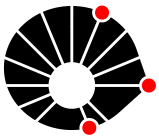
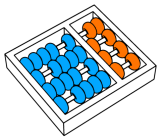




Detecção de anomalias em tempo real

E. R. Carmo *L. F. Bittencourt*

Relatório Técnico - IC-PFG-21-08
Projeto Final de Graduação
2021 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Detecção de anomalias em tempo real

Esdras Rodrigues do Carmo Luiz Fernando Bittencourt*

Resumo

Este trabalho tem como objetivo expor uma implementação de um detector de anomalias em *streaming* altamente escalável e disponível. Serão apresentados os requisitos para um algoritmo ser considerado um detector de anomalias em tempo real e exemplos de algoritmos que satisfazem esse requisito. A partir disso, será implementado um dos algoritmos utilizando processamento em *streaming* com Apache Kafka para maximizar a performance, escalabilidade e disponibilidade da solução. A implementação será avaliada em um ambiente de nuvem na AWS com métricas de consumo de memória, CPU, *throughput* e latência.

1 Introdução

Detecção de anomalias é um assunto recorrente quando se trata de monitoramento de sinais emitidos por alguma aplicação computacional, sensores de saúde física, mercado financeiro, carros autônomos, entre inúmeras outras aplicações. Com a crescente exponencial de dados disponíveis, é notável também a crescente busca em monitorar dados que variam com o tempo, tais como os batimentos cardíacos de uma pessoa, a latência de requisição de algum serviço Web ou o valor de uma determinada ação na bolsa de valores.

Uma anomalia pode ser definida como uma ocorrência de um valor em uma série temporal que difere significantemente da distribuição de valores previamente conhecida. Em muitos casos, a ocorrência de uma anomalia deve ser tratada como um alerta de que algo não está saindo como esperado. Dessa forma, um detector de anomalias é um sistema que observa séries temporais e é capaz de identificar se algum valor é anômalo ou não. O tempo de detecção da anomalia é, na maioria dos casos, importante para a prevenção de incidentes.

Neste trabalho serão apresentados alguns algoritmos de detecção de anomalia na seção 2, tomando como foco a implementação baseada no algoritmo de memória sequencial *Hierarchical Temporal Memory* (HTM) [1, 2]. Para a implementação, será utilizado processamento em *streaming* com base no Apache Kafka [3] com foco na escalabilidade da solução para múltiplas séries temporais, na velocidade de processamento de modo a diminuir o tempo em que uma anomalia é detectada e na alta disponibilidade e tolerância a falhas. A implementação será descrita em detalhes na seção 4. Por fim, será apresentado uma metodologia de testes de performance da solução proposta, bem como a análise dos resultados obtidos através de métricas descritas nas seções 5 e 6.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP

2 Trabalhos Relacionados

Existem diversos algoritmos de detecção de anomalias, sendo que alguns possuem particularidades que os tornam menos aplicáveis em aplicações de *streaming* em tempo real. Ahmad et al. descreve as condições necessárias que um algoritmo de detecção de anomalia deve satisfazer para ser aplicável em uma aplicação em tempo real [2]. Além disso, o trabalho apresenta uma implementação de detector de anomalia utilizando o algoritmo HTM (Hierarchical Temporal Memory), apresenta um benchmark baseado em tanto dados reais quanto sintéticos que foi utilizado para comparar a implementação do detector de anomalia com HTM e demais algoritmos conhecidos que satisfazem os requisitos para um detector de anomalias em tempo real.

Higashino et al. apresenta conceitos sobre processamento em *streaming* e processamento de eventos complexos [5], além de criar um simulador de processamento de eventos complexos em nuvem. Este trabalho serve como base na implementação do detector de anomalia de forma performática, disponível e escalável.

Twitter tem uma biblioteca open-source de detecção de anomalia [6], que utiliza um método baseado na detecção de sazonalidades com o algoritmo S-H-ESD. Neste trabalho as anomalias são categorizadas entre globais e locais, sendo uma anomalia global um dado que está fora do esperado considerando a sazonalidade do evento, já uma anomalia local é o dado que está fora do esperado dentro de um padrão de sazonalidade, sem ultrapassar os limites sazonais.

EGADS é um sistema completo, apresentado pelo Yahoo [7], de análise de dados temporais, detecção de anomalias e sistema de alerta. No entanto, a detecção de anomalias é realizada em duas etapas: criar um modelo a partir da análise dos dados históricos e fazer a análise dos dados novos com base no modelo previamente criado. O primeiro passo assume que o dado histórico estará disponível, quebrando assim um dos requisitos para um detector de anomalias em tempo real apresentados por Ahmad et al. [2].

3 Conceitos

3.1 Aplicações em Streaming

Aplicações em *streaming* são aplicações que recebem um fluxo contínuo de dados, usualmente gerados e processados em tempo real. Em contraste com aplicações em *batch*, as aplicações *streaming* não possuem acesso a todos os dados históricos, mas sim recebem de forma ordenada os dados que são produzidos naquele momento.

