitual da computação em névoa



Fonte: Adaptado de [Iorga et al., 2018]

Segundo o NIST, as seis características a seguir são essenciais para distinguir a computação de névoa de outros paradigmas de computação. No entanto, um dispositivo inteligente ou usuário IoT não precisa usar todas as características ao consumir um serviço de computação de névoa.

- **Localização e baixa latência**: A computação na névoa oferece uma latência muito baixa devido aos nós conhecerem sua localização lógica no contexto de todo o sistema, além de conhecerem os custos de latência para se comunicarem com outros nós.

- **Distribuição geográfica**: Em contraste com a nuvem, que é mais centralizada, os serviços e aplicativos visados pela computação na névoa utilizam estruturas amplamente distribuídas, mas geograficamente identificáveis.

- **Heterogeneidade**: A computação em névoa suporta a coleta e o processamento de dados de diferentes fontes e formatos, adquiridos por meio de vários meios de comunicação de rede, tanto abertas quanto proprietárias, cabeadas ou sem fio.

- **Interoperabilidade**: Suporta de forma transparente a integração de certos serviços, como *streaming* em tempo real, através da cooperação entre diferentes provedores. Portanto, os componentes da computação em névoa devem poder interoperar de forma padronizada.

- **Tempo real**: Aplicativos de computação na névoa exigem interações em tempo real em vez de processamento em lote. Os dados são processados a partir de *buffers* em memória RAM, não são previamente armazenados em memória secundária.

- **Escalabilidade**: A computação na névoa é adaptativa por natureza no nível de *cluster*, suportando computação elástica, pool de recursos, alterações de carga de dados e variações de condição de rede.

2.1.2 Atributos de um Nodo da Névoa

Para facilitar a implementação de um sistema que utilize computação em névoa, Iorga et al. [2018] afirmam que os nodos precisam suportar um ou mais dos seguintes atributos:

- **Autonomia**: Capacidade de operar independentemente, tomando decisões locais tanto no nível do nó quanto do *cluster* de nós.

- **Heterogeneidade**: Os nós da névoa apresentam diferentes estruturas de hardware e podem ser implantados em uma ampla variedade de ambientes.

- **Agrupamento hierárquico**: Organização dos nodos em estruturas hierárquicas com diferentes camadas, fornecendo diferentes subconjuntos de funções de serviço enquanto trabalham de forma coordenada.

- **Gerenciabilidade**: Na névoa, os nós podem ser gerenciados e orquestrados por sistemas complexos que podem executar a maioria das operações de rotina automaticamente.

- **Programabilidade**: Obrigatoriamente os nós da névoa são programáveis em vários níveis e por vários interessados, como provedores de equipamentos, operadores de rede, desenvolvedores de aplicações ou usuários finais.

2.1.3 Computação em Neblina - *Mist Computing*

Como visto anteriormente nessa seção, soluções de computação em névoa são adotadas por muitos setores, e os esforços para desenvolver aplicações distribuídas e ferramentas de análise

existem e continuam a se desenvolver. Porém, a necessidade de recursos computacionais geograficamente dispersos e de baixa latência desencadeou a evolução tecnológica para a computação de neblina, ou *mist computing*, promovendo o desenvolvimento de nós mais especializados e dedicados que apresentam baixos recursos computacionais [Iorga et al., 2018]. Esses nós, referidos como nós de neblina, são considerados nós de névoa leves, ou *lightweight fog nodes*. Esses nós que formam a camada de computação de neblina são colocados ainda mais perto dos dispositivos periféricos do que os nodos de névoa mais potentes. Eles colaboram entre si na análise e processamento de dados, geralmente compartilhando a mesma localidade física que os dispositivos inteligentes que eles atendem e que hospedam a aplicação final. A computação em neblina usa microprocessadores e microcontroladores para alimentar os nós da névoa e, potencialmente, os serviços de computação centralizados em nuvem.

2.2 Eventos

Abordagens recentes para modelagem e implementação de sistemas em tempo real utilizam o conceito de eventos. Por exemplo, muitos aplicativos em tempo real envolvem sensores (geralmente distribuídos) que detectam e relatam eventos, que por sua vez, precisam ser analisados em tempo real para detectar padrões que signifiquem alguma oportunidade ou ameaça. Outras aplicações em tempo real precisam monitorar eventos que capturam ações de usuários, detectando padrões de interesse associados às suas atividades e, subsequentemente, registrando essas informações ou gerando alertas [Buyya et al., 2016].

Conforme definido por Etzion and Niblett [2010], um evento é uma ocorrência dentro de um determinado sistema ou domínio. É algo que aconteceu de fato ou é inferido como tendo acontecido nesse domínio. A palavra evento também é usada para significar uma entidade de programação que representa tal ocorrência em um sistema computacional.

De uma forma mais detalhada, Adi and Etzion [2004] afirmam que um evento é uma ocorrência significativa (em alguns domínios), instantânea (acontece em um ponto específico no tempo) e atômica (acontece completamente ou não ocorre). Podem ser classificados como eventos concretos (externos) e eventos inferidos (internos). Eventos concretos são aqueles que acontecem na realidade, geralmente como resultado de uma mudança no estado de um objeto. Exemplos incluem uma pessoa entrando em uma sala de reunião ou uma luz sendo ligada em um prédio. Este tipo de evento normalmente é gerado por sensores, aplicações ou intervenção humana.

Eventos inferidos não acontecem na realidade física, mas podem ser logicamente concluídos, visualizando o entorno (contexto) e o histórico das ocorrências de eventos concretos. Um evento inferido representa a ocorrência de uma situação significativa na realidade física. Por

exemplo, se todos os convidados para uma reunião já chegaram à sala de reuniões, pode-se inferir a situação lógica de que a reunião pode começar. Ou se o consumo de eletricidade em uma casa é muito alto, pode ocorrer o desarme de um disjuntor e consequentemente, o acontecimento da falta de energia.

Em um sistema de processamento de eventos, é possível encontrar muitas instâncias de eventos que possuem estrutura e significado semelhantes, por exemplo, o fluxo de eventos provenientes de um sensor de temperatura. Nesse caso, todos os eventos contêm o mesmo tipo de informação, embora, é claro, cada evento tenha um registro de data e hora único, relate uma leitura de temperatura diferente ou esteja associado a uma localização geográfica específica. Portanto, em vez de definir a estrutura de cada evento individualmente, o correto é especificar a estrutura dessa classe inteira de eventos. Isso é semelhante a declarar um tipo reutilizável em uma linguagem de programação, e essa especificação é definida como *tipo de evento*.

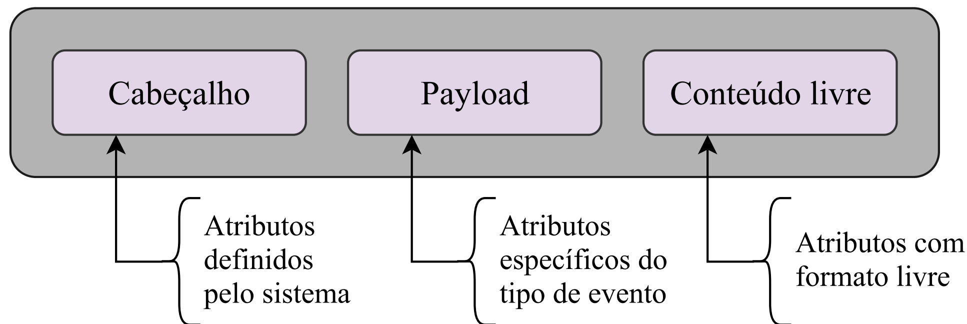
Etzion and Niblett [2010] definem *tipo de evento* como sendo uma especificação para um conjunto de objetos de eventos que possuem a mesma intenção semântica e mesma estrutura. Cada objeto que representa um evento é considerado uma *instância* de um tipo de evento. Um tipo de evento especifica as informações contidas em suas instâncias de eventos, através da definição de um conjunto de *atributos* do evento. Um *atributo* consiste na definição de um *nome* e um *tipo de dado*.

Os atributos de um evento devem ajudar a responder perguntas como: O que aconteceu? Quando aconteceu? Onde aconteceu? Que outras informações estão associadas ao acontecimento? As respostas a estas perguntas podem ser registradas em vários níveis de detalhamento, considerando que cada aplicação pode avaliar que um atributo seja relevante ou não nas suas regras de negócio. Embora a falta de informações seja um grande problema para a correta avaliação de um cenário, oferecer informações em excesso também são um fardo, principalmente para o transporte e armazenamento de dados. Portanto, ao projetar um tipo de evento, é necessário considerar a qualidade e quantidade de informações que será exigida pelos sistemas que o utilizam.

Um evento, tanto externo quanto interno, é representado por uma *instância de evento* que contém as informações necessárias sobre o evento. Essas informações incluem o instante em que o evento ocorreu, dados relevantes para as aplicações que reagem ao evento e dados adicionais necessários para decidir se uma situação (evento inferido) realmente ocorreu. Conforme ilustrado na Figura 3, uma instância de evento pode ser definida como um pacote de dados organizado em 3 blocos básicos: (i) cabeçalho; (ii) payload e (iii) conteúdo livre.

O cabeçalho consiste em meta-informações sobre o evento, por exemplo, seu momento de ocorrência. A palavra "cabeçalho" é usada porque, quando um objeto de evento é serializado em uma mensagem, esses dados geralmente são colocados no início da mensagem. Essas infor-

Figura 3: Estrutura lógica de uma instância de evento, mostrando suas três partes constituintes



Fonte: Adaptado de Etzion and Niblett [2010]

mações são transportadas usando atributos conhecidos e, portanto, podem ser identificadas por um processador de eventos. A segunda parte (*payload*) contém informações específicas sobre a ocorrência em si, e consiste em uma coleção de atributos definidos no tipo de evento. Cada atributo tem um tipo de dados que pode ser simples, como um valor numérico ou uma *string*, ou pode ser uma estrutura de dados complexa. A definição feita no tipo de evento também pode indicar quantas vezes um determinado atributo é necessário ou permitido para aparecer no *payload*. Um evento também pode conter informações de conteúdo livre, que normalmente são dados auxiliares de interesse restrito a alguma aplicação em especial. O tipo de evento determina o conjunto de atributos que podem aparecer no *payload*, mas não restringe a parte de conteúdo livre do evento. Cada instância de evento obrigatoriamente possui um cabeçalho, mas não precisa ter um *payload* nem conteúdo livre.

2.3 Sistemas de Tempo Real

O termo “tempo real” significa que um sistema computacional deve processar os eventos à medida que eles ocorrem e dentro de um intervalo de tempo especificado. Esse intervalo de tempo é tipicamente na ordem de mili, micro ou até nano segundos, dependendo da criticidade do ambiente que está sendo monitorado. Dessa forma, tempo real se refere à capacidade de processar dados à medida que eles chegam, em vez de armazená-los e recuperá-los em algum momento futuro [Buyya et al., 2016].

Assim como a entrada e processamento dos dados ocorre em tempo real, a saída também deve ser reportada em tempo real. Esses sistemas geralmente emitem notificações sobre ocorrências significativas no ambiente, tanto para humanos quanto para outros sistemas, por exemplo disparando um alerta em uma tela, alimentando um fluxo de eventos ou consumindo um *webservice*. Para oferecer processamento em tempo real, Ellis [2014] afirma que um sistema deve atender os seguintes requisitos:

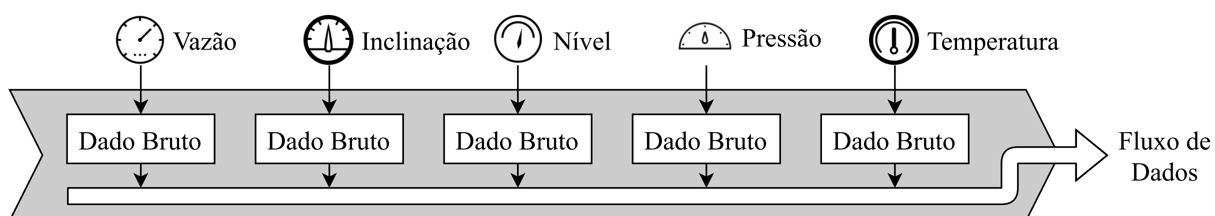
- **Baixa Latência**: Em um sistema em tempo real, a latência se refere ao tempo entre o acontecimento de um evento e o início de seu processamento no sistema. Essa latência normalmente envolve latência de rede e latência de processamento do computador. Os sistemas em tempo real exigem baixa latência para responderem aos eventos dentro dos limites de tempo especificados. Várias estratégias podem ser adotadas para dar suporte a esses requisitos, que incluem:
  - Processamento em memória - necessário para minimizar o atraso no processamento associado ao uso de discos e entrada/saída; esse recurso se torna mais viável a cada dia devido ao custo decrescente da memória;
  - Uso de tecnologia flash para armazenar os dados que não precisam estar na memória principal; esta abordagem aumenta a velocidade de acesso aos dados;
  - Avaliação incremental, isto é, atualização de cálculos e resultados de consultas para cada novo item de dados, sem precisar reavaliar todo o conjunto de dados;
  - Processamento paralelo com conexão de alta largura de banda entre processadores;
  - Busca e processamento antecipado, permitindo acesso mais rápido aos dados de múltiplos fluxos de dados
- **Alta Disponibilidade**: Disponibilidade significa a capacidade de um sistema executar sua função no momento em que for solicitado. Sistemas em tempo real exigem alta disponibilidade, caso contrário, os eventos que chegam do mundo externo não seriam processados imediatamente. Em se tratando de um fluxo de eventos, especialmente com alto volume de dados e alta velocidade de transmissão, não faz sentido armazenar os dados em disco ou *buffer* de memória para processamento subsequente. Para dar suporte a esse requisito de alta disponibilidade, várias estratégias podem ser adotadas:
  - Distribuição do processamento entre vários nós para que, se uma máquina falhar, outra possa assumir;
  - Replicação dos dados para vários servidores, de modo que se uma máquina falhar, os dados ainda podem existir em outra máquina;
  - Processamento redundante de dados, isto é, ter mais de um nó calculando o resultado para um mesmo conjunto de dados. Obrigatoriamente implica na adoção dos dois itens acima.
- **Escalabilidade Horizontal**: Essa característica se refere à capacidade de adicionar servidores a um *pool* existente para melhorar o tempo de resposta ou aumentar a capacidade de processamento. A capacidade de adicionar dinamicamente novos servidores para comportar o volume de dados ou carga de processamento momentânea é de grande importância para que sistemas em tempo real possam garantir que os dados sejam processados dentro de intervalos de tempo especificados.

A escalabilidade horizontal é especialmente importante se não for possível controlar a taxa de entrada de dados. Se um sistema estiver consumindo um fluxo de dados de volume fixo conhecido, ele poderá ser dimensionado estaticamente para garantir que os requisitos em tempo real sejam atendidos. Porém, se a taxa de entrada de dados for variável e imprevisível, o sistema precisa se adaptar dinamicamente a carga momentânea. Observe que a escalabilidade horizontal deve ser contrastada com a escalabilidade vertical, que se refere à capacidade de adicionar recursos a um único servidor para melhorar o desempenho e a capacidade computacional [Ellis, 2014].

2.4 Fluxo de Dados e Fluxo de Eventos

De acordo com Teng et al. [2017], fluxo de dados (Data Stream - DS) é um modelo de dados novo, dinâmico e contínuo, diferente do modelo clássico de banco de dados. Os dados em um DS chegam em tempo real, a qualquer momento, e podem vir por exemplo do mercado de ações, redes sociais, além de diversos sensores acoplados ao corpo humano, meio ambiente, veículos, máquinas, entre outros. A Figura 4 ilustra um fluxo de dados gerados por uma rede de sensores em uma indústria. Nesse caso, cada sensor é responsável por monitorar um fenômeno físico e reportar seu valor instantâneo periodicamente através de um barramento. Dessa forma, o conjunto dos dados de todos os sensores formam um fluxo de dados.

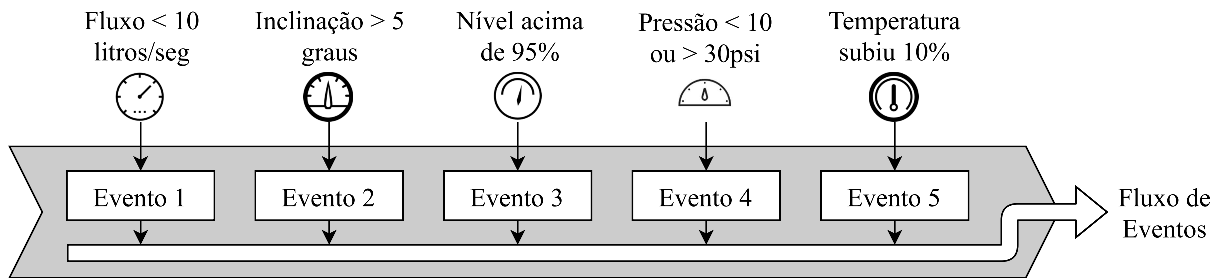
Figura 4: Fluxo de dados gerado por uma rede de sensores



Já o processamento de eventos consiste na realização de operações em instâncias de eventos, conforme chegam em um sistema através de um fluxo de eventos (Event Stream - ES). Operações comuns incluem a leitura, criação, transformação e processamento de eventos [Buyya et al., 2016]. O ES é uma especialização do DS, composta por uma sequência contínua e dinâmica de eventos bem definidos. A principal diferença entre eles, afirmam Teng et al. [2017], é que enquanto DS trabalha com dados brutos, ES trabalha com eventos previamente processados e categorizados. Em essência, o ES é composto por eventos, e cada evento inclui um identificador, atributos do evento e o momento da ocorrência, enquanto o DS é composto por um único dado, sem significado específico.

Por exemplo, a Figura 5 mostra como a aplicação de regras sobre dados de sensores gera um fluxo de eventos. Nesse cenário, os dados brutos são comparados com limites fixos, relativos ou tendências. Quando atingem determinado patamar, um evento é disparado indicando que algo relevante aconteceu. Além do acontecimento propriamente dito, também é reportado o instante em que ocorreu, o local físico ou equipamento associado e demais informações úteis ao sistema.

Figura 5: Fluxo de eventos baseado em regras de monitoramento de um processo industrial



Muitos sistemas de processamento de eventos são criados para reagir a eventos individuais do ambiente, seguindo os princípios da arquitetura orientada a eventos. Conforme Buyya et al. [2016], essas aplicações podem não precisar lidar com propriedades temporais de eventos, seja de eventos únicos ou entre vários eventos, mesmo se os dados associados a um evento tiverem um registro de data e hora. Na verdade, esses sistemas orientados a eventos precisam apenas executar alguma computação ou ação em resposta a um evento. Neste caso, usa-se o termo programação baseada em eventos ou arquiteturas orientadas a eventos.

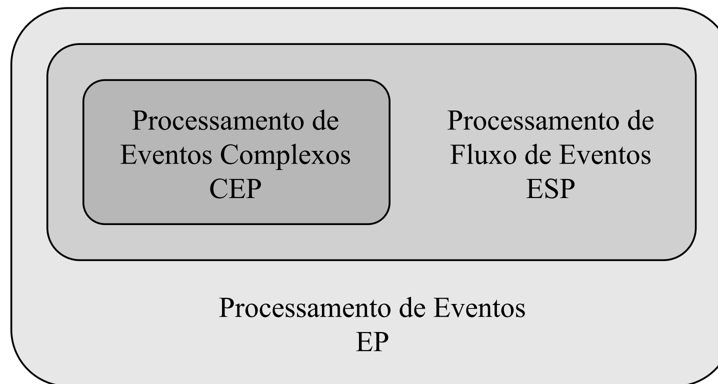
Por exemplo, um sistema de cobrança de pedágio em uma rodovia possui cancelas em todas as entradas e saídas. Quando um veículo entra na rodovia, uma câmera identifica a placa do veículo e gera um evento de entrada associando a localização geográfica ao veículo. Após realizar seu percurso, quando o veículo sai da rodovia a cancela de saída fotografa a placa e gera um evento de saída, vinculando também a localização geográfica. Quando esse evento de saída chega ao sistema de processamento de eventos, ele recupera o evento de entrada, identifica a distância percorrida pelo veículo na rodovia e faz o cálculo do valor devido. Dessa forma, o operador da cancela é notificado automaticamente do valor a ser cobrado do motorista. Como pode-se perceber, o sistema funciona reagindo a eventos pontuais que aconteceram no mundo real.

2.5 Processamento de Eventos

Processamento de Eventos (*Event Processing* - EP) é um paradigma que analisa fluxos de eventos com o objetivo de extrair informações úteis. Como mostrado na Figura 6, pode-se dividir o EP em duas áreas principais chamadas “Processamento de Fluxo de Eventos - ESP” e

"Processamento de Eventos Complexos - CEP". Enquanto ESP suporta análises contínuas como filtragem, agregação, enriquecimento, classificação, junção, entre outros, CEP utiliza técnicas de identificação de padrões sobre sequências de eventos simples, com o objetivo de detectar e relatar eventos compostos ou complexos [Dayarathna and Perera, 2018].

Figura 6: Panorama do processamento de eventos



Fonte: Adaptado de [Dayarathna and Perera, 2018]

Em seu trabalho, Paschke and Vincent [2009] definem uma arquitetura de referência geral para processadores de eventos apresentando entidades elementares e suas relações. A arquitetura de referência apresentada na Figura 7 é uma abstração do EP, com foco nos elementos e relacionamentos necessários para permitir o uso de sistemas baseados em eventos, evitando a dependência de tecnologias específicas. Essa arquitetura descreve quatro entidades diferentes, com papéis bem definidos:

- **Originador de Eventos**: Também chamado de *Produtor de Eventos* ou *Emissor de Eventos*, é um conjunto possivelmente distribuído de fontes de eventos em que eventos, geralmente mais atômicos, mas também eventos de alto nível, ocorrem ou são detectados a partir de fluxos. As origens de eventos são, por exemplo, processos, fluxos de trabalho, serviços, aplicativos, fluxos de mensagens, armazenamentos de dados temporais, etc [Paschke and Vincent, 2009].

- **Modelador de Eventos**: É a entidade responsável pelas definições dos padrões de eventos, isto é, o modelador implementa os tipos de eventos cujas ocorrências (eventos concretos ocorridos ou produzidos na origem ou eventos que não deveriam ocorrer) são necessários para disparar eventos complexos, que são processados no *Processador de Eventos*. O modelador também especifica a interface entre o Processador de Eventos e as fontes [Paschke and Vincent, 2009].

- **Processador de Eventos**: Um *Processador de Eventos* é uma plataforma, possivelmente distribuída, que processa eventos. Possui diversas entradas para recebimento de dados

vindos dos *Originadores de Eventos*, os processa e em seguida publica ou transporta ativamente os resultados para os *Consumidores de Eventos*. O *Processador de Eventos* compreende a infraestrutura técnica que suporta o processo de seleção de eventos, agregação, classificação dentro de hierarquias e abstração de eventos para gerar eventos complexos de maior nível de interesse [Paschke and Vincent, 2009].

- **Consumidor de Eventos**: É o elemento responsável por receber os eventos, simples ou complexos, que foram previamente processados e distribuídos pelo *Processador de Eventos*. Ele pode consumir, usar ou reagir a eventos via regras de reação. Por meio de suas reações, um *Consumidor de Eventos* pode criar eventos adicionais e, portanto, também atuar como um *Originador de Eventos* que pode vir a ser usado pelo próprio *Processador de Eventos* mais adiante [Paschke and Vincent, 2009].

Figura 7: Arquitetura de referência para processamento de eventos

Fonte: Adaptado de [Paschke and Vincent, 2009]

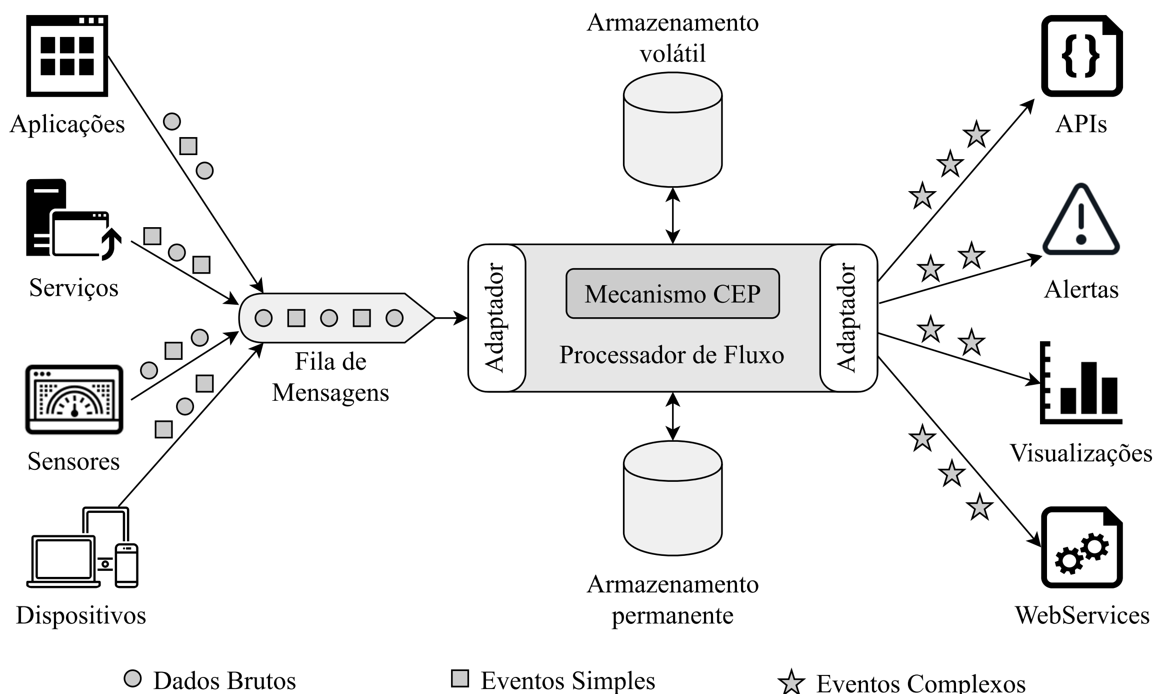
2.6 Complex Event Processing (CEP)

O termo Processamento de Eventos Complexos (Complex Event Processing - CEP) surgiu em 1995 a partir de uma pesquisa liderada por David Luckham na Universidade de Stanford [Luckham et al., 1995]. O projeto chamado *Rapide*[1] tinha como objetivo encontrar falhas em

[1] `http://complexevents.com/stanford/rapide/`

projetos de *hardware* através de simulação de eventos discretos. Um sistema CEP é uma infraestrutura de software que normalmente consiste em uma camada de adaptadores que podem se conectar diretamente a fontes de dados e canalizam informações a um mecanismo de análise. Funcionam alimentados por fluxos de dados de entrada coletados a partir das mais diversas fontes, conforme visualizado na Figura 8. Utilizam algoritmos e regras para identificar tendências e padrões interconectados em tempo real, chamados de *Eventos Complexos*. Em seguida, disponibilizam os dados processados para aos usuários interessados através de um fluxo de eventos de saída [Leavitt, 2009].

Figura 8: Cenário de uso de um processador de fluxo de eventos com módulo CEP



Fonte: Adaptado de [Dayarathna and Perera, 2018]

Os usuários podem especificar a lógica de negócios usada para filtrar e analisar eventos por meio de interfaces de programação. A linguagem utilizada é chamada "Linguagem de Processamento de Eventos", ou *"Event Processing Language* - EPL", descrita em detalhes na seção 2.7. A partir da EPL, um servidor CEP avalia a lógica de negócios, executa algoritmos de análise de fluxo de eventos e transfere os resultados continuamente para os usuários. Isso pode ser feito usando um painel baseado na Web ou uma notificação em tempo real, como comunicações por e-mail, serviço de mensagens curtas (SMS), ou publicando dados em um sistema através de técnicas de fila de mensagens, *push notification* ou *publish/subscribe*.

Leavitt [2009] afirma que os mecanismos do CEP funcionam a partir da análise de eventos de várias maneiras, por exemplo, por meio de correspondência de padrões, inferência, análise de grafos e modelos que suportam a causalidade implícita e explícita de eventos. O algoritmo básico de correspondência de padrões trabalha procurando uma sequência potencial entre os

eventos de entrada, dentro de uma janela de eventos deslizante. O algoritmo então determina se existe um padrão, de qual classe ele é e quais eventos pertencem a ele. Para ajudar com o processo, as ferramentas CEP podem filtrar informações irrelevantes e enriquecer eventos com dados externos. Em seguida, eles agregam e correlacionam informações dos vários eventos de entrada para identificar ocorrências de eventos complexos.

Sistemas CEP podem ser usados em inúmeras aplicações. Por exemplo, podem analisar eventos em uma rede de computadores, sistemas de pagamento eletrônico, monitoramento da rede elétrica, bancos de dados, transporte, logística, entre outros, para determinar se estão sendo alvo de um ataque, se estão sendo executados de maneira adequada ou se estão apresentando problemas relativos a sobrecarga de recursos computacionais. Além de trabalhar com dados reais, os sistemas CEP também podem ser alimentados por simuladores para determinar se situações hipotéticas seriam inviáveis ou ineficientes do ponto de vista computacional, causariam problemas legais, violariam políticas corporativas ou contratos de nível de serviço (*Service Level Agreement* - SLA) dos clientes [Leavitt, 2009].

Do ponto de vista de sistemas distribuídos, CEP aborda dois pré-requisitos cruciais para construir sistemas escaláveis e dinâmicos. Primeiro, dissocia provedores e receptores de informação. Nem os provedores precisam conhecer o conjunto de receptores, nem os receptores precisam conhecer o conjunto de dados de entrada ou fontes de eventos. Segundo, os sistemas CEP não apenas mediam informações em forma de eventos entre provedores e consumidores, mas suportam a detecção de relacionamentos entre eventos, por exemplo, relações temporais que podem ser especificadas pela definição de regras de correlação, frequentemente chamadas de “Padrões de Eventos”. Através da agregação e composição, novos eventos complexos podem ser gerados e usados posteriormente para derivar eventos mais abstratos. Além disso, a correlação gradual de eventos pode ajudar a reduzir a carga de mensagens, e assim, contribui para um sistema de informações altamente escalável [Etzion, 2016].

2.7 Event Processing Language (EPL)

Um sistema de processamento de eventos (*Event Processing System* - EPS) necessita de uma linguagem específica para expressar suas regras de processamento. Essa linguagem é conhecida como “Event Processing Language” (EPL), e pode ser classificada de acordo com o tipo de sistema que a executa. A EPL pode ser baseada em consultas, baseada em regras ou baseada em programação. Cada uma dessas abordagens é descrita a seguir:

- **Baseada em Consultas**: Geralmente, os EPSs baseados em consulta suportam uma EPL estendida a partir da linguagem de banco de dados relacional SQL para consultar dados de eventos. De acordo com Chen et al. [2000], essas instruções são geralmente chamadas

de consultas contínuas. Enquanto as consultas SQL tradicionais são disparadas a partir de uma ordem bem definida e atuam sobre dados persistentes, as consultas contínuas ficam o tempo todo ativas, sendo executadas constantemente e aplicadas a fluxos de eventos dinâmicos. Para lidar com fluxos de eventos dinâmicos que podem apresentar novos dados a cada instante, um recurso comum entre essas linguagens baseadas em consulta são as operações sobre janelas deslizantes [Liu et al., 1999]. As janelas deslizantes são usadas para dividir o fluxo de eventos em segmentos, de modo que esses segmentos possam ser manipulados e analisados pelo sistema de forma assíncrona. Conforme Kajic [2010], existem 2 modelos de janelas deslizantes:

  - **Modelo orientado por tempo**: a janela de eventos é reavaliada somente no final de cada etapa de tempo. O padrão CQL (Continuous Query Language) adota esse modelo.
  - **Modelo orientado por tupla**: a janela é reavaliada toda vez que uma nova tupla chega. O padrão StreamSQL adota esse modelo.

