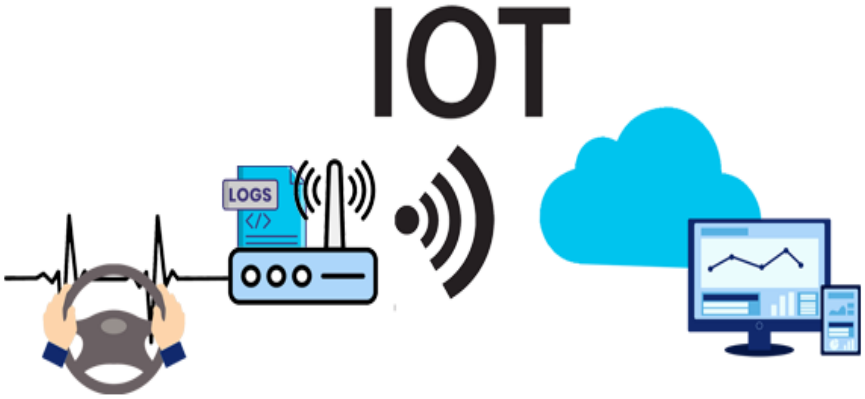
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores



Monitorização de sistemas embebidos para aplicações críticas

José Pedro de Jesus Ganilha

Licenciado

Trabalho de Projeto para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Orientadores : Doutor Tiago Miguel Braga da Silva Dias
Doutor André Ribeiro Lourenço

Júri:

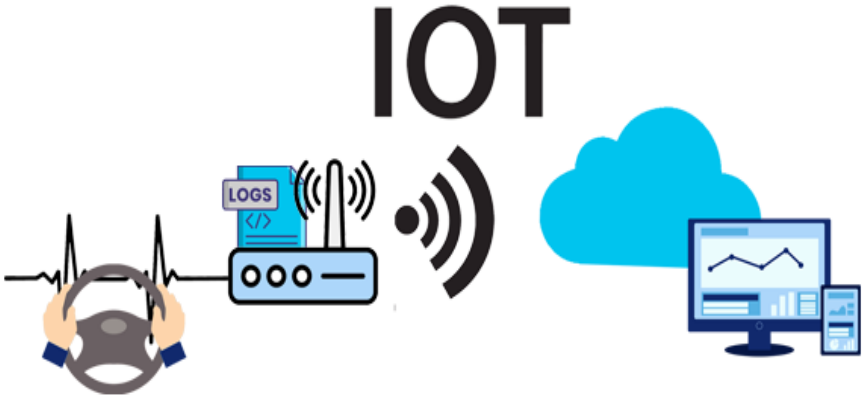
Presidente: Doutor Carlos Jorge de Sousa Gonçalves

Vogais: Doutor David Miguel Ramalho Pereira
Doutor Tiago Miguel Braga da Silva Dias

Dezembro, 2023

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores



Monitorização de sistemas embebidos para aplicações críticas

José Pedro de Jesus Ganilha

Licenciado

Trabalho de Projeto para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Orientadores : Doutor Tiago Miguel Braga da Silva Dias
Doutor André Ribeiro Lourenço

Júri:

Presidente: Doutor Carlos Jorge de Sousa Gonçalves

Vogais: Doutor David Miguel Ramalho Pereira
Doutor Tiago Miguel Braga da Silva Dias

Dezembro, 2023

Agradecimentos

Gostaria de expressar os meus sinceros agradecimentos à minha família, por todo o carinho e suporte durante o meu percurso académico.

Ao Instituto Superior de Engenharia de Lisboa (ISEL) e docentes por todos os ensinamentos e lições que, certamente, me irão seguir pelo meu percurso académico e profissional futuro.

Aos meus orientadores, por todo o apoio prestado.

À CardioID, pela oportunidade de participar num projeto inovador.

A todos os meus amigos, pelo acompanhamento e pela amizade que me motivou a alcançar os meus objetivos.

Resumo

Os sistemas informáticos encontram-se em constante evolução, seja para a adição de novas funcionalidades ou para a correção de problemas no seu funcionamento. Para acompanhar este crescimento, são necessários sistemas de registo de eventos também conhecidos por *logging*, da sua designação em língua inglesa, para monitorizar e apoiar o diagnóstico dos problemas e falhas no funcionamento das aplicações funcionais, tanto em ambiente de teste como de produção.

Estas necessidades também existem para as soluções *Internet of Things* (IoT), no entanto, nesta área da informática é ainda mais imperioso tomar boas decisões para a realização de um processo de *logging* eficaz, e que não afete o funcionamento normal destes sistemas. Por outro lado, o ambiente *Cloud* tem-se revelado essencial para qualquer processamento externo efetuado sobre os *logs* e respetiva visualização da informação em *dashboards*.

O sistema *CardioWheel* é uma solução IoT de monitorização do estado dos utilizadores durante a atividade de condução criada pela empresa *CardioID* que requer o desenvolvimento de um sistema de *logging* para rastrear e observar o fluxo de dados na aplicação. A ocorrência de informação sensível, referente aos dados biométricos relativos ao estado do utilizador, obriga a uma decisão ainda cuidada na construção das diferentes componentes do sistema de *logging*.

O trabalho realizado no âmbito desta dissertação visou, portanto, investigar e propor uma solução que, cumprindo com as limitações dos sistemas IoT, permitisse uma boa dinâmica de monitorização do *CardioWheel*.

A solução desenvolvida utiliza a biblioteca *logging* disponibilizada pela *framework Espressif* para guardar a informação gerada em memória externa, posteriormente enviada após o fim da sessão de *logging* presente ou, em caso de falha, após a terminação de

sessões futuras. A biblioteca *CycloneSSH* foi, portanto, adaptada para garantir a integridade e a segurança no envio do ficheiro de sessão que contém os eventos *log* da sessão. Em ambiente *Cloud*, o *Logstash* foi utilizado para interpretar, transformar e redirecionar a informação recebida para posterior análise e visualização na plataforma *Opensearch*. Os resultados obtidos da implementação demonstraram-se encorajadores face à limitações mencionadas na medida que, após a introdução da solução, não ocorreu um grande impacto em nenhum dos pontos críticos dos dispositivos IoT. Porém, existem melhorias a efetuar com uma finalidade de otimizar o ciclo de vida da declaração *log* no sistema.

Palavras-chave: Logging, IoT, Cloud, CardioWheel

Abstract

Informatic systems find themselves in constant evolution, being by the addition of new functionalities or by bug fixing. To keep up with its growth, the introduction of an event or logging recording system is required to monitor and support the diagnosis of problems and failures in functional applications in test and production environments.

This need also exists in IoT solutions and good decisions must be made to guarantee an effective logging process that doesn't impact the normal functioning of the system. On the other hand, the use of a Cloud environment has been shown beneficial for both external processing made on logs and the respective visualization of information in dashboards.

CardioWheel is an IoT solution created by CardioID that monitors the overall state of its user during driving activity and requires the development of a logging system to track and observe data flow in the application. The occurrence of sensitive user information demands a carefully planned design of the different components in the logging system.

The work carried out within the scope of this dissertation aims to investigate and propose a solution that complies with all the mentioned limitations in IoT systems to allow a dynamic monitoring process of CardioWheel.

The proposed solution uses the logging library provided by the Espressif framework to save the necessary information in external memory, to be eventually sent at the end of the current logging session or, in the case of failure, future sessions. We adapted the CycloneSSH library to guarantee integrity and safety during the transmission of the file that contains the session logging events. In the Cloud environment, we used Logstash to interpret, transform, and redirect the received information to be later viewed and analyzed on the OpenSearch platform.

The described implementation was encouraging as it had minimal impact on the critical points of the IoT devices. Although, it is still possible to further optimize the life cycle of the logging event in the system.

Keywords: Logging, IoT, Cloud, CardioWheel

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ACL *Access Control List*. 44
APT *Advanced Persistent Threat*. 43
ARC *Automated Root Cause*. 56
AS2 *Applicability Statement 2*. 71, 72
ASF *Adaptive Semantic Filtering*. 28
AST *Abstract Syntax Tree*. 27
ATM *Automatic Teller Machines*. 50
AWS *Amazon Web Services*. 52

Bi-LSTM *Bi-Direcional Long Short-Term Memory*. 42
BLE *Bluetooth Low Energy*. 65

CRUD *Create, Read, Update, Delete*. 76

Davis *Dynatrace AI causation engine*. 52
DBSCAN *Density Based Spatial Clustering*. 46
DFS *Distributed File System*. xxiv
DHCP *Dynamic Host Configuration Protocol*. 45
DLQ *Dead-Letter Queue*. 75
DNS *Domain Name System*. 43
DoS *Denial of Service*. 41
DRAM *Dynamic Random-Access Memory*. 87, 89, 90

EDI *Electronic Data Interchange*. 71, 72

ESP-IDF *Espressif IoT Development Framework*. xii, 61, 62, 63, 67, 68, 69, 72, 86, 89, 90, 91

F2FS *Flash-Friendly File System*. 90
FAT *File Allocation Table*. 69
FTP *File Transfer Protocol*. 71
FTPS *File Transfer Protocol Secure*. 71

GAUL *Gestalt Analysis of Unstructured Logs*. 31
GCP *Google Cloud Plataform*. 52
G-SWFIT *Generic Software Fault Injection Technique*. 15

HDFS *Hadoop Distributed File System*. 45
HELO *Hierarchical Event Log Organizer*. 30
HRV *Heart Rate Variability*. 65
HTTP *Hyper Text Transfer Protocol*. 71
HTTPS *Hyper Text Transfer Protocol Secure*. 71

IDF *Inverse Document Frequency*. 39
IDS *Intrusion Detection System*. 41
IoT *Internet of Things*. vii, viii, ix, x, 1, 2, 3, 30, 55, 57, 58, 60, 72, 87, 91
IPLMoP *Iterative Partitioning Log Mining*. 28, 30, 33
IRAM *Internal Random-Access Memory*. 87, 89, 90

JSON *JavaScript Object Notation*. 77, 80

KSS *Karolinska Sleepiness Scale*. 65

LCDM *Logging Context Description Model*. 21
LDA *Latent Dirichlet Allocation*. 26
LoRa *Long Range*. 2
LSTM *Long Short-Term Memory*. 41
LTF *Log Tensor Factorization*. 46
LTTng *Linux Trace Toolkit, next generation*. 53

MDN *Message Disposition Notification*. 71

NLP *Natural Language Processing*. 42

OTS *Off-The-Shelf*. 17

PARIS *Principle Atom Recognition In Sets*. 28
PCA *Principal Component Analysis*. 40, 41
PCC *Pearson Correlation Coefficient*. 28
POP *Parallel Log Parsing*. 33

RADIUS *Remote Authentication Dial In User Service*. 44
RAM *Random Access Memory*. 34, 61
RAS *Reliability, Availability and Serviceability*. 49
REST API *Representational State Transfer Application Programming Interface*. 76
RR *Round Robin*. 62
RRCF *Robust Random Cut Forest*. 78, 79

SFTP *SSH File Transfer Protocol*. 71, 72
SMP *Symmetric Multiprocessing*. 61
SNTP *Simple Network Time Protocol*. 65, 69, 86, 88
SSD *Solid-State Drive*. 90
SSH *Secure Shell*. 71, 72, 73, 74, 88
SSL *Secure Sockets Layer*. 71, 75
STE *Statistical Template Extraction*. 46
SVM *Support Vector Machine*. 38, 41

TACACS+ *Terminal Access Controller Access-Control System Plus*. 44
TLS *Transport Layer Security*. 71, 75

UART *Universal Asynchronous Receiver-Transmitter*. 63
UML *Unified Modeling Language*. 17

Glossário

Apriori Algoritmo para mineração de conjuntos de itens e para associação de regras de aprendizagem em bases de dados relacionais . 28, 30

Buffering Utilização temporária de memória num dispositivo para mover informação de um objeto para outro . 12

Churn Rate Valor resultante da divisão do *Churned LOC* de uma porção do código por outra que visa medir o esforço da manutenção necessária da porção do código em questão, por exemplo a divisão da quantidade de *logs* alterados/removidos pela quantidade de código total . 6, 8

Churned LOC Quantidade de código que é rescrito ou eliminado num curto período de tempo após ser escrito . xxiii, 6

Elastic Stack Conjunto de ferramentas, particularmente o *ElasticSearch* e o *Kibana*, para enriquecimento, armazenamento, análise e visualização de informação . xvi, 53, 54, 55, 74

Fault Injection Técnica de teste que consiste em observar o comportamento de um sistema informático após se induzir uma falha . 8, 15

Hard Coded Valor estático que se encontra inserido no código . 13

Lucene Motor de pesquisa utilizado para sistemas que requerem capacidades de indexar e/ou procurar . 32, 53, 76

MapReduce Modelo de programação utilizado para processar e gerar grupos de dados de grandes dimensões através de um *Distributed File System* (DFS) que oferece armazenamento distribuído, sendo tipicamente constituído por um procedimento distribuídos para filtrar e sortear e outro para reduzir, ou sumarizar, operações . xv, 31, 32, 44, 45, 48

N-Gram Consiste numa sequência contígua de n itens de uma determinada amostra de texto . 26

open source *Software* desenvolvido sob uma licença que permite aos utilizadores o direito de estudar, alterar e distribuir a tecnologia para qualquer entidade, com qualquer objetivo . 15, 54, 74, 76

Espaço Euclidiano Espaço vetorial real de dimensão finita munido de um produto interno . 42

1

Introdução

1.1 Enquadramento

Novos sistemas informáticos são diariamente desenvolvidos e disponibilizados para uma variedade crescente de áreas e aplicações, muitas delas sensíveis a falhas técnicas. As soluções *Internet of Things* (IoT) constituem um exemplo deste tipo de sistemas e correspondem a uma rede de objetos ciberfísicos, designados por *things* e que podem ser desde objetos domésticos a ferramentas industriais, e que integram sensores, *software* e outras tecnologias com a finalidade de se ligarem e trocarem dados com outros dispositivos ou sistemas via *internet*. Devido ao modelo escalável, flexível e eficiente proporcionado, a *Cloud* é o ambiente normalmente utilizado pelos sistemas IoT para a entrega e o processamento dos dados produzidos por este tipo de dispositivos. Neste tipo de sistemas informáticos, também designados por sistemas distribuídos, existe uma grande necessidade de monitorizar e diagnosticar problemas em sistemas remotos ao longo de todo o ciclo de vida de um produto. Em desenvolvimento tradicional de *software* existem dois tipos de ambientes associados ao ciclo de vida de uma aplicação. No *ambiente de desenvolvimento* decorre a elaboração e validação de novas funcionalidades. Por outro lado, no *ambiente de produção* encontra-se uma versão estável e em utilização da aplicação. Durante a fase de desenvolvimento do software, há várias ferramentas disponíveis para auxiliar a tarefas do engenheiro de software, tal como *debuggers* (funcionalidades tipicamente disponíveis em plataformas de desenvolvimento, como o *Eclipse* e o *IntelliJ*, que apoiam na procura e resolução de *bugs*, defeitos

ou problemas que impedem o correto funcionamento do código), porém, *logs* são muitas vezes a única fonte de dados acessível em ambiente de produção para apoiar o despiste e a correção de problemas no funcionamento.

Logging, por sua vez, corresponde ao ato de guardar um registo de eventos que ocorrem num determinado sistema computacional, como problemas, erros ou informação relativa a operações momentâneas [22]. Os eventos podem ocorrer no sistema operativo ou para qualquer *software* e são guardados em mensagens ou entradas de *logs* que, por sua vez, são utilizados para monitorizar e compreender o comportamento do sistema em execução. Um sistema de *logging* está dedicado a facilitar e auxiliar na deteção de falhas via análise da informação gerada. A informação utilizada pode derivar diretamente do sistema de *logs* ou de metodologias de processamento, para destacar determinadas características. As decisões durante o desenvolvimento e planeamento de um sistema de *logs* estão, desejavelmente, comprometidas com a aplicação e o seu ambiente de execução.

Logging é uma prática frequentemente excluída no âmbito do desenvolvimento de um novo sistema e, embora se tenha notado diversos estudos empíricos para sistematizar a introdução de um sistema de *logging* num sistema informático [22], um *standard* para adicionar esta funcionalidade ainda está por definir. Dito isto, algumas das bibliotecas mais utilizadas no desenvolvimento de *software* já oferecem a funcionalidade de gerar dados para incluir em sistemas de *logging* e impõem as suas práticas que podem ou não ir de acordo com outras bibliotecas [22]. Esta inconsistência pode resultar em dificuldades quando surge a necessidade de processar todos estes dados. As funcionalidades de *logging* das bibliotecas também não se aplicam ao código da aplicação em questão e cabe aos engenheiros de software decidir onde e como implementar, de uma forma otimizada e coerente, os *logs* que mais beneficiem o produto final e desenvolver um subsistema dedicado ao *logging* onde se consiga criar, modificar e observar todas as informações geradas pelos *logs*.

Em contexto IoT, a implementação do sistema *logging* está limitada também pelas restrições dos sistemas quanto à sua capacidade de processamento, armazenamento de dados, consumo de energia e ritmo de transferência de dados para a *Cloud*. O acesso remoto aos *logs* é especialmente difícil de alcançar devido a limitações de tráfego encontradas nestes ecossistemas. As tecnologias de transmissão de dados utilizadas, como por exemplo *Long Range* (LoRa) ou SIGFOX, seguem protocolos de comunicação rígidos no que respeita à quantidade de informação enviada por cada dispositivo. As limitações a nível do *hardware* obrigam a que o processamento sobre os *logs* seja externo ao dispositivo.

A *CardioID* é uma empresa que desenvolve dispositivos IoT que exploram o potencial de utilizar o sinal cardíaco para monitorizar e acompanhar o estado dos utilizadores e que pretende agora introduzir sistemas de logging nos seus produtos para melhorar os seus processos de desenvolvimento e manutenção. Um eletrocardiograma corresponde ao registo elétrico do sinal cardíaco e dispõe de informação discriminatória suficiente para identificar indivíduos, estando diretamente relacionado com a sua fisiologia, condutividade da pele, singularidades genéticas e posição e forma do coração. A *CardioID* utilizou a sua tecnologia para implementar o *CardioWheel* que consiste num sistema IoT integrado no volante de um veículo para aferir a fatiga e disposição geral do utilizador, em tempo real, durante a atividade de condução. O sistema embebido utilizado inclui um SoC da Expressif, o ESP32, que é o núcleo de todo o processamento da *CardioWheel*. Tal como qualquer outro dispositivo IoT, o *CardioWheel* apresenta limitações ao nível da capacidade de processamento, da bateria e do armazenamento. Para garantir o funcionamento correto do sistema em questão é, portanto, necessária a implementação de um sistema de *logging* robusto e compatível com as limitações e necessidades que acompanham esta aplicação.

1.2 Objetivos

A presente dissertação visa o estudo e o desenvolvimento de um sistema *logging* adaptado às limitações encontradas em sistemas IoT. Neste desígnio, tem-se como objetivo utilizar as melhores práticas de como, onde e que dados registar e como efetuar o *parsing*, armazenamento e envio desses dados para a *Cloud*. Por outro lado, é para o ambiente *Cloud*, tem-se como principal objetivo a definição de uma metodologia de processamento dos dados recebidos para uma melhor visualização e compreensão dos pontos de falha e causas no sistema.

1.3 Organização do documento

Esta dissertação está organizada em três capítulos.

Neste primeiro capítulo é apresentado o problema em estudo, com o propósito de enquadrar e contextualizar o trabalho e elencar os seus objetivos.

No segundo capítulo são introduzidos os conceitos chave para a compreensão dos temas subjacentes ao trabalho realizado e discutem-se as principais abordagens, técnicas e estratégias existentes para implementar sistemas de *logging*.

No terceiro capítulo apresenta-se a abordagem proposta para conseguir os objetivos mencionados.

Por fim, no quarto capítulo, são demonstrados os resultado obtidos provenientes da solução desenvolvida.

2

Revisão do Estado de Arte

Logging, no seu estado puro, trata-se de uma forma de escrita sequencial que descreve o estado do sistema com um determinado nível de detalhe [22]. Esses níveis podem ser:

- *Trace*, para apresentação da informação mais detalhada que se consegue obter e que é apenas utilizado em casos raros onde é necessária visibilidade total da execução do sistema;
- *Debug*, que inclui a informação utilizada, idealmente, para diagnosticar falhas e resolução de problemas, ou em ambientes de teste, sendo essa informação obtida é menos detalhada que a informação obtida pelo nível *Trace*;
- *Info*, que apenas lista informações sobre a execução do sistema necessárias para denotar alterações do seu estado, entre outros eventos benignos recorrentes num sistema informático;
- *Warn*, que visa guardar informação de eventos inesperados, um problema ou uma falha que pode afetar processos do sistema;
- *Error*, para guardar informações sobre estados inesperados do sistema que possam ter comprometido a sua normal execução;
- *Fatal*, quando se pretende guardar informações sobre estados críticos do sistema que possam ter comprometido a sua normal execução;

Um dos aspetos mais importantes na implementação de uma solução de *logging* é o tipo de informação a apresentar aos responsáveis pela manutenção e/ou desenvolvimento de um sistema informático tendo em conta que se pretende obter informação de boa qualidade sem degradar o desempenho do sistema. Pouca informação poderá indicar uma falha derivada de uma causa ambígua ou levar à desvalorização de determinados *logs* por falta de contexto. Por outro lado, informação em excesso pode ter impacto tanto no armazenamento como no desempenho dos serviços existentes, dado que existem sempre recursos alocados ao *logging*. Sendo certo que uma análise eficiente e exata de *logs* depende da qualidade da informação reunida, no entanto, nem sempre é possível alinhar com antecedência as necessidades desejadas destes dados [22].

Neste capítulo apresenta-se um levantamento dos estudos empíricos e das abordagens que têm sido propostas nos últimos anos para o desenvolvimento e/ou utilização de sistemas de *logging*.

2.1 Estudos Empíricos

Em 2012, Yuan, Park e Zhou efetuaram o primeiro estudo empírico direcionado à práticas de *logging* com referência a quatro projetos desenvolvidos em C/C++ [11]. O estudo foi efetuado sob o *software* mais usado no merdade para a respetiva categoria, nomeadamente *Apache httpd*, *OpenSSH*, *PostgreSQL* e *Squid*. Estes software incluíam capacidade de instanciação de servidores dado que tendem a disponibilizar serviços críticos para outras aplicações e os utilizadores, o que os torna um bom alvo de estudo.