Sendo assim, um detector de anomalias em *streaming* deve satisfazer os seguintes requisitos como definidos em [2]:

1. As predições devem ser feitas *online*, ou seja, o algoritmo identifica o estado x_t como anômalo ou não antes de receber o estado x_{t+1}

2. O algoritmo deve aprender continuamente sem recorrer nem armazenar os dados históricos após o processamento destes.

3. O algoritmo deve ser não-supervisionado, sem necessitar de anotações manuais para o aprendizado.

4. A detecção de anomalia deve ser feita assim que o dado é disponível no fluxo.

3.2 Detecção de Anomalias

Uma anomalia pode ser definida como um padrão de dados que não obedece a uma noção bem definida de comportamento normal [8]. Neste trabalho, iremos focar em anomalias de dados temporais e em conformidade com uma aplicação em *streaming* definida na seção 3.1. Desta forma, iremos considerar os nossos dados como sendo uma sequência infinita de valores reais emitidos por um sensor em um dado instante de tempo: $..., x_{t-2}, x_{t-1}, x_t, x_{t+1}, x_{t+2}, ...$

Sendo assim, podemos definir um detector de anomalias simples como uma função ϕ:

$$\begin{aligned} \phi &: \mathbb{R} \to \{\gamma \in \mathbb{R} | 0 \leq \gamma \leq 1\} \\ \gamma_t &:= \phi(x_t) \end{aligned}$$

Sendo γ_t o *score* de anomalia. Escolhendo um limiar $\delta \in (0, 1)$ podemos definir a função anomalia π:

$$\begin{aligned} \pi &: \mathbb{R} \to \{0, 1\} \\ \pi_t = \pi(\gamma_t) &:= \begin{cases} 1, & \text{se } \gamma_t > \delta \\ 0, & \text{se } \gamma_t \leq \delta \end{cases} \end{aligned}$$

No entanto, neste trabalho iremos implementar um algoritmo que aprende continuamente a detectar anomalias. Dessa forma, para valores iguais $x_t = y_s$ em tempos diferentes $t \neq s$, podemos esperar por *scores* diferentes $\phi(x_t) \neq \phi(y_s)$. Sendo assim, iremos incluir na definição da função ϕ o contexto $\vec{W}_t \in \mathbb{R}^n$ que será atualizado conforme novos dados x_t são apresentados ao modelo, através da função θ:

$$\begin{aligned} \phi &: \mathbb{R}^n \times \mathbb{R} \to \{\gamma \in \mathbb{R} | 0 \leq \gamma \leq 1\} \\ \theta &: \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n \\ \pi &: \mathbb{R} \to \{0, 1\} \end{aligned}$$

$$\begin{aligned} \gamma_t &:= \phi(\vec{W}_{t-1}, x_t) \\ \vec{W}_t &:= \theta(\vec{W}_{t-1}, x_t) \\ \pi_t = \pi(\gamma_t) &:= \begin{cases} 1, & \text{se } \gamma_t > \delta \\ 0, & \text{se } \gamma_t \leq \delta \end{cases} \end{aligned}$$

3.3 Kafka

Apache Kafka é uma plataforma de *streaming* de eventos open-source [3]. Neste trabalho, ele será utilizado como barramento de mensagens que a solução utilizará para realizar a detecção de anomalias em tempo real.

No Kafka, as mensagens ou eventos são organizados em tópicos, os quais possuem $n > 0$ partições. Cada partição de um tópico armazena eventos ordenados conforme a ordem de publicação, no entanto a ordenação não é garantida entre partições diferentes de um mesmo tópico. Por conta disso, será utilizado um particionamento que garante que eventos de um mesmo sensor, que terá um identificador único, estará sempre na mesma partição. Sendo assim, será garantido que a solução não irá processar o evento x_{t+1} antes do evento x_t de um mesmo sensor.

Por outro lado, eventos de sensores diferentes poderão estar em partições diferentes, possibilitando assim que a solução escale horizontalmente de modo a detectar anomalias de múltiplos sensores em paralelo. Neste caso, o número de instâncias em paralelo que a solução poderá rodar é limitada pela quantidade de partições que os tópicos dentro do Kafka possui.

3.4 Redis

Redis é um banco de dados em memória capaz de armazenar diferentes estruturas de dados como *strings*, listas, *hashes*, *sets*, entre outras [4]. Os dados armazenados no Redis possuem uma chave e o valor associado a esta chave.

Para o propósito deste trabalho, o Redis será utilizado como armazenamento do estado atual ($\vec{W}_t$) de cada modelo que a solução está rodando. Desta forma, as instâncias em execução da solução poderão falhar e perder os seus dados locais que o estado do modelo ainda estará presente no Redis.

4 Implementação

4.1 Processamento em Streaming

Para a implementação do detector de anomalia em *streaming* foi utilizado a linguagem de programação Python com a biblioteca Faust [9]. Essa biblioteca facilita o desenvolvimento em *streaming* compartilhando similaridades com outras soluções conhecidas, como o Apache Storm [10] e Apache Samza [11].