- **Baseada em Regras**: Conforme descrito por Abraham [2005], um sistema baseado em regras consiste em estruturas "*if-then*", um conjunto de fatos e um módulo processador que controla a aplicação das regras. Essas declarações de regra "*if-then*" são usadas para formular as instruções condicionais que irão identificar uma situação. Uma regra "*if-then*" assume a forma "*se x é A, então y é B*", onde a parte "*if*" é chamada de antecedente ou premissa, enquanto a parte "*then*" é chamada de consequente ou conclusão. Buyya et al. [2016] classifica as regras em dois tipos:

  - **Regras de Produção**: As técnicas de regras de produção originaram-se na década de 1980 na área de sistemas especialistas, também conhecidos como sistemas baseados em conhecimento. As regras têm a forma "*se condição, então conclusão/ação*". Nessa estrutura, sempre que uma condição for modificada na base de regras ou algum dado novo for adicionado ao sistema, a conclusão/ação também deverá ser atualizada.
  - **Regras Evento-Condição-Ação**: A abordagem de regras de evento-condição-ação (*Event-Condition-Action - ECA*) foi desenvolvida para apoiar a necessidade de reagir a diferentes tipos de eventos que ocorrem em bancos de dados ativos [Paton and Díaz, 1999]. Existem três componentes em uma regra do tipo ECA:

    i. **Evento**: especifica o evento que aciona a invocação da regra. O evento em si pode ser uma composição de diferentes tipos de eventos e, nesse caso, ele é chamado de evento composto.
    ii. **Condição**: Consiste nas condições que precisam ser satisfeitas para executar a ação especificada. A condição é verificada apenas na ocorrência do evento especificado.

iii. **Ação**: especifica as ações a serem tomadas nos dados.

Dois usos típicos de regras ECA executados por EPSs incluem detectar e reagir a ocorrências de determinados tipos de padrões de eventos em tempo real e executar alguma lógica de negócios nos fluxos de eventos recebidos.

- **Baseada em Programação**: Os EPSs programáveis fornecem uma ampla gama de funcionalidades para processar eventos complexos ou compostos. Eles normalmente apresentam uma arquitetura de alto nível, utilizando um *framework* para facilitar e padronizar o desenvolvimento do código. Em outros casos, apresentam uma interface gráfica que auxilia o usuário a configurar as regras. Muitos sistemas CEP podem processar qualquer tipo de fluxo de eventos e gerar alertas, ou até consumir um serviço quando detectam um padrão de eventos específico. Os padrões de eventos são declarados usando estruturas predefinidas que definem regras de processamento, como o comprimento de janelas de tempo, tipo de eventos a serem monitorados e seus relacionamentos. Como operações comuns, pode-se destacar:

  - **Expressões de Eventos**: Consiste no bloco de construção fundamental de um EPS, permitindo a especificação de critérios de correspondência para um único evento.
  - **Filtro**: Componente responsável por reduzir o conjunto de eventos a serem processados pelo EPS, gerando um subconjunto de eventos relevantes. Por exemplo, remoção de dados errados ou incompletos.
  - **Transformação**: Alteração dos dados dos eventos de um formato para outro. Possui as seguintes classificações:
    * **Tradução**: Cada evento recebido é processado de maneira independente e depois é enviado para uma saída. Não considera eventos anteriores ou posteriores. Realiza uma operação do tipo "*um evento de entrada, um evento de saída*".
    * **Divisão**: Pega um único evento de entrada, realiza o processamento e envia vários eventos de saída para um fluxo. É classificado como uma operação de "*um evento de entrada, múltiplos eventos de saída*".
    * **Agregação**: Recebe um fluxo de eventos de entrada e produz um único evento de saída, resultado de uma função sobre os eventos recebidos. É considerada uma operação "*múltiplos eventos de entrada, um evento de saída*".
    * **Composição**: Recebe dois fluxos de eventos como entrada e os processa para produzir um fluxo de eventos de saída. Isso se assemelha ao operador de junção "*JOIN*" na álgebra relacional, exceto que ele une fluxos de eventos em vez de tabelas de dados. Caracteriza uma operação "*múltiplos eventos de entrada, múltiplos eventos de saída*".

- **Detecção de padrões de evento**s: Componente responsável por detectar padrões de alto nível examinando uma coleção de eventos (usando correlação, agregação, abstração, etc.). Esta operação pode ser dividida em três etapas:
  * **Pré-detecção**: Os tipos de padrões de eventos são validados (por exemplo, erros de sintaxe e gramática) e transformados em código EPL executável pelo compilador EPL.
  * **Detecção**: O EPL executa o código gerado na etapa anterior monitorando o fluxo de eventos de entrada. Quando uma correspondência do padrão de eventos é identificada, ele gera uma saída correspondente ao padrão identificado.
  * **Pós-detecção**: o EPL armazena todas as ocorrências do padrão de eventos gerada na etapa anterior ao banco de dados de eventos e notifica os destinos relevantes para que realizem as ações necessárias.

3 TRABALHOS RELACIONADOS

Essa seção tem por objetivo apresentar os estudos realizados pela comunidade científica utilizando processamento de dados na borda da rede, identificação de padrões, classificação, detecção de anomalias, identificação de pontos fora da curva, predição de valores e incerteza de dados. Com esse levantamento, pretende-se identificar as técnicas utilizadas e em qual situação melhor se encaixam, e quais problemas ainda não puderam ser resolvidos.

3.1 Metodologia de Pesquisa

A revisão da literatura utilizada como base para desenvolvimento do modelo STEAM adotou os princípios de revisão sistemática para obter reprodutibilidade e resultados de alta qualidade [Biolchini et al., 2005; Keele et al., 2007]. Durante todo o período de estudos, esse trabalho apresentou 3 etapas de pesquisa, iniciando com uma visualização abrangente sobre processamento de dados na borda, depois evoluiu para levantamento de técnicas e características específicas de aplicações executadas na *Fog*, e por fim, foram estudadas aplicações industriais com restrições de tempo de resposta e sem o apoio de serviços externos, como processadores de fluxo ou *CEP Engines*.

O objetivo das pesquisas foi obter uma visão geral do processamento de dados IoT tanto na borda da rede quanto com apoio de *middlewares*. O escopo da pesquisa bibliográfica abrangeu a seleção de artigos de conferências e periódicos relevantes, indexados nas seguintes bases científicas: ACM Digital Library[1], Google Scholar[2], IEEE eXplore[3], ScienceDirect[4], Springer[5] e Scopus[6].

As pesquisas aqui apresentadas foram realizadas nessas bases científicas entre janeiro de 2019 e março de 2021, procurando por artigos publicados a partir do ano 2016, tendo como objetivo identificar trabalhos contendo palavras-chave e alguns termos que deveriam aparecer no título ou resumo dos artigos selecionados. Obrigatoriamente, os trabalhos precisavam mencionar “Event Processing” e “Edge”, que são os domínios de maior interesse. Também foi necessário que os artigos apresentassem pelo menos um dos termos “analytics”, “prediction”, “noise”, “pattern” ou “enrichment”, que são os assuntos mais específicos. A *string* de busca elaborada para as pesquisas pode ser visualizada no quadro a seguir.

[1] `dl.acm.org`
[2] `scholar.google.com`
[3] `ieeexplore.ieee.org`
[4] `sciencedirect.com`
[5] `link.springer.com`
[6] `scopus.com`

“event processing” AND “edge” AND (“analytics” OR “prediction” OR “noise” OR “pattern” OR “enrichment”)

Sobre o conjunto de trabalhos retornados nas pesquisas, foram aplicados três critérios de exclusão: remoção de duplicidades, revisão do título e resumo, e por fim, análise do texto completo. Ao final dessa etapa restaram 29 artigos que exploravam o assunto da forma desejada, e estão listados na Tabela 1. Um resumo de cada um dos trabalhos é apresentado na subseção 3.1.1.

- **Remoção de duplicidades**: Todos os trabalhos foram colocados em uma lista, ordenados por título do trabalho e as duplicidades foram removidas.
- **Revisão do título e resumo**: Foi realizada a leitura dos títulos e dos resumos dos trabalhos, e foram removidos todos aqueles que não estavam relacionados a processamento de eventos na borda ou não abordavam predição, incerteza, detecção de padrões ou anomalias.
- **Análise do texto completo**: Nessa etapa foi feita a leitura completa dos artigos e foram mantidos apenas os trabalhos que além de abordarem processamento de eventos na borda, também utilizavam técnicas de análise de dados, identificação de padrões, predição, eliminação de ruídos ou enriquecimento de dados.

3.1.1 Estado da Arte

A partir do resultado da pesquisa e filtragem dos trabalhos, foi possível obter 29 artigos de grande relevância para o levantamento do estado da arte. Cada um desses trabalhos é apresentado a seguir, destacando suas principais características e técnicas utilizadas.

Liu et al. [2021] propôs um método de detecção de anomalias usando K-Means e C-Means sobre uma janela deslizante, executado em um nodo na borda da rede. Eles monitoraram vários sensores em tempo real dentro de uma mina subterrânea. Em Yin et al. [2020], os autores desenvolveram um algoritmo para detecção de anomalias usando intervalo de confiança, variação de intervalo e mediana de uma janela deslizante sobre um conjunto de dados de sensores. Este algoritmo calculado na borda da rede também pode distinguir a origem das anormalidades.

Com o objetivo de detectar anomalias em uma montadora de automóveis, De Vita et al. [2020] desenvolveram um *framework* arquitetural usando técnicas de FFT (Fast Fourier Transform) sobre vibração, ANN e K-Means. Bourelly et al. [2020] propôs um algoritmo para detec-

Tabela 1: *Corpus* da literatura contabilizando 29 trabalhos

Autor	Título	Ano
Liu et al.	Edge Computing for Data Anomaly Detection of Multi-Sensors in Underground Mining	2021
Yin et al.	A Distributed Sensing Data Anomaly Detection Scheme	2020
De Vita et al.	A Novel Data Collection Framework for Telemetry and Anomaly Detection in Industrial IoT Systems	2020
Bourelly et al.	A preliminary solution for anomaly detection in water quality monitoring	2020
YR and Champa	IoT Streaming Data Outlier Detection and Sensor Data Aggregation	2020
Liu et al.	Noise removal in the presence of significant anomalies for industrial IoT sensor data in manufacturing	2020
Khairnar et al.	Industrial Automation of Process for Transformer Monitoring System Using IoT Analytics	2020
Galanopoulos et al.	Improving IoT Analytics through Selective Edge Execution	2020
Greco et al.	An edge-stream computing infrastructure for real-time analysis of wearable sensors data	2019
Ali et al.	Middleware for Real-Time Event Detection andPredictive Analytics in Smart Manufacturing	2019
Symeonides et al.	Query-Driven Descriptive Analytics for IoT and Edge Computing	2019
Babazadeh	Edge analytics for anomaly detection in water networks by an Arduino101-LoRa based WSN	2019
Harth et al.	Predictive intelligence to the edge: impact on edge analytics	2018
Oyekanlu et al.	Towards statistical machine learning for edge analytics in large scale networks: Real-time Gaussian function generation with generic DSP	2018
Lujic et al.	Adaptive Recovery of Incomplete Datasets for Edge Analytics	2018
Küfner et al.	Lean Data in Manufacturing Systems: Using Artificial Intelligence for Decentralized Data Reduction and Information Extraction	2018
Widya et al.	A oneM2M-Based Query Engine for Internet of Things (IoT) Data Streams	2018
Bharath et al.	Large Scale Stream Analytics Using a Resource-Constrained Edge	2018
Portelli et al.	Leveraging Edge Computing through Collaborative Machine Learning	2017
Ilapakurti et al.	Adaptive edge analytics for creating memorable customer experience and venue brand engagement, a scented case for Smart Cities	2017
Tsai et al.	Distributed analytics in fog computing platforms using tensorflow and kubernetes	2017
Sirojan et al.	Intelligent edge analytics for load identification in smart meters	2017
Oyekanlu	Predictive edge computing for time series of industrial IoT and large scale critical infrastructure based on open-source software analytic of big data	2017
Harth et al.	Quality-aware aggregation and predictive analytics at the edge	2017
Alam et al.	Enabling Far-Edge Analytics: Performance Profiling of Frequent Pattern Mining Algorithms	2017
Xu et al.	EAaaS: Edge Analytics as a Service	2017
Bhargava et al.	Using Edge Analytics to Improve Data Collection in Precision Dairy Farming	2016
Kartakis et al.	Adaptive Edge Analytics for Distributed Networked Control of Water Systems	2016
Cheng et al.	Geelytics: Enabling On-Demand Edge Analytics over Scoped Data Sources	2016

ção de anomalias no monitoramento da qualidade da água. Eles usaram as técnicacs One-Class Classifier SVM, Isolation Forest, and Elliptic Envelope para detectar um conjunto predefinido de substâncias comumente consideradas perigosas e indicativas de um uso anômalo de água.

YR and Champa [2020] desenvolveu uma estrutura para agregação de dados e detecção de outliers, processando dados de 54 sensores, alegando que as imprecisões e ruídos dos sensores tornam difícil definir e antecipar o comportamento dos dados. Eles usaram Principal Component Analysis (PCA) e R-PCA. Liu et al. [2020] apresentou um algoritmo que calcula a distância

qui-quadrado em uma janela deslizante, realizando detecção de anomalias e remoção de ruído para dados de sensores IoT industriais em um processo de fabricação. Sensores instalados em um compressor coletaram dados de temperatura, velocidade e vibração. Para processar fluxos de dados de sensores vestíveis, [Greco et al., 2019] desenvolveu uma infraestrutura de computação de borda que permite a análise em tempo real de dados provenientes de sensores vestíveis. Eles usaram o algoritmo Hierarchical Temporal Memory, Node-RED, Apache Flink e Apache Kafka.

Um *middleware* para monitorar processos industriais usando análise de dados IoT foi proposto por Khairnar et al. [2020]. O trabalho consistiu na leitura de temperatura, vibração e umidade, além do treinamento de uma RNA (Rede Neural Artificial) para previsão de mau funcionamento de máquinas. Um algoritmo dinâmico e distribuído foi desenvolvido por Galanopoulos et al. [2020] com o objetivo de melhorar a execução de análises de dados em dispositivos IoT, com instâncias mais robustas rodando em cloudlets. Eles usaram técnicas KNN (K-Nearest Neighbours) e CNN (Convolutional Neural Network) para reconhecimento e classificação de imagens.

Um *framework* para análise de dados históricos combinada com análises e previsões de dados em tempo real foi proposto em Ali et al. [2019]. Os autores usaram Multiple Linear Regression, Support Vector Regression, Decision Tree Regression and Random Forest Regression. Em Symeonides et al. [2019] os autores propuseram um modelo de consulta declarativa com o objetivo de abstrair a complexidade da definição de regras de análise de dados. Eles propuseram uma gramática e um *framework* para computação de borda para alcançar latência, robustez e até mesmo requisitos de privacidade desejados.

O trabalho realizado por Harth et al. [2018] propõe um mecanismo de inteligência preditiva, leve e distribuído, executado na borda da rede. Se baseia na capacidade dos nós de borda monitorarem a evolução de diversas séries temporais vindas dos sensores, juntamente com dados de contexto, e determinar localmente através de técnicas de predição se uma informação deve ou não ser propagada adiante. Também trabalha com reconstrução de dados para evitar a solicitação de informações para outros nós, reduzindo o tráfego de rede. Utiliza a técnica de Média Móvel Ponderada Exponencialmente (Exponentially Weighted Moving Average - EWMA) para predição de um passo à frente, e também utiliza vetores de reconstrução para estimar um dado que não foi coletado. Como resultado, foi verificada uma diminuição significativa da comunicação de rede entre os dispositivos de borda com o custo de uma baixa taxa de erro, considerada aceitável pelos autores.

O objetivo do estudo realizado por Oyekanlu et al. [2018] era desenvolver um método de aproximação para obtenção de uma função de distribuição Gaussiana para dispositivos de baixo poder computacional e pouca memória. De acordo com os autores, muitos algoritmos de Inte-

ligência Artificial utilizam uma função de distribuição Gaussiana, que exige recursos computacionais não disponíveis em *hardwares* limitados, como microcontroladores ou Sistemas em um Chip (System on a Chip - SoC). Os autores aplicaram uma técnica de corte de uma onda senoidal programando um Processador de Sinais Digitais (Digital Signal Processor - DSP) na linguagem C. Como resultado, a comparação estatística usando o Erro Quadrático Médio (Mean Squared Error - MSE) entre a técnica proposta pelos autores em relação a função de distribuição Gaussiana do Matlab, mostrou uma baixa taxa de erro até o intervalo de confiança de 95%. Essa técnica comprova a viabilidade usar um aproximador de função Gaussiana em um dispositivo com restrição de recursos, como é o caso de muitos equipamentos de computação em borda de rede.

Um mecanismo recursivo semiautomático para recuperação de dados incompletos foi proposto por Lujic et al. [2018]. Esse mecanismo executado na borda da rede permite a recuperação eficiente de dados incompletos aplicando diferentes técnicas de predição para múltiplas lacunas de dados, utilizando especificações feitas pelo usuário. As técnicas utilizadas foram ARIMA, Suavização Exponencial e TBATS (Trigonometric seasonality, Box-Cox transformation, ARMA errors, Trend and Seasonal). Para avaliar o modelo proposto, os autores aplicaram a técnica sobre um conjunto de dados de consumo de energia elétrica em casas e prédios inteligentes, contendo várias lacunas devido a falhas no sistema de monitoramento. O resultado do experimento mostrou que a técnica é capaz de identificar múltiplas lacunas e recuperar conjuntos de dados incompletos, reduzindo erros de predição de dados em pouco mais de 82% e reduzindo o tempo de execução em até 52%.

O trabalho realizado por Küfner et al. [2018] foi a proposta de um modelo de análise descentralizado para classificar estados operacionais em plantas de produção utilizando Redes Neurais Artificiais (Artificial Neural Networks - ANN) em sistemas embarcados. O experimento consistiu em realizar a medição de vários indicadores de uma máquina elétrica, como corrente, potência e temperatura. Em seguida, foi feito o treinamento de um *Multilayer Perceptron* para classificar e identificar as etapas operacionais de um processo de manufatura. Para todos os modelos de avaliação, obteve-se uma taxa média de detecção entre 99,68% e 99,99% e uma precisão de classificação entre 99,82% e 100%.

Bhargava et al. [2016] propõem o uso de uma nova técnica de *Fog Computing* chamada *Edge Mining* para compactar dados agrícolas dentro de uma rede de sensores sem fio (Wireless Sensor Network - WSN). Ao contrário das técnicas convencionais de compactação de dados, o *Edge Mining* não apenas otimiza o uso de memória do dispositivo do sensor, mas também cria uma base para a possibilidade de resposta em tempo real do sistema. Nessa proposta foi utilizado o L-SIP, uma das técnicas do *Edge Mining* que fornece *feedbacks* em tempo real, permitindo a reconstrução precisa dos dados originais do sensor nas tarefas de compressão e descompressão de dados. Os algoritmos *Edge Mining* convertem os dados brutos dos sensores

em vetores de estado e reduzem o uso de memória, armazenando apenas as instâncias que não podem ser previstas a partir das estimativas anteriores usando um dado modelo de aproximação.

Um novo modelo colaborativo de aprendizado de máquina na borda da rede IoT foi proposto por Portelli and Anagnostopoulos [2017]. Os autores definiram dois objetivos para esse modelo: (i) obter resultados de predição altamente precisos sobre tarefas analíticas de regressão e (ii) proporcionar escalabilidade para a rede de borda IoT. Para isso, foi utilizado um modelo de Regressão Linear (Linear Regression - LR) e Quantização Vetorial Adaptativa (Adaptive Vector Quantization - AVQ). Funciona da seguinte maneira: cada sensor ou dispositivo computacional independentemente treina um modelo de regressão linear *on-line* local, que é então transferido para o sistema *back-end*. Isso garante a eficiência da rede, evitando transmitir os dados em si, transmitindo apenas os metadados correspondentes aos parâmetros do modelo.

Kartakis et al. [2016] desenvolveram um sistema de localização de vazamento de água de ponta a ponta, que utiliza processamento de borda e permite o uso de nós sensores movidos a bateria. Esse sistema combina um algoritmo leve executado na borda para detecção de anomalias com base em taxas de compressão, e um algoritmo de localização eficiente baseado na teoria de grafos. Também foram utilizadas as técnicas de média móvel (Moving Average - MA), filtro de Kalman e comparação com *thresholds*. O cenário de avaliação foi elaborado implantando sensores não intrusivos medindo dados vibracionais em uma rede de abastecimento de água real. Para a detecção de anomalias, foram gerados vazamentos e estouros na rede de forma controlada. Como resultado, a detecção de anomalias ocorreu dentro do tempo esperado e com localização física precisa, com erro máximo de 0,5 metros. Outro ponto a destacar foi a redução da comunicação de dados em 99% comparada com a comunicação periódica tradicional.

No trabalho desenvolvido por Widya et al. [2018] foi proposto um mecanismo de consulta distribuído baseado no *oneM2M*, chamado OMQ (oneM2M-based query engine). O objetivo principal do OMQ é fornecer funcionalidades cruciais de processamento de consultas para aplicativos IoT sobre o *middleware* oneM2M. A ideia consiste em dividir a consulta, identificar o nó IoT responsável por coletar os dados e enviar a parte da consulta ao respectivo nó IoT. Isso resulta em processamento distribuído a partir de uma consulta definida inicialmente no *middleware*. Para validar o mecanismo proposto, foi utilizado um conjunto de dados IoT reais e outro sintético, indicando que o OMQ pode processar com eficiência e eficácia as consultas do usuário, reduzindo o tempo de execução da consulta e o uso da largura de banda de dados entre os nós da rede IoT.

O trabalho realizado por Ilapakurti et al. [2017] consiste em uma abordagem inovadora para implementar um sistema HVAC inteligente (Heating, Ventilation and Air-Conditioning) que melhora tanto a experiência dos frequentadores do local quanto a eficiência operacional por meio da aplicação de tecnologias de Big Data, Processamento de Borda e sensoriamento IoT. O

sistema usa um algoritmo leve executado na borda para detecção de anomalias, tendo como base o filtro de Kalman. Para monitoramento das condições ambientais do local, usa uma matriz com gráficos do tráfego de pedestres. Foram feitos testes reais utilizando uma placa de prototipagem *SJOne-Board* que utiliza um microcontrolador ARM Cortex M3 rodando o sistema operacional FreeRTOS. A aplicação realiza monitoramento ambiental e ajusta a temperatura, umidade e até possui um odorizador para liberar perfume no ambiente, melhorando a experiência dos frequentadores do local.

Bharath Das et al. [2018] construíram um *framework* que distribui eficientemente tarefas de processamento de *streams* para os vários dispositivos de borda disponíveis na rede, com base na proximidade da fonte de dados do sensor e na capacidade de processamento do dispositivo. Para isso, utiliza o *Seagull*, que é uma extensão do *framework Cowbird*. O *Seagull* foi construído para permitir a análise de *streams* em larga escala em nodos de borda distribuídos. Para avaliar o *framework* proposto, foram realizados diversos experimentos analíticos com *streams* em ambientes de borda e *fog*, usando vários dispositivos IoT conectados para simular um cenário de cidade inteligente.

Um método para geração, configuração e gerenciamento automático de tarefas de processamento de *streams* foi proposto por Cheng et al. [2016]. Esse método foi planejado para executar análise de borda sob demanda sobre fontes de dados com escopo geográfico em uma configuração baseada em *Cloud-Edge*. Com base nesse método, os autores implementaram um novo sistema de processamento de *streams* chamado *Geelytics*. Para validar o sistema, foram criadas três aplicações: (i) identificação da menor temperatura em um determinado intervalo de tempo a partir de um conjunto de sensores distribuídos geograficamente; (ii) detecção de *outliers* das temperaturas lidas individualmente nos mesmos sensores; (iii) contagem de pessoas em uma aglomeração, utilizando detecção de rostos a partir de imagens de câmeras de vídeo. Como resultados das avaliações preliminares, foi possível identificar a redução da largura de banda em 99% na aplicação (iii) de reconhecimento e contagem de faces quando comparada a um cenário de processamento na nuvem. Na detecção de *outliers*, aplicação (ii), foi possível obter uma latência de apenas 10 milissegundos após o recebimento da temperatura a partir do sensor, e ocorreu economia de 65% do custo de processamento na aplicação (i), para identificação da menor temperatura, utilizando compartilhamento de resultados intermediários, evitando processamento duplicado.

O trabalho proposto por Tsai et al. [2017] consiste em uma plataforma de computação de névoa (*fog*) para análise distribuída. Essa plataforma integra recursos do data center aos dispositivos finais. Os autores se concentraram na implementação de uma plataforma que suportasse análises complicadas como o *Deep Learning*, para evitar o envio de uma grande quantidade de dados brutos para análise em poderosos data centers. Para realizar essa tarefa, eles utilizaram como tecnologias o *TensorFlow* como modelo de programação, *Docker* como ferramenta de

virtualização e *Kubernetes* como ferramenta de orquestração. Para validar a proposta, foram testadas duas aplicações: (i) uma aplicação para detecção de objetos em imagens em tempo real chamada YOLO (You Only Look Once), que implementa redes neurais, e (ii) outra aplicação para detecção de poluição do ar, trabalhando com 4 sensores distribuídos geograficamente. Os aplicativos foram divididos em várias partes e colocados em vários dispositivos. Os resultados mostraram que executar análises de maneira distribuída pode melhorar em até 54,1% o desempenho e os contêineres do *Docker* elevam em menos de 5% a carga em termos de CPU, memória RAM e número de imagens processadas por minuto.

O objetivo do trabalho realizado por Sirojan et al. [2017] foi preencher a lacuna existente na literatura em termos de redução de custos, escalabilidade e melhoria da precisão das metodologias NILM (Non-Intrusive Load Monitoring), propondo uma abordagem de análise de borda inteligente. Para isso, foi utilizada a análise de *wavelets* para detecção de transientes, extraindo componentes de alta frequência da forma de onda fornecida por um sensor. Depois, uma rede neural *feedforward* de três camadas foi utilizada para classificação de 4 padrões de carga: ventilador, lâmpada fluorescente, lâmpada incandescente e laptop. A proposta foi implementada na plataforma de *hardware embedded myRIO-1900* da fabricante National Instruments, que consiste em um ARM Cortex-A9 de núcleo duplo de 667 MHz. Os resultados experimentais mostraram que a abordagem proposta pode alcançar cerca de 99% de precisão na identificação da carga, bem como cerca de 99,9% de redução de dados comparado a um modelo de processamento em nuvem ou servidor centralizado.

O trabalho realizado por Oyekanlu [2017] foi elaboração de um banco de dados usando *SQLite* para monitoramento de condições de máquinas industriais, equipado com um dicionário pequeno o suficiente para caber na memória restrita de dispositivos de borda. Esse sistema tem como objetivo evitar que dados excessivos de máquinas industriais e de redes inteligentes sejam enviados para a nuvem, já que apenas relatórios de falhas e recomendações necessárias devem ser enviados. Para isso, foi utilizado o processador de sinais digitais (Digital Signal Processor - DSP) TMS320C2000 C28x da *Texas Instruments*, implementando algoritmos para cálculo de curtose, assimetria, raiz quadrada média e fatores de crista, analisando dados de vibração mecânica em máquinas industriais. Estes dados são então analisados utilizando o banco de dados embarcado, e somente condições especiais são reportadas ao sistema na nuvem, gerando uma redução muito significativa no tráfego de rede.

A proposta apresentada por Babazadeh [2019] consiste em uma nova arquitetura distribuída para análise de dados e algoritmos aplicados à detecção de anomalias de infraestrutura. Para isso, foram utilizadas técnicas para compressão de dados, filtro de Kalman e *thresholds* para detecção de anomalias. Os testes foram realizados utilizando uma rede de sensores *wireless* baseada na plataforma *Arduino101-LoRa*, utilizando toda a banda de frequência ISM (Industrial, Scientific and Medical) e também especificamente a frequência de 433 MHz. Como parte vital

do sistema, a biblioteca de compactação de dados sem perdas LZO (Lempel–Ziv–Oberhumer) foi adaptada para ser implementada nos nós sensores com recursos limitados. Os nós sensores avaliam sua própria taxa de compactação de dados em vez dos dados brutos para detectar anomalias. Depois, é aplicado um filtro de Kalman sobre a taxa de compactação de cada pacote de dados. Quando a taxa fica fora de determinados patamares de *threshold*, uma anomalia é detectada e é disparado um evento.

Harth and Anagnostopoulos [2017] propõem um modelo de processamento analítico de borda focado na qualidade e otimização do tempo, e que suporte uma modelagem preditiva para comunicação de dados eficiente. A ideia se baseia na capacidade dos nós de borda decidirem inteligentemente quando e quais dados devem ser liberados e processados, a fim de minimizar a sobrecarga de comunicação e maximizar a quantidade de resultados analíticos. A técnica utilizada foi regressão linear multivariada (Multivariate Linear Regression - MLR). Para validar a proposta, foi realizada uma simulação usando um conjunto de dados com parâmetros de qualidade do ar refletindo 12 nós sensores e atuadores (Sensing and Actuator Nodes - SANs) de uma rede de borda conectada através de nós de borda (Edge Nodes - EN).

Um estudo experimental sobre o perfil de desempenho de algoritmos de mineração de padrões frequentes foi realizado por Alam et al. [2017]. Os autores propuseram um *framework* teórico que não apenas suportasse a mineração de dados de *streams*, mas também garantisse a ciência de contexto e execução adaptativa de algoritmos de mineração de padrões frequentes em ambientes móveis. Para isso, foram utilizadas as técnicas de computação no extremo da borda (*far-edge computing*) e algoritmos de mineração de padrões frequentes (Frequent Pattern Mining - FPM). Como resultado, foi gerada uma tabela com dados de consumo de memória e tempo de execução de 21 algoritmos utilizando 3 conjuntos de dados (*datasets*) com características distintas, sendo executados em 3 *smartphones* diferentes.

O trabalho realizado por Xu et al. [2017], intitulado EAaaS (*Edge Analyticas as a Service*), consiste um serviço analítico escalável para permitir a análise de borda em tempo real em cenários de IoT. Neste trabalho, foi proposto um modelo analítico baseado em regras unificadas para facilitar os esforços de programação do usuário na especificação da lógica analítica baseada em regras. Além disso, também foi projetado e implementado um mecanismo de borda de alto desempenho para aplicar a análise baseada em regras nos fluxos de dados de entrada dos dispositivos. EAaaS foi implementada como parte da plataforma *IBM Watson IoT*. Os protocolos de comunicação utilizados entre os dispositivos e a nuvem foram MQTT e RESTful API. Para testar a proposta, foi criada uma aplicação para monitorar e analisar diversos fluxos de dados vindos de sensores implantados em lanchas de corrida. O resultado da análise era utilizado em tempo real pela equipe de apoio para passar instruções ao piloto. A utilização do EAaaS como *framework* para desenvolvimento de uma aplicação analítica para IoT proporcionou a redução do tempo em 40% em relação a utilização da plataforma padrão *IBM Bluemix Cloud*.