Os autores identificaram que o processo de *logging* ocorria, maioritariamente, de forma arbitrária sem qualquer especificação rigorosa ou procedimento sistemático [12]. Verificaram que as práticas dependiam unicamente da experiência do programador e que, por sua vez, as aplicava ao modificar declarações de *logs* provenientes de bibliotecas de *logging* pré-existentes para uso específico [11]. É referenciado o conceito de *Churned LOC* [13] para analisar algumas métricas relacionadas com estas práticas, de onde se destacam dois tipos de alterações dos *logs*: consistentes e não planeadas [12].

O estudo derivou os seguintes pontos focais [11]:

- Em média, existe uma densidade de 30 linhas de código por *log* e o *Churn Rate* de linhas de *logs* equivale ao dobro de linhas de código. Estes índices indicam que existem recursos dedicados à manutenção das mensagens de *logs* e é uma prática bem presente no quotidiano dos programadores. A Figura 2.1 revela a densidade desta métrica em diferentes *softwares*.

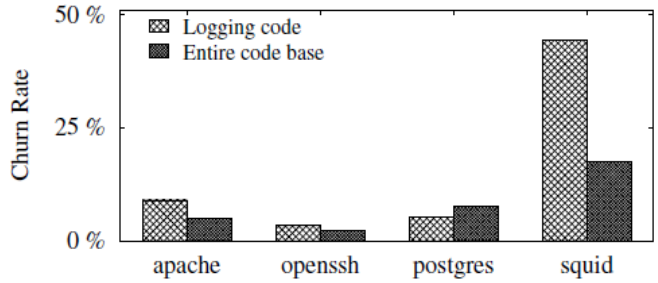


Figura 2.1: Comparação da densidade de *logging* [11]

Software	Total	Reflexão posterior	Seguidas de outras alterações de código		
			Condicional	Variáveis	Funções
Apache	3035	810 (27%)	1491	64	220
OpenSSH	3459	1446 (42%)	1703	284	26
Postgres	15455	5489 (35%)	9153	746	167
Squid	5546	1431 (26%)	2951	930	224
TOTAL	27485	9076 (33%)	15748	2024	637

Tabela 2.1: Média de *logs* alterados sem planeamento, por revisão [11]

- Mensagens de *logging* contribuem para o aumento do tempo de resolução de, em média, 2,2 vezes.
- 33% das modificações efetuada aos *logs*, por revisão, não foram planeadas (Tabela 2.1) e 36% da totalidade das mensagens de *logs* foram modificadas pelo menos uma vez sem planeamento (Tabela 2.2). Estas métricas apontam para uma abordagem mais subjetiva e arbitrária, o que induz problemas na qualidade do código.
- 98% das modificações foram efetuadas ao conteúdo das mensagens, sendo as restantes dedicadas a mover ou eliminar.
- Foram encontradas dificuldades na atribuição dos níveis de detalhe, sendo observadas 2389 modificações tendo-se também identificado uma tendência para subestimar a natureza crítica dos eventos na medida que 72% destas modificações derivaram da alteração de ocorrências críticas (como *error* ou *fatal*) para não

Mensagens Log	Apache	OpenSSH	Postgres	Squid	TOTAL
Modificadas	605	628	3128	1106	5367
Total	1838	3407	6052	3474	14771
Percentagem	40%	18%	52%	30%	36%

Tabela 2.2: Média de *logs* alterados sem planeamento [11]

críticas e vice-versa. Este tipo de decisões durante o *logging* pode ser otimizada com uso a ferramentas auxiliares, como *Fault Injection* [19].

Em 2017 foi efetuado um estudo semelhante em 21 projetos desenvolvidos usando a tecnologia Java, em diferentes categorias [12]: *server-sided*, *client-sided* e *SC-based*. Comparativamente ao estudo anterior, retiraram-se as seguintes conclusões:

- O tempo de resolução de *bugs* foi realisticamente maior quando na presença de mensagens de *logging*. Porém, estes dados foram inconclusivos devido à falta de fundamentos relativos ao impacto de *logging* na resolução de *bugs*.
- Em média, verificou-se uma densidade de 51 linhas de código por *log*, um aumento significativo. Denota-se ainda uma grande variabilidade de densidade entre diferentes projetos e diferentes categorias, tendo tendência a aumentar em projetos de maiores dimensões.
- Destaca-se um *Churn Rate* muito semelhante, no entanto verifica-se um maior número de ocorrência de *logs* eliminados ou movidos.
- A percentagem de modificações consistentes verificou-se ser significativamente menor, baixando de 63% para 50%. Esta percentagem mantém-se baixa em projetos *client-sided*, 38%, e *SC-based*, ou *supporting-component based*, 29%.

Em 2019, em [14] foram exploradas explorou estas dinâmicas em diversas aplicações projetadas em ambiente *mobile* para analisar as práticas utilizadas e como estas se comparam com outros ambientes, tendo-se obtido os seguintes resultados comparativamente aos estudos anteriores:

- Em média, verificou-se uma densidade de 479 linhas de código por cada *log*, o que indica que a prática de *logging* é muito menos evidente em aplicações móveis.
- O *Churn Rate* de *logs* é de 0,7 vezes o *Churn Rate* da totalidade do código, que aponta ao facto de não existirem tantos recursos alocados à manutenção dos *logs*.
- A maioria do nível de detalhe dos *logs* é referente a ocorrências nas categorias *debug* e *error*, enquanto que a maioria das ocorrências de *logs* nas aplicações anteriormente apresentadas são da categoria *info*.

Este estudo também demonstrou o impacto negativo da existência de *logging* nos dispositivos e como isso afeta o seu desempenho. De uma forma geral, verificou-se uma

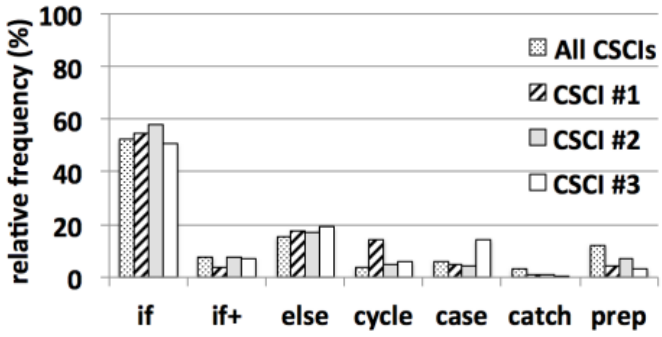


Figura 2.2: Estudo de declarações utilizadas [15]

correlação muito forte entre a degradação dos tempos de resposta, da autonomia da bateria e da capacidade de cálculo exigida ao processador do dispositivo.

Estes trabalhos refletem sobre o esforço de desenvolvimento associado à definição e ajuste do nível de detalhe e do conteúdo das mensagens dos sistemas de logging. Este esforço pode ser mitigado com o desenvolvimento de padrões de boas práticas ao identificar modificações comuns [11]. Porém sem *standards* definidos, surgem obstáculos no *logging* da informação.

Esta dificuldade é explorada por *Pecchia* no estudo [15] que visa explorar e destacar as diferentes práticas aplicadas em sistemas de *logging*. Os resultados foram derivados das práticas implementadas por 100 programadores e *testers* num projeto constituído por três linhas de produtos: *middleware* (CSCI #1), *business logic* (CSCI #2) e *human-machine interface* (CSCI #3). Um dos primeiros resultados observados revela que o padrão de código mais utilizado pelas equipas é a declaração *if*, maioritariamente utilizada para uma comparação direta entre variáveis, conforme se ilustra na Figura 2.2. Apesar de se terem adotados diferentes procedimentos de *logging* em linhas de produtos distintas, estes resultados foram transversais pelas diferentes equipas, indicando mecanismos muito semelhantes na sua implementação.

Todos os padrões mencionados anteriormente, entre outros, foram analisados para determinar quais os objetivos analíticos da entrada de *log*. Desta avaliação destacaram-se três propósitos: *State Dump*, destinado a reportar o conteúdo de variáveis e/ou estruturas de dados; *Execution Tracing*, que trata de notificar se uma determinada porção do código foi executada; e *Event Reporting*, cujo objetivo é providenciar informações sobre um determinado evento que foi executado ou função que foi atingida (como por exemplo eventos de erro). Os objetivos listados mostraram ainda que não existia um consenso entre as diferentes linhas de produto, e que as técnicas de logging foram sendo ajustadas de acordo com as necessidades. Esta relação é demonstrada na Figura 2.3

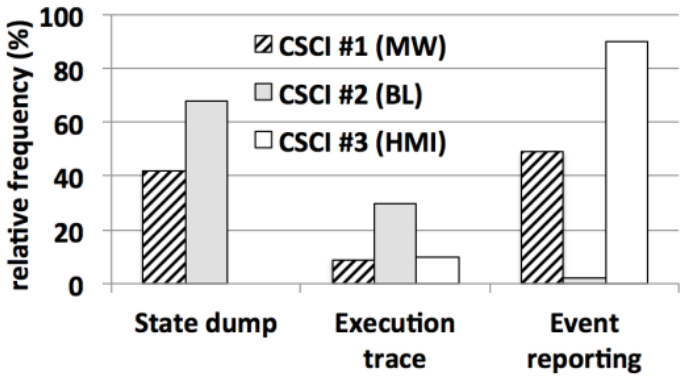


Figura 2.3: Estudo do propósito das mensagens de *logs* [15]

Os autores também concluíram que algumas das práticas aplicadas pelos programadores causavam impactos negativos na análise dos *logs*. O artigo demonstra o uso de três formatos aplicados em *timestamps* pelas diferentes linhas de produção e que, a uniformização deste formato, pode contribuir para o desenvolvimento de funcionalidades com a capacidade de otimizar ou até automatizar a procura de *logs* em determinadas situações. Os autores apontam também para a falta de convenção na nomenclatura de variáveis e na descrição de pares chave-conteúdo (que se referem à representação de informação, conteúdo, de uma determinada variável, chave), que dificultam a análise dos *logs* quer seja manual ou automática. Nota-se ainda alguma inclinação, por parte dos programadores, de reportar conteúdos inteiros de estruturas de dados complexas, como *arrays*, que, embora possam ser úteis para visualizar a informação, não facilitam a implementação dos tópicos já referidos dado que estes *logs* podem ocupar diversas linhas de texto. Para concluir, é indicado um dos pontos mais importantes da secção, que explica a inexistência de indicações de severidade das mensagens de *log* [15].

Numa tentativa de uniformizar algumas destas nomenclaturas, *Chen Zhi* demonstra a possibilidade de pré-definir configurações de *logs* em projetos baseados em *Java* [16]. Estes projetos utilizam funcionalidades como a representação *XML* existente nos ficheiros *pom.xml*, em *Maven*, para definir um padrão universal das mensagens. Na Figura 2.4 denota-se a definição de um padrão composto por um *timestamp*, o nível de detalhe, o *logger* e a mensagem.

Os *loggers* são as entidades, ou classes, responsáveis por capturar os pedidos de *log* e os redirecionar para o *appenders*. A nomenclatura dos *loggers* segue uma hierarquia e, portanto, é necessário escolher uma boa convenção para os organizar. De entre todas as representações das mensagens estudadas, destacaram-se duas convenções de nomenclatura mais utilizadas: por tópico e por *package*, sendo esta a mais utilizada. Utilizar a convenção por *package* implica que o *logger* seja gerado com o nome da classe

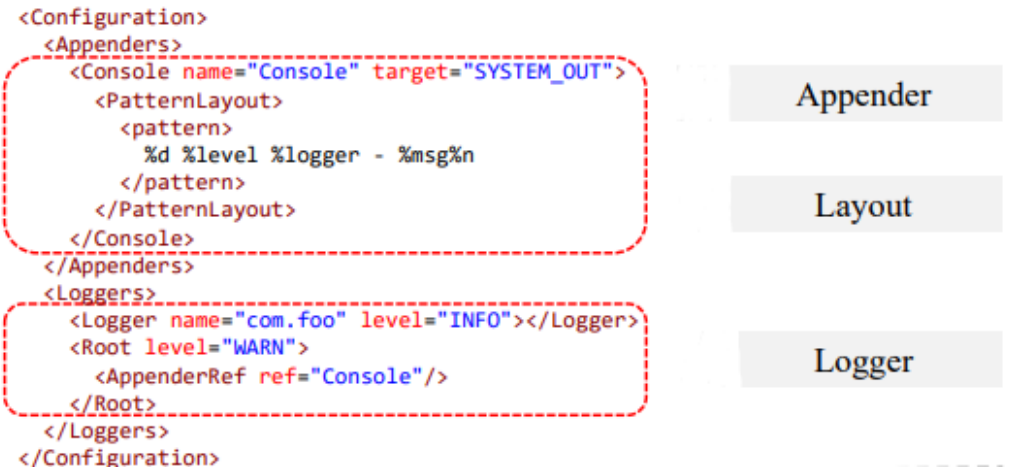


Figura 2.4: Exemplo de configuração de mensagens de *logs* [16]

ou da *package* que, por sua vez, não clarifica suficientemente o seu contexto quando diferentes atividades de *logging* ocorrem na mesma *package* ou classe. A aplicação da nomenclatura por tópico visa diversificar o contexto do *logger* indicando sempre um tópico por cada atividade. Tendo em conta a sua origem, os *loggers* podem ainda ser internos ou externos, que por sua vez são derivados de bibliotecas externas adicionadas no projeto e, neste estudo, contribuíram para 1/3 da totalidade dos *loggers*.

Os *appenders*, ou *handlers*, são responsáveis por registar os pedidos nos destinos correspondentes. Cada biblioteca pode ter um ou mais tipos de *appenders*, cujos destinos podem variar entre um ficheiro de texto, uma tabela em uma base de dados, entre outros. No estudo foram identificados os seguintes tipos de *appenders*, listados por preferência [16]:

- *file appender*, dado que a maioria dos dados foram guardados em ficheiros de texto, em dispositivos físicos;
- *asynchronous appender*, inseridos para efetuar os pedidos de *logging* de uma forma assíncrona, muitas vezes com intenção de diminuir o impacto no sistema;
- *console appender*, neste caso a informação não é persistida e apenas é apresentada na consola.

Alguns *appenders* que visam persistir informação, tal como o *file appender*, podem comprometer entradas de *logs* quando ocorre uma falha no destino de armazenamento, por exemplo no caso de um ficheiro demasiado grande. Para contornar estes riscos é aplicada a prática de *Rolling Policies*, que consiste em arquivar a informação gerada quando determinada condição é alcançada. Um exemplo deste dinamismo é arquivar um ficheiro ao atingir um tamanho pré-definido.

Characters	Freq.	Brief Description
%n	346	line separator character
%msg	334	developers supplied message
%date	313	the date (time, time zone, etc.)
%mdc	246	developers supplied thread-context message
%level	224	developers supplied logging level
%logger	190	developers supplied logger name
%ex	42	control the depth of stack trace
%thread	36	the name of the thread
%line	13	the line number in source file
%class	6	the fully qualified class name of the caller
Other	9	such as the method of the caller

Tabela 2.3: Exemplos de *layouts* [16]

Os *layouts* convertem e formatam a informação passada no pedido. Os *layouts* mais utilizados podem-se verificar na Tabela 2.3.

O estudo apresenta também uma perspetiva de como as configurações de *logging* evoluem paralelamente ao projeto. Um dos principais tópicos abordados, indicam alterações na estrutura lógica da configuração onde se distinguem duas vertentes [16]:

- Adição e/ou remoção de *loggers* e *appenders*, que constituem a maioria das alterações. Uma parte desta vertente resulta de adaptações em forma de revisões ao código categorizado e/ou às configurações de *logging* correspondentes. As restantes adições e remoções ocorrem por necessidade de separar determinadas mensagens para facilitar a perceção do processamento do sistema.

- Modificação entre *loggers* e *appenders*. A associação entre *loggers* e *appenders* pode ser implícita ou explicita, sendo que neste último caso é adicionado conteúdo extra pelo *appender*. Mais de 90% de *loggers* configurados com associações explicitas removem associações implícitas para evitar comportamentos indesáveis durante o *logging*. Nos projetos estudados, as modificações em associações explícitas foram efetuadas para trocar entre um modo de *logging* dedicado a ambiente de produção e um modo a ambiente de desenvolvimento, ou para trocar entre *appenders* síncronos e assíncronos.

Modificações no armazenamento referem-se a alterações efetuadas nas especificações de um *appender* existente para ajustar o comportamento do armazenamento da informação e pretendem aumentar a confiabilidade e o desempenho. Parte das modificações são efetuadas em parâmetros de configuração dos *appenders* destinados a controlar armazenamento, e podem distinguir-se em dois tipos: capacidade, para definir o limite de tamanho dos ficheiros de *logs* e a quantidade de ficheiros; execução, para ativar funcionalidades avançadas como assincronização ou *Buffering*. As restantes modificações

Migration Path	Freq.
Time based rolling → Size based rolling	84
Time based rolling → Size and time based rolling	54
No rolling → Time based rolling → Size based	3
No rolling → Time based rolling → Size and time based rolling	3
Size based rolling → Time based rolling	2
No rolling → Size based rolling	1

Tabela 2.4: Modificações na *rolling policies* [16]

referem-se à alteração das *rolling policies* mencionadas anteriormente (Tabela 2.4). São maioritariamente dedicadas a migrar condições temporais para condições de armazenamento, pois, em implementações baseadas em condições temporais, o limite de espaço de armazenamento físico era ultrapassado.

Algumas alterações são efetuadas nas variáveis definidas na configurações (que, em execução, são substituídas por valores reais), sendo que grande parte destas alterações são dedicadas a adicionar variáveis novas. As restantes alterações resultam da substituição de variáveis por valores *Hard Coded*, ou seja, valores estáticos que se encontram inserido no código de um programa.

Os *loggers* e *appenders* possuem também *thresholds* para facilitar a filtragem dos níveis de detalhe, desta forma, qualquer pedido de *log* com um nível abaixo do *thresholds* especificado é cancelado. A maioria das modificações efetuadas aumentam o *threshold*, especialmente de *debug* para *info*, quando determinados componentes em produção se encontram estáveis, e de *info* para *error* ou *warn*, quando a existência de mensagens triviais excessivas afeta a eficiência dos diagnósticos. As restantes ocasiões levam à diminuição do *threshold* e a alterações para impedir herança dos níveis entre *loggers*.

As alterações no *layout* encontradas pelo estudo podem ser distinguidas em três subcategorias [16]:

- Aperfeiçoamento do *layout*, quando se adiciona informação extra à mensagem, nomeadamente referente a processos independentes do programa;
- Simplificação do *layout*, onde é removida informação descontextualizada e redundante, assim como ajustes em informações relativas à localização (como a linha ou a classe do evento);
- Normalização do *layout*, não altera o conteúdo da mensagem, mas a sua forma de visualização.

Estes dois estudos consolidam as conclusões já extraídas em artigos anteriores através do nível de detalhe que foi possível derivar das práticas comuns utilizadas nestes

projetos, alertando à importância da uniformização de nomenclaturas em sistemas de *logging*.

Frameworks são essenciais para acelerar o desenvolvimento de aplicações de larga escala e, por sua vez, dependem de *logs* para monitorizar a execução de tais aplicações. *Weiyi Shang* [17] acredita que tais declarações de *logs* oferecem uma grande utilidade na compreensão da qualidade do código. Por análise das modificações efetuadas sobre estas declarações em plataformas de *software*, como *Hadoop* e *JBoss*, foi possível identificar temáticas que mostram o valor no estudo da relação entre caraterísticas de *logging* e qualidade do código, das quais se destacam:

- Inclusão de *logs* de *debug* para diagnosticar inconsistências durante a execução;
- Adição e modificação de *logs* de acordo com alterações nas implementações.

O estudo menciona que métricas de produto, como número de linhas de código, descrevem o estado estático de determinado sistema, dado que o contexto é obtido de uma versão específica do sistema, enquanto que métricas de processo, como número de alterações efetuadas, capturam o histórico de desenvolvimento do sistema. Ambas as métricas resultam em bons indicadores de código de qualidade e, portanto, são aplicadas vertentes dos dois tipos. No que respeita a métricas de produto, foi utilizada a densidade de *log* e média de nível de detalhe como parâmetros de estudo. Por outro lado, foi utilizada a média de linhas relacionadas com *logs*, adicionadas ou removidas após determinada alteração, e a frequência de alterações efetuadas para resolver problemas com modificações em *logs*. Através da análise de tais métricas, concluíram-se as seguintes relações entre caraterísticas dos *logs* e qualidade de código:

- Ficheiros de código que incluem declarações *log* destacam-se com uma maior densidade de defeitos encontrados em ambiente de produção;
- Correlação positiva entre ficheiros de código com linhas de código adicionadas pelos programadores e ficheiros de código com defeitos encontrados em ambiente de produção;
- Métricas relacionadas com *logs* complementam métricas de produto e de processo no diagnóstico de defeitos encontrados em ambiente de produção.

2.2 Estratégias de Diagnóstico de *Logs*

A informação presente em *logs* é esperada ser informativa e útil. No entanto, alguns estudos verificaram que, em certas situações, o sistema de *logging* é incapaz de detetar

Acronym	Explanation
OMFC	Missing function call
OMVIV	Missing variable initialization using a value
OMVAV	Missing variable assignment using a value
OMVAE	Missing variable assignment with an expression
OMIA	Missing IF construct around statements
OMIFS	Missing IF construct plus statements
OMIEB	Missing IF construct plus statements plus ELSE before statem.
OMLAC	Missing AND clause in expression used as branch condition
OMLOC	Missing OR clause in expression used as branch condition
OMLPA	Missing small and localized part of the algorithm
OWVAV	Wrong value assigned to variable
OWPFV	Wrong variable used in parameter of function call
OWAEP	Wrong arithmetic expression in parameter of a function call

Tabela 2.5: Lista de operadores de falha [36]

falhas de *software* que podem compremeter o correto funcionamento das aplicações. A técnica de *Fault Injection* insere-se nas soluções designadas a avaliar a eficácia do sistema de *logging* quando exposto a determinadas condições.