O processamento é separado em estágios conforme as etapas existentes no algoritmo apresentado na seção 4.2. A transferência das mensagens entre os estágios e feita pelo Kafka [3] utilizando a biblioteca Faust. Cada estágio possui seu próprio tópico com as mensagens de entrada daquele estágio, e cada tópico possui $N \geq 1$ partições que irá limitar a quantidade de mensagens processadas em paralelo. Dessa forma, um tópico com N partições poderá ter no máximo N instâncias em paralelo consumindo suas mensagens.

Para garantir a ordenação do processamento, cada mensagem no Kafka possui uma chave com o *id* do modelo ou sensor. Essa chave é utilizada pelo algoritmo de particionamento do Kafka para garantir que mensagens do mesmo sensor sejam enviadas à mesma

partição. Dessa forma, os valores enviados por um mesmo sensor será processado pela mesma instância, além de ter a garantia que as mensagens serão processadas na mesma ordem que foram produzidas.

Os estágios *Spatial Pooler*, *Temporal Memory* e *Anomaly Likelihood* possuem aprendizado online e por isso necessitam armazenar seu estado durante todo o processamento. Dessa forma, como o particionamento ocorre de forma que mensagens do mesmo sensor estarão na mesma partição, pode-se manter um estado global em cada réplica do processador para cada sensor que essa réplica processa. Por exemplo, se o sistema estiver possuir 6 sensores enviando métricas e 2 réplicas de processamento, podemos esperar que cada réplica processe 3 sensores, armazenando somente os estados desses sensores.

Para garantir escalabilidade e disponibilidade, o estado de cada sensor também é mantido em uma instância de Redis [4], juntamente com as configurações específicas de cada modelo. Para garantir que o estado esteja atualizado, os processadores enviam de tempos em tempos um *snapshot* do estado atual dos modelos para o Redis, além de sempre que um processador recebe dados de um novo sensor, ele busca pelo estado mais atualizado desse sensor no Redis, caso exista. Dessa forma, se no cenário de 6 sensores desejarmos escalar de 2 para 6 réplicas de processamento, automaticamente as novas réplicas irão buscar do Redis o estado mais atualizado dos modelos sem necessitar de um reprocessamento de todo o fluxo de dados para alcançar o estado atual do algoritmo.

O diagrama com o fluxo das mensagens para o processamento está representado na Figura 1. A implementação pode ser encontrada no GitHub [12].

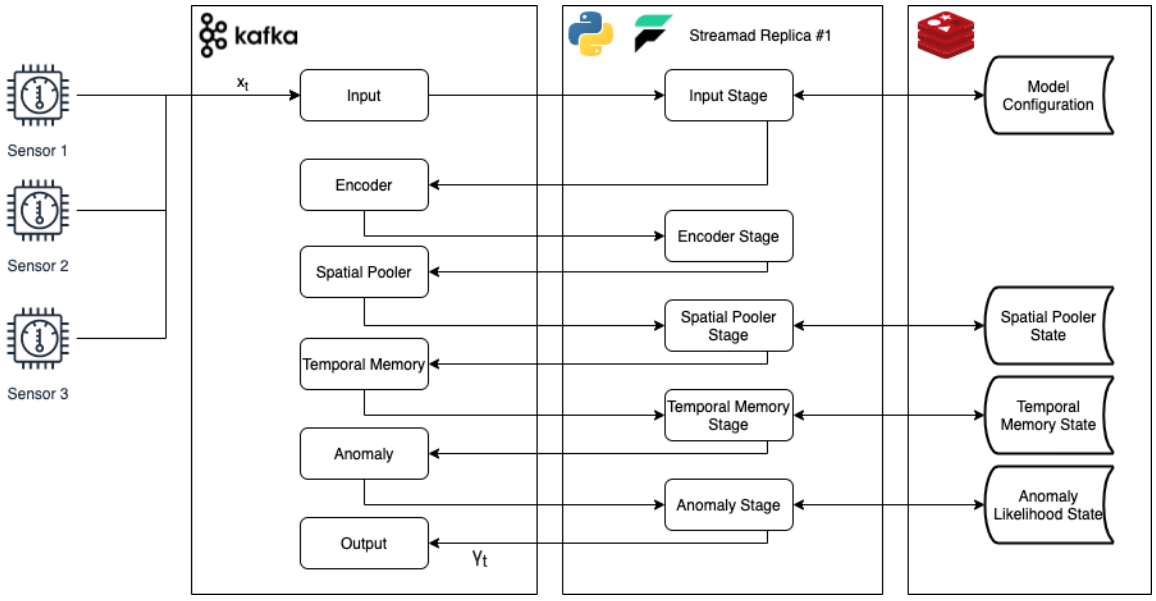


Figura 1: Implementação do detector de anomalias em tempo real

4.2 Detector de Anomalia

Para a implementação do detector de anomalias em tempo real utilizou-se do algoritmo baseado em HTM [2]. Este algoritmo consiste em receber uma entrada x_{t-1}, utilizar um *encoder* para gerar o vector binário esparso $a(x_{t-1})$ e a partir deste vetor, o algoritmo fará uma previsão da próxima entrada, representado por π_{t-1}. Quando a próxima entrada x_t entra no algoritmo, ele irá comparar a previsão feita anteriormente π_{t-1} com o *encoder* da nova entrada $a(x_t)$. A diferença entre a entrada atual e a previsão feita anteriormente serve como base para um cálculo da probabilidade de anomalia, além de ser utilizada como aprendizado para o próprio modelo.