3.2 Discussão Sobre os Trabalhos Relacionados

A seleção dos trabalhos relacionados teve como objetivo fazer o levantamento do estado da arte para entender como a comunidade científica está trabalhando atualmente em relação a processamento de fluxos de dados e de eventos na borda da rede IoT. A Tabela 2 apresenta a relação de todos os trabalhos selecionados e uma compilação de diversas características. Nessa tabela, é possível visualizar facilmente a proposta de cada trabalho, os objetivos e as técnicas utilizadas. As características são discutidas nos parágrafos seguintes, com a apresentação de gráficos que facilitam o entendimento.

Tabela 2: Trabalhos relacionados e suas principais características

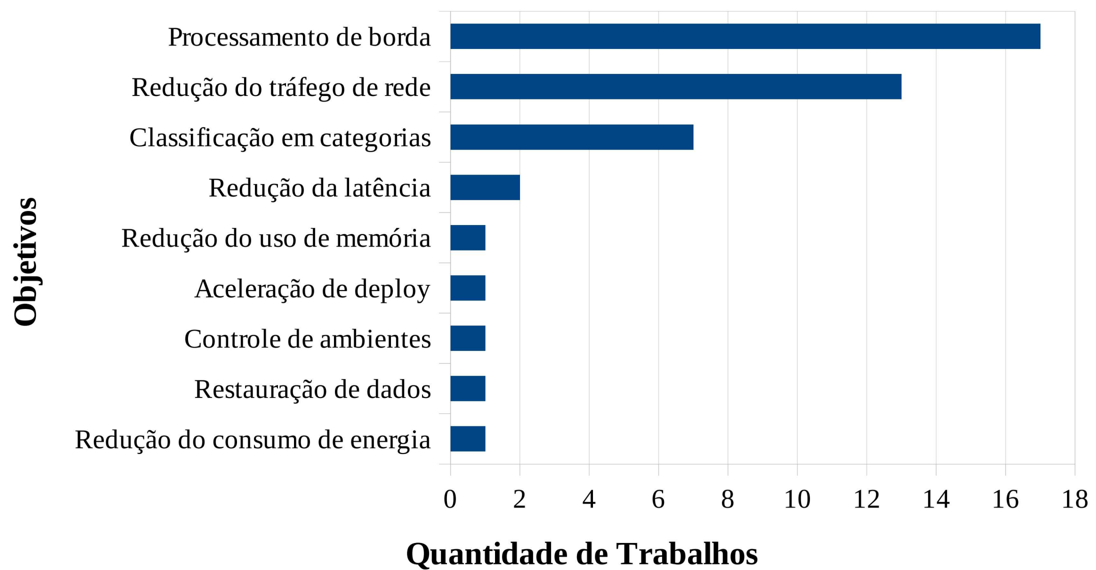
Autor	Ano	Proposta	Objetivo				Funcionalidade		
			Rede	Borda	Classe	Outro	Predição	Padrão	Outlier
Liu et al.	2021	Algoritmo	✓	✓	✓				✓
Yin et al.	2020	Algoritmo		✓					✓
De Vita et al.	2020	Arquitetura		✓				✓	✓
Bourelly et al.	2020	Algoritmo			✓			✓	
YR and Champa	2020	Framework	✓		✓		✓	✓	✓
Liu et al.	2020	Algoritmo		✓					✓
Khairnar et al.	2020	Arquitetura		✓		Latência	✓		
Galanopoulos et al.	2020	Algoritmo	✓	✓	✓			✓	
Greco et al.	2019	Arquitetura		✓				✓	✓
Ali et al.	2019	Framework		✓			✓	✓	
Symeonides et al.	2019	Framework		✓		Latência		✓	
Babazadeh	2019	Arquitetura	✓			Energia		✓	✓
Bharath et al.	2018	Framework		✓			✓	✓	
Harth et al.	2018	Arquitetura	✓				✓		
Küfner et al.	2018	Algoritmo			✓			✓	
Lujic et al.	2018	Algoritmo				Restauro	✓		✓
Oyekanlu et al.	2018	Algoritmo		✓					
Widya et al.	2018	Arquitetura	✓	✓					
Alam et al.	2017	Framework	✓	✓				✓	✓
Harth et al.	2017	Arquitetura	✓	✓			✓	✓	✓
Ilapakurti et al.	2017	Algoritmo				Ambiente	✓		
Oyekanlu	2017	Arquitetura	✓		✓			✓	✓
Portelli et al.	2017	Algoritmo	✓				✓		
Sirojan et al.	2017	Framework			✓			✓	
Tsai et al.	2017	Arquitetura	✓	✓				✓	
Xu et al.	2017	Arquitetura				Deploy			
Bhargava et al.	2016	Algoritmo				Memória		✓	✓
Cheng et al.	2016	Arquitetura	✓	✓				✓	
Kartakis et al.	2016	Algoritmo	✓	✓				✓	✓

Legenda:

Ambiente - Controle de ambientes internos
Borda - Processamento de borda ou distribuído
Classe - Classificação em categorias
Deploy - Acelerar deployment de aplicações
Energia - Redução do consumo de energia
Latência - Redução da latência
Memória - Redução do uso de memória
Outlier - Detecção de outliers
Padrão - Identificação de padrões
Predição - Predição de valores
Rede - Redução do tráfego de rede
Restauro - Restauração de dados incompletos

O objetivo principal dos trabalhos pode ser visualizado na Figura 9, onde se destacam os esforços para realizar processamento de dados na borda da rede, com 17 trabalhos publicados. O processamento na borda, além de reduzir o tráfego de rede entre os dispositivos e a nuvem, também faz com que uma tomada de decisão possa ser feita com mais rapidez. Isso ocorre pois a latência na comunicação tende a ser diminuída devido ao descongestionamento da rede ou até eliminada, pois os dados e algoritmos passam a estar presentes no próprio dispositivo local.

Figura 9: Objetivo principal dos trabalhos

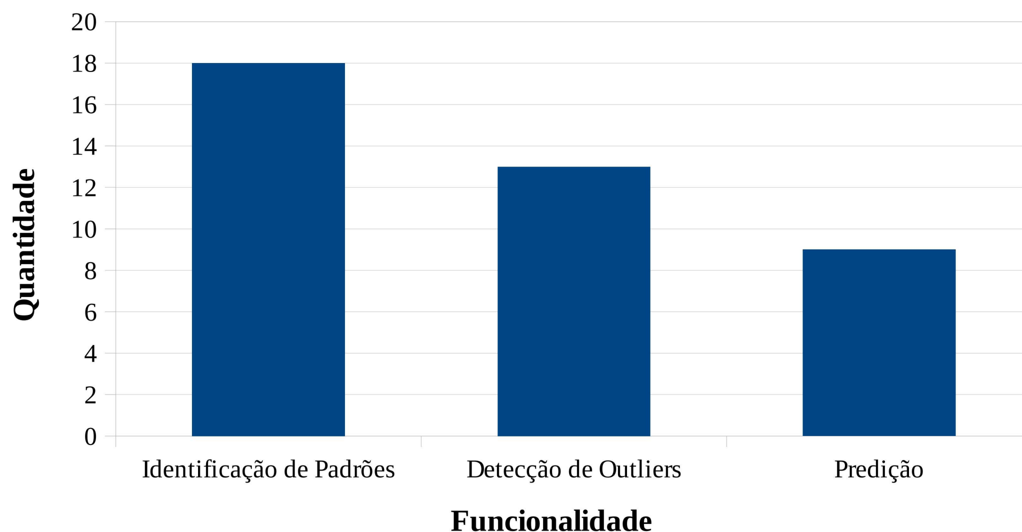


Em seguida, foram identificados 13 trabalhos voltados à diminuição do tráfego de rede, mostrando que o ambiente da Internet das Coisas e a computação ubíqua são um fato, e que os sensores e equipamentos de borda estão transmitindo cada vez mais dados, ocasionando sobrecargas e gargalos nas redes de transmissão. Na terceira colocação, foram identificados 7 trabalhos voltados a classificação de dados em categorias, com o principal objetivo de identificar situações como sobrecarga em rede de energia, etapa de operação em máquinas ou falhas em um processo produtivo. Dois trabalhos apresentaram como objetivo principal a redução da latência, justamente para disponibilizarem resultados mais rapidamente e conseguirem tomar decisões o mais perto possível do tempo real.

Apenas um trabalho apresentou uma proposta para recuperação de dados utilizando reconstrução de séries temporais, um trabalho para redução no consumo de memória em dispositivos com recursos limitados, outro trabalho voltado a redução no consumo de energia em dispositivos alimentados por bateria, um trabalho voltado a monitoramento e controle de ambientes internos, como lojas, escritórios e *halls* e também um único trabalho voltado a acelerar o *deployment* de aplicações que envolvem sensoriamento, monitoramento e processamento de dados em tempo real.

Para atingirem seus objetivos principais, os trabalhos apresentam funcionalidades bem definidas, conforme pode ser visualizado na Figura 10. Aproximadamente 31% dos trabalhos possuem alguma funcionalidade referente a predição de valores. A predição foi utilizada principalmente para antecipar a decisão de propagar dados entre nós da rede ou recompor dados incompletos em uma série temporal. Já a identificação de padrões foi utilizada em aproximadamente 62% dos trabalhos. Com ela, foi possível desenvolver aplicações para classificação de estados, compactação de dados e comunicação de rede eficiente. Em 44% dos trabalhos foram identificadas técnicas relativas a detecção de *outliers*, ou seja, identificação de dados que não condizem com o padrão de comportamento passado. Com essas técnicas foi possível desenvolver soluções para reconstrução de dados ausentes, identificação de anomalias em processos industriais e em infraestrutura e também reduzir o tráfego de rede.

Figura 10: Funcionalidades presentes nos trabalhos



3.3 Técnicas e Ferramentas Utilizadas na Borda

Os trabalhos selecionados na etapa de levantamento do estado da arte fazem uso de diversas técnicas ou ferramentas para detecção de padrões, identificação de *outliers* e predição de valores. As técnicas identificadas com maior frequência são apresentadas a seguir, com uma breve descrição.

- **Média Móvel**: Conhecida pela sigla MA (*Moving Average*), a média móvel consiste no cálculo da média aritmética simples sobre um subconjunto de tamanho fixo de uma série temporal. Os valores contidos neste subconjunto são permanentemente alterados à medida que novos dados são adicionados à série. Assim, a média móvel expressa a

tendência dos dados no intervalo de tempo analisado [Box and Jenkins, 1970]. Técnica utilizada por [Kartakis et al., 2016].

- **Média Móvel Ponderada Exponencialmente**: Representada pelo termo EWMA (*Exponentially Weighted Moving Average*), esta média móvel também é calculada sobre um subconjunto de dados passados de uma série temporal. A diferença é que a média não é simples, mas ponderada com fatores que decaem exponencialmente em relação ao valor mais recente. Dessa forma, a EWMA considera que o valor mais recente tem mais influência no resultado final da média, e o valor mais antigo afeta menos o resultado final [Gardner Jr, 1985]. Técnica utilizada por [Harth et al., 2018; Oyekanlu et al., 2018; Lujic et al., 2018].

- **Modelo Autorregressivo Simples ou Multivariado**: Um modelo autorregressivo (AR) prevê um comportamento futuro baseado no comportamento passado. O modelo especifica que a variável de saída depende linearmente de seus valores anteriores acrescido de um termo estocástico conhecido como ruído branco. Um modelo $AR(p)$ é um modelo autorregressivo em que valores passados são usados como variáveis de predição. Esses valores afetam os resultados de períodos seguintes de acordo com p, que é chamado de ordem do modelo [Box et al., 2015]. É classificado como "simples" quando é considerada apenas uma variável de entrada, e "multivariado" quando mais de uma variável é considerada no modelo. Técnica utilizada por [Portelli and Anagnostopoulos, 2017; Harth and Anagnostopoulos, 2017].

- **ARIMA**: Os modelos ARIMA (*Autoregressive Integrated Moving Average*), introduzidos por Box and Jenkins [1970], são os modelos estatísticos mais populares e eficazes para a previsão de séries temporais. Um modelo ARIMA combina três processos diferentes: uma função autorregressiva (AR), uma função de média móvel (MA) regressiva e uma parte integrada (I). Esses processos são baseados no princípio fundamental de que os valores futuros de uma série temporal são gerados a partir de uma função linear de observações passadas, adicionadas a termos de ruído branco. Técnica utilizada por [Oyekanlu et al., 2018; Lujic et al., 2018].

- **TBATS**: O modelo TBATS é utilizado para modelagem e predição de séries temporais complexas utilizando diversas técnicas: *Trigonometric seasonality, Box-Cox transformation, ARMA residuals, Trend and Seasonal*. Entre elas, destacam-se a transformada Box-cox, que trata os dados sem linearidade e faz com que a variância se torne constante. Em seguida, o modelo ARMA em residuais trabalha com o problema de auto correlação. Por fim, a Sazonalidade Trigonométrica lida com períodos sazonais não inteiros, períodos não aninhados e dados de alta frequência [de Livera et al., 2011]. Técnica utilizada por [Oyekanlu et al., 2018; Lujic et al., 2018].

- **Análise de *Wavelets***: A transformada *wavelet* é uma técnica matemática que pode decompor um sinal em múltiplos níveis de resolução menores, controlando os fatores de escala e deslocamento de uma única função de onda original. Em uma explicação simplista, com essa técnica é possível identificar as diversas componentes de uma onda nos parâmetros frequência, amplitude e fase. Com isso, é possível recompor um sinal, fazer previsões de valores e identificar *outliers* [Foufoula-Georgiou et al., 1995]. Técnica utilizada por [Sirojan et al., 2017].

- **Redes Neurais Artificiais**: Conhecidas pela sigla ANN (Artificial Neural Networks), são estruturas computacionais usadas para modelar uma ampla gama de problemas não lineares. Uma vantagem significativa dos modelos ANN sobre outros modelos não lineares é que as ANNs são aproximações universais, que podem estimar uma grande classe de funções com um alto grau de precisão. Redes retroalimentadas de camada única são o modelo mais utilizado para modelagem e previsão de séries temporais. Necessitam de uma etapa de treinamento antes de serem utilizadas. Técnica utilizada por [Küfner et al., 2018; Tsai et al., 2017].

- **Deep Learning**: Consiste em uma abordagem específica usada para construir e treinar redes neurais artificiais (ANN). O termo *deep* (profundo) se refere ao processamento dos dados em diversas camadas, até serem finalmente convertidos em uma saída. Do contrário, a maioria dos algoritmos modernos de aprendizado de máquina são considerados "superficiais" porque as entradas acabam passando por poucos níveis de processamento até se obter uma saída. O *deep learning* permite que modelos computacionais compostos de múltiplas camadas de processamento aprendam representações de dados com múltiplos níveis de abstração. Com essa técnica, é possível descobrir uma estrutura complexa em grandes conjuntos de dados usando o algoritmo *backpropagation*. Como exemplo de aplicação, as redes convolucionais são usadas no processamento de imagens, vídeo, fala e áudio, enquanto redes recorrentes são aplicadas em dados sequenciais, como texto e fala [Rusk, 2016].

- **Mineração de Padrões Frequentes**: Também conhecida como FPM (*Frequent Pattern Mining*), Han et al. [2007] a define como sendo uma técnica utilizada para identificar subconjuntos, subsequências ou subestruturas que aparecem em um conjunto de dados com frequência não inferior a um limite especificado pelo usuário. FPM foi proposta pela primeira vez por Agrawal et al. [1993] para análise de cestas de mercado, com o objetivo de encontrar relações entre itens. Por exemplo, pão e leite são comprados em uma mesma transação em muitos casos. Técnica utilizada por [Alam et al., 2017].

- **Filtro de Kalman**: De acordo com Welch et al. [1995], o filtro de Kalman é um conjunto de equações matemáticas que constitui um processo recursivo eficiente de estimação, uma vez que o erro quadrático é minimizado. Através da observação de uma variável denomi-

nada “variável de observação”, outra variável não observável, denominada “variável de estado” pode ser estimada eficientemente. Além disso, podem ser estimados os estados passados, o estado presente e até mesmo realizar previsões de estados futuros. O filtro de Kalman é um procedimento aplicável quando os modelos estão escritos sob a forma espaço-estado. Além disso, o filtro de Kalman permite a estimação dos parâmetros desconhecidos do modelo através da maximização da verossimilhança via decomposição do erro de previsão [Kalman, 1960]. Técnica utilizada por [Kartakis et al., 2016; Ilapakurti et al., 2017; Babazadeh, 2019].

- **Compressão de dados LZO**: O *Lempel–Ziv–Oberhumer* é um algoritmo de compactação de dados sem perdas, escrito em ANSI C. Oferece compressão muito rápida e descompressão extremamente rápida, de acordo com testes de desempenho realizados com diversos algoritmos de compressão e descompressão [Sadler and Martonosi, 2006]. Inclui diversos níveis de compactação, resultando em tempo de processamento mais lento ou mais rápido. Apresenta também uma opção chamada *miniLZO*, um subconjunto muito leve da biblioteca LZO com foco em aplicações destinadas a equipamentos com restrição de memória ou energia. A NASA utiliza este algoritmo embarcado na sonda *Curiosity*, atualmente em operação no planeta Marte. Técnica utilizada por [Babazadeh, 2019].

- **L-SIP**: O protocolo L-SIP (*Linear Spanish Inquisition Protocol*) reduz o custo de energia de uma rede de sensores sem fio (Wireless Sensor Network - WSN) transmitindo apenas informações inesperadas. A principal vantagem do L-SIP é que ele transmite um vetor de estados estimado em vez de leituras reais e individuais dos sensores. O L-SIP pode ser ajustado de acordo com uma precisão desejada, normalmente levando a menos pacotes transmitidos e, consequentemente, menos consumo de energia [Goldsmith and Brusey, 2010]. Técnica utilizada por [Bhargava et al., 2016].

Estas técnicas e ferramentas apresentadas são um exemplo de como problemas relacionados a cálculo de tendência de uma série temporal, predição de valores futuros, identificação de probabilidade ou incerteza, classificação de dados, reconhecimento de padrões e detecção de *outliers* podem ser resolvidos.

3.4 Lacunas e Oportunidades de Trabalho

O levantamento do estado da arte revelou alguns desafios a serem resolvidos nos próximos anos e tendências naturais na evolução dos sistemas de processamento de fluxos de dados ou eventos. As questões mais relevantes são apresentadas a seguir.

3.4.1 Incerteza de Dados e Eventos

Entre todos os trabalhos pesquisados, nenhum explorou questões relativas a probabilidade, incerteza ou confiabilidade dos dados processados, embora esse assunto tenha sido levantado por alguns autores como sendo uma área crítica e que necessita de estudos futuros [Wasserkrug et al., 2008], [Mao and Tan, 2015], [Akila et al., 2016] e [Dayarathna and Perera, 2018]. Nessa linha, Etzion and Niblett [2010] introduzem o termo “Processamento de Eventos Inexatos”, que podem ser atribuídos a 3 possíveis razões:

- **Incerteza se um evento realmente ocorreu**: Alguns eventos que ocorreram no mundo real podem não ter sido reportados, enquanto outros eventos que foram relatados podem não ter ocorrido. Várias razões podem induzir a esse tipo de incerteza.

  i. **Uma fonte mal-intencionada**: Um evento pode ser o resultado direto ou indireto de uma tentativa de sabotar o sistema, tanto do ponto de vista material (dano ou interferência física nos sensores), como virtual (ataque *hacker* contra o sistema informatizado).

  ii. **Uma fonte não confiável ou imprecisa**: Um produtor de eventos, como um sensor, pode funcionar incorretamente e indicar que um evento ocorreu, mesmo que não tenha ocorrido. Da mesma forma, um produtor de eventos pode falhar em sinalizar a ocorrência de um evento que de fato ocorreu. No caso de eventos derivados, problemas no design ou na implementação do agente que deriva esse evento podem fazer com que ele crie eventos derivados falsos ou não crie eventos logicamente válidos.

  iii. **Projeção de anomalias temporais**: Como um processador de fluxo de eventos (ESP) recebe dados das mais variadas fontes, inclusive distribuídas fisicamente, é possível que aconteça o processamento de eventos em uma ordem que não seja consistente com a ordem real da ocorrência dos eventos. Isso pode fazer com que um agente crie eventos derivados que não deveriam ter sido criados ou deixe de criar eventos que deveriam ter sido criados.

- **Conteúdo inexato no *payload* de um evento**: O conteúdo inexato ocorre quando o cabeçalho ou o *payload* de um objeto que representa um evento não são consistentes com uma ou mais características do evento que aconteceu na realidade. As razões para isso normalmente são fontes imprecisas, por exemplo, um sensor não calibrado, com mau contato elétrico ou sofrendo interferência mecânica ou eletromagnética pode gerar dados que não condizem com a realidade. Outras fontes de imprecisão que afetam sensores são a variação extrema de temperatura, umidade, taxa de amostragem inadequada ou medição indireta feita por um cálculo de aproximação.

- **Identificação inexata de eventos complexos**: As regras inseridas em um sistema CEP para identificar eventos indiretos não são imunes a falhas. Como toda a linguagem de programação, a EPL também é programada por uma pessoa, que irá fazer uma abstração da realidade, criar um modelo lógico-matemático e configurar as regras no sistema. Visto que programação não é uma ciência exata, dois fenômenos de falha podem ser observados nesse caso:

  i. **Falso positivo**: refere-se a casos em que um evento representando uma situação foi emitido por um sistema de processamento de eventos, mas a situação não ocorreu na realidade. Falsos positivos podem levar a reações que não deveriam ser realizadas, ser prejudiciais ou dispendiosas tanto do ponto de vista de recursos materiais quanto financeiros.

  ii. **Falso negativo**: refere-se a casos em que uma situação ocorreu na realidade, mas o evento que representa essa situação não foi emitido pelo sistema de processamento de eventos. Falsos negativos também são prejudiciais, e dependendo do ambiente, podem ser mais prejudiciais que os falsos positivos por deixarem de reportar uma situação crítica que exige reação imediata.

3.4.2 Componente Embarcado

Hoje em dia, a tecnologia de processamento de eventos é fornecida basicamente de duas maneiras. A primeira delas é como uma plataforma de processamento de eventos especializada, cujo objetivo principal é oferecer suporte a diversos aplicativos que precisem processar eventos. É bastante comum ser disponibilizada como SaaS (*Software as a Service*).

Como segunda forma, a funcionalidade de processamento de eventos é incorporada em outro software, *middleware* ou aplicativo como um componente empacotado, que oferece recursos de processamento de eventos através de uma biblioteca pronta e autônoma. Etzion and Niblett [2010] preveem que, no futuro, até 80% do mercado de processamento de eventos será embarcado de alguma forma. Isso permite que o processamento seja feito de forma distribuída, não apenas entre servidores ou estações de trabalho, mas também na borda e até mesmo nos próprios sensores, *wearables* ou dispositivos de uso pessoal.

3.4.3 Proatividade

O processamento de eventos hoje é usado principalmente de maneira reativa, no qual um sistema executa uma ação como resultado de um evento ou de uma série de eventos que já

ocorreram. Já na computação proativa, por outro lado, a ênfase está na detecção antecipada de estados indesejáveis para que possam ser eliminados ou, pelo menos, mitigados antes que surjam consequências indesejáveis, geralmente com impactos negativos. A computação proativa envolve outras tecnologias além do processamento de eventos, como a análise preditiva para identificar possíveis resultados futuros e decidir uma ação adequada a ser tomada.

Nessa linha, Fournier et al. [2015] afirmam que a "Computação Proativa Orientada a Eventos" é um novo paradigma, no qual uma decisão não é tomada devido a solicitações explícitas dos usuários, nem é feita como uma resposta a eventos passados. Em vez disso, a decisão é acionada de forma autônoma pela previsão de estados futuros. A computação proativa orientada a eventos supera as atuais arquiteturas reativas orientadas a eventos na capacidade de lidar com incertezas, eventos futuros e tomada de decisões em tempo real.

4 MODELO STEAM

Este capítulo apresenta e discute o modelo STEAM, assim como as principais contribuições desse trabalho. STEAM é um acrônimo formado a partir de *Stream Enrichment and Analysis in the Mist for the Internet of Things*. Consiste em um *framework* que fornece um conjunto de técnicas de análise de dados executadas na borda da rede por dispositivos com limitação de recursos, como baixo poder de processamento, pouca memória e conectividade lenta ou limitada. Tem como objetivos principais identificar padrões ou exceções, probabilidade ou incerteza de dados, filtragem de conteúdo, transformação ou padronização de dados, extração de estatísticas e previsão de valores futuros, além de propor a padronização da estrutura do pacote de dados e de nomenclaturas. Na entrada, é auxiliado por um módulo de abstração de dispositivos e coleta de dados, que realiza a captura dos dados vindos de sensores e demais dispositivos e os encaminha para processamento. Na saída, utiliza um conector de protocolos para disponibilizar os dados processados em diferentes padrões de comunicação, para que as aplicações cliente tenham acesso aos dados de forma padrão e transparente.

O capítulo está organizado da seguinte maneira. Inicialmente são apresentadas as diretrizes do projeto, discutindo os pontos chave que guiaram o projeto do modelo. Depois, é mostrada a arquitetura STEAM, iniciando com uma visão geral conceitual e seguindo com uma explicação detalhada de cada camada, módulo e sub-módulo, assim como as entradas, processos realizados e saídas. Por fim, é apresentada em detalhes a implementação do *framework* STEAM com documentação completa da sua API.

4.1 Diretrizes do Projeto

Esta seção apresenta as questões conceituais que guiaram o processo de elaboração da arquitetura e do modelo STEAM.

4.1.1 Formatos de Dados e Comunicação com Sensores

Em um ambiente de computação ubíqua como a IoT, os dados são coletados e transmitidos a partir das mais variadas fontes, como sensores ambientais ou corporais, telemetria de máquinas ou equipamentos, monitoramento de linhas de produção ou distribuição, estoque, logística, entre outros. Essas fontes de dados apresentam uma grande variabilidade em relação a formatos e comportamentos. Quando se trabalha com monitoramento de fenômenos físicos como temperatura, umidade e pressão, os dados normalmente são representados como um simples número

inteiro ou fracionário, e são transmitidos como uma série temporal, onde cada pacote representa uma medição. Como exemplo, a Figura 11 representa um *frame* de dados no formato ASCII emitido por um sensor de temperatura, transmitido através de um protocolo serial via RS-232. É iniciado pelo caractere STX (Start of Text), seguido por um número fracionário que representa a temperatura instantânea, depois a letra “C” que representa a unidade de medição em graus Celsius e finaliza com o caractere ETX (End of Text).

Figura 11: Pacote de dados simples

(STX)	2	3	.	6	C	(ETX)

Já o monitoramento de uma máquina em um processo industrial pode resultar no conjunto de diversas medições diretas ou indiretas, com dados brutos ou tratados, e normalmente são organizados em um formato estruturado e transmitidos utilizando um protocolo de transporte mais robusto. No exemplo da Figura 12 temos um conjunto de dados no formato JSON [Bray, 2014] representando resultados referentes a análise da composição química de um insumo, reportado por um equipamento cromatógrafo de gás. Aqui é possível identificar uma estrutura padronizada, com identificadores bem definidos e valores com tipos de dados específicos, como texto, números, data e hora. Nesse caso, o equipamento utiliza o protocolo TCP/IP para envio dos dados a um software dedicado, que coleta os resultados e os armazena em um banco de dados relacional para posterior processamento.

Figura 12: Pacote de dados complexo

```
{
  "Sample ID": 26048,
  "Instrument ID": "Saturn 2100D",
  "Acquisition Date": "2012-04-23T18:25:43.511Z",
  "Compounds": [
    {
      "Name": "Clorobenzeno",
      "Amount": 14.734,
      "Unit": "ppb"
    },
    {
      "Name": "Tolueno",
      "Amount": 11.626,
      "Unit": "ppb"
    }
  ]
}
```

Tendo em vista essa variabilidade nos formatos de dados e protocolos de comunicação, o modelo STEAM propõe um módulo de abstração de dispositivos e coleta de dados que reconhece os mais diversos formatos e protocolos das fontes de dados, captura os pacotes enviados e os disponibiliza de uma maneira padronizada para processamento.

4.1.2 Heterogeneidade e Limitações dos Dispositivos

Além da variedade de fontes de dados, este ambiente também tem como característica a heterogeneidade de dispositivos. Alguns microcontroladores embarcados, como AVR, ARM, Cortex-M, ESP, MIPS, PIC32, são muito limitados quanto a recursos computacionais. Utilizam processadores de 8, 16 ou 32 bits, executando a uma frequência de *clock* inferior a poucas dezenas de MHz, com um único núcleo, não permitindo múltiplas *threads* de execução. É comum que microcontroladores embarcados disponibilizem entre 1k a 256k bytes de memória flash programável e menos de 4k bytes de memória RAM. Em alguns casos, temos disponíveis apenas algumas dezenas de bytes de memória RAM. Normalmente as aplicações executadas por estes dispositivos são softwares embarcados, projetados e compilados especificamente para as plataformas. Também é possível utilizar sistemas operacionais dedicados como Apache Mynewt, Contiki, FreeRTOS, RIOT, Tiny OS, etc.

Já outros equipamentos são considerados computadores completos, como os *Single Board Computers*, que chegam a disponibilizar versões com processadores de até 64 bits, 1.5GHz de *clock*, 8 núcleos, 4GB de RAM, GPU e conexão Ethernet, Wi-fi e Bluetooth. Devido a maior complexidade de hardware destas plataformas que oferecem diversos módulos embarcados, elas necessitam de um sistema operacional mais elaborado para gerenciar acesso ao hardware, memória, gerenciamento de processos, entre outros. Alguns dos sistemas operacionais utilizados são Android, Minix, Raspbian, Ubuntu Mate, Windows Iot Core, entre outros.

Levando em consideração a grande variedade de dispositivos e suas limitações de recursos, o modelo STEAM é agnóstico quanto a arquitetura de hardware e econômico quanto ao consumo de recursos como memória e CPU.

4.1.3 Variabilidade no Processamento de Dados

Os dados capturados a partir do monitoramento de um ambiente e transmitidos por uma rede de sensores podem ser muito variados. Cada fenômeno monitorado pode apresentar um padrão de comportamento distinto ao longo do tempo, e em muitos casos, esses padrões se repetem ciclicamente. A análise dos dados em tempo real revela informações valiosas para o entendimento dos processos. Ao processar os dados brutos na origem, é possível identificar situações que necessitem de uma tomada de decisão imediata, podendo ser o simples encerramento de um processo industrial, o disparo de uma ordem de carregamento ou em alguns casos, o acionamento de um alarme de incêndio. Com a análise de dados, é possível identificar padrões ou exceções, probabilidade ou incerteza de dados, filtragem de conteúdo, transformação ou padronização de dados, extração de estatísticas e previsão de valores futuros.

No sentido de facilitar o processamento dos dados e extrair informações relevantes ao processo, o modelo STEAM disponibiliza um módulo de análise de dados em tempo real com as seguintes funcionalidades, mas não se limitando a elas: (i) Reconhecimento de padrões; (ii) Detecção de anomalias; (iii) Predição; (iv) Média móvel simples e exponencial; (v) Desvio padrão; (vi) Incerteza ou confiabilidade; (vii) Filtragem; (viii) Transformação. Algumas destas funcionalidades são relativamente simples de serem implementadas, como médias móveis, desvio padrão, filtragem e transformação. Porém, reconhecimento de padrões, detecção de anomalias, predição e incerteza exigem algoritmos mais elaborados. Existem muitas bibliotecas *open source* portáveis desenvolvidas por terceiros utilizando a linguagem C/C++/Python. Elas podem ser utilizadas parcial ou integralmente, ou ainda serem modificadas durante o desenvolvimento do protótipo.