Em [36] apresenta-se um estudo realizado usando três sistemas *open source*: *Apache Web Server*, *TAO Open Data Distribution System* e *MySQL Database Management System*. Nestes sistemas foram observadas quais as causas mais comuns de falha por consequência de uma *Fault Injection* assim como ineficiências no processo de *logging*. Esta abordagem de teste permitiu destacar três consequências da injeção de determinada falha: terminação inesperada do sistema, a execução do sistema é dada como terminada e nenhuma informação é referida nos *logs*; o sistema mantém um estado estável (consequências silenciosas), no entanto nenhuma informação é declarada dentro de um tempo limite razoável; outras diversas falhas, como por exemplo valores errados entregues aos utilizadores. Foi utilizada a técnica *Generic Software Fault Injection Technique* (G-SWFIT) [19], uma abordagem prática à injeção de falhas que emula classes das falhas de *software* mais frequentemente observadas. Na Tabela 2.5, são listados todos os operadores avaliados no estudo.

No sistema mais relevante, *Apache Web Server*, verificou-se que a grande maioria das injeções, independentemente do operador utilizado, não foi captada pelo sistema de *logging*. Obteve-se a distribuição de consequências demonstradas na Figura 2.5. As terminações inesperadas (*halt*) são maioritariamente provocados por alterações na memória e, em média, metade das falhas são capturadas apenas porque um dos processos filho do sistema termina, o que significa que os restantes processos são capazes de detetar o problema, embora sem grandes detalhes. As consequências silenciosas (*silent*) divergem entre erros algorítmicos provocados por ciclos infinitos e falhas atribuídas a recursos do sistema operativo. Caso não ocorra *halt* nem *silent*, a consequência é considerada como *content*, um exemplo desta falha é a produção de um mau *output*.

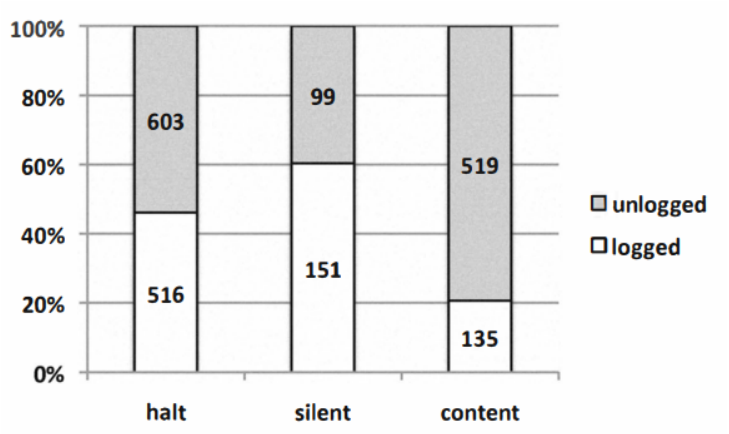


Figura 2.5: Distribuição de consequências de injeções [36]

Dado que foram obtidos resultados semelhantes nos restantes sistemas, dedicados a diferentes soluções informáticas, é seguro concluir que a cobertura dos mecanismos de *logging* estudados não ultrapassam 50% dos casos situacionais, embora esta seja fortemente afetada por erros associados ao sistema operativo [36].

Posteriormente, em [37], *Cinque*, expõe imperfeições em mecanismos de *logging* tradicionais e como estes dependem de padrões de código simples, sem o apoio de um modelo sistemático de falhas. O estudo vai ao encontro a algumas conclusões derivadas em [15] onde o padrão mais utilizado, que constitui 70% dos casos, visa detetar erros através de verificações situadas no final de determinada porção de código. Os padrões utilizados provaram-se imprecisos devido à forma como interagem com o fluxo de execução e podem provocar falsos negativo quando o erro ocorre e o *log* não é produzido. Os erros que reúnem estas condições podem ser ignorados pelos mecanismos de *logging* e levar a falhas não relatadas. Exemplos desta ocorrência podem caracterizar-se como erros de *timming*, que alterem o fluxo da execução ao ponto de impedir que a instrução de teste seja atingida, como por exemplo um ciclo *for* ou *while* infinito. Existem, porém, outros tipos de erros que ocorrem devido a declarações incorretas de informação estática ou associados a decisões efetuados pelos programadores na informação que pretendem guardar, assim como o seu nível de severidade.

Este estudo propõe uma alternativa às práticas apresentadas anteriormente que, ao invés de aplicar as soluções de *logging* numa fase final do ciclo de desenvolvimento, as implementações são formalizadas através de regras definidas sob um modelo sistemático de erros, construído através de artefactos (Figura 2.6).

Este modelo é representado através de entidades que identificam componentes da arquitetura do sistema que os programadores pretendem analisar via *logging*. Estas entidades são baseadas em artefactos de alto e baixo nível projetados numa fase de

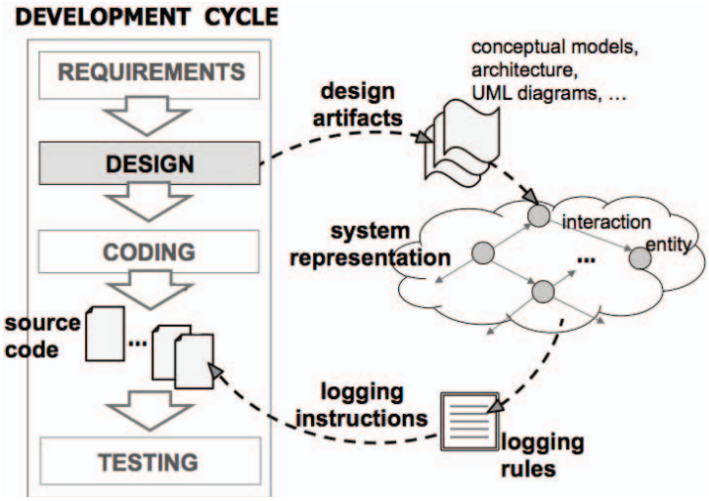


Figura 2.6: *Rule Based Logging* [37]

planeamento, isto significa que podem englobar camadas inteiras de uma arquitetura ou módulos específicos dessas camadas. Podem ainda ser representadas recorrendo a diferentes visualizações do sistema como *packages*, diagramas de classe em *Unified Modeling Language* (UML) ou classes singulares.

Cada entidade possuí um conjunto de serviços invocados por componentes externos, como outras entidades e componentes *Off-The-Shelf* (OTS). Cada serviço pode induzir diversas ações e o fluxo de dados pode permanecer na entidade local ou ser transferido para um componente externo, caso um seja invocado (interação). O processamento pode finalizar ao retornar um valor ou um código de erro origem. Os modos de erros estabelecidos pela arquitetura tomam partido das diferentes componentes e relações que constituem o sistema para derivar falhas que, em mecanismos tradicionais, não seriam listadas [37]:

- *Service Error* (SER), tem como objetivo prevenir que serviços invocados não sejam terminados destinando-se a erros de *timming*;
- *Complaint Error* (CMP), constituí errors que ocorrem na terminação de um serviço, correspondendo ao método tradicional de *logging*;
- *Interaction Error* (IER), destinado a sinalizar quando uma interação criada por uma entidade não foi finalizada;
- *Crash Error* (CER), utilizado para avisar aquando a terminação abrupta de uma entidade, quer seja por um serviço ou outro *trigger*.

Na Figura 2.7 apresentam-se as novas métricas obtidas para o mecanismo de *logging*

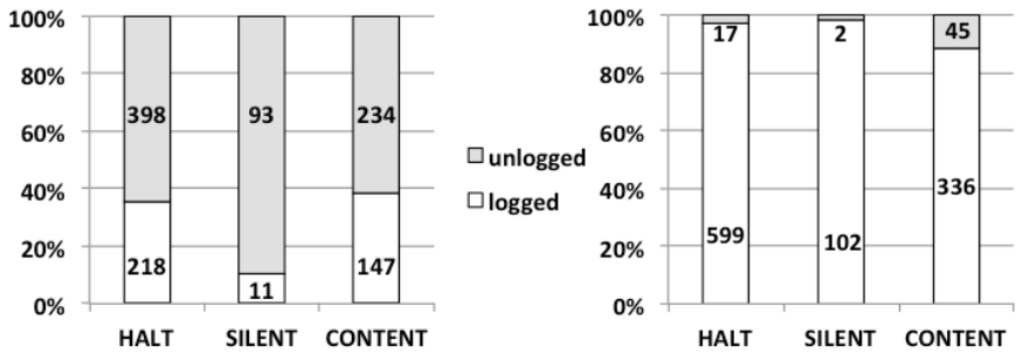


Figura 2.7: Comparação entre a metodologia tradicional e a proposta, respetivamente [37]

tradicional em relação às métricas da arquitetura proposta, ainda no sistema *Apache Web Server*.

Uma abordagem alternativa ao diagnóstico de *logs* é proposta por *Yuan* em 2012 designada *LogEnhancer* [38]. Esta ferramenta é utilizada para modificar todas as mensagens de *logs* de um determinado *software* com o intuito de acumular informação relevante e auxiliar no diagnóstico da falha. Este algoritmo é automaticamente ativado quando é definido um ponto de entrada para um *log*. A estratégia consiste numa análise do código do sistema, no contexto da mensagem de *log*, para identificar variáveis causalmente ligadas ao problema e decorre através de três tarefas:

- *Uncertainty Identification*, esta análise identifica fluxo incoerente e variáveis que não foram resolvidas utilizando o *log* original. Do ponto da mensagem original, é realizada uma análise via engenharia inversa para identificar todas as condições que terão de ter ocorrido para resultar no *trigger* da mensagem de *log*. Ao utilizar as condições como indicações, é inferida a razão através de uma análise do fluxo de dados. Este processo é ilustrado na Figura 2.8.

- *Value Selection*, nesta tarefa são selecionados valores chave que resolvam a coerência do fluxo detetado na fase anterior. Estes valores são derivados através de algoritmos de teste do conteúdo das variáveis nas diferentes condicionais existentes, que são por sua vez sorteados de acordo com o número de condicionais relevantes. Uma tabela de resultados é, depois, gerada para ser utilizada na fase seguinte.

- *Instrumentation*, através do acesso às tabelas geradas para cada entrada de *log*, a informação é adicionada ao conteúdo da mensagem.

A ferramenta foi utilizada em oito aplicações diferentes e provou-se eficiente na deteção de variáveis relevantes em condições de *logging*. Cerca de 95% das variáveis

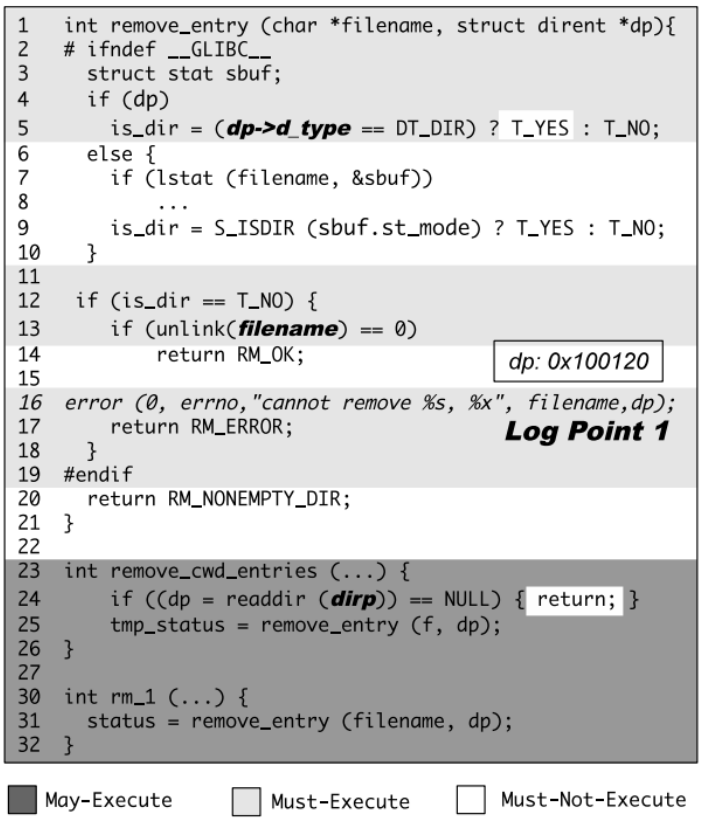


Figura 2.8: *Uncertainty Identification* [38]

colocadas manualmente nas mensagens pelos programadores foram detetadas automaticamente pela ferramenta, o que resulta em uma maior independência do sistema para o *logging* de variáveis (esta característica sinaliza a importância no *logging* informativo e é mencionada brevemente em [11]). Verificou-se, também, que informação adicional era frequentemente compilada nas mensagens que, por sua vez, reduziu o nível de incerteza na identificação das causas [38].

2.3 Sistemas de *Log*

Durante o desenvolvimento de determinado sistema, é necessário tomar algumas decisões que, de alguma forma, afetam a qualidade da informação gerada pelas mensagens de *log* [22]. A revisão do estado da arte mostrou que, normalmente, os programadores utilizam a experiência pessoal e dinâmicas de tentativa e erro para aplicar as práticas de implementação [11][12][14]. No entanto, para se conseguirem práticas eficientes, é necessário responder a três desafios: Como inserir o *Log*? Quando registar em *Log*? e Qual informação registar em *Log*? [41][42].

2.3.1 Como inserir o *Log*?

Esta primeira questão aborda a decisão de como encontrar o melhor ponto de comunicação para um *log*, atendendo a que informação excessiva proveniente de diversos *logs* pode enviesar a compreensão do leitor e/ou afetar o desempenho geral do sistema. Nesta medida, os programadores necessitam ser seletivos [41][42].

Um primeiro estudo, por *Hassani* [40], veio responder à questão de como apresentar os *logs* através da apresentação de algoritmos de verificação. O estudo afirmou que, em 78% dos casos, os *logs* são adicionados e modificados por programadores diferentes, o que dificulta a procura de um especialista quando um problema derivado de um *log* é identificado e pode propagar outros comportamento indesejáveis quando este é alterado por um programador que não é o original. Apesar destes problemas serem rapidamente resolvidos uma vez reportados (menos de 2 semanas), alguns deles só são identificados após um longo período do tempo, o que sugere o auxílio de uma ferramenta de deteção destes comportamentos. A origem dos problemas é distinguida de acordo com as categorias seguintes, por ordem de ocorrência: *logs* inapropriados (como variáveis ou literais incorretos); ausência de declarações, em casos quando um determinado evento ou declaração em falta é requerida pelos programadores; nível de detalhe inapropriado; problemas de configuração das bibliotecas; problemas de execução (ocorrem falhas ou comportamentos indesejados durante a execução do sistema aplicacional); informação excessiva; alterações das bibliotecas.

Com base em padrões estudados e verificados em algumas destas categorias, o artigo providencia algoritmos de verificação (*checker*) que visam alertar os programadores para possíveis comportamentos indesejáveis resultantes da construção da mensagem. Foram desenvolvidos quatro tipos de algoritmos para quatro tipos de problemas [40]:

- *Incorrect log levels*, mensagens com níveis incorretos de detalhe podem enviesar os utilizadores dos *logs* com informação a mais ou a menos. Um algoritmo inteligente é treinado a associar palavras chaves com determinado nível de detalhe pela observação das ocorrências de diferentes palavras/frase e/ou variáveis nas mensagens. Um maior número de ocorrências reflete uma maior certeza da relação com o nível de detalhe (*Shannon entropy*).

- *Missed exception message*, corresponde aos blocos que não possuem declarações de *logs* ou que não descrevem a exceção neles contida. Este algoritmo sugere ao programador para considerar estas alterações consoante o histórico do código local.

- *Log level guards*, algumas bibliotecas possuem um condicional de guarda para evitar que *logs* sejam gerados, caso *logging* esteja desativado em determinado nível da arquitetura. Porém, algumas bibliotecas não disponibilizam um condicional, o que implica a sua inserção manual. Este algoritmo recomenda a modificação de acordo com a as bibliotecas de cada ficheiro e a existência do condicional.

- *Typos*, referem-se a erros ortográficos presentes nas mensagens. Um corretor ortográfico foi implementado com auxílio de um filtro para termos utilizados no sistema que, embora não sejam reconhecidos pelo corretor, são pretendidos pelos programadores, também considerados falsos positivos (um exemplo relevante para o filtro é a repetição de palavras).

Uma abordagem alternativa foi seguida no desenvolvimento do *LogTracker* [41], uma ferramenta que deriva automaticamente regras de revisões de *logs* através da exploração da correlação entre o contexto dos *logs* e respetivas modificações que, por sua vez, recomenda candidatos de revisões baseados nestas regras. Distribui-se em quatro tarefas diferentes:

- Extração das semânticas em contexto dos *logs*, é a componente que mais afeta a eficiência da ferramenta. Semânticas semelhantes podem corresponder a diversas representações sintáticas em contexto *log*. Algoritmos tradicionais de extração das semânticas em código funcional falham quando aplicados em contexto de *logging*, pelo que foi desenvolvida uma nova estratégia: *Logging Context Description Model* (LCDM). Esta estratégia tem como finalidade modelar as condições de verificação e as variáveis de cada *log*, tuplo <c,v>.

- Resgatar modificações em *logs*, que é outro pré-requisito para compreender comportamentos nas revisões de *logs*. O entendimento destas modificações é aferido da diferença entre a estrutura sintática das duas iterações de determinado *log*, tentando eliminar interferência tal como linhas vazias ou comentários.

- Exploração de regras de revisão de *logs*, que resultam da combinação dos dois pontos anteriores, e produzem um tuplo <c,v,e>. Caso uma porção de código partilhe contexto semelhante a um tuplo gerado pelo LCDM, então podem ser-lhe associadas modificações relacionadas com esse tuplo (e vice-versa).

- Aplicação das revisões, em que são utilizadas as regras diferidas para localizar fragmentos de código candidatos e, posteriormente, recomendar as modificações.

Boyuan também desenvolveu uma ferramenta de deteção de anti padrões: *LCAnalyzer* [42]. Estes anti padrões referem-se a falhas recorrentes que impedem uma boa compreensão e manutenção dos *logs*, tendo sido encontrados seis tipos:

- *Nullable Objects*, os conteúdos dinâmicos de um código de *logging* são gerados durante a execução do sistema, no entanto, o valor gerado pode ser *null*. Esta ocorrência pode desencadear exceções quando determinadas condições não são desenvolvidas com esta possibilidade em consideração.
- *Explicit Cast*, consiste em forçar a conversão de um determinado objeto para um tipo específico. Pode levar a uma falha global do sistema se o tipo especificado não mostrar compatibilidade com o objeto (por exemplo converter um bloco de texto para um número).
- *Wrong Verbosity Level*, refere-se ao nível de verbosidade que controla os tipos de informação apresentada na saída, que podem ser FATAL, ERROR, WARN, INFO, DEBUG e TRACE, sendo TRACE o maior nível de verbosidade. A utilização de um determinado nível impede que os níveis superiores não sejam utilizados. Este anti padrão pode resultar em informação redundante, o que significa que o nível tem de ser selecionado de acordo com o objetivo dos programadores.
- *Logging Code Smells*, indicam mau desenvolvimento e escolha incorreta de implementação, identificando-se, principalmente, através de código de *logging* duplicado. Estas situações dificultam a manutenção das declarações de *logs*, dado que pode escalar em inconsistências na informação apresentada se todas as instâncias não forem atualizadas em concorrência. Desta forma, a resolução mais prática deverá ser a utilização de um método ou uma variável universal para efetuar o *logging*.
- *Malformed Output*, corresponde ao formato da mensagem de *log*, que pode não estar humanamente percetível.

Zhenhao estudou cinco tipos de duplicações de código, referentes ao *code smell* anteriormente descrito [43]. A ferramenta *DLFinder* foi desenvolvida com o intuito de detetar automaticamente estes cinco tipos, que podem ser identificados através de detecção de vários tipos de incoerências:

- Entre blocos *catch* do mesmo *try*, que pode dificultar o diagnóstico do tipo de exceção quando as mensagens são semelhantes. Para todas as ocorrências verificadas, informação relativa ao tipo, mensagem e *stack trace* da exceção não são

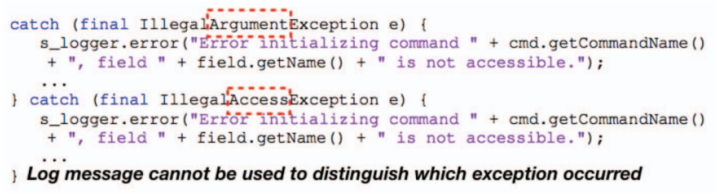


Figura 2.9: Duplicação em blocos *catch* [43]

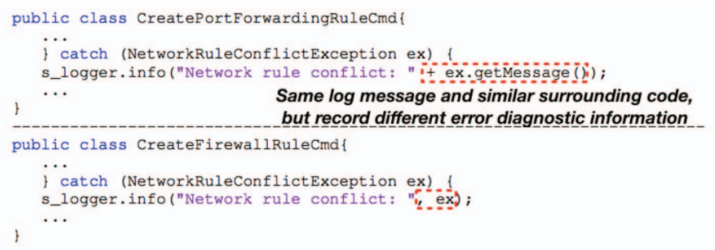


Figura 2.10: Duplicação em informações de diagnóstico de erro [43]

descritas. Na Figura 2.9 apresenta-se dois blocos *catch* distintos que executam código equivalente.

- Em informações de diagnóstico do erro, ocorrendo em instâncias de pontos de *logging* onde as mensagens e o código circulante são muito semelhantes, embora o tipo de erro reportado seja diferente (Figura 2.10).
- Na construção da mensagem, de log causada por referências incorretas ao contexto do erro. Acontece, tipicamente, quando o programador copia a lógica do *log* para um cenário diferentes sem efetuar as alterações necessárias (Figura 2.11).
- Wm polimorfismo, referindo-se à repetição desnecessária de código de *logging* (Figura 2.12).
- No nível de detalhe, que indica que níveis são utilizados em circunstâncias semelhantes (Figura 2.13).