A Figura 2 representa os blocos necessários para o algoritmo de detecção de anomalia, enquanto que a Figura 3 especifica o *encoder* e *spatial pooler* necessários para o restante do algoritmo.

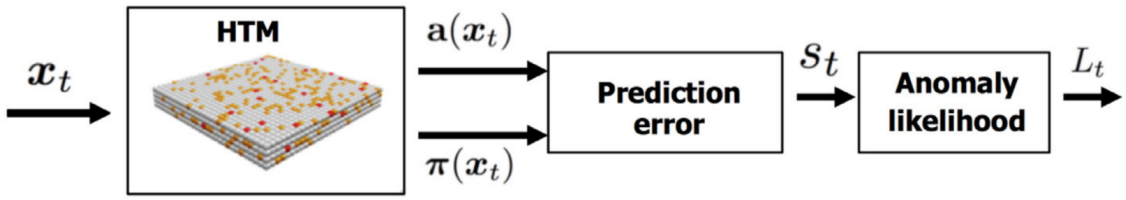


Figura 2: Diagrama de um detector de anomalia baseado em HTM

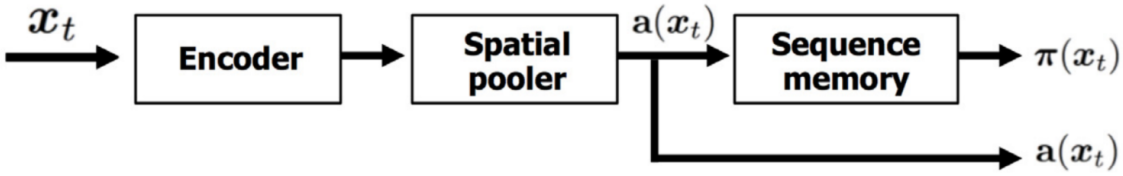


Figura 3: Diagrama detalhado do algoritmo HTM

O último passo do detector de anomalias é chamado de *Anomaly Likelihood*, que é o cálculo do *score* de anomalia baseado no erro da predição do algoritmo HTM. Neste caso, é utilizado uma janela deslizante dos últimos W erros calculados pelo algoritmo e calculado a probabilidade do erro estar a mais de um desvio padrão nesta janela deslizante, a saída do algoritmo é o complemento desta probabilidade: $L_t = 1 - Q\left(\frac{\tilde{\mu}_t - \mu_t}{\sigma_t}\right)$.

Na Figura 4 pode-se analisar o comportamento do algoritmo HTM e do *anomaly likelihood* em um cenário com muito ruído. Percebe-se que no início os picos de latência são identificados como ruído tanto na saída do HTM quanto na saída do *anomaly likelihood*. Porém, conforme os picos se tornam mais frequentes, o *anomaly likelihood* tende a diminuir a probabilidade de anomalia. No entanto, se a densidade de ruído do sensor se altera, ou seja, mais picos identificados por período de tempo, a probabilidade de anomalia aumenta.

Dessa forma, o algoritmo proposto possui características que o faz identificar anomalias mesmo em sensores com alto ruído.