4.1.4 Variedade de Protocolos de Comunicação IoT

O ambiente IoT é muito heterogêneo quanto a formatos de dados e protocolos de comunicação. Isso não é exclusividade apenas dos sensores e dispositivos, mas também das aplicações que consomem os dados gerados, tanto *middlewares* quanto aplicações cliente final. Cada protocolo utilizado para transporte dos dados entre a borda da rede e a aplicação final possui características distintas quanto a formato de pacotes e sequência de mensagens transmitidas. Alguns protocolos são mais elaborados, mantendo a conexão ativa ou garantindo um QoS, enquanto outros são mais simples, apenas despachando um pacote de dados endereçado a um destino sem garantir a entrega.

Para permitir a comunicação de dados entre a borda e as aplicações disponíveis na rede, o modelo STEAM prevê um módulo conector de protocolos que encapsula os dados e os transmite utilizando protocolos padrão, como MQTT, XMPP, LIME, AMQP, CoAP, STOMP, entre outros.

Em resumo, as seguintes diretrizes guiaram o projeto do modelo STEAM:

- Abstração de dispositivos e aquisição de dados, permitindo a leitura de dados das mais variadas fontes, formatos e protocolos. As fontes não precisam ser modificadas;

- Agnóstico quanto a arquitetura de hardware e econômico quanto ao consumo de recursos computacionais, visto que dispositivos de borda são heterogêneos e limitados em recursos como memória, CPU e I/O;

- Utilização de diversas técnicas de análise de dados com objetivo de enriquecer os fluxos de dados brutos com informações adicionais de interesse das aplicações cliente, além da capacidade de realizar tomada de decisão e disparar eventos;

- Disponibilização dos dados processados em uma estrutura padronizada, mas em diferentes protocolos de comunicação, para que as aplicações cliente tenham acesso aos dados de forma padronizada e transparente.

4.2 Arquitetura STEAM

Essa seção apresenta a arquitetura do modelo STEAM descrevendo suas camadas, módulos e componentes. Inicialmente, é apresentada uma visão geral para introdução dos conceitos e entendimento dos fluxos macro, destacando as diferenças do modelo tradicional com dependência de computação em nuvem em relação ao modelo STEAM. Em seguida, cada módulo é descrito de forma detalhada, apresentando suas entradas, processamentos, saídas, e como se comunicam entre si.

4.2.1 Arquitetura IoT versus STEAM

As aplicações IoT tradicionais apresentam uma arquitetura dependente de processamento em nuvem. Já o modelo STEAM foi elaborado em uma arquitetura que não dependa obrigatoriamente de serviços de computação em nuvem, embora possa trabalhar em conjunto com processamento externo. A Figura 13 ilustra o comparativo das duas arquiteturas, e na sequência, são apresentadas as 4 camadas e suas características.

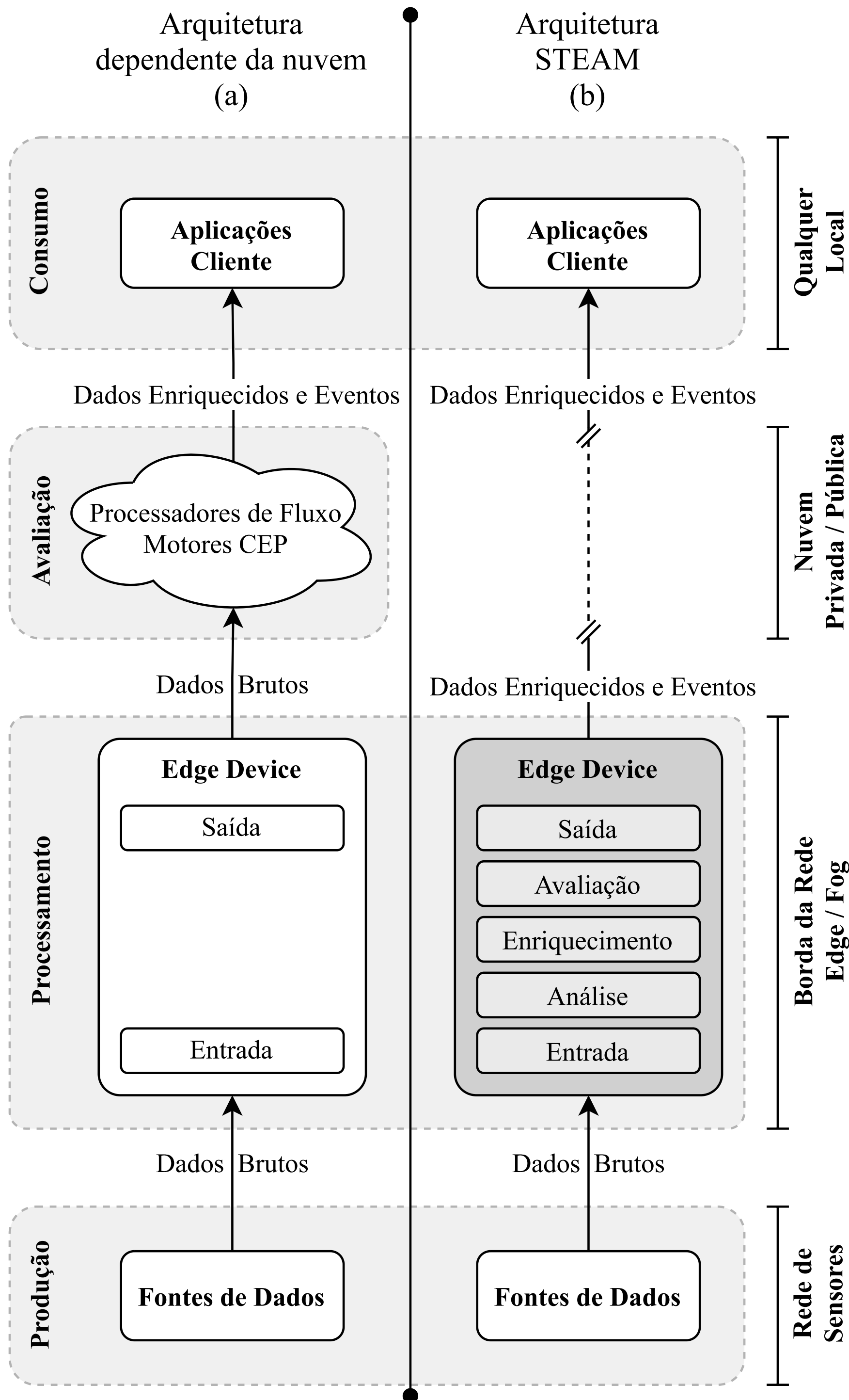
- **Produção**: Uma aplicação IoT típica começa com a produção de dados, representados como fontes de dados brutos genéricos, transmitidos por redes de sensores. Os dados podem ser produzidos tanto por sensores quanto por aplicativos simples, incorporados em hardware dedicado. A arquitetura STEAM não interfere nem modifica essa camada.

- **Processamento**: Depois de coletados, os dados brutos são processados por um *gateway* na borda da rede. Em uma arquitetura IoT tradicional, o *gateway* apenas encapsula os dados em um protocolo padrão e os retransmite para *middlewares* disponíveis em nuvens públicas ou privadas. Já na arquitetura STEAM, o processamento e análise dos dados são feitos na borda, aplicando os conceitos de *Fog Computing*.

- **Avaliação**: Na arquitetura tradicional, a etapa de análise dos dados para detecção de eventos ocorre na camada de avaliação. Essa tarefa é executada por serviços como Processadores de Fluxo e *CEP Engines* hospedados na nuvem. Na arquitetura STEAM essa camada é opcional, visto que a tarefa de avaliação e detecção de eventos ocorre no dispositivo de borda.

- **Consumo**: A camada final é o consumo de dados, realizado por aplicações cliente sendo executadas em qualquer local, tanto LAN, cloud ou até mesmo em dispositivos móveis usando uma rede de telefonia. Mais uma vez, a arquitetura STEAM não interfere nem modifica essa camada, garantindo uma integração transparente entre a produção e o consumo dos dados.

Figura 13: Comparativo entre as arquiteturas. Em (a) temos a arquitetura IoT tradicional, dependente de computação em nuvem. Nela, os dados são gerados pelos sensores, coletados por dispositivos de borda e transmitidos diretamente para serviços de processamento e análise de dados na nuvem. Por fim, são consumidos por aplicações cliente. Já na arquitetura STEAM (b), os dados brutos gerados pelos sensores são coletados e processados na borda, disponibilizando dados enriquecidos e eventos diretamente para as aplicações cliente através de protocolos padrão.



4.2.2 Modelo Detalhado

O modelo STEAM é apresentado na Figura 14 em cinco camadas distintas, distribuídas, independentes, mas interconectadas entre si, descritas a seguir.

i. **Abstração de Dispositivos e Coleta de Dados**: Visto que as fontes de dados podem conter formatos diversos e utilizar protocolos de comunicação variados, a camada de abstração de dispositivos e coleta de dados tem por objetivo oferecer um interfaceamento transparente com os dispositivos e sensores. Para isso, utiliza protocolos abertos ou proprietários, como RS-232, Modbus, OPC, TCP, UDP, HTTP, entre outros para realizar a captura dos dados que se apresentam nos mais diversos formatos, como ASCII, JSON, XML e até binário. Assim que os dados são recebidos, eles são organizados em um formato padrão e disponibilizados para o módulo de análise. Este formato padrão consiste em um JSON contendo os atributos "id", "value", "unit"e "timestamp". No exemplo da Figura 15, temos um *frame* de dados em formato ASCII transmitido via comunicação seria RS-232 representando a medição da temperatura 23,6 graus Celsius. Em seguida, este *frame* é convertido em um formato padrão JSON, alimentando os atributos com os valores disponíveis.

ii. **Análise de Dados**: O módulo de análise de dados consiste em um conjunto de sub módulos que realizam as mais variadas operações sobre os dados brutos previamente recebidos e padronizados, com o objetivo de identificar padrões de comportamento ou exceções, calcular a probabilidade ou incerteza de um dado, realizar filtragem de conteúdo com base em diversos critérios, transformar dados em diferentes escalas ou padrões, extrair estatísticas ou prever comportamento futuro. Conforme pode ser visualizado na Figura 16, o módulo de análise recebe um pacote de dados que foi previamente padronizado pela camada de abstração de dispositivos e comunicação de dados. Sobre este pacote são aplicadas diversas técnicas de análise, como detecção de *outliers*, filtragem, predição, entre outros, e o resultado é disponibilizado de forma padronizada como um objeto JSON contendo um único atributo e seu respectivo valor. Este resultado é usado posteriormente na camada de enriquecimento. Os sub módulos da camada de análise utilizam as técnicas identificadas durante o levantamento do estado da arte, descrito na seção 3.3, mas não se limitam a elas. Atualmente, são disponibilizadas as seguintes funcionalidades de análise:

   - **Pré processamento**: Modificação dos dados de entrada para se enquadrarem em algum padrão como nomenclaturas, conversão de unidades, transformação de escalas lineares, logarítmicas ou exponenciais, normalização, etc;
   - **Filtragem**: Técnica para descartar valores que não se enquadram em um ou mais critérios, podendo ser tanto fixos quanto variáveis. Por exemplo, a filtragem pode considerar *thresholds* constantes pré-estabelecidos ou variáveis, como valores percentuais em relação a máximos ou mínimos, valores fora da faixa média mais ou menos desvio padrão, entre outros;

Figura 14: Na arquitetura STEAM, os dados gerados pelos sensores na camada de produção são processados e analisados por uma aplicação executada na borda da rede. Esses dados são então enviados para aplicações cliente, *middlewares* ou integradores, podendo ser executados tanto em uma nuvem pública quanto privada.

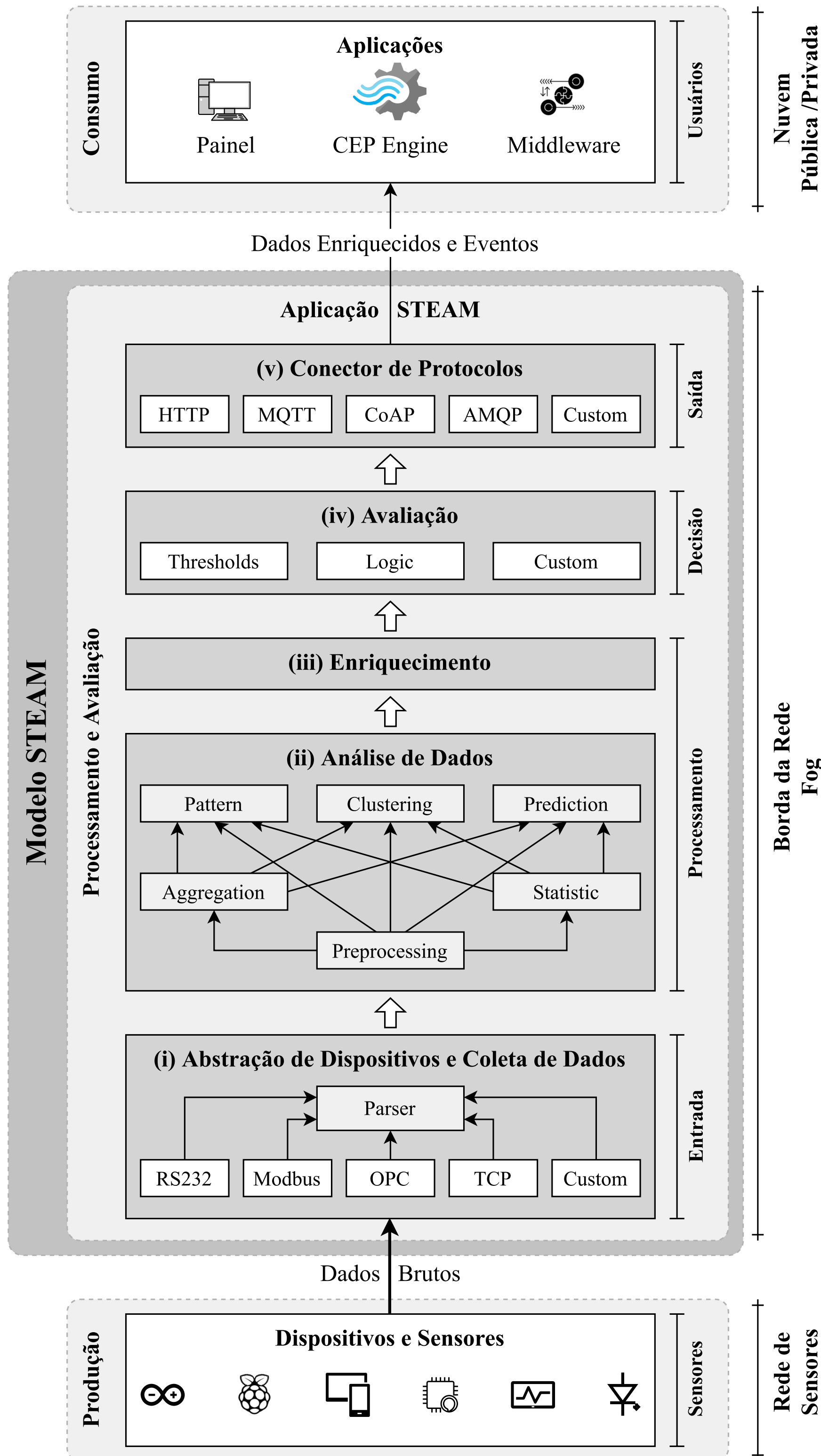
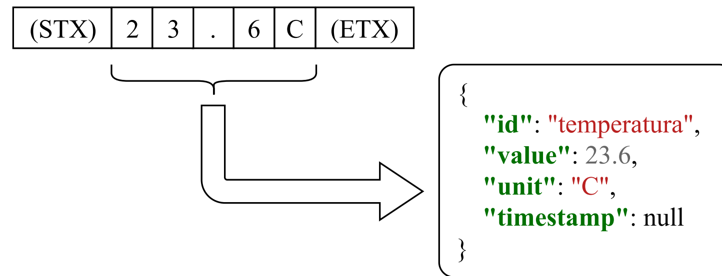
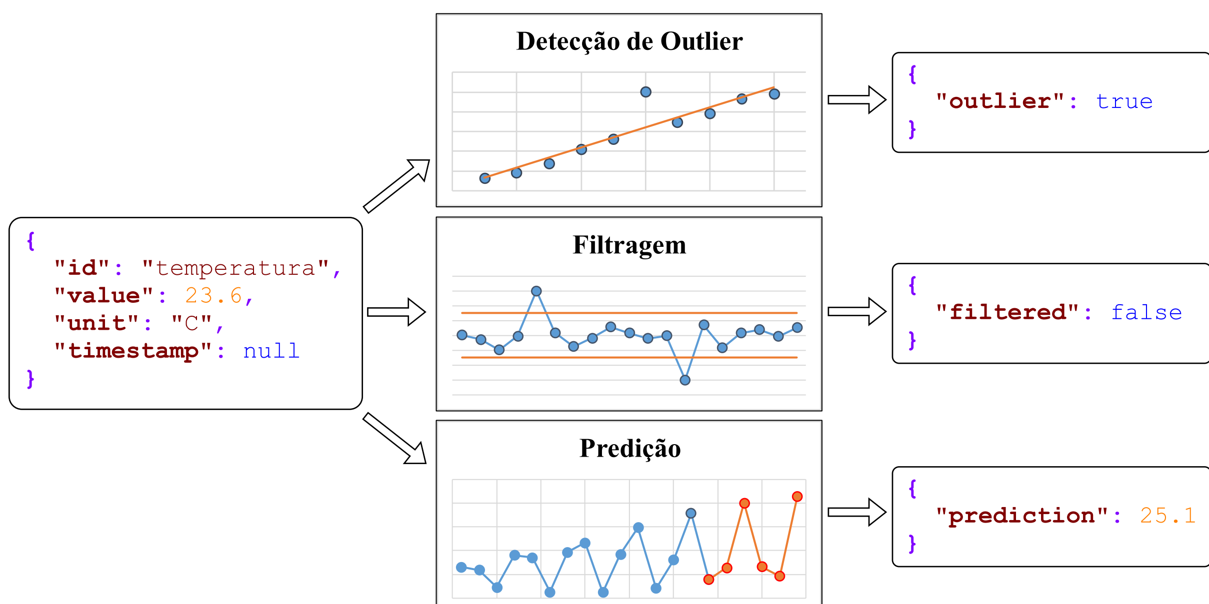


Figura 15: Dados recebidos em formato ASCII e convertidos para JSON



- **Desvios**: Aplicação de técnicas para calcular indicadores de dispersão sobre os dados de entrada, como variância e desvio padrão. Pode obedecer uma janela deslizante de tempo ou de quantidade de valores, ou seja, o cálculo pode utilizar os últimos n dados ou os dados do instante atual até x unidades de tempo atrás;
- **Médias**: Cálculo do valor médio dos dados de entrada utilizando diversas técnicas, como por exemplo, média aritmética, média ponderada simples ou exponencial, média harmônica, média geométrica, moda, mediana, entre outras. Também é comum utilizar janelas deslizantes sobre os dados de entrada;
- **Incerteza**: A determinação da incerteza de um valor é um processo que envolve muitas variáveis, como qualidade do sensor, resolução do conversor analógico/digital além da possibilidade de sofrer influência de fatores ambientais como temperatura, pressão, umidade, luminosidade, etc. O sub módulo de incerteza tem por objetivo determinar um grau de confiabilidade do valor medido em relação a critérios específicos da aplicação;

Figura 16: Análise dos dados e extração de informações relevantes



- **Detecção de Anomalias**: O objetivo desse sub módulo é detectar quando um dado não corresponde a um padrão pré-estabelecido fixo ou a um comportamento dinâmico da série temporal. Em alguns casos, é necessário treinar previamente um modelo ou ajustar parâmetros em tempo real, visto que a natureza do comportamento dos dados pode mudar com o tempo sem que isso represente uma anomalia;
- **Reconhecimento de Padrões**: Este sub módulo aplica técnicas matemáticas e estatísticas para identificar padrões de comportamento dos dados, como por exemplo, estabilidade, tendência ascendente, descendente e suas respectivas taxas de evolução, mudança de tendência, uma sequência de valores abaixo ou acima da média, etc;
- **Predição**: A partir da modelagem do padrão de comportamento passado, este sub-módulo oferece a capacidade de prever um valor futuro a um ou vários instantes a frente do tempo atual, juntamente com um grau de confiabilidade de que esta predição ocorra. Aqui é possível utilizar técnicas como ARIMA, TBATS ou redes neurais;

iii. **Enriquecimento**: A camada de enriquecimento consiste em adicionar informações relevantes ao pacote de dados original antes que ele seja transmitido para a etapa de avaliação. Já que durante o levantamento do estado da arte não foi identificada nenhuma iniciativa de padronização de nomenclaturas, o modelo STEAM propõe uma relação de identificadores com nomes sugestivos e seus respectivos tipos de dados. As nomenclaturas utilizadas para os atributos de saída podem ser visualizadas na Tabela 3.

Tabela 3: Padronização das nomenclaturas utilizadas pelo módulo de enriquecimento

Atributo	Obrigatório	Tipo	Descrição
id	sim	texto	Identificador do pacote ou da fonte de dados
value	sim	qualquer	Valor do dado
unit	não	texto	Unidade de medição do dado
timestamp	não	data-hora	Data e hora da leitura do dado
average	não	numérico	Média aritmética simples
ewaverage	não	numérico	Média exponencialmente ponderada
median	não	numérico	Mediana
variance	não	numérico	Variância
stdeviation	não	numérico	Desvio padrão
certainity	não	numérico	Fator de confiabilidade
slope	não	numérico	Declividade
outlier	não	booleano	Valor fora do padrão
filtered	não	booleano	Valor fora dos critérios de aceitação

Além disso, o módulo de enriquecimento precisa garantir um formato padrão para o pacote de dados de saída. Mais uma vez, não foi identificado nenhum trabalho que definisse uma estrutura padronizada. Dessa forma, o modelo STEAM também propõe uma estrutura de dados que pode ser visualizada na Figura 17, documentada como um JSON *schema* [Pezoa

et al., 2016]. Este padrão JSON *schema*[1] ainda não está oficializado, mas está em estudo pela IETF[2]. Ele é dividido em dois grupos:

- *properties*: define os nomes e tipos das principais propriedades do pacote de dados, sendo que apenas "id" e "value" são obrigatórias. Estas propriedades vêm do pacote gerado pela camada de abstração de dispositivos e comunicação de dados;
- *aditionalProperties*: define a lista de propriedades secundárias, com seus nomes e tipos. Todas elas são opcionais, visto que são geradas sob demanda pela camada de análise;

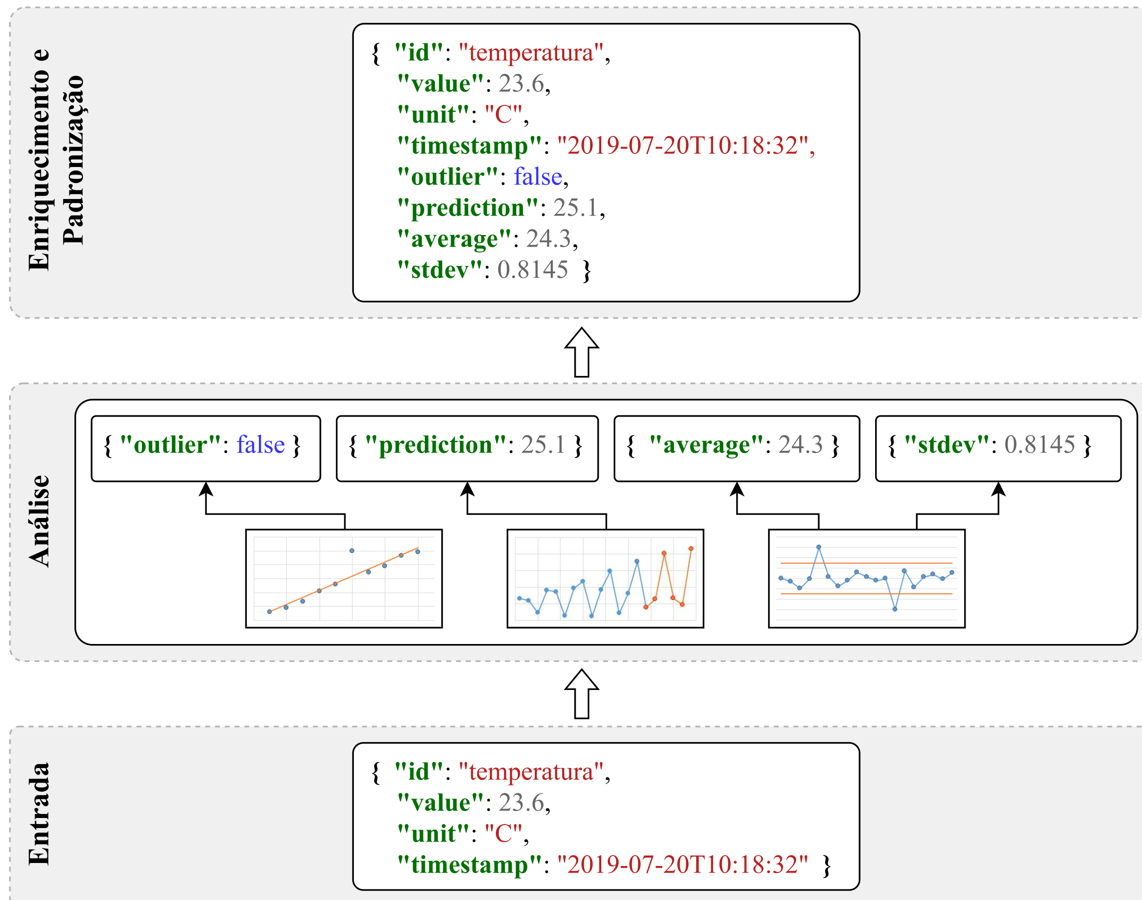
Figura 17: *Schema* JSON que define o formato padrão de saída

```
{
  "type": "object",
  "properties": {
    "id":                 {"type": ["integer", "string"]},
    "value":              {"type": "any"},
    "unit":               {"type": ["string", "null"]},
    "timestamp":          {"type": ["string", "null"],
                           "format": "date-time"}
  },
  "required": ["id", "value"],
  "additionalProperties": {
    "properties": {
        "average":     {"type": "number"},
        "ewaverage":   {"type": "number"},
        "median":      {"type": "number"},
        "variance":    {"type": "number"},
        "stdev":       {"type": "number"},
        "certainity": {"type": "number"},
        "slope":       {"type": "number"},
        "outlier":     {"type": "boolean"},
        "filtered":    {"type": "boolean"}
    }
  }
}
```

A Figura 18 ilustra um exemplo fictício do processo de enriquecimento e padronização dos dados. Como entrada, são utilizados o pacote disponibilizado pela camada de abstração de dispositivos e comunicação de dados e todos os dados gerados pela camada de análise. Então, é montado um único pacote JSON que em seguida é disponibilizado para a camada de conector de protocolos.

[1] https://json-schema.org
[2] https://tools.ietf.org/id/draft-handrews-json-schema-validation-01.html

Figura 18: Enriquecimento e padronização dos dados



iv. **Avaliação**: A quarta camada avalia regras, lógica, comparação de limites e realiza análises personalizadas para fornecer detecção de eventos e tomada de decisões. Por exemplo, nesta etapa, é possível identificar comportamentos, ruídos, outliers e decidir se devem ou não ser enviados dados ou mensagens de alerta para aplicativos clientes ou comandos para atuadores localizados na rede de sensores em situações específicas. A camada de avaliação também pode atuar como um filtro, visto que nela é possível implementar regras que identificam comportamento normal ou anormal, e somente transmitir dados no caso de anormalidade ou em situações bem específicas. Essa etapa também permite a elaboração de regras customizadas.

v. **Conector de Protocolos**: A camada de saída do modelo STEAM é o conector de protocolos. Visto que ela é responsável por disponibilizar os dados enriquecidos para as aplicações, são utilizados protocolos padrão no ambiente IoT para oferecer o máximo de interconectividade entre as camadas de Rede de Sensores e LAN/Nuvem. Utiliza como entrada os pacotes de dados no formato JSON disponibilizados pelo módulo de Enriquecimento e os adiciona como *payload* de diversos protocolos. De acordo com estudos realizados por Al-Fuqaha et al. [2015]; Yassein et al. [2016]; Karagiannis et al. [2015]; Hedi et al. [2017], os protocolos mais utilizados são AMQP, CoAP, DDS, MQTT, RESTFUL Services, WAMP

e XMPP. A Figura 19 ilustra a pilha de protocolos utilizada por dispositivos IoT, desde sua camada física até chegar na aplicação final [Naik, 2017]. No caso do módulo Conector de Protocolos disponibilizado no modelo STEAM, a interface com a camada física se dá através da Ethernet ou Wi-Fi, e a interface com a camada de aplicação se dá através de protocolos padrão.

Figura 19: Pilha de protocolos IoT

Camada de Aplicação	Aplicação IoT					
	HTTP	XXMP	DPWS	SOAP	CoAP	MQTT
Camada de Transporte	TLS			DTLS		
	TCP			TCP/UDP		
Camada de Rede	6LoWPAN			IPSec		
	RPL					
	IPv4 ou IPv6					
Camada de Enlace Camada Física	Ethernet	Wi-Fi	Bluetooth BLE	RFID NFC	GSM LTE	RS-232

Fonte: Adaptado de [Naik, 2017]

5 METODOLOGIA

Esse capítulo apresenta a metodologia de avaliação do modelo STEAM. A seção 5.1 descreve a implementação de um *framework* com diversas classes e funções. Em seguida, na seção 5.2 são apresentadas as métricas de avaliação da *Aplicação 1* executada sobre uma arquitetura *Edge-Cloud*. Na seção 5.3, são apresentadas as métricas utilizadas para avaliação da *Aplicação 2* em um cenário envolvendo apenas *Edge*.

5.1 Implementação do *Framework* STEAM

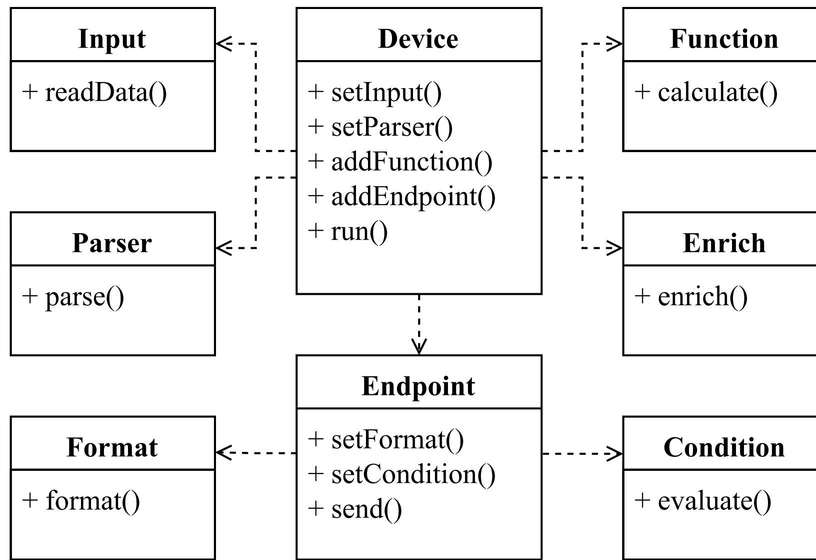
Para validação do modelo STEAM, foi desenvolvido um *framework* 100% funcional, com o objetivo de permitir o desenvolvimento de aplicações IoT para serem executadas na borda da rede, oferecendo análise de dados em tempo real, tomada de decisão e enriquecimento de fluxos de dados. Implementado em Python 3.8 por ser uma linguagem leve e portável para várias plataformas, foi estruturado em diversas classes e funções. Além disso, existem muitos algoritmos, classes e bibliotecas desenvolvidos em Python por terceiros que são disponibilizados como *open source*. O projeto STEAM está hospedado no GitHub, e seu código fonte está acessível no endereço `https://github.com/steam-project/steam`. A seguir, são apresentados os detalhes da implementação do *framework* STEAM, como diagrama de classes, diagrama de sequência e algoritmos.