Relativamente ao nível de detalhe, geralmente os programadores encontram grandes

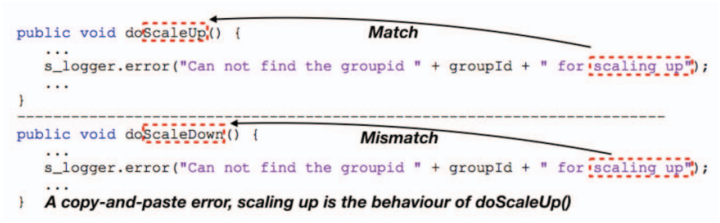


Figura 2.11: Duplicação na construção de mensagens [43]

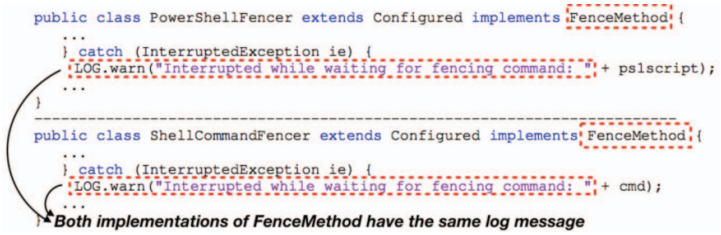


Figura 2.12: Polimorfismo [43]

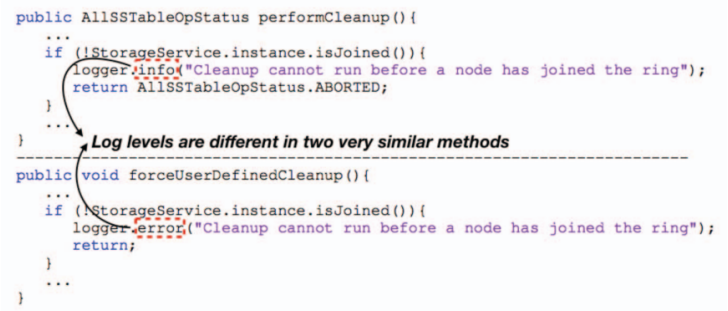


Figura 2.13: Duplicação no nível de detalhe [43]

dificuldades em estimar os custos e benefícios de cada nível. Normalmente, é muito difícil automatizar, de uma forma universal, o processo de atribuição dos detalhes em mensagens. No entanto, é possível atingir este objetivo se as práticas utilizadas forem consistentes, conforme advoga *Heng* em [49]. Neste estudo, dada a diversidade na distribuição do tipo de detalhe em diferentes projetos, foram extraídas cinco métricas essenciais para prever o nível de detalhe apropriado: características das declarações de *logs*, tipo de blocos (*if*, *catch*, entre outros), característica do ficheiro onde o *log* é guardado, alterações de código associadas ao *log* e histórico de alterações da classe. O estudo concluiu que os tipos de blocos utilizados, assim como as declarações de *logs* já implementadas e o conteúdo de novas declarações, têm um papel importante no nível de detalhe nas declarações mais recentes. É também mais provável ser afetado pelo código base atual que o histórico do seu desenvolvimento.

Numa abordagem complementar, *Han Anu* demonstra através da ferramenta *VerbosityLevelDirector* que, para além de beneficiar das características do código, é possível utilizar a intenção subentendida do *log*. Esta intenção é, por sua vez, extraída de comentários que podem, ou não, enriquecer o resultado [50].

2.3.2 Quando registar em *Log*?

Esta questão refere-se ao fornecimento de informação útil ao utilizador de uma forma clara e coerente. Como referido anteriormente, não é desejavel apresentar informação

a menos, que resulta em falta de contexto relativo à execução do sistema, nem informação a mais, que pode afetar o desempenho e o custo do armazenamento, com o risco de gerar informação redundante e aumentar custos de manutenção. Para aplicar um bom equilíbrio, os programadores necessitam de efetuar decisões educadas à questão de onde colocar o *log*.

De acordo com *Qiang* [46], para entender as práticas utilizadas, é necessário primeiro categorizar como efetuar o *logging* de trechos de código. Das categorias encontradas verificou-se uma distribuição equilibrada entre dois grupos de informação: situações inesperadas e pontos de execução. As situações inesperadas ocorrem quando a execução do sistema leva a uma falha e, o *log*, é efetuado através de verificações via *assert*, retorno de valores e/ou exceções estes *logs* são úteis para entender a origem do problema. Os pontos de execução são ocorrências esperadas e informativas sobre o estado do sistema, tipicamente através de *logging* por ramos (como declarações *if* ou *switch*). Quando uma falha ocorre, os programadores utilizam os *logs* relacionados com falhas inesperadas para identificar a origem. Uma vez encontrada a origem, o procedimento será rastrear até à raiz do problema utilizando pontos de execução chave, que fornecem informação relativa ao caminho e estado da execução.

De forma a reduzir o esforço na decisão de onde colocar as declarações de *logs*, *Jieming Zhu* propôs uma *framework* informativa que fornece orientações durante a fase de desenvolvimento: *LogAdvisor* [47]. Esta framework infere as práticas mais comuns de colocação das declarações através da análise de instâncias atualmente existentes no código, que são, por sua vez, sugeridas aos utilizadores. Numa fase inicial, a ferramenta filtra os troços de código relevantes pela pesquisa por invocações de métodos de *logging*, como por exemplo, *Console.Writeline()*. Dos troços identificados, são extraídas três características:

- Estrutura, útil para aferir informação contextual com recurso ao tipo de erro utilizado, por exemplo, semânticas que se podem concluir do estudo das exceções (*FileNotFoundException*), ou por métodos associados e de onde é possível deduzir a sua funcionalidade (por exemplo por observação do nome dos métodos);
- Textual, através da análise de diversas componentes do código, como variáveis e tipos, que é efetuada através do modelo saco-de-palavras para extrair características;
- Sintática, para conseguir a contextualização e utilizando diversas operações sintáticas como a atribuição e/ou retorno de valores como -1/null/*false*/*empty*, existência de declarações *throw*, declarações *try* aninhadas e blocos vazios (dentro de uma declaração *catch* por exemplo).

Tendo as características identificadas, é então filtrada qualquer informação redundante ou irrelevante. Este tratamento pode ser feito limitando o mínimo de ocorrências aceites pela ferramenta. Todas as características são depois inseridas no modelo de treino do *LogAdvisor*.

Num outro estudo [48], *Heng* menciona a importância de considerar a funcionalidade do troço do código ao efetuar sugestões de *logging*, pois certas funcionalidades implementadas têm uma maior probabilidade de necessitar de declarações. Para o estudo dos tópicos utilizados no sistema foi utilizado *Latent Dirichlet Allocation* (LDA) ao nível dos métodos para detetar consistências linguísticas em grupos de código observados. A relevância dos tópicos detetados foi calculada quantitativamente através da densidade de *log*. Verificou-se uma maior frequência de *logs* relacionados com tópicos de "Conexão"e "Construção de *strings*". Além disso, não se apresentaram variações significativas dos tópicos detetados em sistemas de diferentes tecnologias.

2.3.3 Qual informação registar em *Log*?

Esta questão leva a uma discussão relativa à forma de desenvolver e manter as descrições de mensagens de boa qualidade, dificultada pelo forte acoplamento que têm com o código base [41][42]. Descrições inapropriadas são problemáticas pois atrasam o processo de análise [44].

Pinjia He [44] discute que, com a recente popularidade das plataformas de partilha de código (como *GitHub*), *software* moderno pode ser constituído por componentes escritos por diversos programadores. A adaptação do estilo de *logging* é dificultada, não só devido a esta variedade, como também pelo facto de estar em constante evolução. Mesmo utilizando *frameworks* de suporte a esta tarefa, os programadores continuam a ter de efetuar as suas decisões. O estudo utilizou um modelo de linguagem *N-Gram* para categorizar três descrições de *logging*:

- Descrição detalhada das ações ou intenções do programa envolvente que, por sua vez, se podem subdividir em: descrições que informam sobre o comportamento de declarações que precedem o *log*, colocadas, normalmente, no final de blocos ou funções; descrição do estado da ação a decorrer para rastrear o progresso; e descrição do procedimento seguinte.

- Descrições causais de determinado erro ou exceção, sendo normalmente utilizadas em blocos *try-Catch* e declarações *if* (para descrever erros relacionados com variáveis).

- Descrições semânticas, normalmente para informar do conteúdo de variáveis ou funcionalidade, uso e parâmetros de determinada função chamada.

Dos modelos gerados foi possível concluir que existe grande repetibilidade nas descrições em *logs*, o que possibilita a implementação de ferramentas autónomas. No entanto, os modelos não podem ser utilizados entre projetos, o que significa que existe uma grande variedade de repetibilidade [44].

De uma forma semelhante à ferramenta *LogTracker*, *Zhongxin Liu* [45] propôs uma rede neuronal com o intuito de derivar regras utilizadas para descrever variáveis em declarações de *logs*. Os argumentos das declarações *log* são extraídos para, então, serem interpretados com as seguintes regras:

- Se o argumento for uma constante, então é ignorado;
- Se o argumento for um identificador, por exemplo *requestId*, é gravado;
- Se o argumento for uma invocação de um método sem argumentos, por exemplo *request.toString()* ou *this.request*, é gravado o primeiro identificador diferente de *this*;
- Se o argumento for uma invocação de um método com argumentos, por exemplo *Integer.toString(position)*, as variáveis são extraídas iterativamente utilizando as restantes regras;
- Se o argumento não cumprir nenhum das regras anteriormente descritas, como por exemplo *("start at" + length)*, todas as regras são aplicadas iterativamente na *Abstract Syntax Tree* (AST) do argumento.

Enquanto ferramentas, como *LogEnhancer*, utilizam o código como *input* para enriquecer declarações *log* já implementadas, esta proposta tenciona completamentar tais soluções e auxiliar os programadores na introdução de novas declarações de *log*. O modelo mostrou ainda uma maior eficácia quando comparado com outros modelos de referência.

2.4 Infraestrutura de *Logs*

A infraestrutura que suporta o processo de análise de *logs* desempenha um papel importante na implementação de um sistema de logging, na medida que pode requerer a agregação e seleção de grandes volumes de dados, o que causa uma forte dependência

do modo de representação da informação. Extrair conteúdo de dados não estruturados para um formato percetível não é trivial. Outras preocupações importantes incluem a capacidade de processar a informação nos *logs* para análise em tempo real/não real, assim como de escalar com o aumento de volume dessa informação. Dito isto, pode destacar-se duas categorias de estudo: *Log parsing* e *Log storage* [22].

2.4.1 *Parsing* de *Logs*

De acordo com *Aharon* [51], existem dois desafios fundamentais para transformar *logs* em informação compatível para uso automatizado, um referindo-se à tradução das mensagens para um dicionário de eventos e o outro à procura de padrões para obter representações concisas dos processos no *logs*. Foram propostas duas soluções para encarar estes desafios, respetivamente: um algoritmo de agrupamento de texto sequencial para criar um dicionário online e o algoritmo *Principle Atom Recognition In Sets* (PARIS) para descobrir padrões de eventos (*atoms*) que tendem a ocorrer frequentemente. Para alguns sistemas distribuídos, a análise da informação tem de ser feita localmente dado que transferência de dados via rede pode provar-se dispendiosa, e existe ainda a dificuldade de criar um dicionário consistente para todos os *logs* a serem gerados em diferentes máquinas. O processamento do algoritmo PARIS está também limitado temporalmente, uma restrição que deve ser ajustada de acordo com o projeto em questão.

Liang [53] também construiu um dicionário de palavras-chaves dedicado a *logs* de erro. Numa fase inicial, a metodologia permite construir um dicionário pela extração de todas as palavras-chave provenientes das descrições, cuja extração é controlada de acordo com diversas regras semânticas, como remoção de pontuação e substituição de padrões. Todas as descrições são, depois, convertidas em vetores binários onde cada elemento do vetor corresponde a uma palavra-chave. Usando estes vetores foi possível correlacionar dois eventos através do *Pearson Correlation Coefficient* (PCC) [3]. Utilizando o método *Adaptive Semantic Filtering* (ASF) foi possível determinar se dois eventos são redundantes não apenas através do contexto semântico, mas também do intervalo temporal em que ocorrem, dado que dois *logs* de erros gerados em um curto espaço de tempo podem estar relacionados independentemente da sua correlação.

Em uma abordagem semelhante, *Makanju* introduziu *Iterative Partitioning Log Mining* (IPLMoP) [52]. Contrariamente a muitas soluções que utilizam o algoritmo *Apriori* [1], esta não depende da frequência de ocorrência dos padrões para derivar resultados. Para além disso, uma solução por uso do *Apriori* é também ineficiente a extrair padrões de maior comprimento. Este novo algoritmo é utilizado para agrupamento de

Figura 2.14: Partição por posição do *token* [52]

informação em *logs* através do particionamento iterativo de um conjunto de exemplares. Em cada passo do processo de partição, as partições resultantes aproximam-se de conter apenas as mensagens relevantes, produzidas pelo mesmo formato. Na primeira iteração, a partição trabalha sob a impressão que as mensagens de *log* que têm o mesmo formato têm também o mesmo número de *tokens*. Por exemplo, de um grupo com a descrição "Conexão de *"', que contém três *tokens*, é possível derivar que todas as suas instâncias, "Conexão de 1.1.1.1"ou "Conexão de 0.0.0.0", têm o mesmo número de *tokens*. Tendo os *tokens*, a próxima iteração procura a posição que contém menor variedade de valores para serem divididos, como se verifica na Figura 2.14.

Na iteração final, a partição é efetuada por uma procura de relações bijectivas (1-1) entre conjuntos de *tokens* únicos, em duas posições selecionadas. Se a relação se verificar, está implícito que existe uma relação forte entre dois elementos e os *logs* a que correspondem estes elementos são filtrados para uma nova partição. Ao observar o exemplo prático na Figura 2.15, nada se pode aferir da relação entre a terceira e quarta posição, dado que 3552 mapeia 3552 e 3534, e 3534 mapeia 3552 e 3311. Este tipo de relações M-M podem então ser ignoradas ou separadas em relações 1-M no caso de resultarem de uma partição do primeiro ou segundo passo. Para as restantes relações, através da

```
Fan speeds 3552 3552 3391 4245 3515 3497
Fan speeds 3552 3534 3375 4787 3515 3479
Fan speeds 3552 3534 3375 6250 3515 3479
Fan speeds 3552 3534 3375 **** 3515 3479
Fan speeds 3311 3534 3375 4017 3515 3479
```

Figura 2.15: Relação M-M [52]

proporção entre valores únicos nos conjuntos e o número de linhas que possuem esses valores nas posições correspondentes, é possível interpretar M como um valor constante ou variável (de acordo com limites de aceitação pré-definidos). Uma vez tendo o particionamento concluído, idealmente todas as partições representam um grupo, por outras palavras, cada mensagem de *log* foi produzida utilizando o mesmo formato. O formato da linha consiste em uma linha de texto constituída por valores constantes e valores variáveis, representados por um "*".

Devido à introdução de alterações ao *software* ou a reconfigurações, é normal a ocorrência de novos eventos em *logs* durante o percurso de vida de um sistema. Isto introduz uma dificuldade aos algoritmos para aprender padrões e modelos, pois todas as técnicas de análise têm de ser capazes de detetar este tipo de alterações. Muitos dos algoritmos de mineração referidos e presentes no mercado encontram dificuldades em atualizar dinamicamente grupos de eventos. Neste sentindo, *Gainaru* apresentou o *Hierarchical Event Log Organizer* (HELO) [52], capaz de explorar, de uma forma precisa, padrões de tipos de eventos em *logs* gerados por super-computadores. Esta ferramenta tem duas vertentes, uma *online* e outra *offline*, sendo que esta última se assemelha à solução implementada no artigo. Por outro lado, a vertente *online* oferece a vantagem de remodelar dinamicamente grupos previamente encontrados com novos eventos classificados [54]. A ferramenta mostrou superar outras soluções como *Apriori* e IPLMoP, sem comprometer custos computacionais

Tendo em conta que os sistemas IoT são altamente limitados em recursos, *Hamooni* indica que, para um *parsing* otimizado e automatizado das mensagens de *log*, a solução necessita da capacidade de reconhecer padrões e relacionar esses padrões com eventos [55]. Desta forma foi desenvolvida a ferramenta *LogMiner* que se destaca nas seguintes características associadas ao reconhecimento de padrões:

- Sem supervisão, não deverá começar de um estado desconhecido, sem necessidade de supervisão humana;
- Heterogeneidade, diferentes mensagens de *logs* são geradas de diferentes sistemas, pelo que o mecanismo de reconhecimento deverá descobrir todos os formatos possíveis independentemente da sua origem;

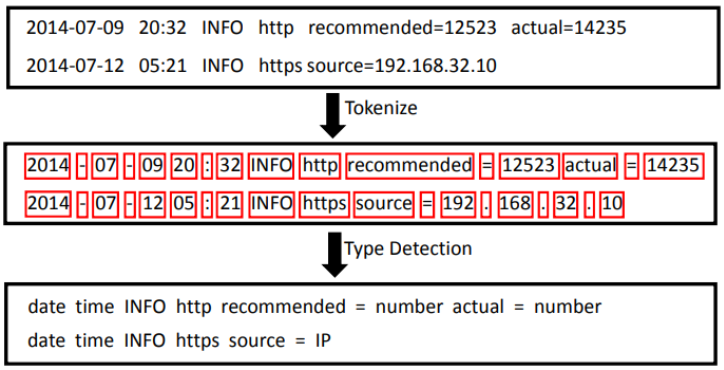


Figura 2.16: Separação em *tokens* e tipos [55]

- Eficiência, o ritmo de processamento tem de ser sempre maior que o ritmo de criação de mensagens de *log*;
- Escalabilidade, deverá ser capaz de processar grandes quantidades de *logs* sem o risco de contenção no processador e na memória.

A ferramenta proposta partilha algumas semelhanças com soluções de agrupamento de eventos já discutidos anteriormente, no entanto, com alguma adaptabilidade para sistemas limitados. Cada mensagem de *log* é apenas processada uma única vez na fase de pré processamento, que se identifica pela separação de *tokens* e distinção destes em tipos (Figura 2.16).

Uma vez processado, o evento é inserido em um grupo de acordo com o nível de semelhança que tem com o representante do grupo, que é selecionado por ser a primeira ocorrência no grupo. Caso não se verifique semelhança com nenhum dos representantes, é criado um novo grupo, no qual este novo evento será o representante. Neste mecanismo, o representante é o único elemento persistido o que permite um elevado ritmo de processamento de candidatos sem alocar memória. Os autores implementaram também uma medida de *MapReduce* para paralelizar o processamento que, por sua vez, ultrapassa resultados obtidos por medidas tradicionais sequenciais. A Figura 2.17 demonstra exatamente as diferenças avaliadas nestes dois tipos de abordagens.

Pin Zhou propôs a ideia de comparar *logs* inteiros em vez de destilar o problema (referenciado no *log*) em *tokens* [56]. Esta ideia é apropriada através do desenvolvimento do sistema *Gestalt Analysis of Unstructured Logs* (GAUL), capaz de oferecer um diagnóstico refinado o suficiente para derivar uma conclusão direta por comparação do histórico do projeto alvo, sem qualquer dependência de formato. Durante a análise, a ordem das mensagens é também ignorada, evitando cálculos complexos para determinar dependências e correlações temporais. O sistema divide-se em três fases:

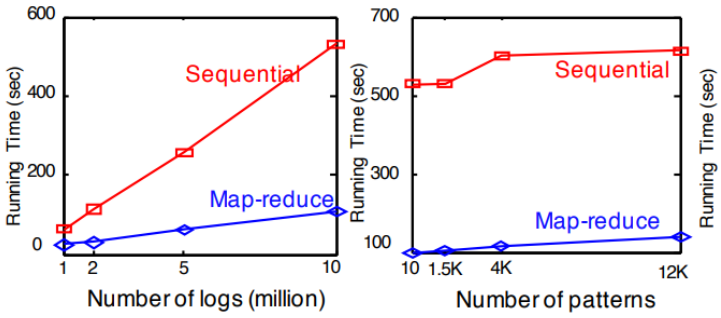


Figura 2.17: Abordagem *MapReduce* versus Sequencial [55]

- *Build Index*, através do motor *Lucene* cada *log* de problemas previamente diagnosticados é indexado como um documento associado a um identificador de problema e de grupo;
- *Update Index*, os *logs* de diagnósticos recentes, e respetivos identificadores, são continuamente alimentados no histórico enquanto que problemas antigos e irrelevantes são removidos via o identificador de grupo;
- *Search Index*, o sistema compara os *logs* de problemas recentemente diagnosticados com prévios através do *Lucene* via *queries*. Se equivalências em demasia forem verificadas, o problema é descartado como ruído. Quanto menor o número de equivalência, maior é a sua afinidade com determinado identificador de grupo.

Para uma maior eficiência, diversas técnicas são aplicadas durante o processo descrito [56]. Para se obter semelhança, é calculado o somatório de todas as semelhanças entre um novo problema e um grupo. Porém, este processo pode ser mais eficiente se apenas forem considerados os melhores elementos do grupo. Outras técnicas incluem a filtragem de elementos com um acumulado de semelhanças baixo e tolerância a ruído (através da remoção de informações temporal e sequencial, de problemas duplicados, entre outros.

Numa abordagem mais distante, *Liang Tang* menciona que informação ao nível das palavras (uma característica bem presente nas soluções anteriores) não é suficiente para derivar conclusões aceitáveis numa dinâmica de grupos [57]. *Logs* constituem mensagens de tamanho relativamente pequeno mas de vocabulário variado. Consequentemente, é possível que mensagens do mesmo evento apresentem palavras totalmente diferentes. Face a esta limitação, o autor propõe a framework *LogTree* que utiliza o formato e estrutura dos *logs* brutos para gerar grupos de eventos. Esta *framework* utiliza um analisador de linguagens para obter uma representação hierárquica, em árvore, da

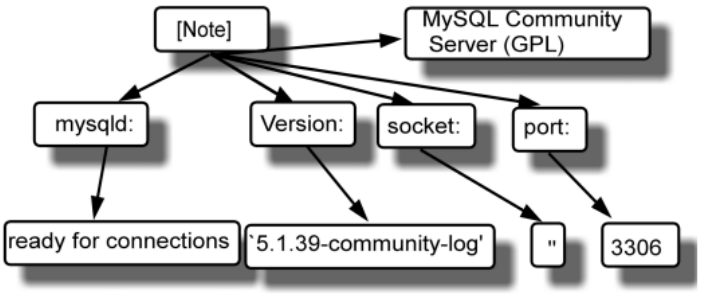


Figura 2.18: Exemplificação da hierárquia de uma mensagem [57]

estrutura das mensagens (Figura 2.18). A semelhança entre duas mensagens é aferida pelo número de nós partilhados entre duas mensagens, sendo que nós de alto nível são mais relevante na discriminação da mensagem. Outro método para esta aplicação é via a análise de segmentos do *log*, símbolo como ":"ou "["são importante para identificar o *template* da mensagem.