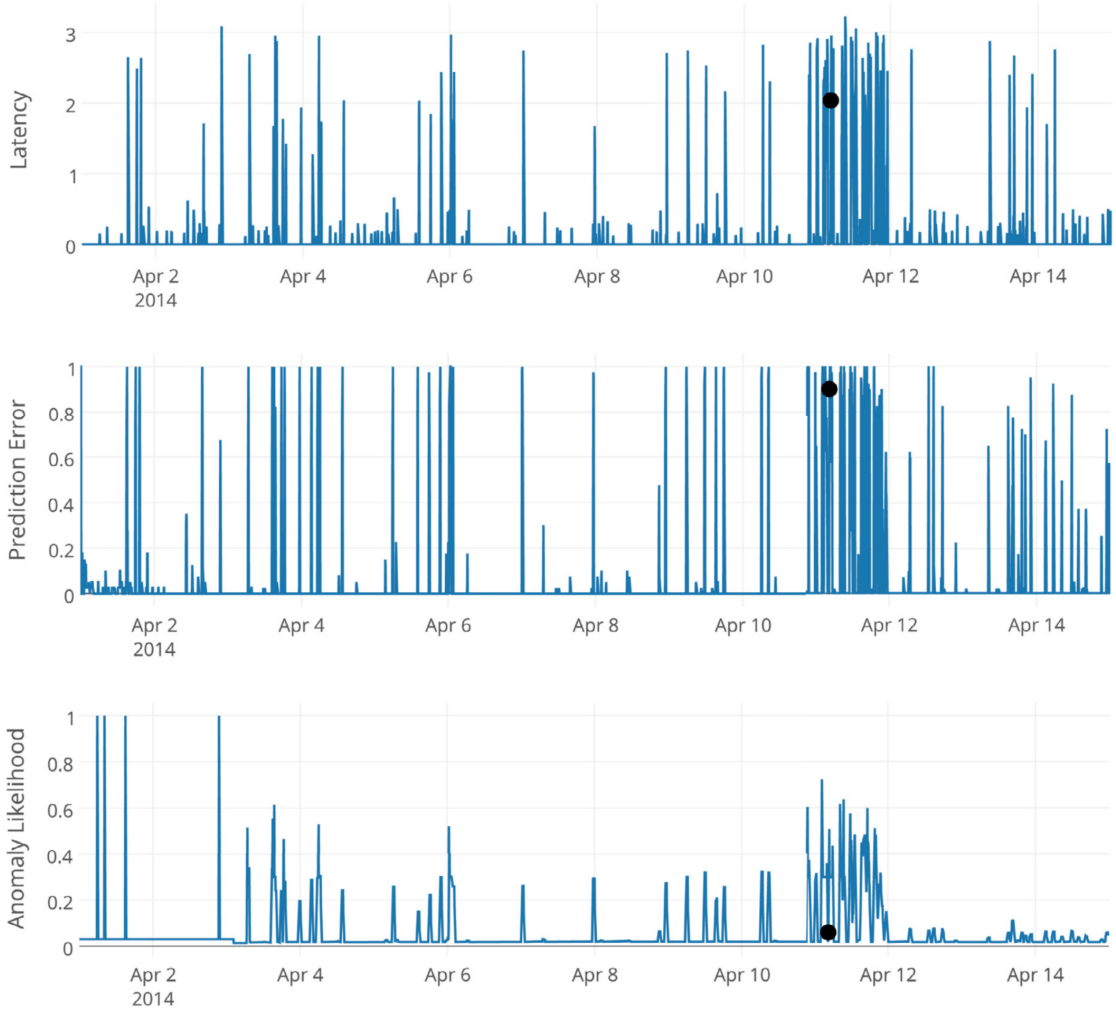


Figura 4: Aplicação do algoritmo HTM e *anomaly likelihood* em um sensor pouco previsível e com muitos ruídos

5 Metodologia

Para avaliação da solução proposta, foi utilizado um ambiente em nuvem AWS (Amazon Web Services) com máquinas virtuais EC2 [13]. Todas as máquinas virtuais foram provisionadas dentro da mesma rede e em uma mesma zona a fim de diminuir a latência de rede. Utilizou-se uma instância com Kafka e Redis e até quatro instâncias exclusivas para a execução do detector de anomalias, que iremos chamar de *streamad* [12]. Para os serviços Redis e Kafka, a quantidade de memória na máquina é importante devido ao alto uso do

recurso por esses serviços, no entanto o uso de CPU permaneceu baixo durante todos os testes. Já para a execução do algoritmo, o uso de CPU é limitado em 1 *core* por réplica, devido à implementação utilizando *event loop* [14] em uma única *thread*. O uso de memória, no entanto, é proporcional ao número de sensores que cada réplica está processando, dado ao armazenamento do estado de cada algoritmo em memória.

As instâncias utilizadas na nuvem são detalhadas na Tabela 1.

Tabela 1: Tipos de instância para ambiente em nuvem

Quantidade	Tipo de Instância	# vCPUs	Memória	Descrição
1	t3.large	2	8 GiB	Redis e Kafka
4	c5.large	2	4 GiB	Réplicas do Streamad

Para medir a escalabilidade e performance da implementação foram realizado duas análises:

- Análise em *streaming*: Tem como objetivo simular um caso real de execução do sistema, com dados sendo produzidos por 8 sensores artificias em paralelo a cada 100 milissegundos. Desta forma, teremos um fluxo constante de 80 mensagens por segundo na entrada no sistema.

- Análise em *batch*: Tem como objetivo forçar o sistema chegar ao ponto de saturação. Para isso, são gerados todos os dados presentes no *benchmark* de anomalias de Numenta [15] na categoria *artificialWithAnomaly* e *artificialNoAnomaly*. Nesta análise, todos os dados de entrada ficaram disponíveis no tópico do Kafka fazendo com que o tempo de geração do dado não tenha influência no tempo de processamento.

Ambas as análises foram realizadas em 3 variações de escala: 1, 2 e 4 réplicas.

O estado do algoritmo em todas as análises foi salvo a cada 1 minuto, sendo possível observar uma periodicidade de acréscimo de latência devido ao *commit* do estado do algoritmo.

Para a geração dos dados, cada réplica da aplicação expôs dados de uso de CPU, memória, histograma de latência de cada estágio e quantidade de mensagens processadas por segundo em cada estágio. Para a coleta das métricas, foi utilizado o Prometheus [16], e para a visualização, o Grafana [17].

6 Avaliação

Para a coleta dos dados presentes nesta seção, foi utilizado o Prometheus com configuração de coleta a cada 15 segundos, a fim de garantir maior granularidade nas métricas.