5.1.1 Diagrama de Classes

A Figura 20 apresenta o diagrama de classes do *framework* STEAM, e na sequência, a descrição de cada classe e suas funcionalidades.

- **Device**: É a classe principal de uma aplicação STEAM, armazenando os dados, processando funções lógicas e analíticas e organizando todo o fluxo de execução do programa.
- **Input**: Classe abstrata que faz interface com a rede de sensores. O *framework* STEAM estende esta classe para criar recursos de aquisição de dados específicos, suportando vários protocolos de comunicação industrial, como RS232, Modbus, OPC, etc.
- **Parser**: A classe *Parser* padrão lida com *frames* de dados brutos com um ou vários valores, separados por um único caractere ou string. É possível estender essa classe, adicionando a capacidade de interpretar *frames* de dados brutos com estruturas complexas.

Figura 20: Diagrama de classes do *framework* STEAM



- **Function**: Classe base para realizar análise de dados. O *framework* STEAM estende essa classe para fornecer uma rica biblioteca de processamento de dados. Até o momento, foram implementadas as seguintes classes: *Min*, *Max*, *Sum*, *Count*, *Mean*, *Median*, *EWMA*, *StDev*, *Slope*, *Arima*, and *Equation*.

- **Enrich**: Classe que lida com o processo de enriquecimento do fluxo de dados, atualizando os pacotes de dados brutos dos sensores com os dados processados retornados das funções analíticas.

- **Condition**: Classe que avalia uma condição, indicando a ocorrência de um evento. O *framework* STEAM oferece atualmente as classes *EquationCondition*, *MissingValueCondition* e *ThresholdCondition*. É possível estender a classe *Condition* para oferecer avaliação de codições personalizadas e detecção de eventos complexos.

- **Format**: Classe que formata o pacote de dados enriquecido antes de enviá-lo aos aplicativos cliente. O *framework* STEAM estende esta classe fornecendo um conjunto de formatos padrão, como *MessageFormat*, *CSVFormat*, *TSVFormat*, *JSONFormat*, e *WSO2Format*.

- **Endpoint**: Classe base para implementar a camada de saída de uma aplicação STEAM, definindo o destino dos fluxos de dados, mensagens e eventos processados e enriquecidos. Por padrão, o *framework* STEAM fornece as classes *FileEndpoint* e *HTTPEndpoint*, permitindo o armazenamento de arquivos e envio de mensagens HTTP, respectivamente. É possível estender essa classe para criar serviços de publicação customizados, implementando protocolos como MQTT, AMQP e CoAP, por exemplo.

5.1.2 Diagrama de Sequência

O diagrama de sequência de uma aplicação STEAM pode ser visualizado na Figura 21. Quando a aplicação é iniciada, é instanciado um objeto *Device*, responsável por gerenciar todas as configurações, processamentos, lógicas e fluxos de dados. O método *setInput()* define a camada de coleta de dados através de um objeto da classe *Input*. Por exemplo, se o sensor utiliza comunicação serial, devem ser especificados os parâmetros de porta, velocidade, paridade, bits de dados e bits de parada. Se for comunicação via *socket*, precisa informar o protocolo e a porta. Depois, a aplicação define qual objeto *Parser* será utilizado para interpretação e padronização dos dados brutos chamando o método *setParser()*.

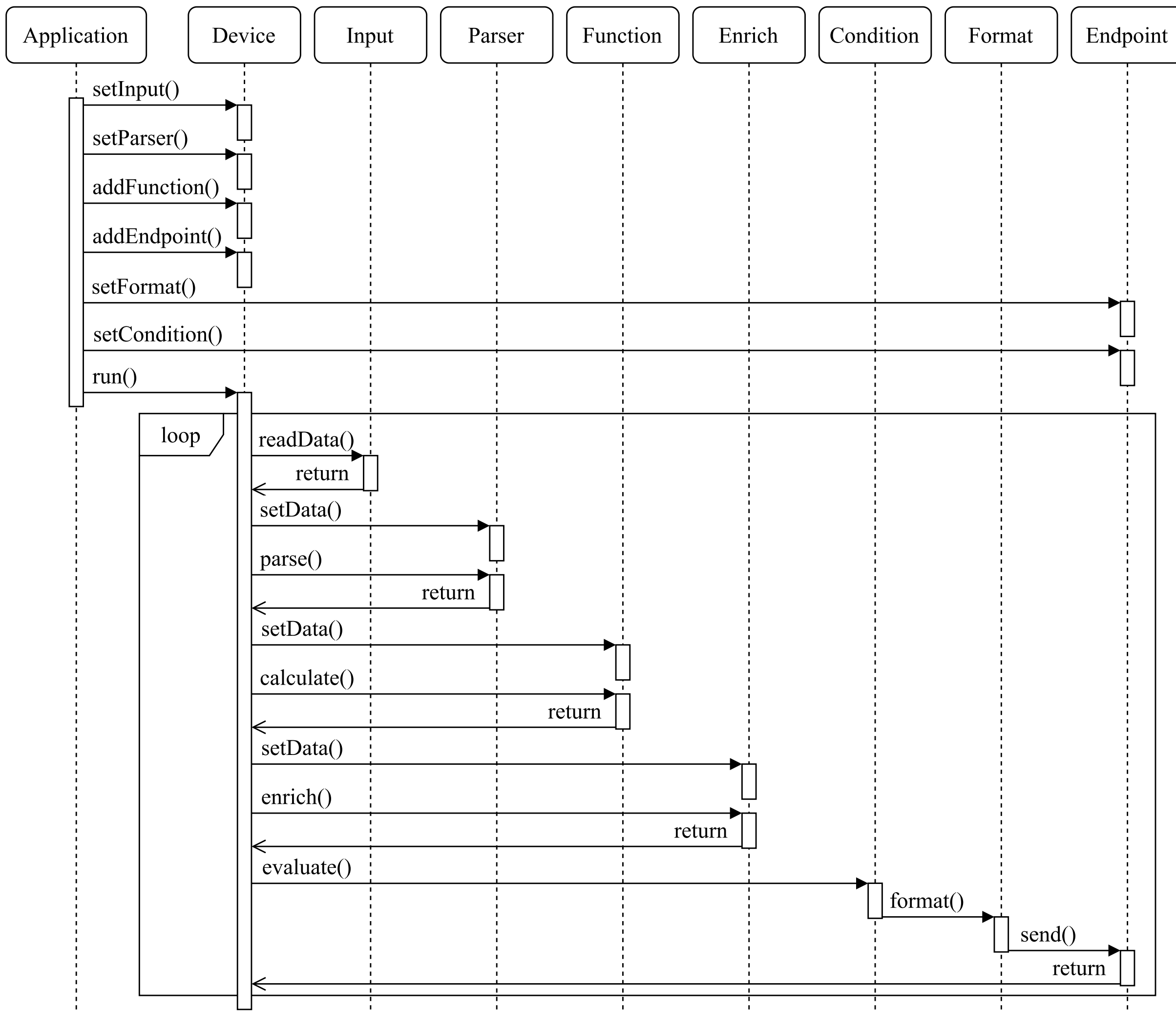
O método *addFunction()* adiciona um ou mais objetos *Function* à aplicação. Cada objeto é responsável por aplicar uma função analítica específica sobre os dados de entrada, gerando um resultado que será incluído mais adiante no pacote de dados enriquecidos. Na sequência, são definidos um ou mais objetos *Endpoint* através da chamada do método *addEndpoint()*. Os objetos *Endpoint* mais tarde farão o envio dos dados para as aplicações cliente.

Para cada objeto *Endpoint*, é definido um formato de apresentação passando um objeto *Format* como parâmetro da função *setFormat()*. Os formatos disponibilizados por padrão pelo *framework* STEAM são *MessageFormat*, *CSVFormat*, *TSVFormat*, *JSONFormat* e *WSO2Format*. Também para cada objeto *Endpoint*, o método *setCondition()* define a condição de envio dos dados utilizando um objeto *Condition*. *MissingValueCondition* gera um evento ao identificar falta de dados em um fluxo, *ThresholdCondition* identifica situações em que um valor fica aquém ou além de limites inferiores ou superiores, tanto fixos quanto dinâmicos, e *EquationCondition* permite criar condições de detecção de eventos personalizadas.

Ao disparar o método *run()*, a aplicação passa o controle da execução para o objeto *Device*, responsável por executar a aplicação em um *loop* infinito. Este *loop* inicia com a execução do método *readData()* do objeto *Input*, que aguarda a chegada dos dados a partir da comunicação com a rede de sensores. Assim que um pacote de dados é recebido pela camada de aquisição de dados, ele chama o método *setData()* do *Parser*, e em seguida, recebe os dados em um formato padrão através da chamada da função *parse()*. Os dados padronizados são passados para um ou mais objetos *Function* através do método *setData()*. A função *calculate()* é disparada para cada objeto *Function*, que processa os dados e retorna o resultado obtido. Os resultados das funções analíticas são enviados para o objeto *Enrich* através do método *setData()*, e retornados pela função *enrich()*.

Nesse instante, os dados brutos de um pacote foram processados, e todos os resultados analíticos estão disponíveis no pacote de dados enriquecido. Então, o método *evaluate()* do objeto *Condition* é executado para avaliação de todo o cenário e identificação de eventos. Caso

Figura 21: Diagrama de sequência de uma aplicação STEAM



a condição seja avaliada como verdadeira, o pacote de dados enriquecido é enviado para o objeto *Format()* através do método *format()*, e finalmente é enviado para a aplicação cliente através do método *send()* do objeto *Endpoint*. Com isso, é encerrado o ciclo de processamento de um pacote de dados e o *loop* retorna para a instrução inicial, aguardando a chegada de um novo pacote de dados para repetir novamente toda a sequência de processamento.

5.1.3 Algoritmo de uma Aplicação STEAM

O *framework* STEAM, além de disponibilizar uma biblioteca de classes e funções para facilitar o desenvolvimento de aplicações, também padroniza o fluxo do processamento de dados. O Algoritmo 1 demonstra o funcionamento do método *run()* da classe *Device*, que controla todas as aplicações STEAM. Na linha 1 ocorre a captura dos dados dos sensores, e a aplicação permanece ativa, recebendo e processando dados enquanto eles existirem. Entre as linhas 2 e

4 ocorre a interpretação dos dados brutos e padronização do pacote de dados de entrada. Da linha 5 até 9 são executadas todas as funções de cálculos associadas ao dispositivo, seguido do enriquecimento dos pacotes de dados. A linha 10 acessa os objetos *Endpoint*, e para cada um deles, executa a avaliação da condição de envio dos dados. Caso a condição seja avaliada como verdadeira, os dados são enviados para o *endpoint* específico. No final, a execução retorna para a linha 1 e aguarda a chegada de um novo pacote de dados brutos.

Algoritmo 1: Processo de execução de uma aplicação STEAM

Entrada: Dados brutos recebidos da rede de sensores. Linha 1.
Saída: Dados padronizados e enriquecidos enviados para os clientes. Linha 12.

```
 1: while line ← input.readData() do
 2:   parser.setData(line)
 3:   parsed ← parser.parse()
 4:   buffer.addData(parsed)
 5:   for f in device.functions do
 6:     data ← f.calculate()
 7:     enrich.setData(data)
 8:     enrich.enrich()
 9:   end for
10:   for e in device.endpoints do
11:     if e.evalCondition() then
12:       e.send()
13:     end if
14:   end for
15: end while
```

5.1.4 Application Programming Interface - API

Para desenvolver um aplicativo STEAM, é necessário conhecer as classes, estruturas, métodos, parâmetros e valores de retorno das funções. Esta seção descreve a interface de programação de aplicativos (API) do *framework* STEAM, usando a seguinte sintaxe:

<classe>:<método>(<parâmetro:tipo>):<retorno>

- **Device**: *Classe*
  - **Device(*batchlen: int*)**: Método construtor do objeto. O parâmetro *batchlen* representa a quantidade de pacotes de dados armazenados em um *buffer* interno, posteriormente usados por uma lista de objetos *Function*;

- **setInput(*Input*)**: *void* - Método para definir o objeto *Input* do dispositivo, responsável por coletar dados de sensores ou outros dispositivos de borda.
- **setParser(*Parser*)**: *void* - Método para definir o objeto *Parser*, projetado para analisar, extrair dados relevantes de *frames* de dados brutos do sensor e formatar os dados em um pacote padrão.
- **addFunction(*Function*)**: *void* - Método para adicionar um ou vários objetos *Function* ao dispositivo. Essas funções são executadas quando cada pacote de dados é recebido dos sensores.
- **addEndpoint(*Endpoint*)**: *void* - Método para adicionar um ou vários objetos *Endpoint* ao dispositivo. Um único aplicativo STEAM pode lidar com vários *endpoints*, despachando mensagens para vários aplicativos cliente a partir de um único fluxo de entrada.
- **run()**: *void* - Método que inicia a aplicação STEAM, rodando em *loop* até o recebimento de um sinal de fim de transmissão, interrupção ou fechamento.

• **Input**: *Classe*

- **readData()**: *boolean, string* - Função que escuta, monitora e realiza leituras de sensores constantemente, de acordo com o protocolo de comunicação implementado. Ele retorna dois parâmetros: o primeiro é um booleano, indicando se o processo de aquisição de dados ocorreu com sucesso ou não, e o segundo parâmetro são os dados lidos, em formato de string com os dados brutos.

• **Parser**: *Classe*

- **Parser(*unit: string, separator: string, columns: string[]*)**: Método construtor do objeto com três parâmetros opcionais. O parâmetro *unidade* é a unidade de medida padrão, usada quando o *frame* de dados brutos não contém esta informação. O parâmetro *separator* é um caractere ou string que separa cada dado em um *frame* de dados com valores múltiplos. Finalmente, *columns* é uma coleção de identificadores para nomear os dados analisados. Se nenhum nome de coluna for fornecido, eles são identificados como *valor*.
- **setData(*data: string*)**: *void* - Método para alimentar o objeto *Parser* com os dados recebidos anteriormente pelo objeto *Input*. O parâmetro *data* é o pacote de dados padrão retornado pela função *Input.readData()*.
- **parse()**: *boolean, string* - Função que analisa os dados brutos do sensor, extraindo informações relevantes como a unidade de medida e o valor ou valores medidos. Retorna dois parâmetros, um booleano indicando se o processo de análise foi executado com sucesso ou não, e os dados extraídos do *frame*, em uma string formatada como um pacote de dados padrão.

- **Function**: *Classe*
  - **Function(*id: string, batchlen: int, attribute: string*)**: Método construtor do objeto com três parâmetros obrigatórios. O parâmetro *id* é o identificador da função, *batchlen* é a quantidade de pacotes de dados usados pelo processo de cálculo da função, e *attribute* é o identificador da coluna, cujos valores são selecionados dos pacotes de dados e processados pela função.
  - **calculate()**: *key:value* - Esta função executa o método de cálculo específico. Ela seleciona uma quantidade *batchlen* a partir de uma coleção de *attribute* dos últimos pacotes de dados recebidos e armazenados em um *buffer*, faz o cálculo e retorna um elemento *chave:valor*, onde *chave* é o *id* da função e *valor* é o resultado do processo de cálculo.
- **Enrich**: *Classe*
  - **Enrich(*packet, packets*)**: Método construtor do objeto com dois parâmetros obrigatórios. O parâmetro *packet* é uma referência ao pacote de dados recebido mais recentemente dos sensores, enquanto o parâmetro *packets* é uma referência a um *buffer* com *batchlen* quantidade de pacotes.
  - **setData(*data: string*)**: *void* - Método para alimentar o objeto *Enrich* com o resultado da etapa de análise. O parâmetro *data* é o elemento *chave:valor* retornado pelo método *Function.calculate()*.
  - **enrich()**: void - Método que enriquece o último pacote de dados recebido dos sensores com o resultado retornado pelass funções analíticas, ou seja, atualiza o parâmetro **packet* com o conteúdo do parâmetro *data*, ambos parâmetros recebidos pelo construtor.
- **Condition**: *Classe*
  - **Condition()**: Método construtor do objeto. Como é uma classe abstrata, ela foi estendida para fornecer três condições personalizadas. O construtor da classe *EquationCondition* recebe um parâmetro *string* contendo uma equação; o *MissingValueCondition* recebe um único ou uma lista de *ids* de colunas para verificar seus conteúdos; e o construtor do *ThresholdCondition* é alimentado por um único ou uma lista de *ids* de colunas, seguido por um único ou uma lista de limites inferiores e, finalmente, por um único ou uma lista de limites superiores.
  - **evaluate()**: *boolean* - Função que realiza a avaliação da condição, retornando um booleano indicando se um evento foi detectado ou não.

- **Format**: *Classe*
  - **Format(*columns: string[]*)**: Método construtor do objeto. O parâmetro obrigatório *columns* representa uma lista de *id*s das colunas usadas pelo processo de formatação da mensagem de saída.
  - **format()**: *string* - Função que formata o pacote de dados, retornando-o como uma string.
- **Endpoint**: *Classe*
  - **Endpoint()**: O construtor da superclasse não tem parâmetros, mas as classes estendidas são personalizadas para qualquer método de saída desejado. O construtor *FileEndpoint* recebe o nome do arquivo como um parâmetro, e o construtor *HTTPEndpoint* recebe a URL do serviço para postar os dados processados.
  - **setFormat(*Format*)**: *void* - Método que configura o objeto *Format*, utilizado para compor a apresentação dos dados de saída antes de enviá-los ao destino.
  - **setCondition(*Condition*)**: *void* - Este método atribui um objeto *Condition* ao *Endpoint*. Quando avaliada, uma condição pode indicar a ocorrência de um evento e decidir se uma mensagem deve ser enviada ao seu destino ou não.
  - **send()**: *void* - Este método envia a mensagem ao destino definido na extensão da classe *Endpoint*.

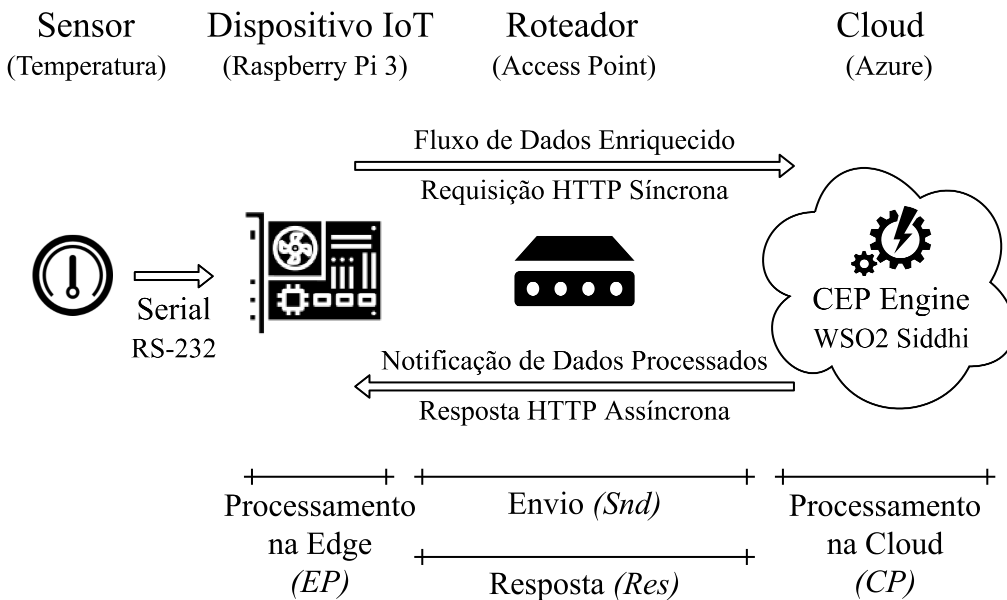
5.2 Métricas da Aplicação 1 - Edge-Cloud

Para validação do modelo STEAM, foram previstas 2 aplicações, descritas em detalhes mais adiante no capítulo 6. Para avaliar a qualidade da *Aplicação 1* entre Edge e Cloud, foram definidas três métricas: *Precisão*, *Consumo de Tempo* e *Tamanho do Pacote de Dados*. *Precisão* é medida comparando os valores calculados na borda pelo aplicativo STEAM com os valores retornados pelo *CEP Engine* calculados na nuvem. Uma vez que um dos objetivos deste trabalho é trazer a análise de dados da nuvem para a borda, os resultados de ambos os processos devem ser os mesmos.

Para medir a precisão de acordo com a Figura 22, cada pacote de dados processado na borda *(EP)* é identificado unicamente *(UID)* e enviado *(Snd)* para o *CEP Engine* para processamento na nuvem *(CP)*. Assim que *CP* estiver completo, é enviado de volta um pacote de resposta assíncrona *(Res)* para a *Aplicação 1*, identificado com o mesmo *UID* de origem. Então, os

pacotes *EP* e *CP* são combinados por *UID* e os resultados do processamento na borda pela aplicação STEAM são comparados com o resultado do processamento na nuvem pelo *CEP Engine*. Dessa forma, a métrica de precisão consiste na contagem de valores incompatíveis comparando os pacotes *EP* e *CP* durante todo o teste.

Figura 22: Infraestrutura da *Aplicação 1* desenvolvida para avaliar um aplicativo STEAM e comparar com um aplicativo híbrido *Edge-Cloud*



Em relação à métrica *Consumo de Tempo*, é esperada uma diminuição significativa comparando um aplicativo com processamento exclusivamente de borda *(EP)* em relação a um aplicativo dependente de processamento de nuvem *(CP)*. Embora um aplicativo STEAM exija mais processamento na extremidade (EP) e, consequentemente, gaste mais tempo processando localmente do que um aplicativo híbrido *Edge-Cloud*, espera-se que a eliminação da transmissão de dados *(Snd + Res)* resulte em um intervalo de tempo mais curto para se obter o mesmo resultado final, além da eliminação completa de *timeouts* e perda de pacotes. Dessa forma, a métrica *Consumo de Tempo* para a *Aplicação 1* contabiliza apenas o tempo *EP*, representado na Equação 5.1, enquanto no aplicativo híbrido *Edge-Cloud*, o tempo necessário para a obtenção de um resultado é calculado pela soma dos tempos individuais *EP*, *Snd*, *CP* e *Res*, conforme definido na Equação 5.2.

$$Edge = EP \tag{5.1}$$

$$Hybrid = EP + Snd + CP + Res \tag{5.2}$$

Por fim, a métrica *Tamanho do Pacote de Dados* registra a quantidade de dados gerados pelo processo *Enriquecimento* e os compara com um pacote de dados brutos. O aumento do tamanho do pacote de dados impacta diretamente no tráfego de rede, e pode ser um problema de acordo com o protocolo usado entre a borda e a nuvem. Portanto, foram analisados não apenas o aumento total no tamanho dos dados, mas também quantos bytes em média cada dado enriquecido injeta no pacote.

5.3 Métricas da Aplicação 2 - Edge

A *Aplicação 2* foi projetada para utilizar uma arquitetura de processamento exclusivamente na borda da rede, conforme ilustrado na Figura 23. Nesse cenário, os aplicativos STEAM recebem, processam e publicam os dados, tanto salvando os resultados em um arquivo de log local quanto os enviando para um painel de controle implementado em Node-RED[1] rodando em um laptop conectado à rede local via Wi-Fi. O aplicativo de painel de controle simplesmente recebe os dados por meio de um serviço HTTP e os exibe em um gráfico de linhas ou área de texto, sem realizar nenhum processamento de dados.

Para a avaliação da *Aplicação 2*, foi elaborada uma metodologia de micro-benchmark com três métricas: *Uso de CPU / Memória*, *Tempo de Processamento* e *Razão Saída/Entrada*, representados na Figura 24. Para ler o uso de CPU e memória do sistema, foram usados os métodos *cpu_percent()* e *virtual_memory()* da biblioteca *psutil* do Python, respectivamente. Esses valores foram medidos ao final do fluxo de processamento de cada pacote recebido, indicando o consumo de CPU e uso de memória durante as tarefas. Para a métrica *Tempo de Processamento*, foi medido o tempo gasto por cada camada de processamento do STEAM por pacote, desde a leitura dos dados brutos até o despacho do pacote enriquecido. Para essa métrica, foi usado o método *time_ns()* da biblioteca *time* do Python, que retorna um número inteiro de nanossegundos desde *epoch*, que é a zero hora do primeiro dia de janeiro de 1970.

Também foi medida a quantidade total de bytes recebidos dos sensores e, posteriormente, enviados para os aplicativos externos. Com essas informações, foi calculada a métrica *Razão Saída/Entrada*, consistindo na razão dos tamanhos dos pacotes de dados de saída em relação à entrada, indicando o fator de aumento ou diminuição sobre o tamanho do fluxo de dados obtido como resultado do processamento da aplicação STEAM. Quando a aplicação encerra, o micro-benchmark salva um arquivo de log contendo as métricas coletadas e calculadas para cada pacote de dados processado. A Figura 25 ilustra um exemplo de arquivo de log onde cada linha representa um pacote de dados processado pelo aplicativo STEAM e as colunas identificam as métricas, separadas por caracteres *tab*.

[1] Ferramenta de programação low-code direcionada a aplicações orientadas a eventos, permitindo a conexão entre dispositivos de hardware, APIs e serviços online. Disponível em: `https://nodered.org/`

Figura 23: Infraestrutura da *Aplicação 2*. Cada um dos 3 sensores envia 1 medição por segundo para o *Nodo* que agrupa, encapsula e retransmite o pacote de dados brutos para o aplicativo STEAM em execução em um Raspberry Pi na borda da rede. O aplicativo STEAM recebe, processa e publica os dados, tanto salvando os resultados em um arquivo de log local quanto os enviando para um painel de controle *Node-RED* executado em um laptop na LAN.

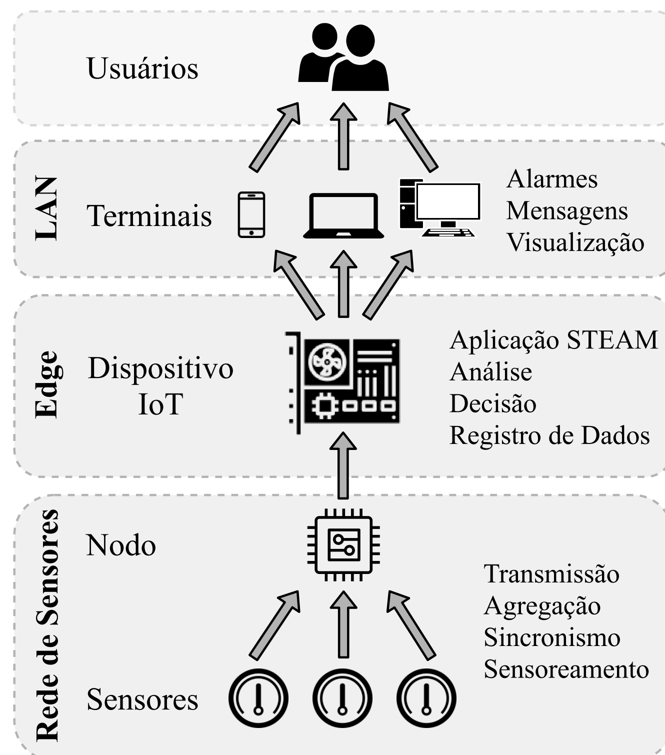


Figura 24: Metodologia e métricas do micro-benchmark para avaliação das aplicações STEAM: enquanto o aplicativo está em execução, são coletadas medições de uso de CPU, memória, tempo de processamento por camada e proporção de tamanho de dados de saída/entrada para cada pacote processado e, finalmente, as métricas são salvas em um arquivo de log.

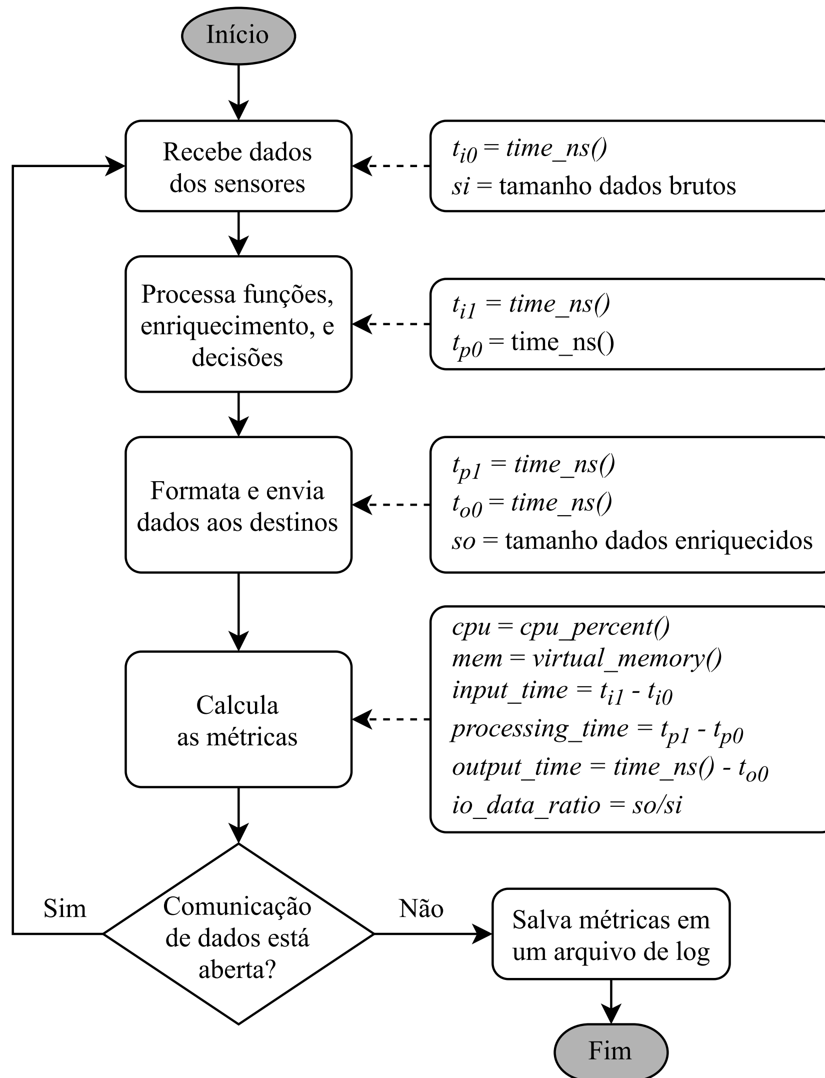


Figura 25: Exemplo de arquivo de log gerado pelo micro-benchmark. Cada linha representa um pacote de dados processado pelo aplicativo STEAM.

id	cpu_%	mem_kb	input_µs	process_µs	output_µs	out_in_ratio
1	0.5	157032448	893014	5906087	117364564	1.451
2	0.9	157057024	804630	7995255	111122838	1.660
3	0.7	157057024	736609	5954071	126767879	0.720
4	1.9	157376512	809838	5945842	102003842	0.640
5	0.7	157315072	749578	5914748	117781123	1.490
6	0.2	157315072	772286	5944070	116796182	0.261
7	2.0	157319168	825255	6272923	115830617	0.319
8	0.7	157306880	784369	5945894	116912275	1.608

6 AVALIAÇÃO E RESULTADOS

Esse capítulo apresenta a implementação de duas aplicações desenvolvidas com o *framework* STEAM, quatro cenários de teste, a execução dos experimentos e os resultados obtidos aplicando as métricas previamente definidas.