Noutro estudo empírico [59], *He Pinjia* concluiu que todas as técnicas de *parsing* abrangidas pelo estudo atingem grande precisão, porém, podem ser melhoradas com algum pré processamento das mensagens. Por outro lado, metodologias baseadas em grupos encontram dificuldades ao escalar com altos tráfegos de informação, exigindo paralelismo. São também alvo de críticas devido à necessidade de afinar parâmetros de comparação [58]. Num estudo subsequente introduziu a solução *Parallel Log Parsing* (POP), que aplica as características introduzidas por IPLMoP em *Spark*, uma plataforma de processamento de dados em larga escala, com a capacidade de paralelizar a solução.

2.4.2 Amazenamento de *Logs*

Quanto mais complexo é um sistema informático, mais exigente será a análise de *logs*. O esforço dedicado a colecionar, guardar e indexar uma grande quantidade de *logs*, muitas vezes tratado no paradigma de *Big Data*, é agravado quando estes são heterogéneos. O *Facebook* tratou cerca de 130 *TeraBytes* diários em 2010 e guardou 300 *PetaBytes* em 2014. Devido ao tamanho destes grupos de dados, considera-se mais apropriado utilizar bases de dados virtuais com sistemas de processamento distribuídos e paralelos em vez de bases de dados convencionais. *Cloud* é um exemplo desta solução, utilizado por muitas das grandes empresas. *Mavridis* investigou o desempenho de duas das soluções *Cloud* mais utilizadas *Spark* [86] e *Hadoop* [20] (Figura 2.19. A maior diferença encontrada entre estas duas plataformas é a forma como o processamento é executado em *Hadoop* a informação é processada no disco, enquanto que em *Spark* é

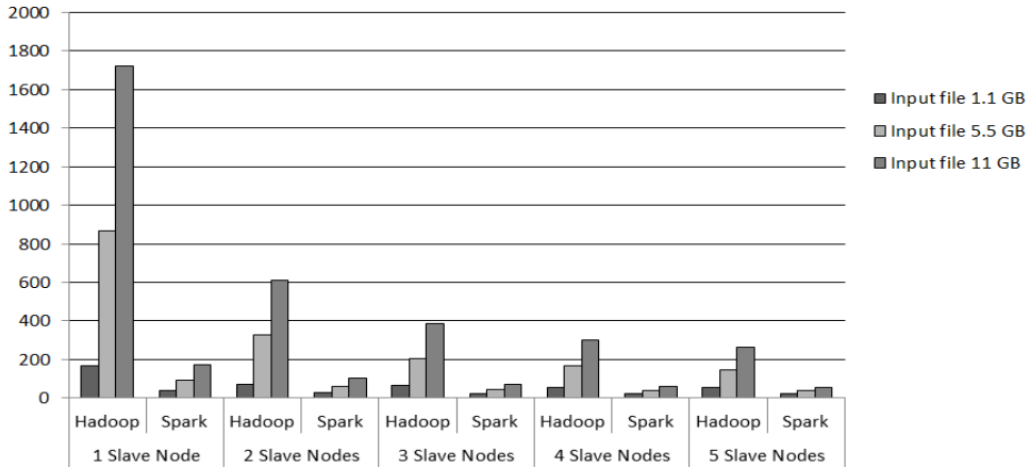


Figura 2.19: *Spark versus Hadoop* [74]

utilizada a *Random Access Memory* (RAM). Esta decisão possibilita tempos de resposta muito menores dado que ocorrem menos ações de escrita e leitura em disco, especialmente para aplicações que efetuam consultas iterativas no mesmo grupo de dados e, embora este tipo de recurso seja volátil, a memória em disco pode ser utilizada como reserva em caso de eventuais falhas.

Spark também oferece diversos módulos de processamento e aprendizagem automática, características apenas encontradas em *Hadoop* via plataformas externas. Dada a complexidade adicional de *Spark* e aos custos extras de alocar informação na RAM, *Spark* deverá ser utilizado em casos de processamento de *stream* de dados e de grande exigência de pedidos para se tornar rentável. Por outro lado, *Hadoop* pode muito bem ser utilizado para operações de informação estática e assíncronas [74].

Para ambas estas plataformas, e muitas outras, a informação é comprimida antes de ser armazenada, uma metodologia que ajuda a reduzir requisitos de armazenamento. No entanto, em muitos dos métodos utilizados, a informação é comprimida em blocos, o que impossibilita que *logs* independentes sejam descomprimidos, característica que se torna ineficiente e afeta a disponibilidade de recursos no disco e no processador do sistema. Em [75], *Lin Hao* menciona que, com a habilidade de descomprimir um *log* de uma forma independente, é possível reduzir tempo na recolha da informação em armazenamento secundário e na descompressão de informação. Para aplicar este tipo de solução é necessário ultrapassar dois desafios. O primeiro desafio deve-se ao funcionamento sequencial das técnicas de compressão tradicionais aproveitando redundância em pequenas distâncias de dados para compressão a um nível individual, estas redundâncias dificilmente podem ser exploradas. Outro desafio resulta da falta de redundância tipicamente ausente em entradas de *logs* individuais. Cowic propõe, um método de compressão de *logs* bem estruturados para compressão e descompressão

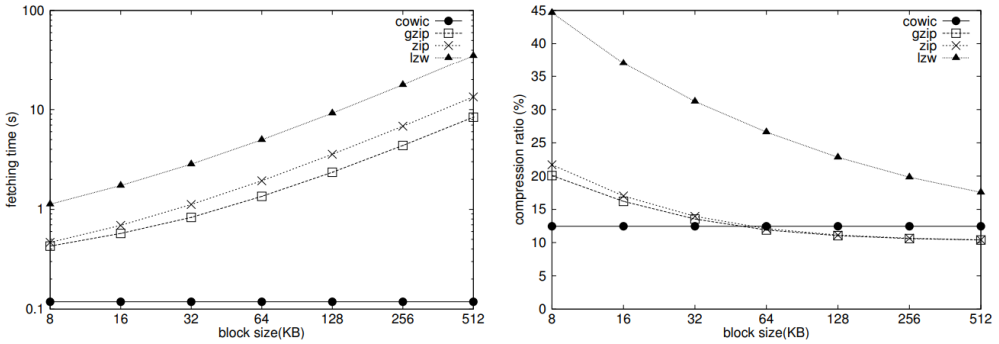


Figura 2.20: Comparação entre *Cowic* e outras técnicas de compressão [75]

individual das mensagens [75]. É explorada uma característica já referida em secções anteriores, que aponta ao facto das mensagens puderem ser subdivididas que resulta em colunas com informações altamente redundantes, dado que existem propriedades partilhadas entre mensagens. Numa fase inicial, uma pequena porção de *logs* é utilizada como *seed* para o treino do modelo de compressão para palavras novas que não ocorrem no modelo, sendo construída uma lista de palavras auxiliar. A Figura 2.20 apresenta uma breve comparação entre *Cowic* e outras técnicas de compressão, avaliando o ritmo de recolhe/compressão em função do tamanho do bloco de *logs* [75].

Mais recentemente, *Liu Jinyang* introduziu uma abordagem diferente com *Logzip* que, através de estratégias já adotadas em estudos anteriores, incluindo agrupamento e mapeamento de *logs* com base no contexto das mensagens, é capaz de detetar informação redundante [76]. Esta informação é, porém, removida e a representação resultante é comprimida em vez da representação original dos *logs*. Os resultados obtidos demonstraram um ritmo de compressão mais elevado comparativamente a outros métodos utilizados, incluindo *Cowic*, bem como uma redução de custos de armazenamento através de uma maior compressão e de custos no consumo de rede durante replicação de instâncias.

2.5 Análise de *Logs*

Após o processamento dos *logs*, a informação resultante é utilizada via técnicas e métodos sofisticados de análise de *logs* para auxiliar a detetar comportamentos inesperados, constrangimentos no desempenho e falhas de segurança. Esta análise de *logs* é aquisição de conhecimento para um determinado propósito, tal que é dificultada pela complexidade dos sistemas que geram a informação [22].

Nesta etapa são abordados os principais contextos onde estas análises se inserem, destacando deteção de anomalias, segurança, origem de causas e previsão de falhas.

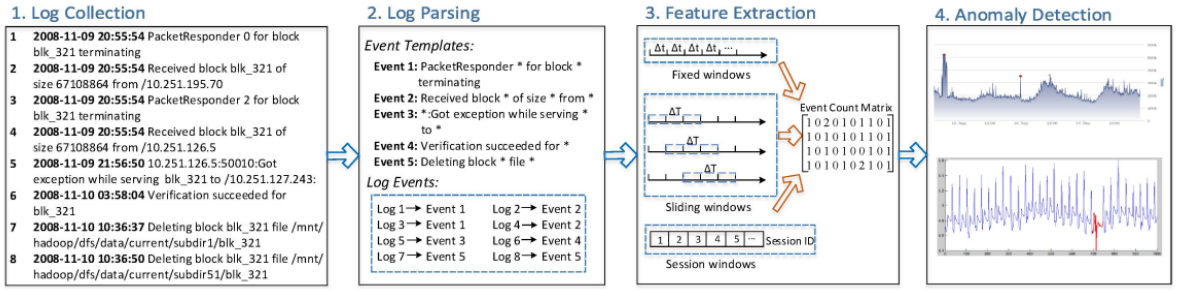


Figura 2.21: Processo de Deteção [35]

2.5.1 Detecção de Anomalias

A análise manual de informação constitui uma tarefa extensa, propícia a erros e, muitas vezes inviável. Para ultrapassar estes problemas, técnicas de deteção de anomalias são habitualmente estabelecidas para descobrir padrões indesejáveis em informação de *logs*. De uma forma geral, o processo de deteção divide-se nas quatro fases apresentadas na Figura 2.21: obtenção dos *logs*, *parsing*, extração de características e eventual deteção de eventos anómalos [22].

Oliner [25] define estes padrões indesejados como alertas, por outras palavras, mensagens nos *logs* do sistema que merecem atenção do administrador do sistema. Este mérito pode dever-se à necessidade de ações imediatas ou por indicação de um problema subjacente. Diversos alertas podem corresponder a sintomas da mesma falha e a falha pode ser devida a qualquer tipo de comportamento, desde um sistema de ficheiro defeituoso a uma conexão transiente que termina um processo. De cinco supercomputadores foi retirada uma coleção de *logs* para ajudar a compreender os desafios na deteção de alertas, dos quais se destacaram:

- Contexto insuficiente, muitas das mensagens são ambíguas sem o contexto externo necessário. O tipo de informação ausente mais indicativa refere-se ao contexto operacional, que ajuda a considerar o elemento humano e os fatores externos que influenciam a semântica da mensagem. Um exemplo de contexto operacional encontra-se representado na Figura 2.22.

- Reporte assimétrico, algumas falhas não se encontram registadas nos *logs* que têm um excesso de informação irrelevante. Cada tipo de falha pode ainda produzir diversas assinaturas de alertas nos *logs*.

- Evolução do sistema, durante o curso de vida de um sistema, qualquer alteração, desde atualizações de *software* a ligeiros ajustes de configuração, pode alterar drasticamente o significado ou carácter dos *logs* que, por sua vez, resulta em

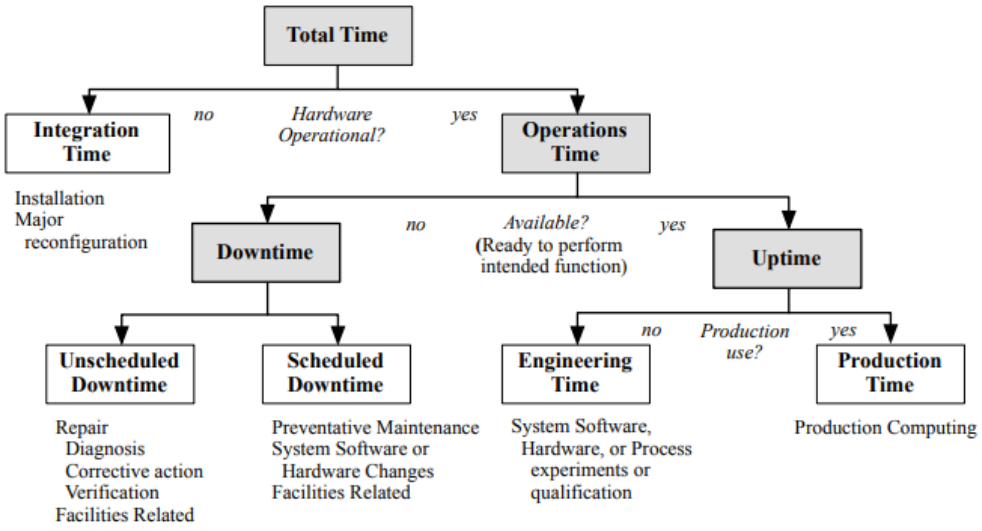


Figura 2.22: Exemplo do contexto operacional [25]

variações de padrões e comportamentos aprendidos. A capacidade de identificar quando estas alterações ocorrem pode desencadear algoritmos de reaprendizagem.

- Correlação Implícita, alguns grupos de mensagens são fundamentalmente relacionados apesar de não existir nenhum indicador explícito para tal.
- Estrutura Inconsistente, os *logs* são geralmente não estruturados e a respetiva semântica e formato podem variar entre sistemas, o que torna a deteção complicada, especialmente quando é realizada *online* e em situações de grande quantidade de informação [28]. Este problema é resolvido utilizando um *parser*, como discutido na secção anterior.
- Mensagens corruptas, foram verificadas mensaens truncadas, pacialmente substítuidas e com *timestamps* incorretos.

Existem diversas soluções para deteção de anomalias, porém, não existe uma comparação documentada relativa às ferramentas disponibilizadas, tornando-se importante decidir a solução a adotar para o contexto de cada situação e não comprometer o esforço envolvido pelos programadores (alterar uma *pipeline* dedicada a deteção de anomalias não é trivial). *Min Du* reconhece esta dificuldade e propõe uma análise detalhada e uma avaliação das diferentes opções disponíveis [35]. De entre as soluções de deteção de anomalias surgem processos supervisionados e não supervisionados. Um processo supervisionado é definido por uma tarefa de *machine learning* que transforma informação de treino classificada num modelo. Esta classificação indica se determinado evento

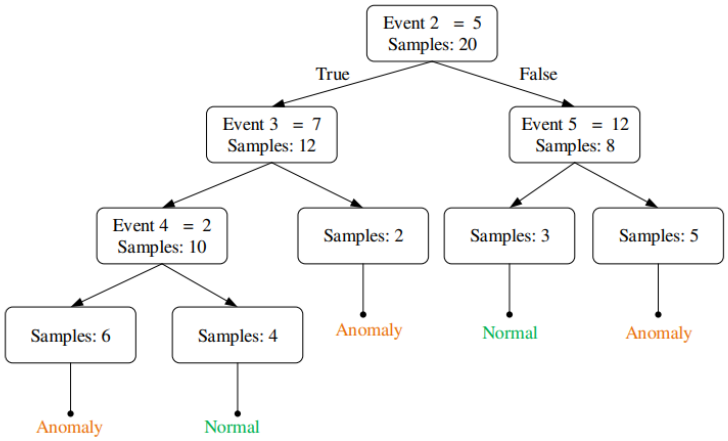


Figura 2.23: Árvore de Decisão [35]

é anormal ou normal, sendo que quanto melhor classificada a informação mais preciso será o modelo resultante. De processos supervisionados destacaram-se as seguintes metodologias:

- Regressão Logística, é um modelo estatístico geralmente utilizado para classificação em que se estima a probabilidade p para todos os estados possíveis (normal ou anormal) de determinado evento via uma função logística construída da informação classificada.

- Árvore de Decisão, consiste num diagrama estruturado em árvore que utiliza ramos para demonstrar o estado previsto de cada amostra. A árvore de decisões é construída de cima a baixo utilizando informação de treino, em que cada *node* da árvore é criado com base no melhor atributo. Por sua vez, é esse atributo escolhido com base na quantidade de informação que o atributo expõe sobre determinada classe de informação, também denominado de ganho de informação, e obtém-se pela diferença entre a entropia do *node* com a média das entropias dos *nodes* filhos. A entropia corresponde a uma medida de incerteza num grupo de observações. Na Figura 2.23, o *node* primário é separado com base no valor do atributo "Event 2"que, por sua vez, é tratado como o melhor atributo. As amostras introduzidas são posteriormente separadas de acordo com o valor definido em cada *node* até à terminação de um ramo, resultando num estado anómalo ou normal.

- *Support Vector Machine* (SVM), consiste na construção de um hiperplano com o intuito de separar classes de amostras num espaço de n-dimensões (Figura 2.24).

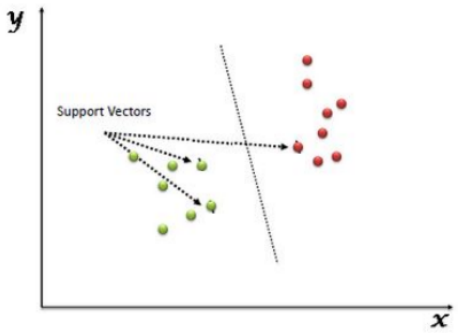


Figura 2.24: *Support Vector Machine*

A construção do hiperplano trata-se de um problema de otimização que maximiza a distância entre o hiperplano e os pontos de dados mais próximos de cada classe (*Support Vector*).

Em processos não supervisionados, a informação de treino não se encontra classificada o que resulta em decisões menos enviesadas, tornando esta opção preferível em ambientes reais de produção. Algumas estratégias de processos não supervisionados incluem [35]:

- Agrupamento de *Logs* consiste em agrupar *logs* com um determinada parâmetro de semelhança. *Qingwei Lin* [26] projetou um exemplo desta aplicação através de um método de agrupamento para identificar problemas em sistemas online denominado LogCluster. A solução requer duas fases de aprendizagem: uma fase de inicialização de informação base e uma fase de aprendizagem online. Na primeira fase, todos os eventos de *logs* são vetorizados e, a cada vetor, é atribuído um peso com base em diversos métodos de atribuição de pesos (tal como *Inverse Document Frequency* (IDF)); posteriormente os *logs* são agrupados com base nas suas semelhanças e em cada grupo resultante um representante é selecionado. A fase seguinte é utilizada para ajustar o agrupamento resultante da fase anterior onde serão introduzidos *logs* iterativamente e comparados com os representantes de cada grupo. Caso a diferença entre o *log* introduzido e determinado representante se verificar superior ao limite de aceitação definido, então é adicionado ao grupo, no entanto, caso não se verifique um grupo compatível, um novo será criado. Após a fase de aprendizagem online o sistema está calibrado para ser executado e um *log* será considerado anómalo caso a distância mínima dos representantes seja inferior a determinado limite.

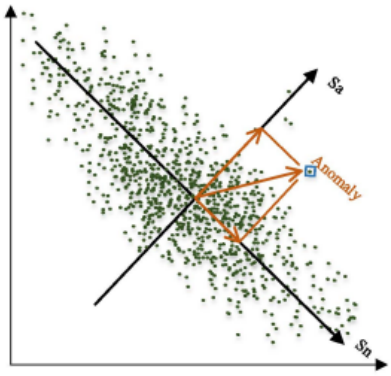


Figura 2.25: *Principal Component Analysis* [35]

- *Principal Component Analysis* (PCA), corresponde a um método estatístico geralmente utilizado para efetuar redução de dimensões. A ideia principal deste mecanismo trata de reduzir a informação para um novo sistema de coordenadas constituído por conjunto de componentes principais, *k*. PCA calcula *k* pela procura de componentes onde se verifica uma maior variedade entre a informação pré processada, desta forma garantindo que a informação reduzida mantém as características principais. *Wei Xu* [27] demonstrou uma primeira abordagem de deteção de anomalias através de PCA. Uma vez efetuado o *parsing* dos *logs* e a extração de características, padrões dominantes são captados na informação resultante para construir subespaços normais (*Sn*). Como se verifica na Figura 2.25, quanto menor correlação existir entre determinado vetor (ou *log*) e o eixo *Sn* maior a probabilidade de se apresentar como uma anomalia, sendo *Sa* os subespaços anómalos.

- Mineração de Invariantes, um exemplo de um invariante demonstra-se em forma de uma relação linear que se mantém durante a execução do sistema sob diversos *inputs* e carga de trabalho. Em linguagem C, por exemplo, um *log* que descreve o *open* de um ficheiro terá de ser seguido de uma descrição do *close* desse mesmo ficheiro. Estes dois eventos idealmente são identificados em pares e a inexistência de uma das partes pode indicar um invariante e, portante, uma anomalia. Para a mineração de tais invariantes, um espaço invariante é primeiro estimado via *Singular value decomposition* que por sua vez determina o número de invariante *r* para posterior mineração no passo seguinte. De seguida é utilizada um algoritmo de *Brute-force search* para delinear os invariantes. Cada sequência de *logs* pode agora ser comparada com o conjunto de invariantes resultando na deteção de uma anomalia caso algum destes não seja respeitado.

Min Du [35] efetua uma comparação de todas estas metodologias. Das hipóteses supervisionadas, a árvore de decisão é a mais intuitiva das soluções mencionadas, sendo possível derivar explicações significativas pela observação do *node* onde ocorreu a anomalia. As regressões logísticas observaram-se incapazes de resolver determinados problemas lineares que, por outro lado, são resolvidos pelo SVM. Os parâmetros do SVM são, no entanto, difíceis de afinar e frequentemente esse processo requer intervenção manual para estabelecer um bom modelo. As soluções não supervisionadas observaram-se com uma maior praticabilidade e relevância devido à inexistência de classificação da informação. A aprendizagem online no mecanismo de agrupamento de *logs* torna esta opção adequada para processamento de um elevado volume de dados. A custo de um tempo de processamento mais demorado, a mineração de invariantes deteta anomalias com uma elevada precisão e proporciona um contexto significativo na interpretação de cada anomalia. PCA foi categorizada como uma estratégia difícil de entender e sensível à informação de treino (o que significa que a precisão na deteção de anomalias depende significativamente da amostra de treino).