6.1 Streaming

O histograma de latência da aplicação e o percentil 95 agrupado por estágio de processamento estão representados na Figura 5.

No gráfico de latência é perceptível o acréscimo de latência que ocorre a cada minuto, devido ao envio do estado da aplicação ao Redis. Além disso, em todos os casos nota-se que o estágio com maior latência é o *input*. Este estágio é o único que sempre se comunica com o Redis para buscar a configuração do modelo, enquanto que os demais recebem esse dado na própria mensagem que o estágio *input* envia, eliminando assim a comunicação para busca de configuração nos demais estágios. Além disso, a comunicação com o Redis é realizada de forma assíncrona e dependente de entrada / saída, fazendo com que o *event loop* possa suspender o processamento de uma mensagem do *input* enquanto a resposta do Redis não está pronta e comece a processar uma mensagem de outro estágio, que é totalmente dependente de CPU.

Em um ambiente com alta concorrência de recursos, como é o caso da Figura 5a, o tempo que o *event loop* leva para retornar ao processamento de entrada / saída é maior em comparação com um ambiente com menos concorrência de recursos, como na Figura 5c. Este tempo fica visível analisando a diferença entre a latência do estágio *input* com os demais nos 3 cenários da Figura 5.

A tabela 2 mostra os mínimos e máximos do percentil 95 da latência de cada estágio, em cada cenário analisado. O primeiro minuto de cada análise foi desconsiderado para que a inicialização dos objetos necessários no processamento não tenha impacto na latência máxima observada no período. É possível notar que, durante o processamento, os estágios *encoder* e *anomaly* são os que possuem menor latência, enquanto o *input* possui a maior latência e maior variação, devido ao comportamento já mencionado do *event loop*.

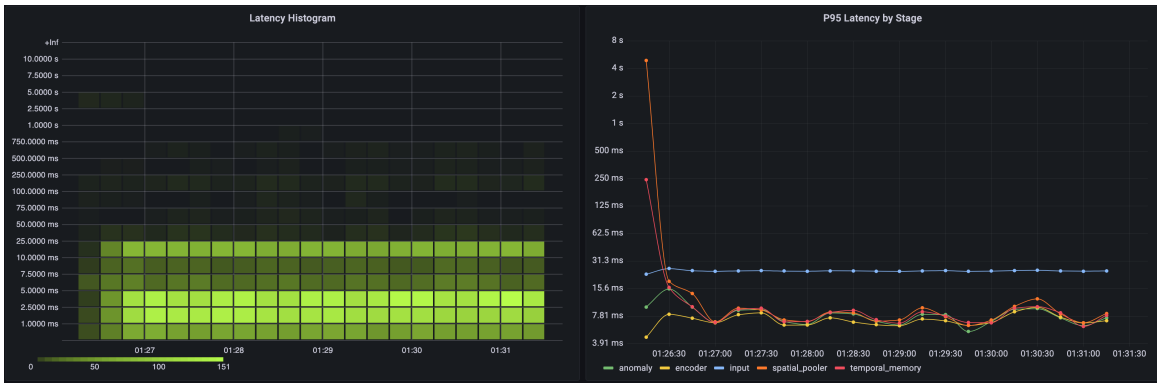
Tabela 2: Intervalo do percentil 95 de latência em milissegundos

#	Input	Encoder	Spatial Pooler	Temporal Memory	Anomaly
1	24,1 - 24,7	6,12 - 9,74	6,16 - 12,1	6,03 - 9,90	5,28 - 9,41
2	6,94 - 24,0	2,91 - 6,97	4,57 - 8,89	4,75 - 8,69	3,84 - 8,63
4	4,74 - 32,6	2,27 - 13,7	4,29 - 13,0	4,80 - 11,3	2,97 - 15,6

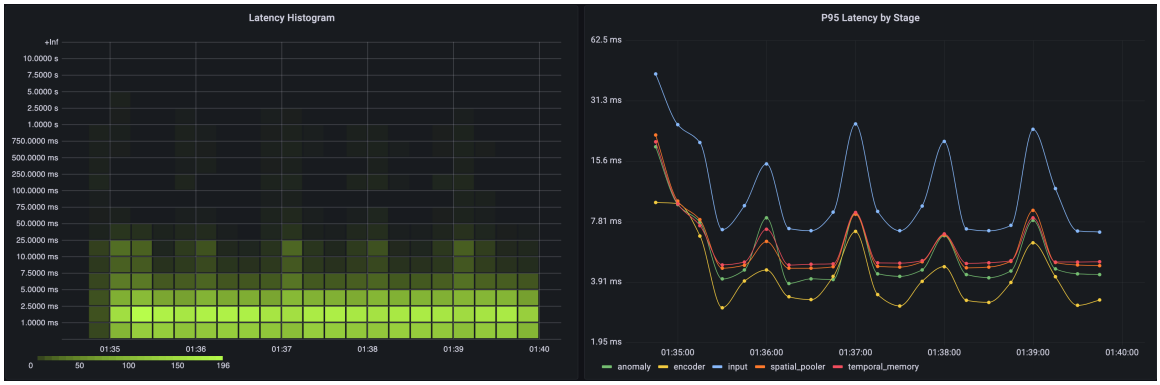
O número de mensagens processadas por segundo e a utilização de recursos computacionais por instância podem ser observados na Figura 6. Em todos os casos, nota-se que o *throughput* por estágio tem uma tendência a ficar constante em 80 mensagens por segundo, demonstrando que o sistema é capaz de processar na mesma taxa que os sensores produzem as mensagens.