Escrever aplicativos STEAM é muito simples em comparação com criar um aplicativo IoT do zero. O *framework* STEAM fornece um conjunto de classes prontas para uso e funções integradas, tornando desnecessário o uso de código estruturado, lógica complexa, estruturas de repetição e instruções condicionais. Os relacionamentos das classes garantem a consistência da aplicação, atribuindo ao desenvolvedor apenas a tarefa de instanciar objetos e configurar parâmetros. A Figura 26 ilustra o aplicativo básico usado em todos os experimentos. A linha 2 é a instanciação do objeto *Device*, configurado para gerenciar uma janela deslizante com as últimas 20 medições do sensor. A linha 5 define a camada de coleta de dados *Input* como uma comunicação TCP na porta 5000, mas também poderia utilizar comunicação serial ou qualquer outro protocolo. As linhas 8 a 11 criam o objeto *Parser*, definindo um caractere *tab* como separador de valores e identificando os nomes das colunas do *frame* de dados brutos recebidos do sensor. As linhas 14 a 16 criam o primeiro *endpoint*, configurando um objeto *HTTPEndpoint* que consiste na URL do fluxo de entrada do painel de controle gráfico desenvolvido em Node-RED[1], formata a saída de dados como *JSONFormat*, e finalmente, associa os objetos ao *Device*. As linhas 19 a 21 criam o segundo *endpoint* através de um objeto *FileEndpoint*, configurando o nome do arquivo de saída e o formato de dados como sendo CSV. A linha 24 inicia a execução do aplicativo.

A partir do código base mostrado na Figura 26, foram construídas duas aplicações para avaliar o modelo STEAM. Elas recebem dados de um nó sensor, executam funções estatísticas, avaliam expressões e, finalmente, enriquecem o fluxo de dados com o resultado do processamento. Em seguida, os aplicativos enviam eventos e o fluxo de dados enriquecido para uma *CEP Engine* hospedada na Azure ou para um painel de controle implementado em Node-RED, executado por um notebook na LAN, traçando gráficos de linhas e exibindo mensagens de eventos relevantes. A seguir, as aplicações são explicadas em detalhes.

6.1 Aplicação 1 - *Edge-Cloud*

O ambiente configurado para a *Aplicação 1* apresenta a seguinte arquitetura, representada previamente na Figura 22 da seção 5.2. Um sensor de temperatura envia constantemente seu valor medido via comunicação serial RS-232 para o aplicativo STEAM sendo executado em

[1] `https://nodered.org/`

Figura 26: Aplicação básica desenvolvida com o *framework* STEAM. Este código-fonte é totalmente funcional, lendo dados de sensores, processando e analisando os dados brutos na borda e posteriormente enviando eventos e o fluxo de dados enriquecido para aplicações cliente

```
# Create the Device object
device = Device(batchlen=20)

# Define the Input method
device.setInput(TCPInput(port=5000))

# Create data parser
device.setParser(
  Parser(
    separator='\t',
    columns=['id', 'timestamp', 'unit', 's1', 's2', 's3']))

# Create the endpoint - Node-RED - Line chart - JSON format
ep_data = HTTPEndpoint(url='http://node-red-server:1880/datastream')
ep_data.setFormat(JSONFormat())
device.addEndpoint(ep_data)

# Create the endpoint - Save to file - CSV format
ep_file = FileEndpoint(file='output.csv')
ep_file.setFormat(CSVFormat())
device.addEndpoint(ep_file)

# Run the application
device.run()
```

um dispositivo IoT, que consiste em uma Raspberry Pi 3 modelo B+ com 1GB RAM e Raspbian OS. A camada *Abstração de Dispositivos e Coleta de Dados* da aplicação STEAM recebe continuamente os pacotes de dados brutos enviados pelo sensor através da comunicação serial RS-232 e os encapsula em um pacote com formato padrão. Depois, o fluxo de dados brutos é enviado para a camada *Análise* e o resultado é mesclado com o fluxo de dados original na camada *Enriquecimento*. Finalmente, a camada *Conector de Protocolos* do aplicativo STEAM envia o fluxo enriquecido para um *CEP Engine* hospedado na Azure. Um *access point* sem fio é responsável por conectar a rede do dispositivo IoT à nuvem.

O *CEP Engine* utilizado foi o *WSO2 Streaming Integrator*[2] executando *Siddhi*[3], e foi instalado em uma máquina virtual hospedada na Azure executando Ubuntu 18.04. O *CEP Engine* recebe os dados vindos da aplicação STEAM como um fluxo HTTP no formato JSON, analisa e processa os dados e envia o resultado de volta ao dispositivo IoT de forma assíncrona.

[2]https://wso2.com/integration/streaming-integrator/
[3]https://wso2.com/products/complex-event-processor/

6.1.1 Cenário de Teste da Aplicação 1

Para avaliar o modelo STEAM em uma arquitetura *Edge-Cloud*, a *Aplicação 1* realizou o monitoramento da temperatura dentro de uma sala limpa de um fabricante de microchips. Um sensor fazia constantemente o envio do valor da temperatura via comunicação serial RS-232 para a aplicação STEAM, cujo trecho de código fonte pode ser visualizado na Figura 27, a uma taxa de transmissão de 20 medições por segundo. O mesmo fluxo de dados do sensor também era enviado ao *CEP Engine* na nuvem, executando a aplicação representada na Figura 28, fazendo os mesmos cálculos sobre a mesma janela deslizante, e retornava de forma assíncrona os resultados para a borda.

Figura 27: Trecho do código fonte da *Aplicação 1* desenvolvida com o *framework* STEAM. Sobre uma janela deslizante dos últimos 20 dados recebidos a partir do sensor, são calculados contagem, somatório, mínimo, máximo, média, mediana e desvio padrão. Em seguida, os dados são enviados para uma *CEP Engine* hospedada na Azure, processados e retornados para a borda da rede, onde é feita a comparação dos resultados.

```
1  # WSO2 CEP Engine Endpoint - Raw Data Input Stream
2  ep_cep = HTTPEndpoint(url='http://[WSO2_machine_IP]:8006/inputStream')
3  ep_cep.setFormat(WSO2Format)
4  device.addEndpoint(ep_cep)
5
6  # Analytical function list
7  device.addFunction(fn.Count())
8  device.addFunction(fn.Sum())
9  device.addFunction(fn.Min())
10 device.addFunction(fn.Max())
11 device.addFunction(fn.Mean())
12 device.addFunction(fn.StDev())
```

6.1.2 Realização do Experimento e Resultados da Aplicação 1

O experimento da *Aplicação 1* consistiu na repetição de 30 testes idênticos, cada um enviando 530 medições às duas aplicações por meio de um fluxo de dados. As aplicações, tanto a executada pelo STEAM quanto pelo *CEP Engine*, calcularam a contagem, soma, mínimo, máximo, média, desvio padrão e indicador de *outlier* em uma janela deslizante de 20 valores. Não foram processados os primeiros 19 pacotes de dados, pois, no início do teste, a janela deslizante ainda não possuía 20 eventos. Do vigésimo evento até o final, a cada novo dado recebido, uma nova janela deslizante era formada, e então, os cálculos eram executados. Dessa forma, o ciclo foi processado 510 vezes a cada teste, 20 vezes por segundo, cada teste durando aproximadamente 25,5 segundos.

Figura 28: Aplicação configurada no *WSO2 Streaming Integrator* hospedado em uma máquina virtual na Azure. Primeiro, é definido o fluxo de entrada que recebe dados por meio de uma postagem HTTP, depois uma consulta executa as funções de agregação e, por fim, os dados processados são enviados de volta para a borda.

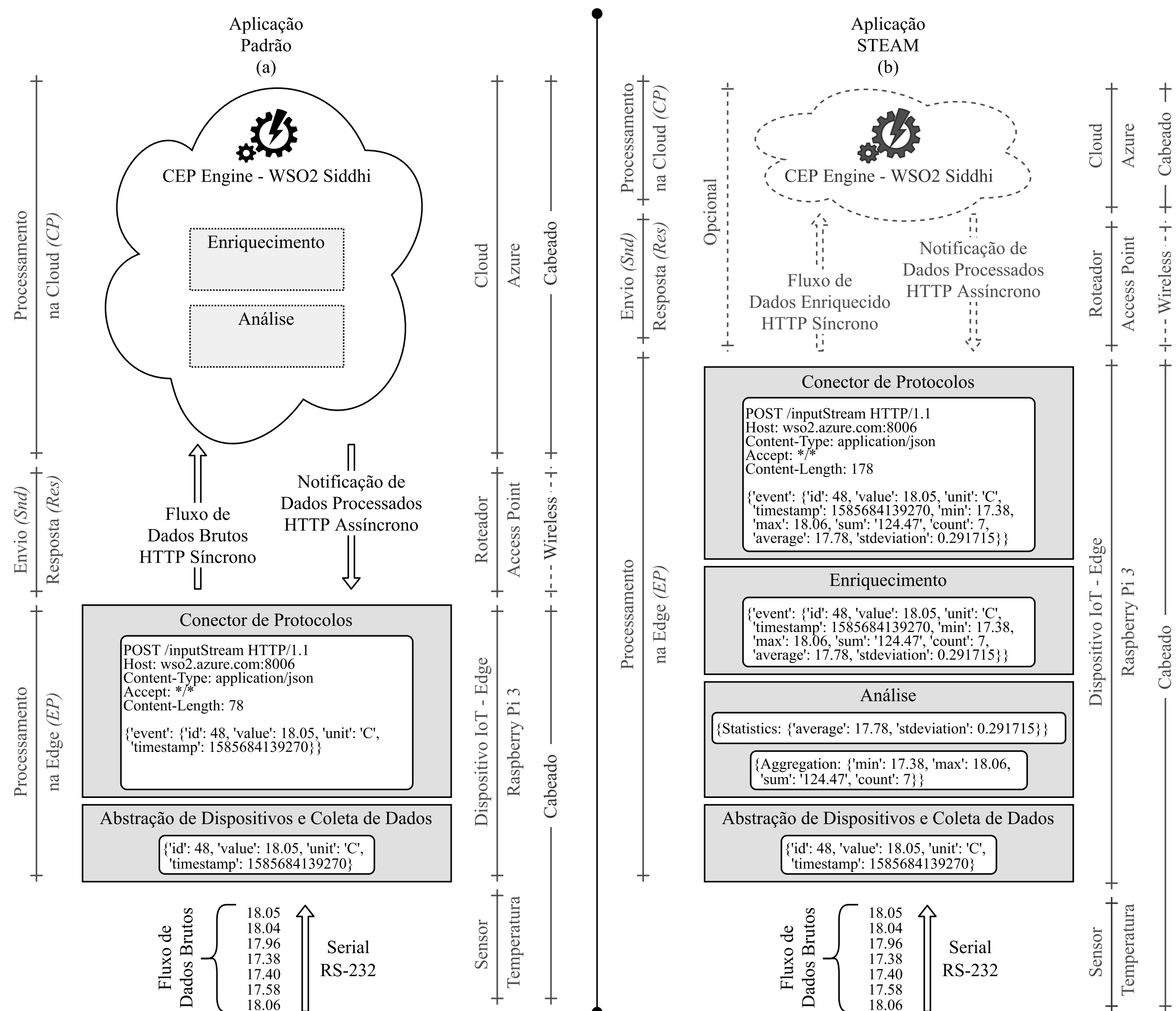
```
-- Input stream, receiving data from edge
@source(type = 'http', basic.auth.enabled = "false", @map(type = 'json',
  receiver.url = "http://0.0.0.0:8006/inputStream"))
define stream InputStream(
  id long, value double, unit string, timestamp long);

-- Aggregate and calculation functions over window length = 20
@info(name='qryAggregate')
from InputStream#window.length(20)
select
  id, count() as countVal, sum(value) as sumVal, min(value) as minVal,
  max(value) as maxVal, avg(value) as avgVal, stdDev(value) as stdDevVal,
  ifThenElse(stdDev(value) > 1.0, true, false) as outlier
insert into AggregateStream;

-- Sends the outcome computed in the cloud back to the edge
@sink(type = 'http', method = "POST", @map(type = 'json',
  publisher.url = "http://[STEAM_IoT_machine_IP]/wso2"))
define stream AggregateStream(
  id long, countVal long, sumVal double, minVal double,
  maxVal double, avgVal double, stdDevVal double, outlier bool);
```

Para se ter uma melhor compreensão de todo o fluxo e transformação dos dados, foi monitorada cada etapa do processamento no aplicativo STEAM, ilustrado na Figura 29, detalhado em (a) *Aplicativo Padrão*, agindo como um *proxy* entre o sensor e a *CEP Engine*, e (b) *Aplicativo STEAM*, realizando análise e enriquecimento dos dados na borda da rede. Neste diagrama, a camada *Abstração de Dispositivos e Coleta de Dados* captura dados brutos do sensor e os converte em um formato padrão, em ambos os cenários. Enquanto em (a) os dados formatados são imediatamente enviados para a nuvem através do *Conector de Protocolos*, onde o fluxo de dados é transmitido de forma síncrona para a *CEP Engine* na nuvem usando o protocolo HTTP, em (b) são realizadas várias etapas de processamento. A camada *Análise* executa funções estatísticas e de agregação sobre o fluxo de dados. Então, o resultado dessa etapa é enviado para a camada *Enriquecimento*, que mescla os dados processados com os dados originais e, finalmente, os envia para o *Conector de Protocolos*. Para ambos os cenários, quando a *CEP Engine* termina os cálculos, ela envia de volta uma notificação ao aplicativo STEAM usando uma solicitação HTTP assíncrona.

Em ambos os cenários, (a) e (b), a *CEP Engine* hospedada na nuvem é responsável pela análise e computação de dados, enviando o resultado de volta para a borda por questões de comparação de dados. Porém, em (b), a aplicação STEAM faz os mesmos cálculos e obtém os mesmos resultados, tornando a *CEP Engine* um elemento opcional.

Recapitulando, as métricas definidas para avaliar a *Aplicação 1*, foram *Precisão*, *Consumo*

Figura 29: Execução das aplicações STEAM em detalhes, começando com a aquisição de dados, análise, enriquecimento e envio para uma *CEP Engine* hospedada na nuvem. Em um aplicativo padrão (a), todas as análises de dados são realizadas na nuvem, enquanto em um aplicativo STEAM (b), o cálculo é realizado na borda, tornando a nuvem um elemento opcional.

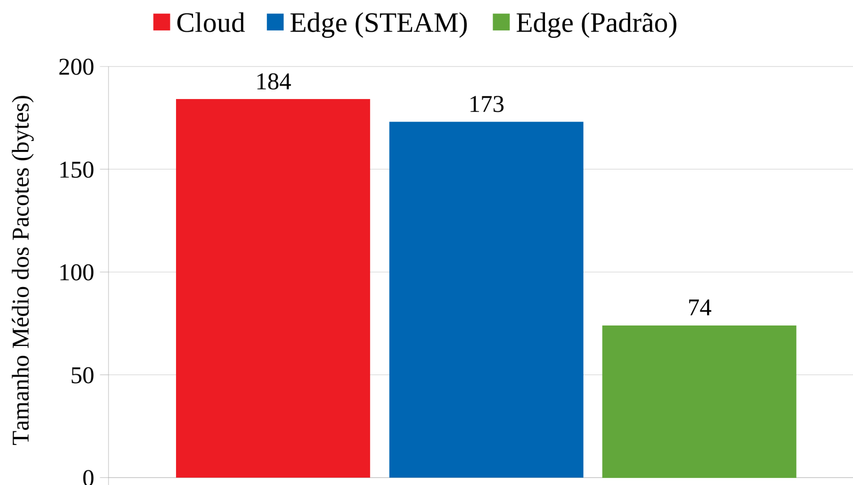


de Tempo e *Tamanho do Pacote de Dados*. Em relação à precisão, foram comparados os resultado dos cálculos feitos na borda com os valores retornados pela *CEP Engine* executada na nuvem, evento a evento, e não foram encontradas incompatibilidades. Assim, foi alcançada uma precisão de 100% nos cálculos. No entanto, foram identificadas algumas perdas de pacotes. De 30 testes com 530 eventos, cada um totalizando 15900 pacotes de dados, apenas 8 não retornaram ao aplicativo STEAM. Analisando os logs detalhadamente, foi identificado que todos os dados chegaram à *CEP Engine* e foram processados corretamente por ela, mas de alguma forma, eles se perderam no caminho de volta para a borda. Vale a pena investigar os motivos desse problema em um trabalho futuro, mas eles estão fora do escopo de pesquisa atual.

Considerando o tamanho do pacote de dados no formato JSON, foi identificado o previsível aumento da quantidade de bytes após o processo de enriquecimento, ilustrado na Figura 30.

Um único pacote de dados brutos com os atributos *id*, *value*, *unit* e *timestamp* soma 74 bytes. O processo de enriquecimento realizado pela aplicação STEAM na borda para os seis atributos *min*, *max*, *sum*, *count*, *average* e *stdeviation* injetou quase 99 bytes no pacote, resultando em uma média de 173 bytes por pacote. Finalmente, esses seis atributos calculados na nuvem resultaram em um pacote de 184 bytes, aumentando o pacote de dados brutos original em 110 bytes. O aumento do tamanho médio dos pacotes devido ao processo de enriquecimento é significativo, chegando a 133,8% para o aplicativo STEAM executado na borda e 148,6% para o aplicativo CEP na nuvem. Mesmo que o aumento no tamanho dos pacotes de dados tenha sido significativo, enviar os dados para a nuvem tornou-se uma tarefa opcional, uma vez que os valores agora são calculados pelo aplicativo STEAM na borda.

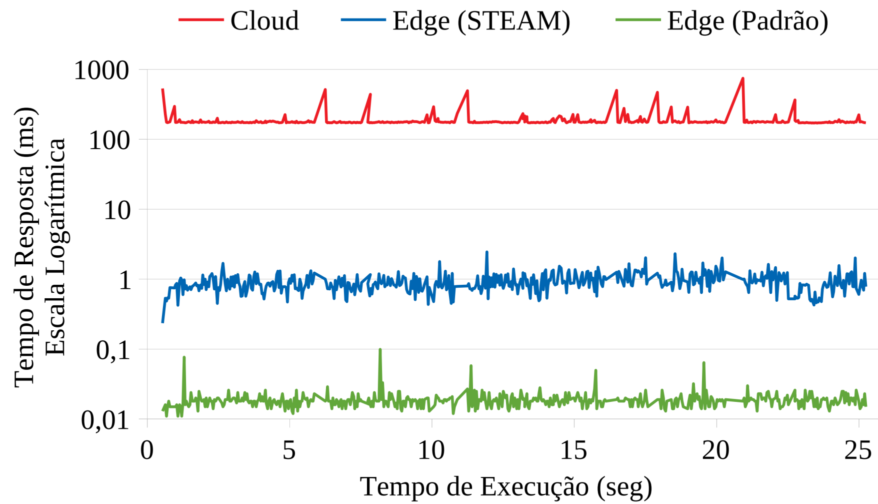
Figura 30: Tamanho médio dos pacotes em bytes. Os pacotes Edge (Padrão) incluem apenas dados básicos, enquanto Edge (STEAM) e Cloud incluem dados enriquecidos



Com relação ao consumo de tempo, a Figura 31 mostra um único teste que exemplifica o comportamento comum de todos os 30 testes realizados. A linha vermelha na parte superior do gráfico é o tempo de processamento no cenário (a) para todos os dados, calculado pela Equação 5.2. A linha verde na parte inferior do gráfico representa apenas o tempo consumido na borda por um aplicativo padrão, de acordo com a Equação 5.1. A linha azul no centro do gráfico é o tempo de processamento gasto pelo aplicativo STEAM no cenário (b), executando na borda, usando a mesma Equação 5.1. O eixo Y representa o tempo de resposta em milissegundos, expresso em uma escala logarítmica devido à grande diferença nas magnitudes envolvidas nos tempos de resposta de borda e nuvem. O eixo X é o tempo de execução do teste, expresso em segundos.

Os valores apresentados na Tabela 4 são os tempos mínimo, máximo, média, mediana e desvio padrão dos tempos envolvidos nos experimentos. A coluna *Híbrido* expressa todo o

Figura 31: Tempo de resposta de um único teste que representa o comportamento comum de todos os testes realizados. O eixo "y"está em escala logarítmica



cenário ilustrado na Figura 29 (a), calculando $EP + Snd + CP + Res$ vezes, enquanto a coluna *Padrão* está considerando apenas EP vezes no mesmo cenário. A coluna *STEAM* também mostra o tempo EP, entretanto, sobre o cenário da Figura 29 (b). O tempo médio de resposta em todos os testes envolvendo todas as solicitações feitas à nuvem foi de 185 ms com um desvio padrão de 45,3323. O tempo mínimo de resposta foi de 171 ms e o máximo de 747 ms. Entre todas as solicitações feitas à *CEP Engine*, 95% duraram de 171 ms a 223 ms, enquanto 85% gastaram entre 171 ms e 180 ms. Analisando o tempo gasto no processamento interno do aplicativo STEAM rodando em um Raspberry Pi 3, foi obtida uma média de 910 µs por pacote de dados processado, com desvio padrão de 275,5413. O tempo de processamento mais rápido foi de 236 µs e o mais lento foi de 2449 µs. Para 95% dos pacotes processados, demorou entre 425 µs a 1361 µs, enquanto 70% do processamento ocorreu de 670 µs a 1200 µs. Comparando exclusivamente o tempo médio de resposta, o aplicativo STEAM foi 230,3 vezes mais rápido do que um aplicativo híbrido *Edge-Cloud*, com a mesma precisão nos cálculos e descartando a dependência de um ambiente de nuvem complexo.

Tabela 4: Tempos de processamento e resposta para os experimentos da *Aplicação 1*

Métrica / Cenário	Híbrido Edge + Cloud	Padrão Edge	STEAM Edge
Mínimo	171 ms	11 µs	236 µs
Máximo	747 ms	99 µs	2449 µs
Média	185 ms	18,88 µs	910 µs
Mediana	176 ms	18 µs	854 µs
Desvio Padrão	45,3323	6,1562	275,54138

6.2 Aplicação 2 - *Edge*

A *Aplicação 2* teve como objetivo monitorar a temperatura de ponto de orvalho em uma planta de uma indústria fabricante de microchips. A infraestrutura utilizada nos experimentos da *Aplicação 2* está representada na Figura 23, apresentada anteriormente na seção 5.3. Foram elaborados 3 cenários de teste, detalhados nas subseções 6.2.1, 6.2.2 e 6.2.3. O *Cenário 1* consistiu no processamento de dados de um único sensor, enquanto o *Cenário 2* e *Cenário 3* trabalharam com 3 sensores em conjunto.

6.2.1 Cenário de Teste 1 da Aplicação 2

Nesse cenário, a rede de sensores consiste em um nó (*Nodo*) recebendo medições de um único sensor (*Sensor*) a uma taxa de transmissão de uma medição por segundo. O *Nodo* retransmite os dados para o aplicativo STEAM em execução em um Raspberry Pi 3 Modelo B+ 1GB de RAM com Raspbian OS (*Dispositivo IoT*) através de uma conexão TCP. O *frame* de dados TCP consiste em uma string ASCII contendo 4 campos separados por caracteres *tab*, descritos na Tabela 5.

Tabela 5: Estrutura do *frame* de dados brutos do Cenário 1 da Aplicação 2

Coluna	Descrição	Tipo
id	Identificador sequencial da medição	Inteiro
timestamp	Data e hora da medição	ISO-8601
unit	Unidade de medição da temperatura de ponto de orvalho	Texto
value	Temperatura de ponto de orvalho medida pelo sensor	Fracionário

Essa aplicação, cujo código fonte está representado na Figura 32, recebe um único valor medido por um sensor a cada segundo, e inicialmente calcula o desvio padrão (linha 35) e a média móvel (linha 34) em uma janela deslizante dos últimos 20 valores, correspondendo a 20 segundos de medições. Para detectar anomalias no fluxo de dados, foi usada a técnica *Statistical Process Control* (SPC), considerando que fontes comuns de variações resultam em uma distribuição normal das amostras, onde a média $\bar{m}$ e desvio padrão σ podem ser estimados, conforme definido nas linhas 37 a 41. Qualquer observação fora desta faixa de controle calculada por $\bar{m} \pm 3\sigma$ é considerada anormal [He and Wang, 2018], e relatada como uma mensagem de aviso, codificada da linha 2 a 19. Todos esses valores e mensagens são armazenados em um arquivo de log local (linhas 22 a 31) e enviados para um painel de controle remoto implementado em Node-RED, traçando um gráfico de linhas e exibindo as mensagens de aviso.

Figura 32: Trecho do código fonte da *Aplicação 2 - Cenário 1* desenvolvida com o *framework* STEAM. Sobre uma janela deslizante dos últimos 20 dados recebidos a partir do sensor, são calculados *thresholds* dinâmicos e disparados eventos. A aplicação também salva dados em um arquivo de log, além de integrar com um painel de controle externo

```
# Node-RED - Above dynamic upper threshold - Message format
ep_upper = HTTPEndpoint(url='http://node-red-server:1880/msgstream')
con_upper = ThresholdCondition(columns='value', upper='upper')
fmt_upper = MessageFormat(
  '''<font color=red>{timestamp} -
  Value {value:.2f} above {upper:.2f}</font><br>''')
ep_upper.setCondition(con_upper)
ep_upper.setFormat(fmt_upper)
device.addEndpoint(ep_upper)

# Node-RED - Below dynamic lower threshold - Message format
ep_lower = HTTPEndpoint(url='http://node-red-server:1880/msgstream')
con_lower = ThresholdCondition(columns='value', lower='lower')
fmt_lower = MessageFormat(
  '''<font color=blue>{timestamp} -
  Value {value:.2f} below {lower:.2f}</font><br>''')
ep_lower.setCondition(con_lower)
ep_lower.setFormat(fmt_lower)
device.addEndpoint(ep_lower)

# Save to log file - Above dynamic upper threshold - Message format
ep_upper_file = FileEndpoint('log_above_upper.txt')
ep_upper_file.setCondition(con_upper)
ep_upper_file.setFormat(fmt_upper)
device.addEndpoint(ep_upper_file)

# Save to log file - Below dynamic lower threshold - Message format
ep_lower_file = FileEndpoint('log_below_lower.txt')
ep_lower_file.setCondition(con_lower)
ep_lower_file.setFormat(fmt_lower)
device.addEndpoint(ep_lower_file)

# Analytical function list
device.addFunction(fn.Mean(format='{:.2f}'))
device.addFunction(fn.StDev(format='{:.2f}'))

device.addFunction(
    fn.Equation(id='upper', format='{:.2f}', equation='mean + 3 * stdev'))

device.addFunction(
    fn.Equation(id='lower', format='{:.2f}', equation='mean - 3 * stdev'))
```

6.2.2 Cenário de Teste 2 da Aplicação 2

Nesse cenário, a rede de sensores consiste em um nó (*Nodo*) recebendo medições de 3 sensores (*Sensor*) a uma taxa de transmissão de 1 medição por segundo por sensor. O *Nodo* agrupa, encapsula e retransmite os dados para o aplicativo STEAM em execução em um Raspberry Pi 3 Modelo B+ 1GB de RAM com Raspbian OS (*Dispositivo IoT*) através de uma conexão TCP. O *frame* de dados TCP consiste em uma string ASCII contendo 6 campos separados por caracteres *tab*, apresentados na Tabela 6.

Tabela 6: Estrutura do *frame* de dados brutos do Cenário 2 da Aplicação 2

Coluna	Descrição	Tipo
id	Identificador sequencial da medição	Inteiro
timestamp	Data e hora da medição	ISO-8601
unit	Unidade de medição da temperatura de ponto de orvalho	Texto
s1	Temperatura de ponto de orvalho medida pelo sensor 1	Fracionário
s2	Temperatura de ponto de orvalho medida pelo sensor 2	Fracionário
s3	Temperatura de ponto de orvalho medida pelo sensor 3	Fracionário

O aplicativo STEAM recebe, processa e publica os dados, tanto salvando os resultados em um arquivo de log local quanto os enviando para um painel Node-RED rodando em um laptop (*Terminais*) conectado à rede local via Wi-Fi. O aplicativo de painel simplesmente recebe os dados por meio de um serviço HTTP e os exibe em um gráfico de linhas ou área de texto, sem realizar nenhum processamento de dados.

Essa aplicação, cujo código fonte está representado na Figura 33, lê um fluxo de dados de entrada contendo as medições de 3 sensores e inicialmente detecta valores ausentes (linha 3), disparando um evento. Depois, da linha 21 a 26, é calculada a variação instantânea da temperatura do ponto de orvalho, comparando o valor atual com a medição anterior para cada sensor. Como os sensores monitoram o mesmo processo industrial, a discordância da taxa de variação entre as medições indica uma anomalia, disparando outro evento conforme definido entre as linhas 29 e 33. Um painel de controle implementado em Node-RED, hospedado em um laptop conectado à rede administrativa da fábrica, recebe os valores capturados dos sensores além dos eventos e dos dados calculados pelo aplicativo STEAM. Um gráfico de linha representa as medições de cada sensor, e uma área de texto exibe as mensagens dos eventos, como medições ausentes e desacordos na variação da temperatura indicada pelos 3 sensores, configurados nas linhas 2 a 8 e 11 a 17.

6.2.3 Cenário de Teste 3 da Aplicação 2

Os cenários de teste 1 e 2 apresentados anteriormente executaram a uma taxa de processamento de 1 pacote de dados por segundo. No entanto, uma contribuição esperada da arquitetura STEAM é permitir o desenvolvimento de aplicações IoT que forneçam resultados em tempo real. Para identificar os limites de velocidade de processamento e consumo de recursos computacionais, os dados dos sensores do *Cenário 2* foram armazenados em um arquivo de texto, e em seguida, foram enviados ao aplicativo do *Cenário 3* na maior taxa de fluxo de dados que ele conseguia processar. Os experimentos foram repetidos 30 vezes para obtenção de um conjunto de resultados confiável.