A deteção de anomalias necessita de ser imediata e oportuna de modo a intervir rapidamente ataques ou problemas de desempenho. *Min Du* [28] apresenta uma primeira proposta, em *deep learning*, a situações provocadas por determinados ataques designada por *DeepLog* e que é uma abordagem de processamento de linguagem natural. Por outras palavras, interpreta mensagens como elementos de uma sequência e procura por padrões e regras gramaticais. Constitui uma rede neuronal que é treinada via *logs* produzidos pela execução normal do sistema e modela a sequência utilizando *Long Short-Term Memory* (LSTM), mecanismo que permite aprender correlações abrangentes e intrincadas e padrões incorporados na sequência de *logs* produzidos. A abordagem pressupõe que os *logs* são seguros e protegidos e assume que determinada adversidade é incapaz de modificar o código e alterar os conteúdos dos *logs*. Foram considerados dois tipos de ataques, um que leva a uma execução imprópria do sistema, causando padrões anormais, como por exemplo *Denial of Service* (DoS), que pode provocar lentidão na execução (que resulta em anomalias de desempenho detetadas através dos *timestamps*) e ataques cujo efeito se pode encontrar em *logs* de sistema, devido às atividades de *logging* de serviços de monitorização de sistemas, como por exemplo *Intrusion Detection System* (IDS). A ferramenta *LogAnomaly* utiliza também LSTM para detetar anomalias, no entanto, aplica-se para padrões sequenciais e quantitativos em *logs* [29].

Zhang [30] menciona que, embora LSTM mostre resultados promissores, a sua utilização dificilmente cobre *logs* em constante evolução. Esta caraterística é partilhada por métodos supervisionados como classificadores SVM, árvores de decisão ou modelos

de regressão logística. A ferramenta proposta, *LogRobust*, visa resolver esta questão. A extração de características divide-se em uma fase de pré-processamento, onde é extraída a semântica via linguagem natural do evento, uma fase de vetorização do evento e uma fase de classificação baseada em atenção. O vetor semântico resultante tem também de representar eventos diferentes com grande discriminação e identificar eventos instáveis com semânticas semelhantes. De forma a suportar *logs* instáveis, é utilizado uma rede neuronal com *Bi-Direcional Long Short-Term Memory* (Bi-LSTM).

Lu [31] propôs também, em *Spark*, um método de análise estatística para detetar a origem de anormalidades. A extração é baseada na parametrização específica da plataforma *Spark* para distinguir entre tarefas anormais relacionadas com a execução, a memória ou o processador. A anormalidade é detetada quando os limites temporais de execução de determinada tarefa, em determinada fase do processamento, são alcançados. Por sua vez, esta fase indica a inclinação probabilística da ocorrência para determinada origem. Métodos semelhantes utilizam ainda técnicas de agrupamento das mensagens (via a separação de *tokens* por exemplo) para inferir a inclinação probabilística [32].

Numa abordagem de mineração de *logs*, *Bertero* [33] aplicou métodos de *Natural Language Processing* (NLP) ao considerar conjuntos de ficheiros de *logs* como linguagens, contrariamente a soluções prévias que dependiam de processamentos pesados para extrair informação relevante dos *logs*. Em essência, esta perspetiva captura grandes quantidades de *logs* relacionados com sistemas em um estado A e B, e estes, por sua vezes, são transformados em vetor multi dimensionais, via algoritmos NLP, para treino de um classificador. O *pipeline* resultante constitui um modelo preciso para determinado estado do sistema. É utilizado, especificamente, o algoritmo *WORD2VEC*, da *google*, para produzir um mapa do conjunto dos ficheiros de *logs* para um Espaço Euclidiano. A ferramenta *LogAnomaly*, referida anteriormente, utiliza uma alternativa ao *WORD2VEC* denominada *TEMPLATE2VEC*, desenvolvida para capturar, não apenas o contexto dos *logs*, mas também a semântica e sintaxe, incluindo sinónimos e antónimos [29].

Aplicações modernas de larga escala, especialmente em ambiente *cloud*, estão sujeitas a alterações constantes derivadas de intervenções esporádicas, como escalamentos, *upgrades*, migrações ou reconfigurações. Estas alterações tornam o sistema complexo e contribuem para diversas falhas operacionais no desenvolvimento e execução das aplicações. Em ambiente *Cloud* é especialmente importante monitorizar o comportamento das operações de aplicações e inspecionar como o fluxo de determinadas ações afeta os recursos do sistema. *Mostafa Farshchi* [34] apresenta uma abordagem estatística com o intuito de identificar a correlação entre o estado dos recursos alocados na *Cloud* e

comportamentos das operações aplicacionais. A confiabilidade das operações de uma aplicação em *Cloud* é então assegurada pelas mencionadas contribuições. Como caso de estudo, são utilizados *Rolling Upgrades* para determinar esta correlação, dado que é uma operação *Cloud* altamente propícia a interferência com outras operações. Estes *Rolling Upgrades* correspondem à atualização de um grupo de máquina virtuais em ambiente *Cloud* após o lançamento de uma nova versão de determinada aplicação. Este processo é efetuado através de uma imagem construída da nova versão da aplicação.

2.5.2 Segurança

A evolução do mundo informático está diretamente relacionada com o surgimento de ataques e de *malwares* cada vez mais sofisticados. De uma forma geral, *logs* podem ser aproveitados para propósitos de segurança, como deteção de ataques ou de intrusões. Devido à natureza dos ataques cibernéticos, uma grande porção da informação utilizada nas soluções que se seguem é retirada da rede via *logs* de aplicações destinadas a esse tipo de restrição [22].

Barse [64] introduziu uma das primeiras abordagens empíricas a explorar os tipos de *logs* necessários para auxiliar a deteção de intrusões. Esta exploração foi executada através de uma *framework* desenvolvida para extrair manifestações de ataques, uma entrada *log* que é adicionada, alterada ou removida por consequência de um ataque que, por sua vez, é utilizada para alarmar comportamentos maliciosos a sistemas de deteção de intrusões. Neste estudo consideraram-se ataques a *OpenSSH*, *Tcpdump* e *Neptune*. *Cristina Abad* [65] nota que também é importante considerar a correlação de todos os componentes envolvidos e analisar os *logs* como um conjunto, pois, tipicamente, um ataque deixa diversas provas da sua presença.

Advanced Persistent Threat (APT) refere-se a uma classe de ataques que pode colocar organizações e governos em risco através de acessos não detetado nos sistemas, onde o atacante tem a capacidade de roubar informações sensíveis durante um longo período de tempo. Neste sentido, *Oprea* [60] apresenta uma *framework* destinada a detetar, o mais cedo possível, infeções APT e *malwares*. É baseada em *belief propagation*, uma estratégia que permite identificar pequenas comunidades de domínios potencialmente indicativos de infeções, comunidades essas que estão sempre limitadas a domínios raros, ou seja, domínios novos detetados no tráfego da empresa e contactados por um pequeno número de *hosts*. O algoritmo é baseado em uma estratégia de passagem de mensagens iterativa entre nódulos e respetivos vizinhos até convergência ou até uma condição de paragem ocorrer, utilizando *logs* capturados de *Domain Name System* (DNS) ou de *web proxies* para construir um modelo de comunicação entre os *hosts*

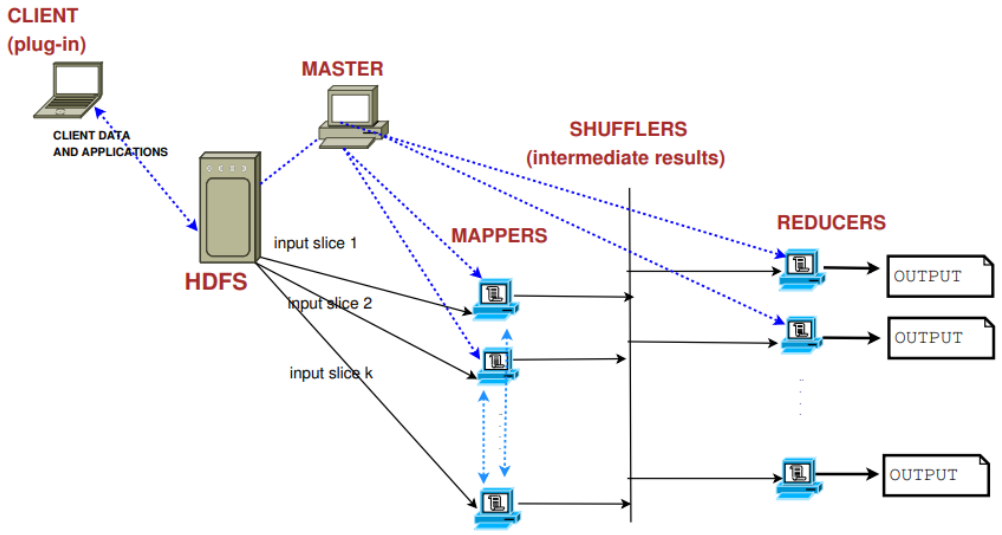


Figura 2.26: Arquitetura *Hadoop* [62]

internos e os domínios externos.

Através da configuração de *Access Control List* (ACL) em todos os *routers*, tráfego não autorizado é bloqueado. De uma forma semelhante, a utilização de tecnologias como *Terminal Access Controller Access-Control System Plus* (TACACS+) e *Remote Authentication Dial In User Service* (RADIUS) garantem que apenas utilizadores autenticados tenham acesso aos *routers*. Dito isto, ACL podem estar mal configuradas e, em equipas de grandes dimensões, um utilizador comprometido pode resultar em diversos problemas de segurança. Para estes casos *Jie Chu* [61] propôs uma camada adicional de defesa com o intuito de supervisionar todas operações efetuadas na rede e, por sua vez, detetar e alarmar atividades suspeitas de uma forma automática viaa análise de *logs* provenientes dos *routers* e/ou *switchs* em tempo real.

De acordo com *Eunjung Yoon* [62] estratégias baseadas em *MapReduce* encontram-se vulneráveis a intervenção por ataques via os participantes dos sistema, quer sejam eles utilizadores ou nódulos distribuídos. Cálculos e comportamentos inesperados podem passar despercebidos devido à falta de controlo sobre funções e processos aplicados em informação dedicada aos utilizadores. Uma abordagem nova foi utilizada para detetar mau uso e ataques em *frameworks* de *MapReduce*, especificamente *Hadoop*, por análise semântica dos *logs* respetivos e de chamadas ao sistema. Em *Hadoop*, esta estratégia é aplicada através de um modelo *master*/*slave*, onde o nódulo *master* tem como objetivo distribuir os diferentes pedidos pelos restantes nódulos, assim como monitorizar o processamento e os resultados de saída. A Figura 2.26 demonstra a arquitetura do *Hadoop*.

Sobre a suposição que os nódulos possuem *hardware* semelhante e o nódulo *master* e o

Hadoop Distributed File System (HDFS) são confiáveis, foi possível estudar os seguintes tipos de eventos [62]:

- Integridade do programa, verificando se a execução das funções de *map*/*reduce* seguem o fluxo correto, determinado por invariantes conhecidas;
- Parâmetros de entrada, verificando se os ficheiros processados das funções *map* originam de localizações/endereços HDFS corretos;
- Resultados de saída, verificando se os ficheiros resultantes das funções *reduce* são mapeados em localizações/endereços HDFS corretos;
- I/O ao nível do *MapReduce*, rastreando e analisando chamadas de sistemas relacionadas com as duas funcionalidades.

Yen [63] menciona que diversos produtos destinados a segurança produzem *logs* com grande diversidade de formatação e maioritariamente incompletos, contraditórios e extensos. O sistema *Beehive*, foi introduzido com a finalidade de extrair informação de *logs* inconsistentes provenientes de tais produtos e detetar atividades suspeitas numa determinada empresa, tal como infeções por *malwares* ou violações de políticas. Este sistema divide-se em três camadas:

- *Parsing*, filtragem e normalização de *logs*, utilizando informação de configurações específicas da rede que possibilita o processamento de determinado conteúdo das mensagens via: *timestamps*; mapas entre endereços IP e *hosts*, obtidos por análise de *logs* de servidores *Dynamic Host Configuration Protocol* (DHCP), IPs estáticos; *hosts* dedicados, máquinas utilizadas por apenas um indivíduo;
- Extração de características, que podem ser de acordo com o destino, o *host*, política ou tráfego. A Tabela 2.6 lista informação extra sobre cada tipo de característica referido;
- Agrupamentos por caraterísticas, para identificar *outliers*.

2.5.3 Análise da Origem de Causas

A deteção de comportamentos anómalos é o primeiro passo para a resolução de problemas num determinado sistema, pelo que é necessário investigar e identificar a origem do comportamento inesperado [22].

Feature Type	#	Description
Destination-Based	1	New destinations
	2	New dests. w/o whitelisted referer
	3	Unpopular raw IP destinations
	4	Fraction of unpopular raw IP dests.
Host-Based	5	New user-agent strings
Policy-Based	6	Blocked domains
	7	Blocked connections
	8	Challenged domains
	9	Challenged connections
	10	Consented domains
	11	Consented connections
Traffic-Based	12	Connection spikes
	13	Domain spikes
	14	Connections bursts
	15	Domain bursts

Tabela 2.6: *Beehive* - Extração de características [63]

Tatsuaki Kimura [66] defende que padrões espácio-temporais podem ser derivados de eventos de uma rede e que a análise de tais padrões fornece informação útil para o diagnóstico de anomalias na rede. Este estudo apresenta duas metodologias para tomar partido destes padrões: *Statistical Template Extraction* (STE), um método de conversão automática de *logs* não estruturados para *templated* primário através de algoritmos de agrupamento estatístico, e *Log Tensor Factorization* (LTF), para desenvolver um modelo estatístico que capta padrões espácio-temporais nas mensagens *log*. O método STE trata de diferenciar cada palavra no evento *log* como um *template* ou um parâmetro através de dois conceitos:

- *Word Scoring*, um mecanismo de pontuação de palavras utilizado para calcular a tendência que a palavra tem para se tornar um *template*. Esta pontuação é determinada pela observação da frequência com que a palavra se encontra em determinada posição na mensagem e do número total de palavras na mensagem.
- *Score Clustering*, uma estratégia de agrupamento para distinguir *templates* de parâmetros com base na pontuação. O estudo utiliza *Density Based Spatial Clustering* (DBSCAN), um algoritmo capaz de gerar grupos de tamanho arbitrário pela observação da densidade de dados fora e dentro do grupo proposto, assim como a deteção de ruído dado que a densidade de pontos é tipicamente mais baixa em áreas de ruído [67].

Para a estratégia LTF, é denotado X como o número de ocorrências de um *log* pela observação de determinado *template*, *host* e *time-window*. X é fatorizado em três componentes [66]:

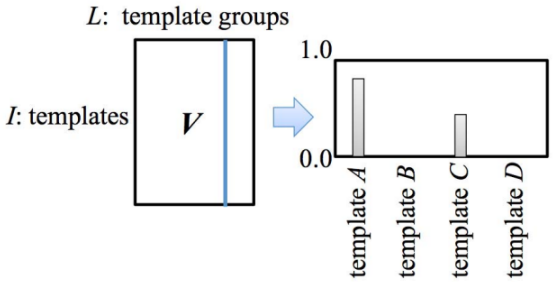


Figura 2.27: Visualização de *V* [66]

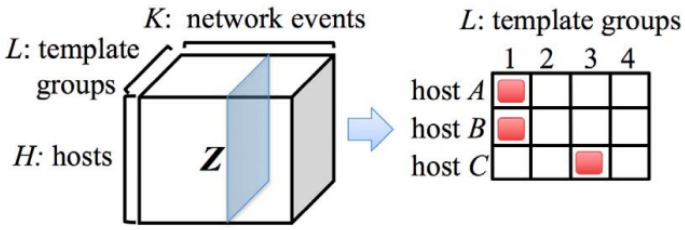


Figura 2.28: Visualização de *Z* [66]

- *V* constitui uma matriz de grupos de *templates* e representa em cada elemento se determinado *template* pertence ou não a determinado grupo de *templates*. A Figura 2.27 demonstra que, para cada vetor *V*, determinado template *I* que pertence ao grupo de *templates L* tem uma relação superior a zero;
- *Z* indica se determinado grupo de *templates L* num *host H* corresponde a determinado evento da rede *K*. Esta relação é demonstrada na Figura 2.28;
- *W* corresponde a uma matriz de peso através da qual é possível entender quando um determinado evento de rede ocorreu.

Em determinado projeto podem ocorrer diferentes processos de conversão de ficheiros de código base em artefactos (em inglês, *builds*) dedicados a cada componente (por exemplo *Automake* e *CMake*) e cada *build* pode ser muitas vezes irreproduzível devido à quantidade limitada de *traces* de execução gerados. Esta inconsistência no comportamento dificulta a análise de origem de falha devido à dificuldade de replicação do caso de estudo. Adicionalmente, tais projetos podem utilizar diversos *scripts* (programas escritos para um sistema de tempo de execução especial que automatiza a execução de tarefas que seriam executadas) para a manutenção das *builds* que, por sua vez, tornam-se difíceis de instrumentalizar para rastrear os comandos executados. *Zhilei Ren* propõe *RepTrace* [68] para reúnir os *traces* das chamadas de sistema dos comandos executados durante as *builds* e efetuar uma análise de causalidade para identificar a origem de falha em determinada *build* irreproduzível. Para tal análise, a ferramenta

constrói um gráfico de dependências de artefactos inconsistentes, baseado em dois tipos de dependências, e procura pelo processo provocativo da falha. Os dois tipos de dependências considerados são: dependência de escrita/leitura, dois processos cumprem tal dependência se um efetuar uma escrita e outro uma leitura; dependência de processo pai/filho, caso determinado processo gere outro processo. A análise começa no artefacto inconsistente e identifica os processos responsáveis pela observação das dependências associadas. Caso existam outros processos com dependência ao processo identificado, e caso esses processos gerem artefactos ou valores de execução inconsistentes, o *RepTrace* prossegue com o rastreamento até a terminação das dependências. O último processo rastreado é considerado então como a causa da falha.

A compreensão e resolução de problemas de sistemas distribuídos em *Cloud* é considerado um problema crítico devido ao paralelismo dos pedidos dos utilizadores. A natureza *multi-tenancy* (modo de operação que se refere a múltiplas instâncias independentes, executadas num ambiente partilhado) em tais ambientes introduz também interferência que, por sua vez, impacta o desempenho. Um estudo por *Aidi Pi* contribuí para a resolução destes desafios com a introdução de *LRTace*, uma ferramenta distribuída para controlo de *feedback* e deteção de falhas [69]. *LRTace* correlaciona os *logs* com métricas de desempenho dos recursos, possível através de virtualização, para posterior caracterização de *frameworks* de análise de dados, como *Spark* e *MapReduce*. Possibilita aos utilizadores reconstruir o fluxo, compreender as *frameworks* e encontrar as anomalias e respetivas origens. Para a caracterização das componentes de uma mensagem de *log* e de padrões entre *logs* distintos, foram propostas *keyed messages*. A *keyed message* corresponde a uma estrutura uniforme de mensagens de *logs*, organizadas em tuplos (valor-chave), e incluíem também informação relativa a métricas de desempenho. O fluxo das aplicações de onde os *logs* se extraíram pode ser reconstruído através de operações como *GROUPBY*, *COUNT* e *SUM*. *LTRace* é constituída por dois componentes: *Tracing Worker* e *Tracing Master*. O primeiro componente encontra-se distribuído em todas as máquinas de processamento e é responsável por colecionar a informação, *logs* e métricas de desempenho, e posterior envio para *Kafka*, uma plataforma de processamento de *streams* [24]. Por outr lado, o *Tracing Master* consome a informação disponível na plataforma, transformada para a estrutura *keyed message* e disponibilizada aos utilizadores.

Edward Chuah apresenta uma outra solução à problemática dos sistemas distribuídos com a introdução da *framework* de diagnóstico, *ANCOR*, baseada no sistema de análise de *logs*, *FDiag* [70]. Contrariamente às duas soluções mencionadas anteriormente, *FDiag* é aplicado à informação gerada por *syslog* e não por análise do desempenho dos sistemas. Adicionalmente, divide-se em duas fases [71]:

- *Space-R*, uma estratégia para identificar padrões em eventos do sistema que ocorrem em todos os *nodes* do sistema distribuído;
- Geração de relatórios para sumarizar os resultados do diagnóstico que fornecem informação relativa aos padrões identificados num determinado período de tempo no sistema distribuído.

ANCOR trata de relacionar anomalias na utilização de recursos com falhas no sistema, através da observação da utilização de determinada recurso, que por sua vez contém estatísticas de I/O, taxas de transferência e utilização de memória virtual a nível dos *nodes* e a nível do processo executado, e de *logs* racionalizados que possuem eventos gerados pelos componentes do sistema distribuído. Através de ambas as observações foi possível desenvolver uma abordagem de duas fases: identificação de anomalias nos recursos do sistema, para um diagnóstico parcial, e análise dos *logs* para um diagnóstico mais preciso [70].