A diferença, no entanto, está na quantidade de mensagens que cada réplica processa nos casos das Figuras 6b e 6c. Isso ocorre porque as mensagens, a partir do estágio *input*, não foram igualmente distribuídas entre as partições do Kafka. Embora o algoritmo de balanceamento tem como objetivo distribuir partições igualmente entre as instâncias, o algoritmo utilizado para particionar as mensagens com base no nome do sensor não foi capaz de distribuir suas mensagens igualmente entre as partições, fazendo com que algumas instâncias recebam mais dados que outras.

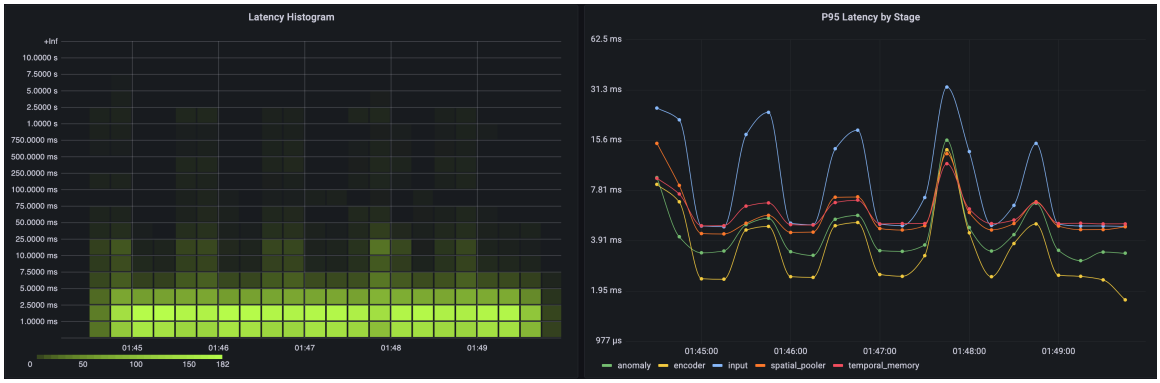
O uso de CPU e memória, na Figura 6, seguem a mesma proporção que a quantidade de mensagens processadas por instância, como esperado.



(a) 1 réplica



(b) 2 réplicas



(c) 4 réplicas

Figura 5: Histograma de Latência e Percentil 95 por estágio em escala logarítmica na análise de *streaming*

6.2 Batch

Para a análise em batch, como mencionado na seção 5, foi utilizado os dados *artificialWithAnomaly* e *artificialNoAnomaly* do benchmark Numenta. A quantidade de mensagens geradas

(a) 1 réplica

(b) 2 réplicas

(c) 4 réplicas

Figura 6: Throughput e utilização de recursos computacionais na análise de *streaming*

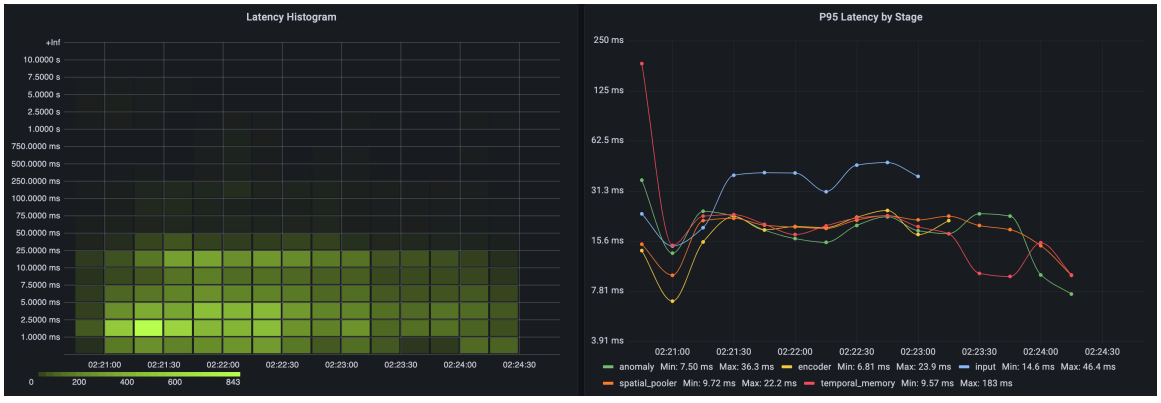
no *input* do modelo para esta análise é de 44.352.

O histograma de latência e percentil 95 das análises neste cenário estão representados na Figura 7. Percebe-se que as latências após a inicialização possuem valores muito similares entre os casos da Figura 7a, 7b e 7c. Isso demonstra o fato de que a aplicação estava operando em saturação nos 3 cenários, i. e., processando o máximo de mensagens possível.