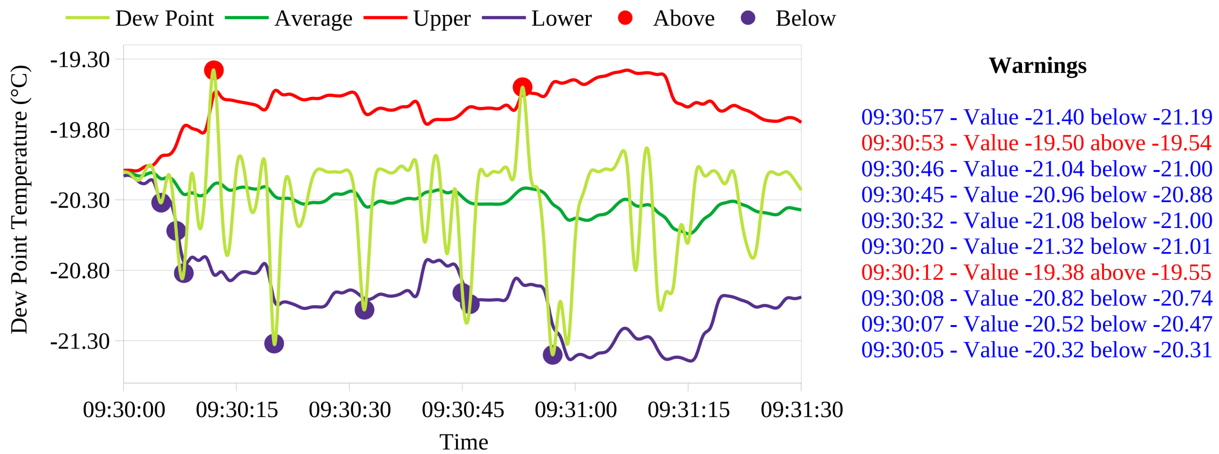
Figura 33: Trecho do código fonte da *Aplicação 2 - Cenário 2* desenvolvida com o *framework* STEAM. Sobre uma janela deslizante dos últimos 20 dados recebidos a partir de 3 sensores, são identificados valores faltantes e desacordo na variação do ponto de orvalho, emitindo eventos em ambos os casos. A aplicação também salva dados em um arquivo de log além de integrar com um painel de controle externo.

```
# Node-RED - Missing value - Message format
ep_missing = HTTPEndpoint(url='http://node-red-server:1880/msgstream')
con_missing = MissingValueCondition(columns=['s1', 's2', 's3'])
fmt_missing = MessageFormat(
  '<font color=blue>{timestamp} - Value missing</font><br>')
ep_missing.setCondition(con_missing)
ep_missing.setFormat(fmt_missing)
device.addEndpoint(ep_missing)

# Node-RED - Slope disagreement - Message format
ep_slope = HTTPEndpoint(url='http://node-red-server:1880/msgstream')
con_slope = EquationCondition('slope_disagree')
fmt_slope = MessageFormat(
  '<font color=red>{timestamp} - Slope disagreement</font><br>')
ep_slope.setCondition(con_slope)
ep_slope.setFormat(fmt_slope)
device.addEndpoint(ep_slope)

# Analytical function list
# Slope of each sensor measurements
device.addFunction(
  fn.Slope(id='s1_slope', batchlen=2, attribute='s1', format='{:.1f}'))
device.addFunction(
  fn.Slope(id='s2_slope', batchlen=2, attribute='s2', format='{:.1f}'))
device.addFunction(
  fn.Slope(id='s3_slope', batchlen=2, attribute='s3', format='{:.1f}'))

# Slope disagreement detection - Equation
device.addFunction(
    fn.Equation(
        id='slope_disagree',
        equation='''max(s1_slope, s2_slope, s3_slope) > 0.1 and
                    min(s1_slope, s2_slope, s3_slope) < -0.1'''))
```

6.2.4 Realização dos Experimentos e Resultados da Aplicação 2

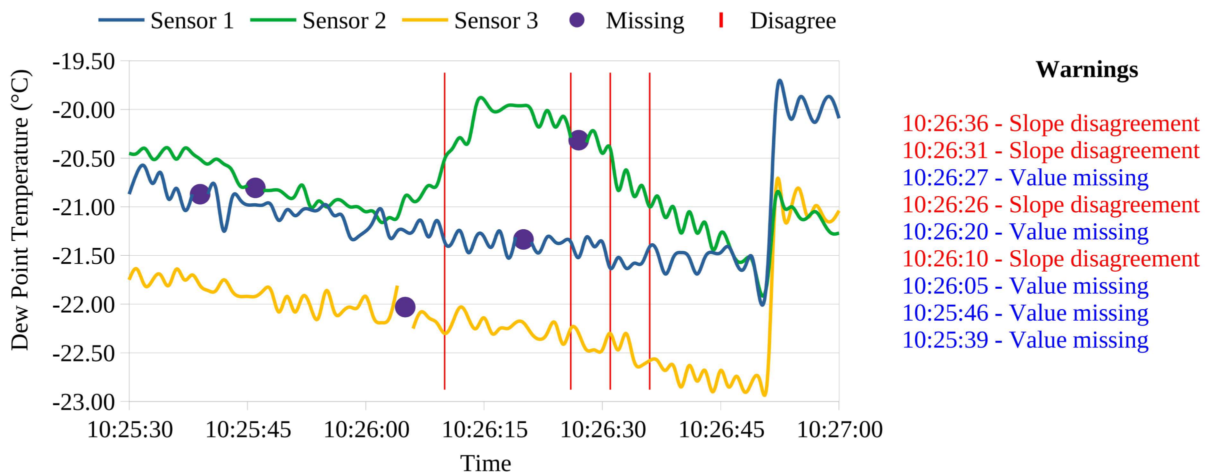
O primeiro resultado obtido na realização do *Cenário 1* é o ponto de vista do usuário, ou seja, dois painéis de controle Node-RED para visualização de dados, um contendo um gráfico de linhas e outro apresentando a exibição de eventos relevantes em formato texto, ilustrados na figura 34. A linha vermelha superior e a linha azul inferior são os *thresholds* dinâmicos superior e inferior, respectivamente, calculados pela equação $\bar{m} \pm 3\sigma$. A linha verde centralizada é a média móvel $\bar{m}$, e a linha azul oscilante é a temperatura do ponto de orvalho lida do sensor. O gráfico também mostra círculos vermelhos onde a temperatura do ponto de orvalho excede o limite superior e círculos azuis onde a temperatura fica abaixo do limite inferior. No lado direito do gráfico, uma área de texto exibe mensagens contendo os eventos detectados pelo aplicativo. Em azul estão os avisos relacionados a valores baixos e em vermelho estão as mensagens associadas a valores altos.

Figura 34: Captura de tela do painel de controle Node-RED para a *Aplicação 2 - Cenário 1*. No lado esquerdo, são visualizadas as linhas dos valores medidos do sensor e sua média móvel, os *thresholds* dinâmicos superior e inferior, além de círculos que indicam quando o ponto de orvalho ultrapassou algum limite. No lado direito, uma área de texto exibe as mensagens dos eventos que ocorreram quando as medições excederam os limites.



O painel de controle do *Cenário 2* é ilustrado na Figura 35. O gráfico está traçando três linhas, representando três sensores de temperatura do ponto de orvalho. Os círculos azuis apontam valores faltantes, indicando a ausência de leitura do sensor ou falha de transmissão, causando a falta de valores nas séries temporais. As linhas verticais vermelhas indicam discordância de declividade entre as medições do sensor, conforme detalhado na seção 5.3. Como o painel anterior, este também exibe mensagens de aviso. Em azul estão os alertas de valores ausentes e em vermelho estão as mensagens indicando desacordo nas declividade das medições.

Figura 35: Captura de tela do painel de controle Node-RED para a *Aplicação 2 - Cenário 2*. No lado esquerdo, são visualizadas as linhas das medições dos três sensores. Os círculos azuis indicam valor ausente e as linhas vermelhas verticais indicam desacordo de inclinação entre os valores do sensor. No lado direito, uma área de texto exibe as mensagens relativas aos eventos previamente apontados pelos círculos e linhas verticais.



Devido aos recursos computacionais limitados, o uso de CPU e memória são indicadores-chave no ambiente IoT. Para ter uma visão geral significativa e confiável do consumo de recursos, cada cenário foi executado 30 vezes, coletando a carga instantânea da CPU do sistema e a memória geral usada. As figuras 36 e 37 mostram os comportamentos típicos do *Cenário 1* e *Cenário 2* da *Aplicação 2*, respectivamente, em relação ao consumo de CPU e memória. Em ambos os casos, a carga média da CPU ficou abaixo de 1 % com picos inferiores a 2,5 %, e o uso médio da memória foi menor que 500kb, com picos abaixo de 800kb, excluindo os *outliers*. Os valores exatos são detalhados na Tabela 7, e a distribuição dos dados é apresentada na Figura 38.

Figura 36: Comportamento típico do uso de CPU e memória para a *Aplicação 2 - Cenário 1*. A carga média da CPU ficou abaixo de 1% com picos inferiores a 2,5 %, e o uso médio da memória ficou em torno de 500kb, com picos abaixo de 800kb, excluindo os *outliers*.

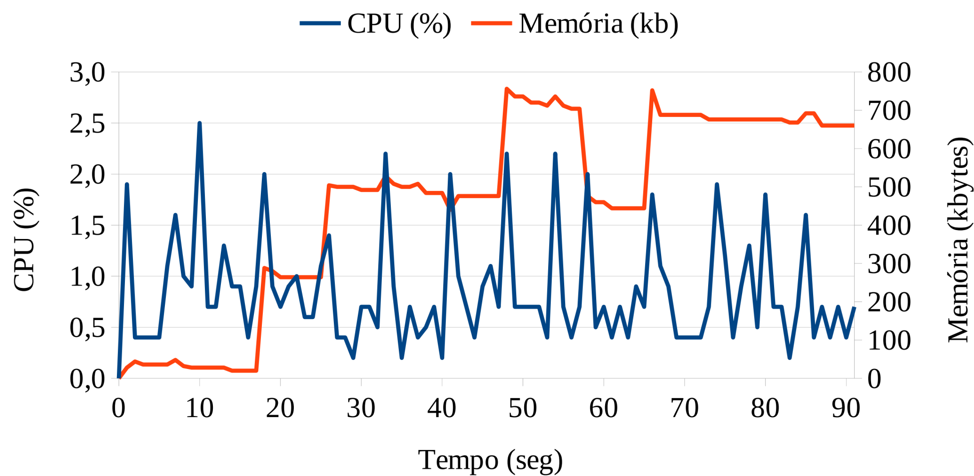


Figura 37: Comportamento típico do uso de CPU e memória para a *Aplicação 2 - Cenário 2*. A carga média da CPU ficou abaixo de 1% com picos inferiores a 2,5 %, e o uso médio da memória ficou em torno de 430kb, com picos abaixo de 600kb, excluindo os *outliers*.

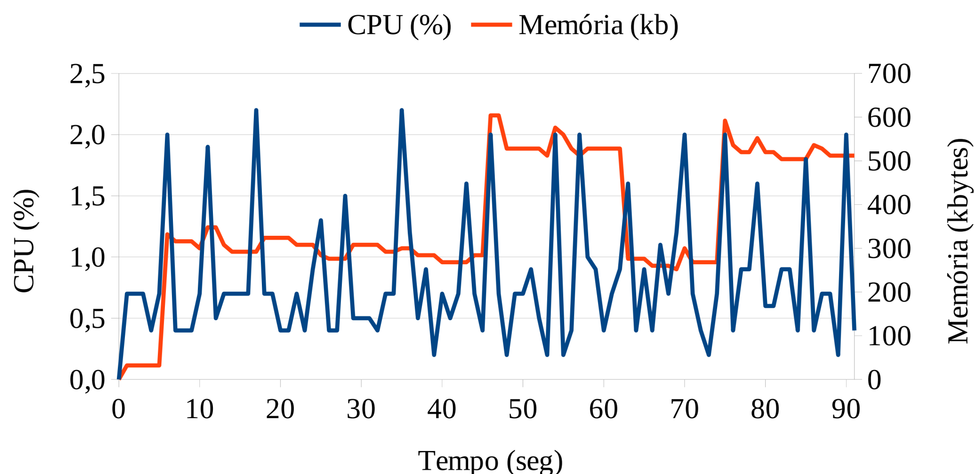
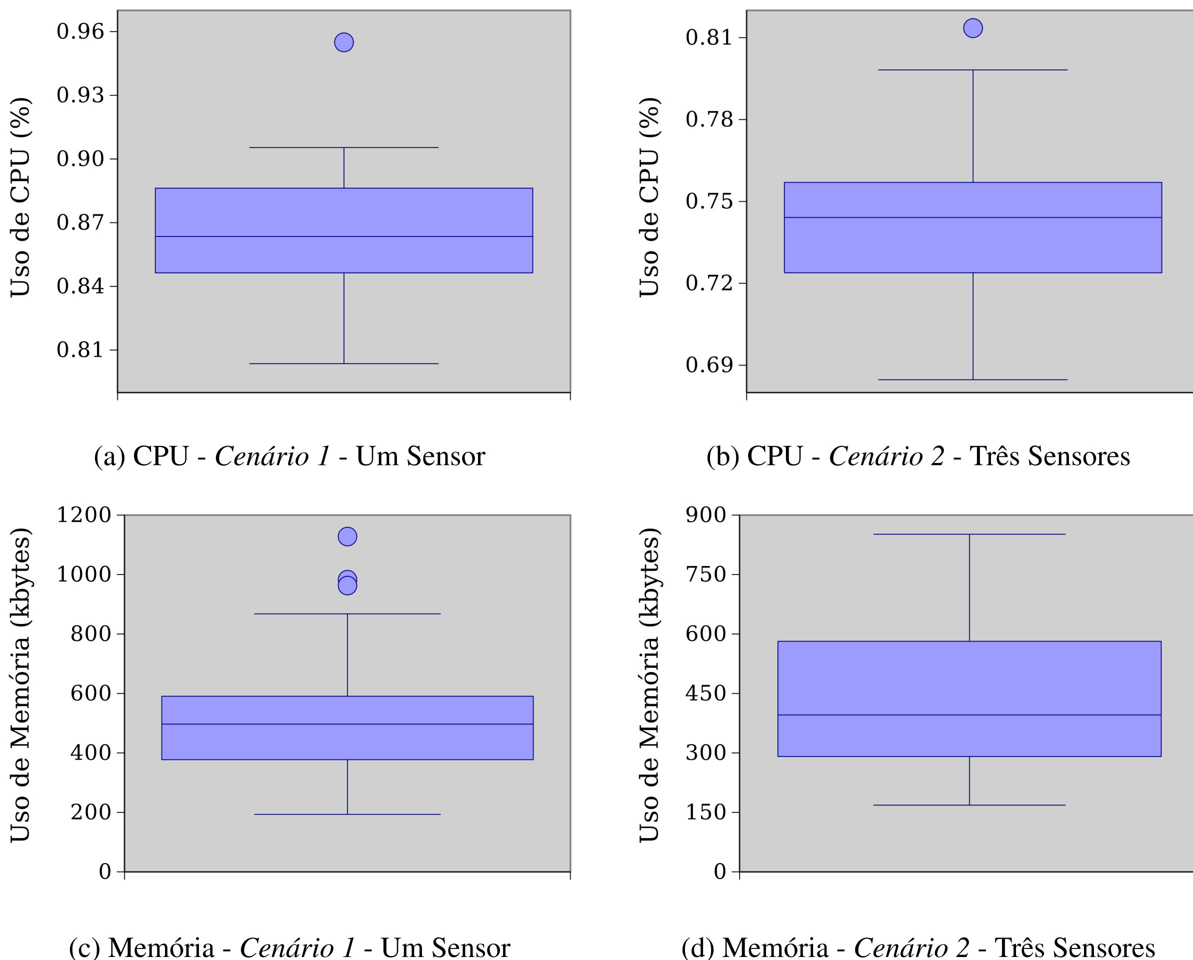


Tabela 7: Uso médio de CPU e memória - compilação de 30 experimentos do *Cenário 1* com 1 sensor e *Cenário 2* com 3 sensores.

Cenário	Cenário 1 - Um Sensor		Cenário 2 - Três Sensores	
Métrica	CPU (%)	Memória (kb)	CPU (%)	Memória (kb)
Mínimo	0,804	193,41	0,685	168,25
Máximo	0,955	1127,93	0,814	851,64
Média	0,867	523,82	0,741	435,64
Mediana	0,864	496,92	0,744	395,78
1º quartil	0,846	377,49	0,724	291,05
3º quartil	0,886	590,72	0,757	581,72

Figura 38: Distribuição de CPU e memória - compilação de 30 experimentos da *Aplicação 2 - Cenários 1 e 2*, processando um pacote de dados por segundo.



(a) CPU - *Cenário 1* - Um Sensor

(b) CPU - *Cenário 2* - Três Sensores

(c) Memória - *Cenário 1* - Um Sensor

(d) Memória - *Cenário 2* - Três Sensores

Os experimentos inicialmente realizados neste trabalho para avaliação de tempos utilizaram uma taxa de processamento de um pacote por segundo. Aplicando a métrica *Tempo de Processamento* descrita na seção 5.3, foram coletados os tempos gastos nas camadas *Input*, *Processing* e *Output* da aplicação STEAM. A figura 39 apresenta a distribuição do tempo gasto por camada

de processamento. A etapa *Input*, responsável por coletar e analisar os dados brutos dos sensores, é a mais rápida de todas, consumindo em média 728 µs. A camada *Processing*, que realiza cálculos e avalia as condições, usou em média 5554 µs para completar as tarefas. *Output*, a camada mais lenta, consumiu em média 108997 µs para formatar e enviar dados aos *endpoints*, que nesse caso, consistia em salvar um arquivo de log local e também enviar o fluxo de dados enriquecido ao painel de controle Node-RED via protocolo HTTP. Proporcionalmente, o procedimento *Input* consumiu 0,63 %, a camada *Processing* consumiu 4,82 %, e o *Output* registrou 94,55 % do tempo gasto no processamento dos pacotes. A tabela 8 apresenta o detalhamento das métricas do tempo de processamento, e a Figura 40 mostra a distribuição dos dados.

Figura 39: Tempo médio gasto por camada de processamento - 30 experimentos da *Aplicação 2 - Cenários 1 e 2*, com um pacote de dados por segundo - escala logarítmica.

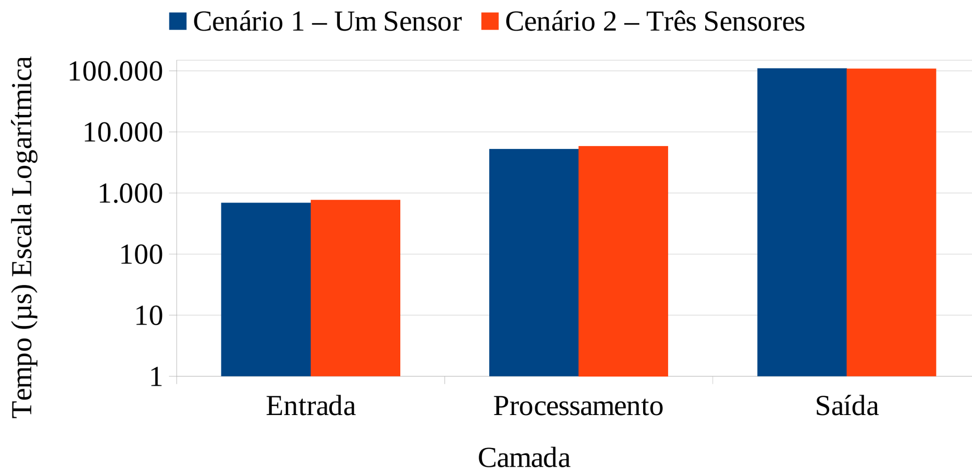
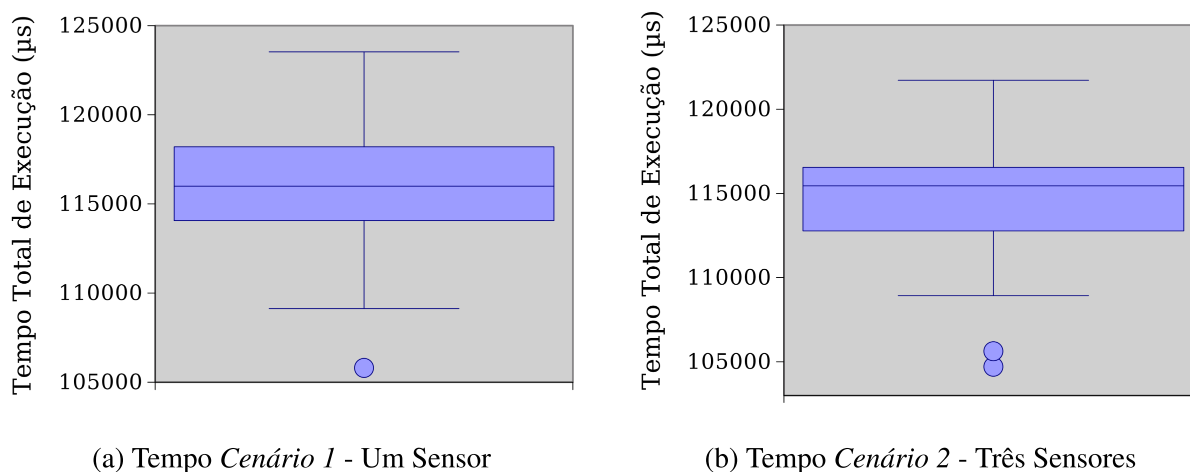


Tabela 8: Tempo médio gasto por camada de processamento - 30 experimentos da *Aplicação 2 - Cenários 1 e 2*, com um pacote de dados por segundo.

Cenário	Cenário 1 - Um Sensor				Cenário 2 - Três Sensores			
Métrica (µs)	Input	Proc.	Output	Total	Input	Proc.	Output	Total
Mínimo	674	5143	99760	105801	746	5727	98169	104713
Máximo	698	5431	117701	123527	810	6120	115166	121722
Média	687	5239	109886	115811	768	5870	108069	114707
Mediana	688	5235	110025	115995	763	5850	108880	115446
1º quartil	680	5199	108107	114061	758	5823	106168	112775
3º quartil	693	5275	112295	118200	773	5884	109873	116548

Os dados brutos recebidos dos sensores geralmente apresentam um tamanho pequeno, contendo apenas informações relevantes em uma estrutura simples. O fluxo de processamento de uma aplicação STEAM calcula novos valores que são incluídos no pacote de dados e formatados como uma estrutura JSON, contendo símbolos, delimitadores e identificadores. Além disso, os dados enviados para os aplicativos clientes podem assumir diversas estruturas e for-

Figura 40: Distribuição do tempo total de execução - compilação de 30 experimentos da *Aplicação 2 - Cenários 1 e 2* com um pacote de dados por segundo.



(a) Tempo *Cenário 1* - Um Sensor

(b) Tempo *Cenário 2* - Três Sensores

matos como XML, HTML e CSV, aumentando o tamanho dos dados publicados. Se por um lado o pacote de dados processado e enriquecido aumenta de tamanho, por outro lado, uma aplicação STEAM pode avaliar expressões e critérios de envio apenas de mensagens relevantes para as aplicações clientes. Esse recurso atua como um filtro, podendo reduzir drasticamente a quantidade de dados transmitidos e, consequentemente, diminuir o tráfego da rede.

Os fluxos de processamento de dados representados nas Figuras 41 e 42 demonstram as diferenças em formatos e tamanhos comparando uma única entrada e o pacote de dados de saída para o *Cenário 1* e *Cenário 2* da *Aplicação 2*. Nesses casos, o aplicativo STEAM recebe o pacote de dados brutos, realiza cálculos e monta o pacote de dados enriquecido no formato JSON. Em seguida, o aplicativo envia o pacote para o painel de controle do Node-RED e, ao mesmo tempo, o converte em uma string *Tab Separated Values* (TSV) e o salva em um arquivo de log.

Na Tabela 9, são apresentadas as diferenças nos tamanhos dos dados detalhados por método de saída e aplicação. Em comparação com os dados brutos, o tamanho do arquivo de log formatado em TSV aumenta entre 141,73% e 182,22% devido à inclusão dos valores calculados. No entanto, a publicação para o gráfico do painel de controle requer um formato JSON, resultando no incremento de 528,65% a 608,18% no tamanho total dos dados, em comparação com os dados brutos. No entanto, só são enviadas mensagens para a área de texto do painel quando um evento é detectado. Isso atua como um filtro sobre os dados processados, diminuindo todo o tamanho dos dados publicados. Nesse caso, o fluxo de dados de saída geral diminuiu para 14,23% e 18,65% dos tamanhos em comparação com o fluxo de dados brutos de entrada.

Figura 41: Transformação dos dados durante o fluxo de processamento da *Aplicação 2 - Cenário 1*. Os dados brutos recebidos do sensor são processados, enriquecidos e convertidos para os formatos TSV, HTML e JSON. Em seguida, são enviados para seus respectivos *endpoints*, que nesse caso são arquivo texto, painel de controle para visualização de eventos e gráficos.

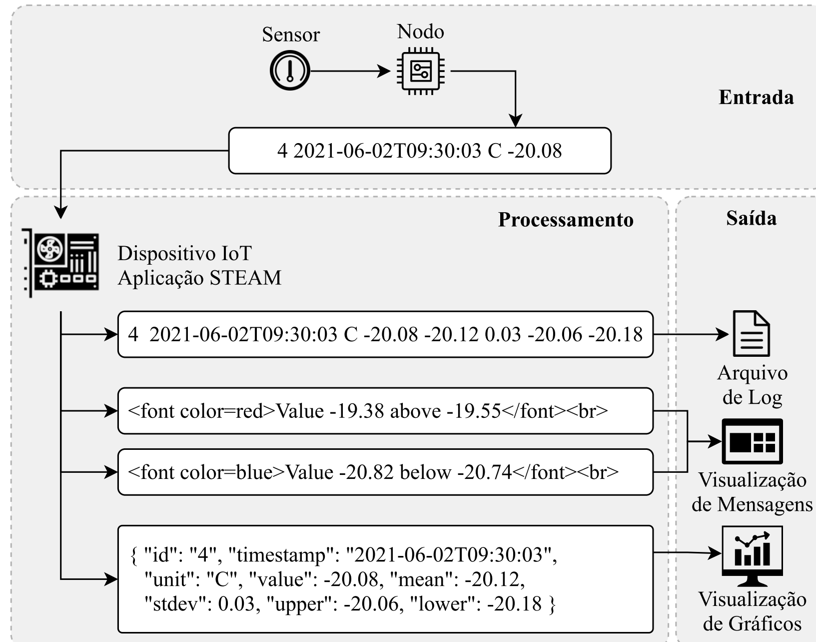


Figura 42: Transformação dos dados durante o fluxo de processamento da *Aplicação 2 - Cenário 2*. Os dados brutos recebidos dos três sensores são processados, enriquecidos e convertidos para os formatos TSV, HTML e JSON. Em seguida, são enviados para seus respectivos *endpoints*, que nesse caso são arquivo texto, painel de controle para visualização de eventos e gráficos.

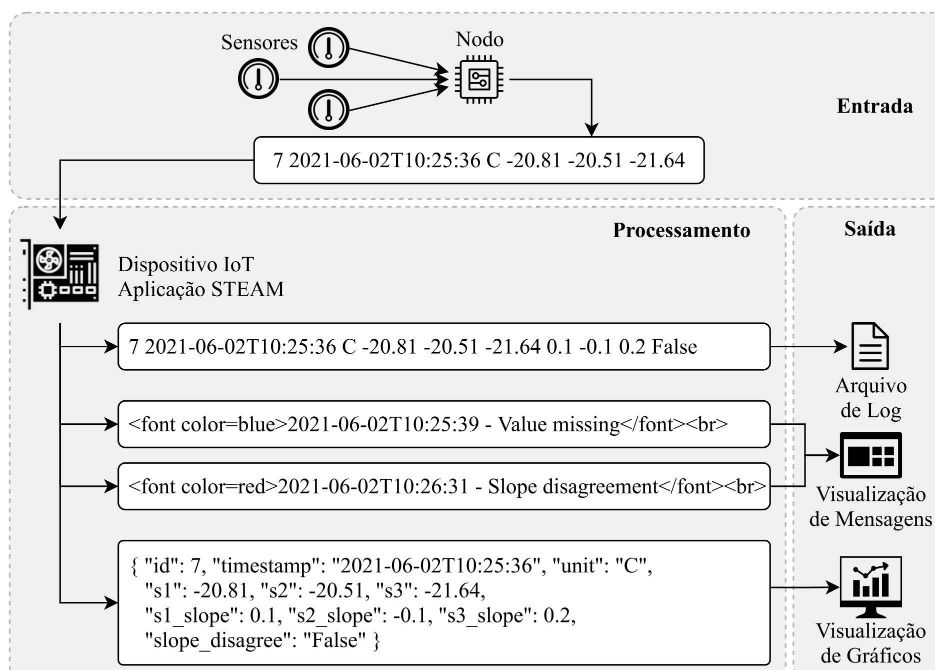
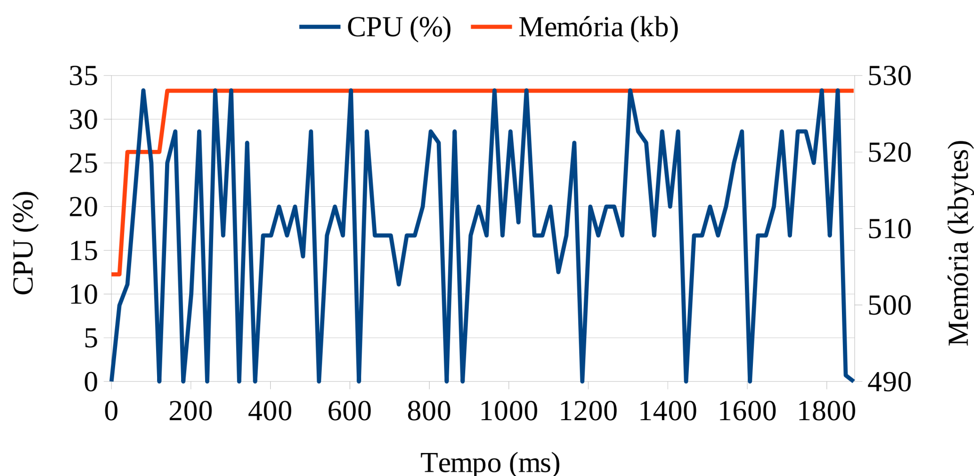


Tabela 9: Razão entre os tamanhos dos pacotes de dados medidos na saída e entrada

Cenário	Cenário 1 - Um Sensor		Cenário 2 - Três Sensores	
Métrica	Tamanho (bytes)	Razão	Tamanho (bytes)	Razão
Dados Brutos de Entrada	2874	-	4146	-
Arquivo de log (CSV)	5237	182,22%	5876	141,73%
Painel de Controle Gráfico	17479	608,18%	21918	528,65%
Painel de Controle Mensagens	536	18,65%	590	14,23%

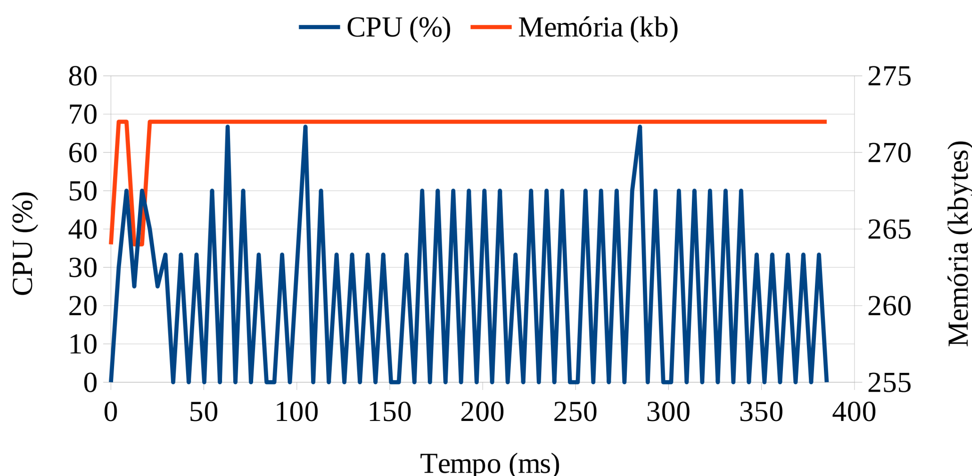
Até o momento, as aplicações implementadas e cenários executados nos experimentos apresentaram baixo consumo de CPU e memória, realizando uma taxa de processamento de um pacote de dados por segundo. No entanto, uma contribuição esperada do modelo STEAM é permitir o desenvolvimento de aplicações IoT com respostas próximas do tempo real. Para identificar os limites de velocidade de processamento e consumo de recursos computacionais, foi elaborado um cenário de teste de estresse, identificado como *Cenário 3*. Inicialmente, os dados dos sensores foram armazenados em um arquivo de texto, e em seguida, o aplicativo foi alimentado a partir desse arquivo na maior taxa de dados que ele pudesse processar. Esse teste foi repetido 30 vezes para se obter um conjunto de resultados confiável. No primeiro teste de estresse, foi utilizada a mesma *Aplicação 2* detalhada na Subseção 6.2.2, mas como dito anteriormente, a leitura dos sensores foi simulada e a transmissão de dados foi forçada ao limite da velocidade de processamento. A Figura 43 ilustra um teste típico desse cenário, no qual a carga média da CPU atingiu 15,4% com picos de 33,3%, e o consumo médio de memória foi 527,04kb. Considerando todos os 30 testes, a carga da CPU registrou 15,4% e a memória 289,81kb em média.