Num outro estudo, *Ziming Zheng* demonstra que a análise de *logs* de *Reliability, Availability and Serviceability* (RAS) pode ser útil na compreensão de falhas em recursos computacionais de alta gama, como *IBM Blue Gene* [72]. Estes *logs*, porém, tipicamente oferecem informação limitada relativa ao sistema e ambiente de operação e a sua utilização torna-se inadequada para um contexto crítico. No entanto, uma análise correlacionada entre *logs* de RAS e *logs* provenientes de processos do sistema torna a abordagem confiável. Esta técnica de análise está dividida em três partes:

- Correlação entre eventos RAS severos do mesmo tipo, que pode ser referido através do código de erro e interrupções em processos, de forma a identificar os eventos que realmente provocam interrupções. Umas das observações concluídas desta identificação aponta ao facto que aproximadamente 20% destes eventos não afetarem os processos.
- Separação entre eventos críticos gerados por *hardware* ou *software* e eventos críticos causados por falhas nas aplicações. Tipicamente, quando um evento interrompe um processo num local específico o utilizador tende a reiniciar o processo, que por sua vez pode ser atribuído a um local diferente pelo *scheduler*. Caso o evento seja causado por um erro aplicacional, o mesmo tipo de evento será reportado na nova localização.
- Remoção de redundância nos registos RAS pelo estudo de correlações entre interrupções dos processos. Esta redundância ocorre quando o *scheduler* atribui *nodes* em falha a processos executados. Por cada processo interrompido, é rastreado o

local de origem e, caso o processo seguinte (do mesmo local) seja interrompido pelo mesmo tipo de evento então o evento é considerado redundante.

2.5.4 Previsão de Falhas

A possibilidade de antecipar falhas em sistemas críticos não representa apenas uma vantagem de valor no mercado, mas também a capacidade de prevenir consequências irrecuperáveis ao negócio. A previsão de falhas é apenas possível após a obtenção de conhecimento suficiente sobre padrões anormais e respetivas causas. A previsão de tais ocorrências obriga a uma monitorização mais pró-ativa e não reativa, como é o caso da deteção de anomalias [22].

Para evitar avarias inesperadas, muitas empresas agendam manutenções regulares aos equipamentos, processo designado por manutenção preventiva. No entanto, este tipo de manutenção não é económica devido aos custos associados à mão-de-obra. *Jianmin Wang* introduziu um estudo em *Automatic Teller Machines* (ATM) com a aplicação de uma alternativa capaz de prever automaticamente quando e que tipo de reparação é necessária, designada por Manutenção preditiva [39]. Esta tarefa coloca diversos desafios: Contextualização do problema para um modelo preditivo; Extração de características e Redundância na informação extraída. O estudo considera este tipo de manutenção como um problema de classificação binária, onde as classes representam a possibilidade de determinado equipamento falhar, ou não, dado um determinado intervalo de tempo. A informação relevante pode ser extraída de diversos contextos:

- Registos de manutenção, pedidos de manutenção em forma de *tickets* que por sua vez possuem data do pedido, as equipas de suporte envolvidas, sintomas reportados, solução aplicada, componentes utilizados;
- *Logs*/mensagens de sistema, estruturados ou não, podem conter identidade do dispositivo, *timestamp*, conteúdo da mensagem, prioridade, código e ação;
- Informação de inventário, dados do fabricante, tipo de máquina, número série, localização e data de instalação;
- Informação de utilização;
- Informação ambiental, temperatura, humidade e níveis de poluição;
- Informação de configuração, a nível de *hardware*, *software* e atualizações efetuados ou por efetuar.

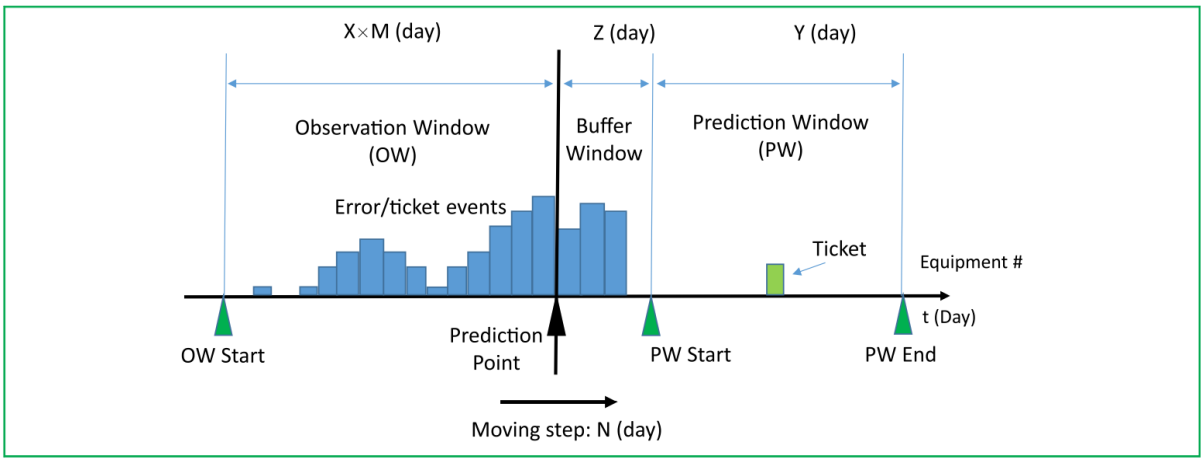


Figura 2.29: Construção de um modelo preditivo [39]

Pela observação de todos estes contextos, os registos de manutenção, *logs*/mensagens de sistema e a informação relativa ao inventário demonstrou a maior utilidade, sendo possível observar dois tipos de eventos de falha: críticos e não críticos. Para a construção do modelo são consideradas várias instâncias de aprendizagem, constituídas por características e um valor (positivo ou negativo), em que cada instância é construída a partir do ponto em que o modelo efetua uma previsão (*Prediction Point*). Antes deste ponto é considerado um período de observação dividido em X sessões com uma lista de registos de manutenção e eventos de falha. Uma sessão tem uma duração de M dias e traduz-se numa unidade de medida para extração de características. O período de previsão, com duração Y dias, é colocado após o ponto de previsão. Após o ponto de previsão, a entidade responsável terá de verificar a ocorrência e, caso um registo de manutenção seja criado, então um valor positivo será atribuído à instância. Este processo é ilustrado na Figura 2.29.

O período entre o ponto de previsão e a resposta à ocorrência, por parte da entidade, denomina-se período bónus. Para controlar o valor mínimo deste período é possível adicionar um período de *buffering* para facilitar tempo suficiente na verificação e criação do evento de manutenção. Por ajuste destes períodos e do ponto de previsão são geradas todas as instâncias de treino e teste para a construção do modelo. A metodologia global é constituída por sete passos: Pré-processamento; Mineração de padrões; Geração de instâncias; Seleção de caraterísticas; Treino do modelo; Teste do modelo; e Avaliação do modelo.

Durante a construção das instâncias foram extraídos diferentes tipos de caraterísticas. Este processo foi realizado e comparado para quatro tipos de algoritmos: *InfoGain*, *GainRatio*, *ReliefF* e *SymmetricalUncert*, dos quais *InfoGain* mostrou uma maior consistência na extração da informação relevante no menor intervalo de tempo. Por outro

lado, foram utilizados quatro algoritmos de classificação para avaliação do modelo sendo estes *XGBoost*, *Random Forest*, *Ada Boost M1* e *LibSVM*, do qual *XGBoost* mostrou melhores resultados [39].

2.6 Plataformas de *Logging*

Nesta secção são analisadas várias soluções disponíveis no mercado que, através de diversos mecanismos, demonstram exemplos de boas práticas referentes aos tópicos abordados e estudados em secções anteriores. Algumas destas soluções, como *Dynatrace* e *Trace Compass*, possuem a capacidade de respeitar as regras de implementação, infraestrutura e análise de *logs*.

Dynatrace é uma plataforma de monitorização que simplifica a complexidade da *Cloud* em empresas e acelera a transformação para o digital [9]. A plataforma oferece informação relativa ao desempenho das aplicações geridas, assim como da infraestrutura subjacente e dos consumidores via um motor dedicado, *Dynatrace AI causation engine* (Davis). Relativamente a mecanismos de *logging*, a plataforma possui as seguintes características [9]:

- Coleciona automaticamente *logs* e eventos de um grupo alargado de tecnologias nativas no sistema como *Java*, *.NET*, *Node*, PHP, C/C++, *Python*, entre outras tecnologias;
- Oferece compatibilidade com alguns ambientes de *Cloud* como *Amazon Web Services* (AWS), *Google Cloud Plataform* (GCP) e *Microsoft Azure*;
- Permite atribuir severidade, estado ou nível aos *logs* de acordo com condições pré-definidas;
- Oferece uma estratégia central e flexível para extrair informação dos *logs*, sendo que as regras definidas para esta extração podem ser inteiramente personalizadas;
- Possuí uma linguagem própria para descrever padrões utilizando *matchers* para comparação de determinados tipos de dados (por exemplo *INTEGER* ou *INT* correspondem a números do tipo *integer*), podendo esta linguagem ser também utilizada para *parsing* do padrão em diversos contextos para uma melhor compreensão, análise ou processamento;

- Visualização simplificada de *logs* via tabelas com a capacidade de filtrar e procurar com diversos campos à descrição do utilizador;
- Uma vez criados os eventos dos *logs*, estes são automaticamente correlacionados a problemas presentes no ambiente do utilizador via os algoritmos de inteligência artificial nativos à plataforma;
- Criação de métrica baseadas em dados de *logs*.

Trace Compass [89] é uma ferramenta utilizada para visualizar *logs* através de métricas e outras estratégias de observação de dados para a posterior analise dos mesmos. É direcionada a projetos *Java*, desenvolvida em *Eclipse* e utiliza a técnica *Tracing* que, de uma forma semelhante a *logging*, consiste em observar e guardar eventos que ocorrem em locais de execução específicos de determinado sistema. A ferramenta oferece suporte para diversas estratégias de correlacionar, filtrar e *parsing* da informação, incluindo algumas características que possibilitam importar/exportar ficheiros e atribuir favoritos. A abordagem de visualização permite que sejam utilizadas perspetivas de organização do projeto, uma extensão da perspetiva padrão do *Eclipse*, de eventos, uma visualização versátil que apresenta eventos em um formato tabular com suporte para procura, filtragem e atribuição de favoritos, estatística de acordo com o tipo de eventos e em histograma para exibir a densidade de eventos em relação ao tempo. Umas das principais funcionalidades implementadas no *Trace Compass* é o *Linux Trace Toolkit, next generation* (LTTng), que permite rastrear, de uma forma altamente eficiente, problemas de desempenho em aplicações e em *kernel* em ambiente *Linux*, assim como resolução de problemas quando envolvem processos concorrentes.

Elastic Stack [10] é uma outra solução, constituída por diversas plataformas, para extração de informação de qualquer origem, em qualquer formato, de uma forma segura e confiável para posterior análise e visualização do conteúdo resultante. A Figura 2.30 sumariza a infraestrutura utilizada por esta perspetiva.

ElasticSearch [10], uma componente central da *Elastic Stack* baseada no motor *Lucene*, constitui um motor distribuído para procura e análise de todo o tipo de informação, incluindo textual, numérica, geoespacial, estruturada e não estruturada, provenientes de diversos tipos de fluxos de informação, como *logs*, métricas de sistemas e aplicações *web*. É mais conhecido pela sua escalabilidade, rapidez, natureza distribuída, simplificação de serviços *REST* e compatibilidade com diversas linguagens de programação e com plataformas *Cloud*, como *Hadoop* [20] e *Spark* [86]. A informação tratada sofre um processo de ingestão, mecanismo pelo qual os dados brutos são processados, normalizados e enriquecidos antes de serem indexados pelo motor *ElasticSearch*. Uma vez

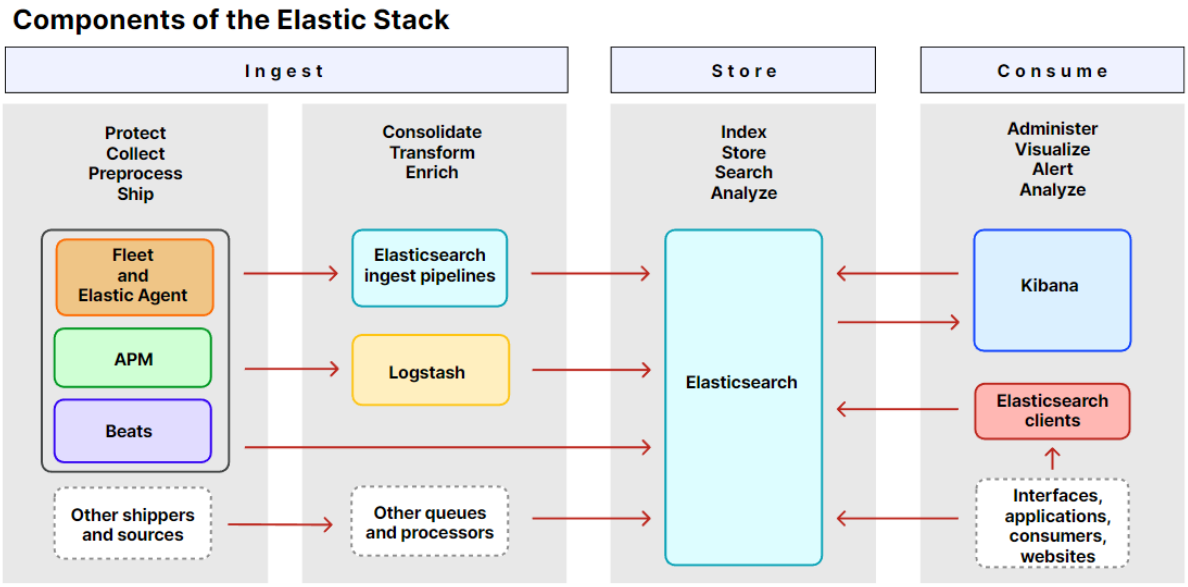


Figura 2.30: Infraestrutura *Elastic Stack* [10]

indexados, é possibilitada a execução de *queries* complexas sobre o conteúdo e utilização de agregações para adquirir simplificações do conteúdo. Um índex é constituído por uma coleção de documentos JSON relacionados entre si. *Kibana* [10], por sua vez, é uma interface de visualização de dados indexados proveniente do motor *ElasticSearch*. Os utilizadores podem observar a informação via diversas formas de visualização, com perspetivas temporais e/ou geográficas. Esta interface possuí também da capacidade de detetar anomalias nos dados e explorar propriedades que as poderão contribuir para tais eventos.

Embora algumas das ferramentas do motor *ElasticSearch* sejam gratuitas e abertas a todos os tipos de utilizadores, muitas requerem uma subscrição mensal aos serviços que oferecem. *OpenSearch* [77] constitui uma bifurcação do motor *ElasticSearch*, distribuído pela *Amazon*, com algumas das funcionalidades disponíveis, tais como mecanismo de segurança, denotando encriptação, deteção de anomalias e prevenção de *ransomwares*, mecanismo de aprendizagem automática, replicação de *clusters*, entre outros, a custo de uma documentação mais escassa. A ferramenta de visualização *OpenSearch Dashboards* foi desenvolvida com base na interface *Kibana* [77].

OpenTelemetry [79], ou OTel, é uma *framework open source* para intrumentação, geração, obtenção e exportação de dados telemétricos como *traces*, *logs* e métricas. A *framework* oferece um *Observability backend*, um sistema dedicado a instrumentalizar determinada aplicação via os dados telemétricos que produz para facilitar a resolução de problemas e dirigido a arquiteturas baseadas em micro-serviços. A Figura 2.31 apresenta uma breve organização da infraestrutura desta *framework*.

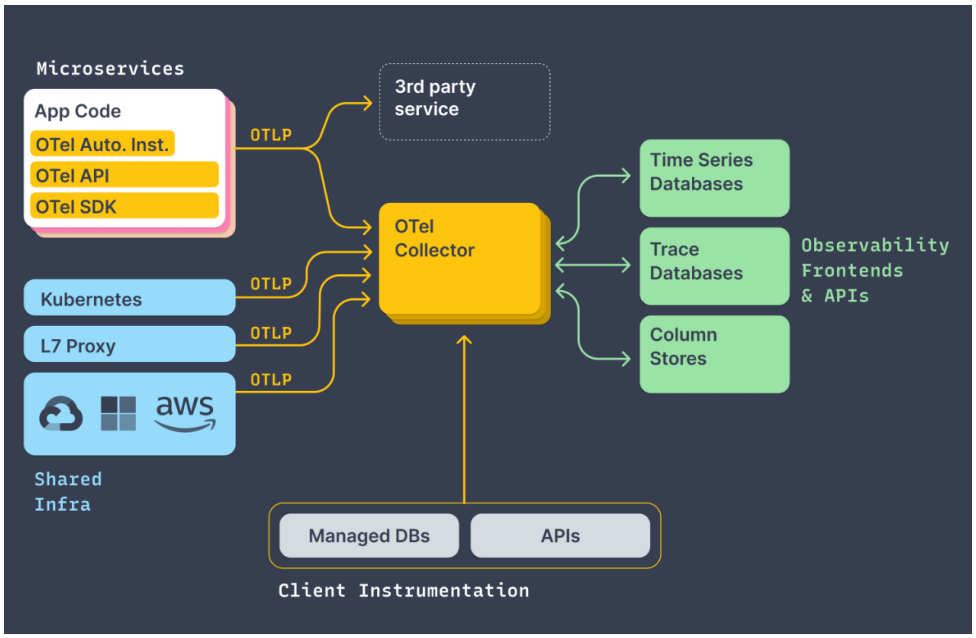


Figura 2.31: Infraestrutura *OpenTelemetry* [79]

Outras plataformas, semelhantes ao *Elastic Stack*, oferecem soluções alternativas, como o *TICK Stack* [88], *OpenStack* [78] e *Cisco AppDynamics* [6]. *TICK Stack* constituí um conjunto de tecnologias (Figura 2.32) de forma a disponibilizar uma plataforma capaz de guardar, capturar, monitorizar e analisar dados temporalmente. De entre estas tecnologias descrevem-se as seguintes:

- *Telegraf*, conjunto de informação sequencial proveniente de diversas origens, como dispositivos IoT;
- *InfluxDB*, base de dados de grande eficiência e desempenho para gestão de grandes volumes de informação;
- *Chronograf*, visualização em tempo real da informação disponível no *InfluxDB*;
- *Kapacitor*, monitorização e notificação baseadas em filtros criados do *InfluxDB* em anomalias presentes nesses filtros.

OverOps [81] constituí uma solução que permite às empresas, que desenvolvem *software*, garantir que alterações ou atualizações no código não impacte o consumidor. Com compatibilidade *Cloud*, oferece integrações CI/CD robustas para garantir confiabilidade do *software* desde a fase de teste à fase de produção. Através do *OverOps* é possível:

- Identificar e detetar problemas críticos induzidos por atualizações das aplicações e receber alertas via *Slack* (um programa de transmissão de mensagens) [85];

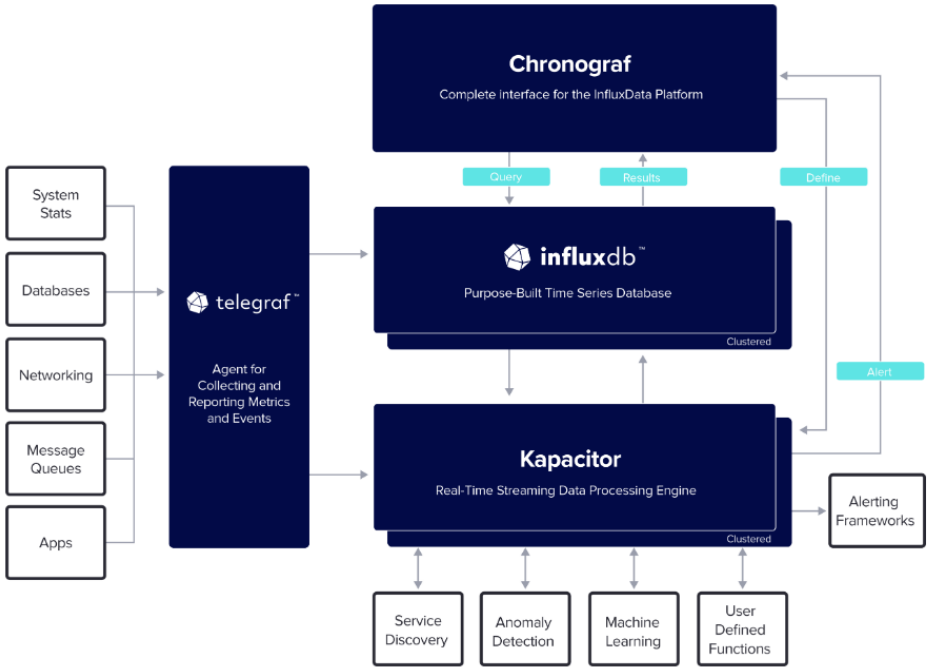


Figura 2.32: Infraestrutura *TICK Stack* [88]

- Bloquear determinadas atualizações com um padrão não confiável e notificar sobre problemas prioritário de forma a evitar que código potencialmente crítico impacte os consumidores;

- Resolução de problemas, erros ou exceções, detetados pela análise da origem via o código, variáveis, estado do ambiente e *logs* de nível *DEBUG*, sendo esta funcionalidade disponibilizada através de menu dedicado *Automated Root Cause* (ARC) e podendo ser aplicada em ambientes de produção ou de teste.

O *Open Zipkin* [80] é um sistema distribuído de rastreamento que auxilia a acumular informação temporal para diagnosticar latências nos serviços das arquiteturas em estudo. A informação pode facilmente ser consultada de acordo com atributos extraídos dos eventos de *log* e, em parte, é automaticamente gerada, complementando o diagnóstico por demonstração de percentagem de tempo dedicado em serviços particulares ou de operações em falha. A interface de utilizador apresenta um diagrama de dependência para a apresentação do número de pedidos rastreados por aplicação que, por sua vez, é útil para identificar comportamentos agregados, como o contexto do erro ou chamadas a serviços descontinuados. O *Open Zipkin* necessita de ser instrumentalizado, normalmente pela configuração de um rastreador ou uma biblioteca de instrumentalização, como *Kafka* [24].

O *ShiViz* [84] assemelha-se a muitos outros motores de visualização de *logs* para sistemas distribuídos, com a particularidade de apresentar a informação via uma diagrama temporal pelo que, da perspetiva interativa do utilizador, é possível observar como os eventos se relacionam uns com os outros.