O número de mensagens processadas por segundo e a utilização de recursos computacionais são mostrados na Figura 8. Observa-se que, entre todos os casos, a aplicação é capaz de processar entre 470 a 640 mensagens por segundo, por instância. A quantidade de mensagens processadas por toda a solução é, no entanto, dependente do número de instâncias

(a) 1 réplica

(b) 2 réplicas



(c) 4 réplicas

Figura 7: Histograma de Latência e Percentil 95 por estágio em escala logarítmica na análise de *batch*

e da quantidade de partições nos tópicos do Kafka.

Na tabela 3, é comparado o tempo de execução da análise nos 3 casos, além da taxa de processamento baseado na quantidade de dados existentes no *input* do sistema. Percebe-

(a) 1 réplica

(b) 2 réplicas

(c) 4 réplicas

Figura 8: Throughput e utilização de recursos computacionais na análise de *batch*

se que a taxa de processamento não é linear com a quantidade de réplicas existentes no sistema. Um dos fatores que contribuíram para que isso acontecesse foi os sensores não estarem igualmente distribuídos entre as partições do Kafka, fazendo com que algumas instâncias recebam mais dados que outras. No gráfico da Figura 8c, é possível perceber que a distribuição de trabalho entre as instâncias não foi igual, dado que uma instância levou aproximadamente o dobro do tempo que as demais para terminar o processamento. Além disso, outro fator possível é que neste cenário foi utilizado 11 sensores diferentes, enquanto que os tópicos do Kafka possuíam apenas 8 partições, forçando assim que algumas partições

recebam dados de mais sensores que outras.

Por outro lado, os casos mencionados que contribuíram para a distribuição desigual dos dados entre as instâncias, tanto na análise em *batch*, quanto na análise em *streaming*, seriam solucionados conforme mais sensores sejam adicionados ao sistema, dado o algoritmo de particionamento padrão do Kafka [18]

Tabela 3: Tempo e taxa de processamento nos 3 cenários analisados

#	Dados Processados	Tempo de Processamento	Taxa de Processamento
1	44.352	8 min. e 30 seg.	86,96 msg/s
2	44.352	5 min.	147,8 msg/s
4	44.352	3 min. e 45 seg.	197,1 msg/s

7 Conclusão

Neste trabalho foram apresentados conceitos de detecção de anomalias, requisitos para um algoritmo de detecção de anomalias em tempo-real, e por fim a análise de uma implementação de um detector de anomalias tolerante a falhas, escalável e com alta disponibilidade.

Os resultados mostram que é possível utilizar algoritmos complexos de detecção de anomalia e implementar uma solução de alta performance e escalabilidade, pronto para ser utilizado em um ambiente real de monitoramento.

Além disso, os resultados mostram como as aplicações Kafka e Redis contribuem para a implementação, sendo possível manter o estado dos algoritmos e transferir mensagens entre os estágios do algoritmo de forma resiliente e performática, mesmo com poucos recursos computacionais. A utilização destes serviços foi essencial para que o sistema se tornasse tolerante a falhas e altamente escalável.

Referências

[1] J. Hawkins, S. Ahmad *Why neurons have thousands of synapses, a theory of sequence memory in neocortex*, Front. Neural Circuits. 10 (2016) 1–13.

[2] S. Ahmad, A. Lavin, S. Purdy and Z. Agha, *Unsupervised real-time anomaly detection for data*. Neurocomputing, Volume 262 (2017)

[3] Apache Kafka `https://kafka.apache.org/`

[4] Redis `https://redis.io/`

[5] W. A. Higashino, M. A. M. Capretz and L. F. Bittencourt, *CEPSim: A Simulator for Cloud-Based Complex Event Processing,* 2015 IEEE International Congress on Big Data, 2015, pp. 182-190.

[6] A. Kejariwal, Twitter Engineering, *Introducing practical and robust anomaly detection in a time series* [Online Blog], 2015, `https://bit.ly/3w4PSJb`

[7] N. Laptev, S. Amizadeh, and I. Flint. *Generic and Scalable Framework for Automated Time-series Anomaly Detection* 2015

[8] V. Chandola, A. Banerjee, and V. Kumar. *Anomaly detection: A survey.* ACM Computing Surveys, 2009

[9] Faust `https://faust.readthedocs.io/en/latest/`

[10] Apache Storm `http://storm.apache.org/`

[11] Apache Samza `http://samza.apache.org/`

[12] Streamad Github `https://github.com/esdrasbrz/streamad`

[13] Amazon Web Services EC2 `https://aws.amazon.com/pt/ec2/`

[14] Python Event Loop `https://docs.python.org/3/library/asyncio-eventloop.html`

[15] Numenta Anomaly Benchmark `https://github.com/numenta/NAB`

[16] Prometheus `https://prometheus.io/`

[17] Grafana `https://grafana.com/oss/grafana/`

[18] Abhishek. *Kafka Partitioner...*, Simply Distributed Blog, `https://bit.ly/3hAx6o0`