Figura 43: Comportamento típico do uso de CPU e memória para a *Aplicação 2 - Cenário 3*, utilizando um fluxo de dados com alta velocidade e publicando dados e mensagens para um painel de controle. A carga média da CPU ficou em torno de 15% com picos inferiores a 35%, e o uso médio da memória ficou em torno de 530kb.



Para o segundo teste de estresse, a publicação de dados e mensagens no painel de controle do Node-RED foi removida, uma vez que a comunicação de rede HTTP é uma tarefa normalmente lenta em comparação com o acesso a recursos locais. Nesse cenário, os dados e mensagens processados foram salvos em um arquivo de log local. Para ilustrar o comportamento típico desse cenário, um teste arbitrário pode ser visualizado na Figura 44, mas o experimento completo também contemplou 30 execuções. A carga média da CPU para este teste específico atingiu 22,4% com picos de 66,7%, e o uso médio de memória foi de 271,74kb. Compilando todos os 30 testes, a carga da CPU atingiu 18,0% e o consumo de memória foi de 196,91kb, em média.

Figura 44: Comportamento típico do uso de CPU e memória para a *Aplicação 2 - Cenário 3*, utilizando um fluxo de dados com alta velocidade e publicando dados e mensagens em um arquivo de log. A carga média da CPU ficou em torno de 22% com picos de 50%, excluindo os *outliers*, e o uso médio da memória ficou em torno de 270kb.



Em relação ao tempo de processamento, o primeiro teste foi concluído em 1827,578 ms, e o segundo em 380,895 ms, em média. Analisando minuciosamente essas medições, foi possível identificar que a diferença significativa de tempo ocorreu no processo de publicação de dados no painel de controle hospedado no computador remoto. Convertendo essas medidas em pacotes processados por tempo, o primeiro cenário com gravação dos resultados em arquivo de log e publicação no painel de controle conseguiu processar 49,79 pacotes por segundo, enquanto o segundo cenário, apenas salvando os dados em arquivo de log, alcançou uma taxa de 238,91 pacotes por segundo. Em outras palavras, ao enviar dados para o painel de controle, cada pacote consumiu 20083 μs, e ao salvar em um arquivo de log local, o mesmo processo durou 4186 μs. A tabela 10 apresenta a compilação do tempo gasto por cada camada de processamento coletada nos 30 testes de estresse.

Comparando os testes de estresse com os testes de caso real, mais especificamente a etapa de saída, foi possível notar uma discordância significativa entre os tempos decorridos no envio de dados para o painel de controle do Node-RED. No teste de caso real identificado como

Tabela 10: Tempo médio gasto por camada de processamento - 30 experimentos da *Aplicação 2 - Cenário 3*, teste de estresse.

Saída	Painel				Arquivo			
Métrica (µs)	Input	Proc.	Output	Total	Input	Proc.	Output	Total
Mínimo	345	3133	11995	15503	354	3306	317	3993
Máximo	427	3891	52688	57005	409	3553	421	4298
Média	369	3302	16413	20083	372	3467	347	4186
Mediana	366	3275	12800	16448	372	3497	339	4214
1º quartil	360	3217	12629	16234	366	3430	332	4120
3º quartil	369	3315	13724	17460	377	3518	360	4251

Aplicação 2 - Cenário 2, enquanto o tempo médio gasto pela etapa de saída foi de 108978 µs, essa mesma etapa realizada no teste de estresse identificado como *Aplicação 2 - Cenário 3*, consumiu 16413 µs em média, processando exatamente os mesmos dados. Analisando o status da rede com o comando *netstat*, foi possível identificar conexões inativas entre a Raspberry Pi e o Node-RED ao enviar um pacote por segundo, ilustrado na Figura 45. No entanto, foi possível verificar três conexões estabelecidas e nenhuma conexão inativa durante os testes de estresse, representado na Figura 46. A necessidade de estabelecer novas conexões após a espera de um segundo resultou em um aumento geral do tempo medido na camada de saída, entretanto, isso não ocorreu no teste de estresse, que usa as mesmas conexões durante todo o experimento.

Figura 45: Status da rede em um teste de caso real - *Aplicação 2 - Cenário 2*: O intervalo de um segundo entre o envio dos pacotes ao painel de controle causou o fechamento da conexão, exigindo a abertura de uma nova conexão, aumentando o tempo gasto no processo de publicação.

```
Proto   Foreign Address       State         PID/Program name
tcp     192.168.1.120:55550   ESTABLISHED   13174/node-red
tcp     192.168.1.120:55556   TIME_WAIT
tcp     192.168.1.120:55558   TIME_WAIT
tcp     192.168.1.120:55560   TIME_WAIT
tcp     192.168.1.120:55562   TIME_WAIT
tcp     192.168.1.120:55564   FIN_WAIT2
tcp     192.168.1.120:55566   TIME_WAIT
tcp     192.168.1.120:55568   FIN_WAIT2
```

Figura 46: Status da rede de um teste de estresse - *Aplicação 2 - Cenário 3*: O curto intervalo de tempo médio entre o envio dos pacotes ao painel de controle manteve a conexão aberta, permitindo o uso de três fluxos de dados simultâneos, diminuindo o tempo gasto no processo de publicação.

```
Proto   Foreign Address       State         PID/Program name
tcp     192.168.1.120:55570   ESTABLISHED   13174/node-red
tcp     192.168.1.120:55572   ESTABLISHED   13174/node-red
tcp     192.168.1.120:55574   ESTABLISHED   13174/node-red
```

6.3 Comparação dos Resultados

Para finalizar a análise dos resultados, também foi feita uma comparação das aplicações STEAM com a literatura apresentada no capítulo 3. Não foi possível fazer análises quantitativas, pois os trabalhos relacionados não forneceram detalhes sobre a implementação dos experimentos, bem como a fonte dos conjuntos de dados. Portanto, foi feita uma comparação qualitativa, apresentada na Tabela 11. Os aplicativos desenvolvidos nos trabalhos relacionados eram voltados para fins específicos, exigindo desenvolvimento *hard-coded* não apenas para a estrutura do software, mas também para entrada, saída, análise e lógica de dados. No entanto, os aplicativos STEAM são para fins gerais, construídos sobre uma API de classes padrão e auxiliados por uma biblioteca de funções analíticas. Além disso, em aplicações STEAM, a aquisição e exportação de dados são gerenciadas respectivamente pelas camadas *Abstração de Dispositivos* e *Conector de Protocolos*, fornecendo abstração e extensibilidade de maneira transparente.

Tabela 11: Comparação qualitativa entre a literatura e o modelo STEAM

Característica	Literatura	STEAM
Aplicação	Específica	Geral
Desenvolvimento	*Hard-Coded*	API de Classes
Análise de Dados	*Hard-Coded*	Biblioteca de Funções
Aquisição de Dados	Específica	Abstração de Dispositivos
Exportação de Dados	Específica	Conector de Protocolos

7 CONCLUSÃO

Tendo como cenário a implementação de aplicações de Internet das Coisas com processamento na borda da rede, essa tese apresentou STEAM, um modelo e *framework* para facilitar o desenvolvimento de aplicações IoT autônomas, permitindo análise de dados em tempo real, enriquecimento de fluxos de dados e tomada de decisão na borda, além de permitir a publicação dos dados processados para uma grande variedade de serviços, formatos e protocolos. O *framework* fornece um conjunto de classes prontas para uso e funções integradas, tornando desnecessário o uso de código estruturado, lógica complexa, loops e instruções condicionais. Esse recurso permite que mesmo não programadores tenham a possibilidade de desenvolver aplicativos IoT ricos, simplesmente configurando parâmetros.

Para demonstrar a viabilidade do modelo e *framework* propostos, foram desenvolvidas duas aplicações e quatro cenários de testes reais em uma indústria de semicondutores, obtendo resultados consistentes. A primeira aplicação consistiu em realizar diversas análises estatísticas na borda sobre um fluxo de dados de um sensor de temperatura, e comparar com o mesmo processamento realizado por uma *CEP Engine* rodando na nuvem. Esse experimento comprovou a autonomia do modelo STEAM, sendo capaz de obter na borda os mesmos resultados calculados na nuvem, possibilitando a eliminação da dependência de serviços externos mais pesados, mais lentos e mais caros.

A segunda aplicação explorou recursos de conectividade e tomada de decisão na borda, fornecendo tempos de resposta muito rápidos na detecção de eventos relevantes ao processo. Nesse experimento, foram utilizados fluxos de dados tanto de um quanto de vários sensores de ponto de orvalho instalados na planta industrial. A aplicação STEAM realizou detecção de medições ausentes, discordância na variação dos valores dos sensores e identificação de valores além ou aquém de limites dinâmicos utilizando princípios de Controle Estatístico de Processo. Além desses eventos, a aplicação também alimentou um painel de controle implementado em Node-RED, com gráficos e exibição de mensagens em tempo real.

7.1 Contribuições

As principais contribuições dessa tese são:

- O modelo STEAM, contendo 5 camadas: i) Abstração de Dispositivos e Coleta de Dados; ii) Análise de Dados; iii) Enriquecimento; iv) Avaliação; v) Conector de Protocolos;
- O *framework* STEAM, oferecendo um conjunto de classes e funções prontas para uso,

tornando desnecessário o emprego de código estruturado, lógica complexa, estruturas de repetição e instruções condicionais ao desenvolver aplicativos IoT para serem executados por dispositivos com poucos recursos computacionais na borda da rede;

- Uma metodologia de micro-benchmark para avaliação de aplicações IoT com três métricas: *Uso de CPU / Memória*, *Tempo de Processamento* e *Razão Saída/Entrada*;
- Validação do modelo e *framework* STEAM através da aplicação exaustiva do micro-benchmark em duas aplicações e quatro cenários de teste reais em uma indústria de semicondutores;

Durante a realização dessa tese, foram publicados 2 artigos diretamente relacionados ao assunto:

- **Gomes, M. M.**, da Rosa Righi, R., da Costa, C. A., e Griebler, D. *Simplifying IoT Data Stream Enrichment and Analytics in the Edge*. Computers & Electrical Engineering, 92:107110,2021. ISSN 0045-7906. doi: `http://dx.doi.org/10.1016/j.compeleceng.2021.107110`
- **Gomes, M. M.**, da Rosa Righi, R., da Costa, C. A., e Griebler, D. *STEAM++ An Extensible End-To-End Framework For Developing IoT Data Processing Applications In The Fog*. International Journal of Computer Science and Information Technology, 2022

Além dos artigo supramencionados, foram publicados outros artigos como co-autor, abordando assuntos diversos, mas que contribuíram diretamente para a realização dessa tese com ideias ou técnicas:

- da Rosa Righi, R., Lehmann, M., **Gomes, M. M.**, Nobre, J. C., da Costa, C. A., Rigo, S. J., et al (2019). *A Survey on Global Management View: Toward Combining System Monitoring, Resource Management, and Load Prediction*. Journal of Grid Computing, 1-30. `http://dx.doi.org/10.1007/s10723-018-09471-x`
- da Rosa Righi, R., **Gomes, M. M.**, da Costa, C. A., Parzyjegla, H., Heiss, H. U. (2019). *Towards Using Cloud Elasticity on the Internet of Things Landscape*. Big Data and Cloud Innovation, v. 3, p. 1-20. `http://dx.doi.org/10.18063/bdci.v3i1.835`
- da Rosa Righi, R., Correa, E., **Gomes, M. M.**, da Costa, C. A. (2018). *Enhancing performance of IoT applications with load prediction and cloud elasticity*. Future Generation Computer Systems. `http://dx.doi.org/10.1016/j.future.2018.06.026`

- da Rosa Righi, R., Suad, F., Correa, E., **Gomes, M. M.**, da Costa, C. A., Bertoldi, L. R. (2018). *Exploring Extensibility and Interoperability in the Internet of Things Landscape*. International Conferences WWW/Internet 2018 and Applied Computing 2018. p. 339-343. `http://toc.proceedings.com/42252webtoc.pdf`

7.2 Trabalhos Futuros

Embora os experimentos tenham demonstrado a viabilidade do modelo e *framework* STEAM para o desenvolvimento de aplicações IoT, foram identificadas diversas oportunidades para aprimoramentos e extensões que podem ser abordadas em trabalhos futuros. A IoT está evoluindo diariamente e se espalhando para diversas áreas como saúde, transporte, agricultura e indústria, enfrentando situações específicas e requisitos desafiadores. Nesse sentido, a utilização do STEAM nessas áreas pode contribuir tanto para a disseminação da IoT quanto para a evolução do modelo e *framework* aqui apresentados.

Por estar na sua primeira versão, o *framework* STEAM pode receber uma ampliação das suas classes e funções para atender a mais protocolos de comunicação, tanto para rede de sensores quanto para aplicações ou serviços cliente, mais funções analíticas e também de avaliação de resultados para processamento de eventos complexos. Outra oportunidade de trabalho seria estender o modelo STEAM para permitir escalabilidade e elasticidade, atendendo dessa forma a demanda dinâmica e crescente de soluções IoT.

Atualmente, o *framework* suporta aplicações com apenas um fluxo de dados. Adicionar suporte para processamento de diversos fluxos em paralelo ampliaria as possibilidades de uso em casos reais, em aplicações com múltiplos sensores.

A última sugestão seria implementar o *framework* em outras linguagens, como C++ por exemplo, para atender dispositivos embarcados com fortes restrições de recursos computacionais como os micro-controladores ESP e ATMega.

REFERÊNCIAS

M. Abe. **Manual de análise técnica: essência e estratégias avançadas: tudo o que um investidor precisa saber para prosperar na Bolsa de valores até em tempos de crise**. Novatec Editora, 2018.

A. Abraham. Rule-based expert systems. **Handbook of measuring system design**, 2005.

A. Adi and O. Etzion. Amit - the situation manager. **The VLDB Journal The International Journal on Very Large Data Bases**, 13(2):177–203, may 2004. ISSN 1066-8888. doi: 10.1007/s00778-003-0108-y.

R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. **ACM SIGMOD Record**, 22(2):207–216, jun 1993. ISSN 01635808. doi: 10.1145/170036.170072.

V. Akila, V. Govindasamy, and S. Sandosh. Complex event processing over uncertain events: Techniques, challenges, and future directions. In **2016 International Conference on Computation of Power, Energy Information and Commuincation (ICCPEIC)**, pages 204–221. IEEE, apr 2016. ISBN 978-1-5090-0901-5. doi: 10.1109/ICCPEIC.2016.7557198.

N. Al Bassam, S. A. Hussain, A. Al Qaraghuli, J. Khan, E. Sumesh, and V. Lavanya. Iot based wearable device to monitor the signs of quarantined remote patients of covid-19. **Informatics in Medicine Unlocked**, 24:100588, 2021. ISSN 2352-9148. doi: https://doi.org/10.1016/j.imu.2021.100588.

A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. **IEEE Communications Surveys & Tutorials**, 17(4):2347–2376, 2015. ISSN 1553-877X. doi: 10.1109/COMST.2015.2444095.

K. A. Alam, R. Ahmad, and K. Ko. Enabling Far-Edge Analytics: Performance Profiling of Frequent Pattern Mining Algorithms. **IEEE Access**, 5:8236–8249, 2017. ISSN 2169-3536. doi: 10.1109/ACCESS.2017.2699172.

M. I. Ali, P. Patel, and J. G. Breslin. Middleware for Real-Time Event Detection andPredictive Analytics in Smart Manufacturing. In **2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)**, pages 370–376. IEEE, may 2019. ISBN 978-1-7281-0570-3. doi: 10.1109/DCOSS.2019.00079.

K. Ashton. That ‘internet of things’ thing. **RFID journal**, 22(7):97–114, 2009.

M. Babazadeh. Edge analytics for anomaly detection in water networks by an Arduino101-LoRa based WSN. **ISA Transactions**, feb 2019. ISSN 00190578. doi: 10.1016/j.isatra.2019.01.015.

R. Bharath Das, G. Di Bernardo, and H. Bal. Large Scale Stream Analytics Using a Resource-Constrained Edge. In **2018 IEEE International Conference on Edge Computing (EDGE)**, pages 135–139. IEEE, jul 2018. ISBN 978-1-5386-7238-9. doi: 10.1109/EDGE.2018.00027.

K. Bhargava, S. Ivanov, W. Donnelly, and C. Kulatunga. Using Edge Analytics to Improve Data Collection in Precision Dairy Farming. In **2016 IEEE 41st Conference on Local Computer Networks Workshops (LCN Workshops)**, pages 137–144. IEEE, nov 2016. ISBN 978-1-5090-2347-9. doi: 10.1109/LCN.2016.039.

J. Biolchini, P. G. Mian, A. C. C. Natali, and G. H. Travassos. Systematic review in software engineering. **System Engineering and Computer Science Department COPPE/UFRJ, Technical Report ES**, 679(05):45, 2005.

C. Bourelly, A. Bria, L. Ferrigno, L. Gerevini, C. Marrocco, M. Molinara, G. Cerro, M. Cicalini, and A. Ria. A preliminary solution for anomaly detection in water quality monitoring. In **2020 IEEE International Conference on Smart Computing (SMARTCOMP)**, pages 410–415. IEEE, 2020. doi: 10.1109/SMARTCOMP50058.2020.00086.

G. E. Box and G. M. Jenkins. Time series analysis forecasting and control. Technical report, WISCONSIN UNIV MADISON DEPT OF STATISTICS, 1970.

G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. **Time series analysis: forecasting and control**. John Wiley & Sons, 2015.

T. Bray. The javascript object notation (json) data interchange format. 2014.

R. Buyya, R. N. Calheiros, and A. V. Dastjerdi. **Big data: principles and paradigms**. Morgan Kaufmann, 2016.

J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In **ACM SIGMOD Record**, volume 29, pages 379–390. ACM, 2000.

B. Cheng, A. Papageorgiou, and M. Bauer. Geelytics: Enabling On-Demand Edge Analytics over Scoped Data Sources. In **2016 IEEE International Congress on Big Data (BigData Congress)**, pages 101–108. IEEE, jun 2016. ISBN 978-1-5090-2622-7. doi: 10.1109/BigDataCongress.2016.21.

M. Dayarathna and S. Perera. Recent Advancements in Event Processing. **ACM Computing Surveys**, 51(2):1–36, feb 2018. ISSN 03600300. doi: 10.1145/3170432.

A. M. de Livera, R. J. Hyndman, and R. D. Snyder. Forecasting time series with complex seasonal patterns using exponential smoothing. **Journal of the American Statistical Association**, 106(496):1513–1527, 2011. ISSN 01621459. doi: 10.1198/jasa.2011.tm09771.

F. De Vita, D. Bruneo, and S. K. Das. A novel data collection framework for telemetry and anomaly detection in industrial iot systems. In **2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)**, pages 245–251. IEEE, 2020. doi: 10.1109/IoTDI49375.2020.00032.

B. Ellis. **Real-time analytics: Techniques to analyze and visualize streaming data**. John Wiley & Sons, 2014.

O. Etzion. Complex Event Processing. In **Encyclopedia of Database Systems**, volume 51, pages 1–2. Springer New York, New York, NY, 2016. ISBN 4971178163. doi: 10.1007/978-1-4899-7993-3_571-2.

O. Etzion and P. Niblett. **Event Processing in Action**. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. ISBN 1935182218, 9781935182214.

E. Foufoula-Georgiou, P. Kumar, T. Mukerji, and G. Mavko. Wavelets in geophysics. **Pure and Applied Geophysics**, 145(2):374–375, 1995.

F. Fournier, A. Kofman, I. Skarbovsky, and A. Skarlatidis. Extending event-driven architecture for proactive systems. In **EDBT/ICDT Workshops**, pages 104–110, 2015.

A. Galanopoulos, A. G. Tasiopoulos, G. Iosifidis, T. Salonidis, and D. J. Leith. Improving iot analytics through selective edge execution. **arXiv preprint arXiv:2003.03588**, 2020.

E. S. Gardner Jr. Exponential smoothing: The state of the art. **Journal of forecasting**, 4(1): 1–28, 1985.

D. Goldsmith and J. Brusey. The spanish inquisition protocol—model based transmission reduction for wireless sensor networks. In **Sensors, 2010 ieee**, pages 2043–2048. IEEE, 2010.

L. Greco, P. Ritrovato, and F. Xhafa. An edge-stream computing infrastructure for real-time analysis of wearable sensors data. **Future Generation Computer Systems**, 93:515–528, 2019. ISSN 0167-739X. doi: 10.1016/j.future.2018.10.058.

J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. **Data Mining and Knowledge Discovery**, 15(1):55–86, jul 2007. ISSN 1384-5810. doi: 10.1007/s10618-006-0059-1.

N. Harth and C. Anagnostopoulos. Quality-aware aggregation and predictive analytics at the edge. In **2017 IEEE International Conference on Big Data (Big Data)**, volume 2018-Janua, pages 17–26. IEEE, dec 2017. ISBN 978-1-5386-2715-0. doi: 10.1109/BigData.2017.8257907.

N. Harth, C. Anagnostopoulos, and D. Pezaros. Predictive intelligence to the edge: impact on edge analytics. **Evolving Systems**, 9(2):95–118, jun 2018. ISSN 1868-6478. doi: 10.1007/s12530-017-9190-z.

Q. P. He and J. Wang. Statistical process monitoring as a big data analytics tool for smart manufacturing. **Journal of Process Control**, 67:35–43, 2018. ISSN 0959-1524. doi: 10.1016/j.jprocont.2017.06.012.

I. Hedi, I. Špeh, and A. Šarabok. IoT network protocols comparison for the purpose of IoT constrained networks. **2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2017 - Proceedings**, pages 501–505, 2017. doi: 10.23919/MIPRO.2017.7973477.

A. Ilapakurti, J. S. Vuppalapati, S. Kedari, S. Kedari, R. Vuppalapati, and C. Vuppalapati. Adaptive edge analytics for creating memorable customer experience and venue brand engagement, a scented case for Smart Cities. pages 1–8. IEEE, aug 2017. ISBN 978-1-5386-0435-9. doi: 10.1109/UIC-ATC.2017.8397583.

M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. Goren, and C. Mahmoudi. Fog computing conceptual model. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, mar 2018.

R. Kajic. Evaluation of the stream query language cql, 2010.

R. E. Kalman. A new approach to linear filtering and prediction problems. **Journal of basic Engineering**, 82(1):35–45, 1960.

V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate. A Survey on Application Layer Protocols for the Internet of Things. **Transaction on IoT and Cloud Computing**, 3(1):11–17, 2015. ISSN 2331-4761. doi: 10.5281/ZENODO.51613.

S. Kartakis, W. Yu, R. Akhavan, and J. A. McCann. Adaptive Edge Analytics for Distributed Networked Control of Water Systems. In **2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)**, pages 72–82. IEEE, apr 2016. ISBN 978-1-4673-9948-7. doi: 10.1109/IoTDI.2015.34.

H. Kaur and R. Kumar. A survey on internet of things (iot): Layer-specific, domain-specific and industry-defined architectures. In X.-Z. Gao, S. Tiwari, M. C. Trivedi, and K. K. Mishra, editors, **Advances in Computational Intelligence and Communication Technology**, pages 265–275. Springer Singapore, Singapore, 2021. ISBN 978-981-15-1275-9.

S. Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

V. Khairnar, L. Kolhe, S. Bhagat, R. Sahu, A. Kumar, and S. Shaikh. Industrial automation of process for transformer monitoring system using iot analytics. In **Inventive Communication and Computational Technologies**, pages 1191–1200. Springer, 2020.

T. Küfner, T. H.-J. Uhlemann, and B. Ziegler. Lean Data in Manufacturing Systems: Using Artificial Intelligence for Decentralized Data Reduction and Information Extraction. volume 72, pages 219–224, 2018. doi: 10.1016/j.procir.2018.03.125.

N. Leavitt. Complex-event processing poised for growth. **Computer**, 42(4):17–20, April 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.109.

C. Liu, X. Su, and C. Li. Edge computing for data anomaly detection of multi-sensors in underground mining. **Electronics**, 10(3):302, 2021. doi: 10.3390/electronics10030302.

L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. **IEEE Transactions on Knowledge and Data Engineering**, 11(4):610–628, 1999. doi: 10.1109/69.790816. cited By 218.

Y. Liu, T. Dillon, W. Yu, W. Rahayu, and F. Mostafa. Noise removal in the presence of significant anomalies for industrial iot sensor data in manufacturing. **IEEE Internet of Things Journal**, 7(8):7084–7096, 2020. doi: 10.1109/JIOT.2020.2981476.

D. C. Luckham. **The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0201727897.

D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. **IEEE Transactions on Software Engineering**, 21(4):336–354, 1995.

I. Lujic, V. De Maio, and I. Brandic. Adaptive Recovery of Incomplete Datasets for Edge Analytics. In **2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)**, pages 1–10. IEEE, may 2018. ISBN 978-1-5386-6488-9. doi: 10.1109/CFEC.2018.8358726.

N. Mao and J. Tan. Complex event processing on uncertain data streams in product manufacturing process. In **2015 International Conference on Advanced Mechatronic Systems (ICAMechS)**, pages 583–588, Aug 2015. doi: 10.1109/ICAMechS.2015.7287178.

B. Meindl, N. F. Ayala, J. Mendonça, and A. G. Frank. The four smarts of industry 4.0: Evolution of ten years of research and future perspectives. **Technological Forecasting and Social Change**, 168:120784, 2021. ISSN 0040-1625. doi: https://doi.org/10.1016/j.techfore.2021.120784.

N. Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In **2017 IEEE International Systems Engineering Symposium (ISSE)**, pages 1–7. IEEE, oct 2017. ISBN 978-1-5386-3403-5. doi: 10.1109/SysEng.2017.8088251.

E. Oyekanlu. Predictive edge computing for time series of industrial IoT and large scale critical infrastructure based on open-source software analytic of big data. In **2017 IEEE International Conference on Big Data (Big Data)**, volume 2018-Janua, pages 1663–1669. IEEE, dec 2017. ISBN 978-1-5386-2715-0. doi: 10.1109/BigData.2017.8258103.

E. Oyekanlu, S. Onidare, and P. Oladele. Towards statistical machine learning for edge analytics in large scale networks: Real-time Gaussian function generation with generic DSP. In **2018 First International Colloquium on Smart Grid Metrology (SmaGriMet)**, pages 1–6. IEEE, apr 2018. ISBN 978-9-5318-4235-8. doi: 10.23919/SMAGRIMET.2018.8369850.

A. Paschke and P. Vincent. A reference architecture for Event Processing. In **Proceedings of the Third ACM International Conference on Distributed Event-Based Systems - DEBS ’09**, page 1, New York, New York, USA, 2009. ACM Press. ISBN 9781605586656. doi: 10.1145/1619258.1619291.

N. W. Paton and O. Díaz. Active database systems. **ACM Comput. Surv.**, 31(1):63–103, Mar. 1999. ISSN 0360-0300. doi: 10.1145/311531.311623.

F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON Schema. In **Proceedings of the 25th International Conference on World Wide Web - WWW ’16**, pages 263–273. ACM Press, 2016. ISBN 9781450341431. doi: 10.1145/2872427.2883029.

K. Portelli and C. Anagnostopoulos. Leveraging Edge Computing through Collaborative Machine Learning. In **2017 5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)**, volume 2017-Janua, pages 164–169. IEEE, aug 2017. ISBN 978-1-5386-3281-9. doi: 10.1109/FiCloudW.2017.72.

N. Rusk. Deep learning. **Nature Methods**, 13(1):35–35, jan 2016. ISSN 1548-7091. doi: 10.1038/nmeth.3707.

H. Sabireen and V. Neelanarayanan. A review on fog computing: Architecture, fog with iot, algorithms and research challenges. **ICT Express**, 7(2):162–176, 2021. ISSN 2405-9595. doi: https://doi.org/10.1016/j.icte.2021.05.004.

C. M. Sadler and M. Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In **Proceedings of the 4th international conference on Embedded networked sensor systems - SenSys '06**, page 265, New York, New York, USA, 2006. ACM Press. ISBN 1595933433. doi: 10.1145/1182807.1182834.

J. Singh, P. Singh, and S. S. Gill. Fog computing: A taxonomy, systematic review, current trends and research challenges. **Journal of Parallel and Distributed Computing**, 157:56–85, 2021. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2021.06.005.

T. Sirojan, T. Phung, and E. Ambikairajah. Intelligent edge analytics for load identification in smart meters. pages 1–5. IEEE, dec 2017. ISBN 978-1-5386-4950-3. doi: 10.1109/ISGT-Asia.2017.8378414.

M. Symeonides, D. Trihinas, Z. Georgiou, G. Pallis, and M. Dikaiakos. Query-Driven Descriptive Analytics for IoT and Edge Computing. In **2019 IEEE International Conference on Cloud Engineering (IC2E)**, pages 1–11. IEEE, jun 2019. ISBN 978-1-7281-0218-4. doi: 10.1109/IC2E.2019.00-12.

P. Teng, G. Li, L. Su, and X. Chen. Adaptive Rule Update Method in Complex Event Process. In **Proceedings - 9th International Conference on Intelligent Human-Machine Systems and Cybernetics, IHMSC 2017**, volume 1, pages 270–275. Institute of Electrical and Electronics Engineers Inc., aug 2017. ISBN 9781538630228. doi: 10.1109/IHMSC.2017.69.

P.-H. Tsai, H.-J. Hong, A.-C. Cheng, and C.-H. Hsu. Distributed analytics in fog computing platforms using tensorflow and kubernetes. In **2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)**, pages 145–150. IEEE, sep 2017. ISBN 978-1-5386-1101-2. doi: 10.1109/APNOMS.2017.8094194.

S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Complex event processing over uncertain data. In **Proceedings of the second international conference on Distributed event-based systems**, pages 253–264. ACM, 2008.

M. Weiser. The computer for the 21 st century. **Scientific american**, 265(3):94–105, 1991.

G. Welch, G. Bishop, et al. An introduction to the kalman filter. 1995.

P. Widya, Y. Yustiawan, and J. Kwon. A oneM2M-Based Query Engine for Internet of Things (IoT) Data Streams. **Sensors**, 18(10):3253, sep 2018. ISSN 1424-8220. doi: 10.3390/s18103253.

X. Xu, S. Huang, L. Feagan, Y. Chen, Y. Qiu, and Y. Wang. EAaaS: Edge Analytics as a Service. pages 349–356. IEEE, jun 2017. ISBN 978-1-5386-0752-7. doi: 10.1109/ICWS.2017.130.

M. B. Yassein, M. Q. Shatnawi, and D. Al-zoubi. Application layer protocols for the Internet of Things: A survey. In **2016 International Conference on Engineering & MIS (ICEMIS)**, pages 1–4. IEEE, sep 2016. ISBN 978-1-5090-5579-1. doi: 10.1109/ICEMIS.2016.7745303.

C. Yin, B. Li, and Z. Yin. A distributed sensing data anomaly detection scheme. **Computers & Security**, 97:101960, 2020. ISSN 0167-4048. doi: 10.1016/j.cose.2020.101960.

S. K. YR and H. Champa. Iot streaming data outlier detection and sensor data aggregation. In **2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)**, pages 150–155. IEEE, 2020. doi: 10.1109/I-SMAC49090.2020.9243509.