2.7 Conclusão

O processo de *logging* oferece um ecossistema rico em informação que, por sua vez, possibilita diversos tipos de análises benéficas para a operação de sistemas complexos. Contudo, o estudo realizado mostra que existem diversos desafios que ocorrem no ciclo de vida da informação *log*. Devido à falta de definição de requisitos, tipicamente, os programadores usam a abordagem tentativa-erro e a sua experiência pessoal para declarar *logs*. Nesta medida, diversos estudo empíricos destacam a importância de instrumentalizar o processo de *logging* para substituir tais práticas. Quando os requisitos são bem definidos, as plataformas de *logging* mostram bons resultado após introduzidas e configuradas à medida dos projetos e recorrendo às técnicas mais recentes, como *machine learning*, para apoiar em decisões no processo *logging*. Dito isto, é ainda desconhecida a implicação que esta estratégia proporciona aos programadores, na medida que os modelos nunca são totalmente precisos e falsos positivos ocorrem frequentemente.

A maioria do esforço de investigação sob infraestruturas de *logging* é dedicado a técnicas de *parsing*. A literatura frequentemente interpreta o *parsing* da informação como um problema de extração de *templates* que, por sua vez, é resolvido por técnicas de agrupamento da informação estática na declaração *log*. A análise de *logs* de sistema, como *Hadoop*, encontra-se extensivamente estudada quando comparada com a de *logs* aplicacionais, que se deve à natureza privada e sensível da informação tratada. De forma a explorar esta vertente dos sistemas *logging*, poderá ser observada a informação *log* gerada em projetos *Open Source*, uma opção limitada pela quantidade de dados, ou por parceria com entidades da indústria, que possibilita a exploração de técnicas recentes em ambiente real. O *parsing* mostra ainda potencial para uma compressão de informação mais eficiente e um melhor armazenamento; no entanto, este processamento seria a nível do dispositivo em contexto IoT, uma proposta limitada considerando os recursos disponibilizados.

Análise de *logs* é um dos temas mais estudados na área de *logging*, onde a informação pode ser interpretada através de grande diversidade de representações que, por sua

vez, possibilita o crescimento de um ambiente competitivo onde os algoritmos desenvolvidos superam outros algoritmos em circunstâncias diferentes.

Relativamente aos sistemas IoT, por observação das práticas sugeridas na literatura, isto é, como, quando e que informação apresentar nas declarações *logs*, é possível derivar e definir boas práticas a utilizar durante o ciclo de desenvolvimento dos projetos. Qualquer processamento aplicado em *logs* após a geração da informação, destacando *parsing*, análise e visualização da informação, deve ser introduzido ao nível do ambiente *Cloud*, de forma a desacoplar processos pesados do ambiente limitativo dos dispositivos IoT.

3

Solução Proposta

Neste capítulo apresenta-se o sistema de *logging* desenvolvido para o *CardioWheel*. Este sistema compreende duas componentes associadas a pontos extremos do ecosistema *CardioWheel*, ilustrado na Figura 3.1. A primeira componente visa a obtenção dos dados de *logging* da *CardioWheel embedded board*, responsável pela aquisição e o pré-processamento do sinal ECG do condutor de um veículo. Estes dados são depois entregues à gateway *CardioWheel* instalada nesse veículo que, entre outras operações, faz o seu envio para um servdor na *cloud*. Este servidor é responsável pelo processamento, análise e visualização dos dados de logging.

Este capítulo está organiado em três secções. Na secção 3.1 analisa-se o elemento ciber-físico do sistema *CardioWheel*, a *CardioWheel embedded board*, apresentando-se as suas componentes *hardware* e *software*. O subsistema de logging desenvolvido para a *CardioWheel embedded board* é descrito na secção 3.2, enquanto na secção 3.3 se discute a

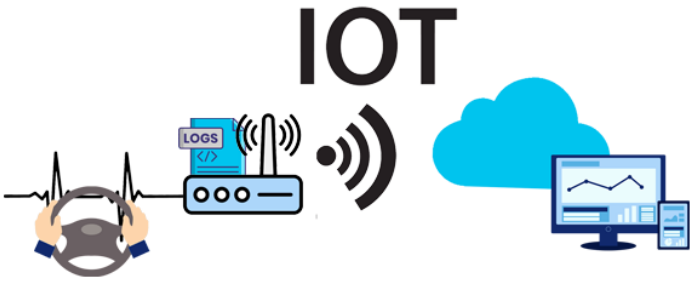


Figura 3.1: Arquitetura do *CardioWheel*

solução desenvolvida para fazer a análise e apresentação da informação de *logging*. Finalmente, na secção 3.4 resume-se as características da solução proposta.

3.1 O sistema *CardioWheel*

O *CardioWheel* é um Sistema Avançado de Assistência ao Condutor (em inglês, Advanced Driver Assistance System - ADAS) que visa a medição do nível de fatiga e a aferição da disposição geral do condutor de um veículo automóvel durante a sua atividade de condução [4]. Esta avaliação é efetuada analisando, em tempo real, o ECG do condutor, que é obtido através do contacto da superfície da sua pele com sensores instalados no volante do veículo pela *CardioWheel embedded board*, conforme se ilustra na Figura 3.2. Estes dados são enviados para a gateway que os agrega e envia, de forma segura, para um servidor na *Cloud* onde são processados com diferentes objetivos, em que se inclui a avaliação do desempenho dos condutores, a sua ecoeficiência e segurança rodoviária.

A utilização do ambiente *Cloud* implica a utilização de recursos computacionais acessíveis através da Internet, como servidores, sistemas de armazenamentos de dados ou bases de dados. A disponibilidade de tais recursos é especialmente importante em ambientes IoT, como o *CardioWheel*, devido às limitações de processamento e armazenamento dos dispositivos ciberfísicos. Desta forma, a *Cloud* é responsável por armazenar os dados gerados por esses dispositivos de forma centralizada, independentemente da quantidade de informação recebida, pois um ambiente *Cloud* é escalável, o que significa que recursos podem ser adicionados e realojados de acordo com as necessidades do sistema. Esta flexibilidade garante uma relação de custo-benefício maior a longo prazo, dado que os utilizadores apenas pagam pelos recursos que estão a consumir. A manutenção dos recursos é ainda mantida pelo fornecedor.

3.1.1 A *CardioWheel embedded board*

A *CardioWheel embedded board* consiste num sistema ciberfísico baseado no mocrocontrolador *ESP32* da *EspressIF* [4], uma empresa de semicondutores conhecida pelo desenvolvimento de microcontroladores acessíveis e com diversas capacidades desejadas no mundo IoT. Nomeadamente, o sistema é implementado por um módulo *ESP32-WROVER-E* [4] e um circuito eletrónico responsável pelo acondicionamento e pré-processamento do sinal ECG, conforme se mostra na Figura 3.3. O módulo *ESP32-WROVER-E* inclui um processador com dois núcleos de processamento com frequência de relógio ajustável entre 80 MHz e 240 MHz, uma interface de cartão SD e todos

Figura 3.2: Instalação da *CardioWheel embedded board* do volante do veículo [4]

Figura 3.3: Placa *CardioWheel* [4]

os periféricos necessários ao correto funcionamento do sistema. Por exemplo, o submódulo Wi-Fi suporta suporta as comunicações sem fios a ritmos até 150 Mbps. A quantidade de memória RAM disponível é 8 MB, mas o módulo inclui também uma memória Flash com uma capacidade de 4 MB.

A implementação original do FreeRTOS era orientada a microcontroladores com um único núcleo de processamento (*single core*). Para suportar sistemas com múltiplos núcleos de processamento foram desenvolvidas versões alternativas do FreeRTOS, como a ESP-IDF FreeRTOS. Esta adaptação oferece modificações significantes a nível de API e *kernel*, como a implementação da arquitetura *Symmetric Multiprocessing* (SMP), onde cada núcleo é executado de forma independente, o que permite a existência de diversas tarefas concorrentes, aumentando o *throughput*. Uma outra caraterística desta arquitetura, que contribui para uma melhor utilização dos processadores, é a aplicação de memória partilhada, o que significa que as tarefas podem alternar os núcleos durante a sua execução. Em todo o caso, também podem ser configuradas para executarem

unicamente num núcleo específico. Uma outra adição por parte da *framework* é relativa à gestão de tarefas, onde diversas regras podem ser definidas durante a inicialização de uma tarefa, como a quantidade de memória alocada, o núcleo de processamento associado e a prioridade na execução. Relativamente à prioridade, as tarefas são agendados de acordo com o nível de prioridade e o núcleo alocado. Relativamente ao funcionamento do *scheduler*, no FreeRTOS, na eventualidade de ocorrerem diversas tarefas com a maior prioridade prontas a executar, a sua execução é periodicamente alternada via um algoritmo *Round Robin* (RR). No entanto, uma tarefa pode estar alocada a um núcleo diferente e incapaz de ser selecionado pelo *scheduler*. Nesta medida, o ESP-IDF FreeRTOS implementa uma versão do RR que prioriza as tarefas não selecionados pelo *scheduler*. Através de métodos de sincronismo, é ainda possível bloquear e desbloquear determinada tarefa por ação de outra tarefa independente. Adicionalmente, as tarefas podem também ser consideradas *idle*, de prioridade mais baixa, e podem também ser terminadas para a libertação de recursos.

É importante mencionar o suporte para o tratamento de *secções críticas*, porções do código que necessitam de ser executadas sem interrupções, por mecanismos de interrupções ou por processos paralelos. São utilizadas para proteger recursos partilhados ou estruturas de dados de acessos concorrentes que, por sua vez, provocam corrupção de dados e comportamentos inesperados. Em ambientes de processamento paralelo, é importante garantir que apenas um núcleo seja capaz de aceder a determinadas secções do código, dados ou outros recursos (e.g. periféricos ou suas estruturas de dados) para manter a integridade e a estabilidade do programa.

3.1.2 A biblioteca *logging* da ESP-IDF

Um projeto criado através da ESP-IDF contém, pelo menos, o componente principal que contém o ponto de entrada da aplicação. Outros componentes podem ser adicionados com bibliotecas e funções reutilizáveis, para uso de um ou mais projetos. Nesta medida, o FreeRTOS é integrado na ESP-IDF como um componente. Outro componente disponível na ESP-IDF é a biblioteca *logging* [18].

A biblioteca *logging* tem como principal propósito substituir a utilização da função *printf* na depuração de erros em programas, para o que oferece várias vantagens, como por exemplo formato configurável das mensagens de log, possibilidade de definição de diferentes níveis de log (DEBUG, INFO, WARNING e ERROR) para controlo do nível de detalhes das mensagens com base nas necessidades específicas de depuração, configuração dinâmica dos níveis de log durante a execução do programa, suporte de

diferentes domínios de log, ou utilização de vários canais para saída das mensagens de log (e.g. consola, ficheiro, canal série ou serviços remotos).

Por definição, a mensagem associada a um evento de log gerado por esta biblioteca tem o seguinte formato:

LEVEL (TIMER) TAG: MESSAGE

em que

- *LEVEL*, o nível do evento que, por sua vez, pode ser, ordenado de nível mais baixo para o mais alto: *ERROR*, *WARNING*, *INFO*, *DEBUG* e *VERBOSE*;
- *TIMER*, um temporizador iniciado após a execução do dispositivo, em milissegundos. Este temporizador, por omissão, é calculado pelo ciclo do processador, o que significa que o temporizador recomeça após o reiniciar, esperado ou inesperado, do dispositivo, pode, porém, ser calculado tendo o ciclo do *hardware* para, por sua vez, persistir até o dispositivo se desligar;
- *TAG*, o contexto, ou etiqueta, associado ao evento *log*;
- *MESSAGE*, o conteúdo do evento *log*.

A biblioteca *logging* permite limitar o nível de *log* executado pelo sistema através da chamada de função *esp_log_level_set(TAG, LEVEL)*, sendo a *TAG* o contexto ou a lista de contextos afetado pela regra (*TAG*="*"para afetar todos os contextos). Cada nível de *log* possuí uma função responsável pela criação do respetivo evento. No caso do nível *ERROR* é utilizada a função *ESP_LOGE(TAG, MESSAGE)*. Por defenição, as mensagens de log são enviadas pela *Universal Asynchronous Receiver-Transmitter* (UART). Porém, pode ser definido um outro canal de saída utilizando a função *esp_log_set_vprintf(func)*, onde *func* descreve a função a executar como alternativa ao comportamento original.

3.2 Subsistema de *logging* da *CardioWheel embedded board*

O subsistema de *logging* desenvolvido para a *CardioWheel embedded board* consiste num novo componente do seu *firmware* implementado recorrendo à biblioteca *logging* incluída na ESP-IDF, conforme se ilustra na Figura 3.4. Nas secções seguintes discutem-se os princípios que nortearam o seu desenvolvimento e apresenta-se o modo de funcionamento do sistema.

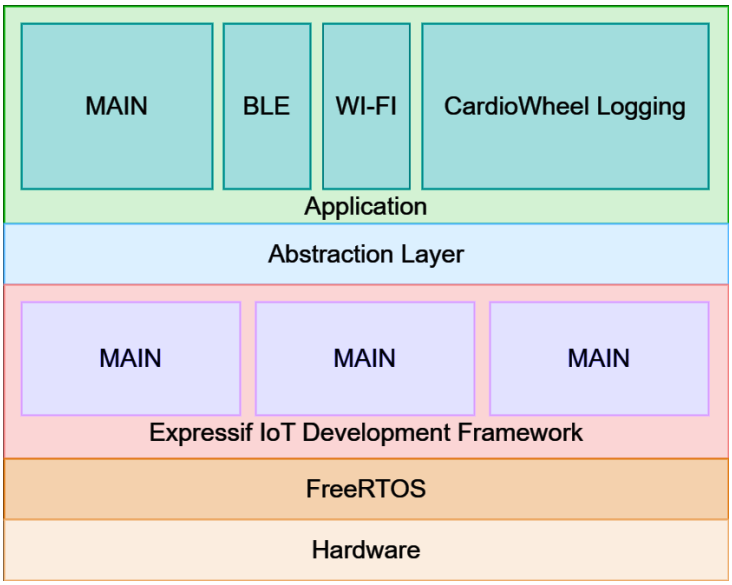


Figura 3.4: Arquitetura de *software* da *CardioWheel*

3.2.1 Análise de requisitos

O estudo do estado da arte sobre *logging* (Capítulo 2.3) mostrou que no desenvolvimento destes sistemas cabe aos programadores definir como, onde e que tipo de informação apresentar em eventos de *logging*. Assim, e numa tentativa de prevenir a ocorrência das problemáticas discutidas no Capítulo 2.3 e do impacto subsequente provocado durante o processamento da informação, definiriam-se um conjunto de regras para a implementação do sistema de *logging* da *CardioWheel embedded board*, tendo em conta a sua especificidade.

3.2.1.1 Qual informação registar em *Log*?

Antes de definir a informação que se deve apresentar em *log*, é necessário destacar que, com este sistema de *logging*, pretende-se efetuar sessões. Cada sessão corresponde a um período de *logging* de aproximadamente três horas, pelo que é necessário definir não apenas a informação relevante que se pretende observar e monitorizar, mas também a periodicidade dos eventos que geram a informação. Uma periodicidade indevida pode provocar o sobcarregamento do dispositivo e, por sua vez, provocar atraso ou falha em processos posteriores, Nesta secção refere-se a informação que deverá ser introduzida em *logs* de acordo com as necessidades da *CardioID* e que tipo de padrões a praticar de forma a facilitar o *parsing* e o processamento posterior da informação. Dito isto, pretende-se introduzir as seguintes informações:

- Informação de inicialização referente ao sincronismo *Simple Network Time Protocol* (SNTP), estado da memória, do cartão SD, dos sensores (Acelerômetro e Eletrocardiograma), do módulo *Bluetooth Low Energy* (BLE), do módulo Wi-Fi e do módulo de processamento. Este evento será introduzido sob o contexto *INIT*;

- Atualizações de *firmware* ao iniciar o dispositivo (contexto *FIRMWARE*);

- Percentagem de bateria disponível, com uma periodicidade de 30 segundos (contexto *BATTERY*);

- Estado da bateria que indica se está, ou não, em carregamento, inserida ou em falha, com uma periodicidade de 30 segundos (contexto *BATTERY*);

- Informação referente ao modo de utilização do volante durante a condução (Sem, Esquerda, Direita, Ambas), com uma periodicidade de 30 segundos (contexto *LOD*);

- Qualidade do sinal e/ou potência de ruído, com uma periodicidade de 30 segundos (contexto *SIGNAL*);

- Evento associado à deteção automática da mudança de condutor (contexto *DRIVER*);

- Estado do cartão de memória (contexto *SDCARD*), que por sua vez indica se o cartão está cheio, *FULL*, com problemas, *CORRUPTED*, ou normal, *NORMAL* (periodicidade de 30 segundos);

- Evento associado a deteção de acelerações súbitas, contexto *SPEEDUP* (o sistema será instalado no volante e permite monitorizar acelerações do veículo e do volante);

- Evento de deteção de eventos associados à monitorização do giroscópio, contexto *GIRO* (o giroscópio permite monitorizar o ângulo e a velocidade angular do volante);

- Eventos de deteção de anomalias na variabilidade do ritmo cardíaco, contexto *HRV* (*Heart Rate Variability* (HRV));

- Eventos associados à variação do *Karolinska Sleepiness Scale* (KSS) que é uma escala que permite que permite quantificar a sonolência, com uma periodicidade de 1 minuto (contexto *KSS*);

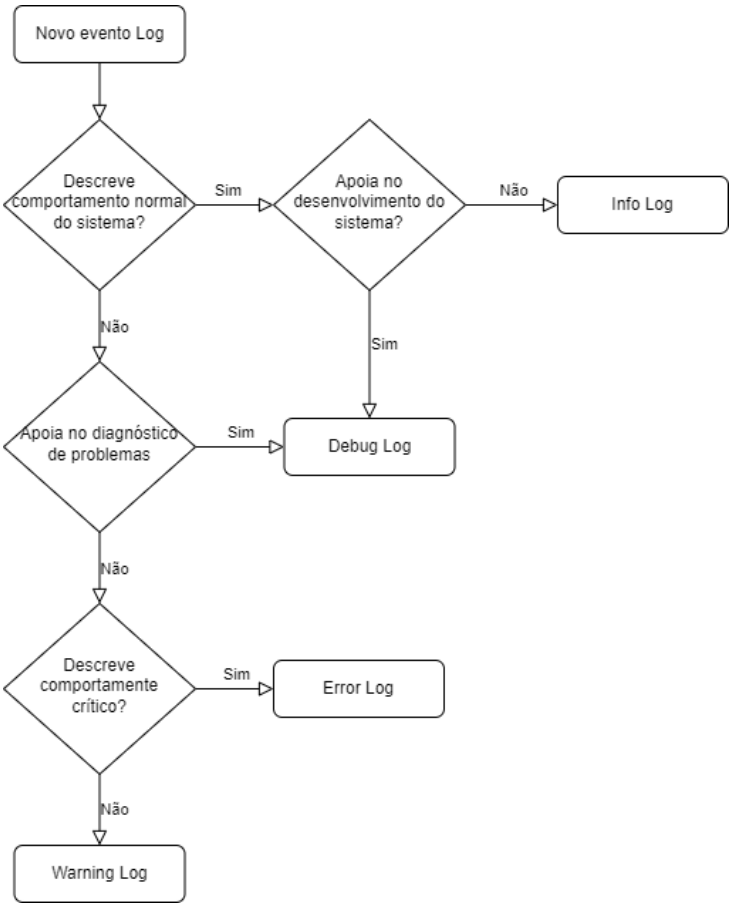


Figura 3.5: Relação entre eventos de log e níveis de severidade

É importante garantir que os eventos *logs* são declarados com um formato percetível, capaz de ser identificado na fase de *parsing*, especialmente quando se tratam de informações descritivas como o estado do sensores. Como exemplo, sugere-se a utilização de "=" e ":" para apresentar o estado de um determinado sensor, portanto, "ECG=OK". Outros formatos poderão ser utilizados, no entanto, deve garantir que a regra seja adicionada à ferramenta de *parsing* e que o padrão utilizado é suficientemente distinto do texto comum.

3.2.1.2 Como inserir o *Log*?

No desenvolvimento do subsistema de *logging*, seguiram-se os seguintes princípios para comunicar corretamente a informação via o evento *log*:

- Nível de detalhe adequado, para o que foi criado o diagrama apresentado na Figura 3.5 que associa os casos mais prováveis de ocorrer durante uma sessão de *logging* aos níveis de severidade adequados;

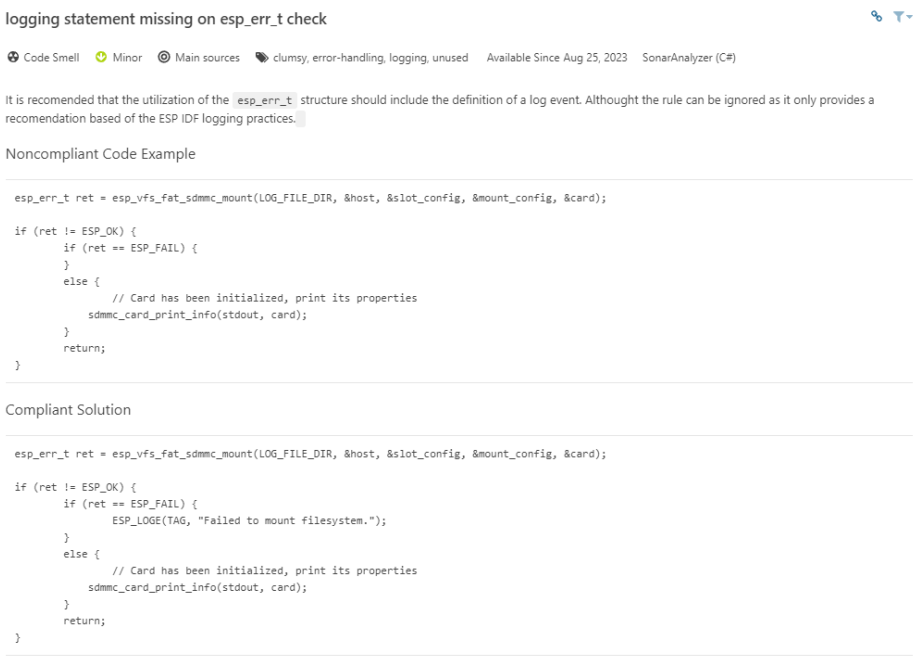


Figura 3.6: Regra personalizada no *SonarQube*

- Ausência de eventos de *log* perante uma exceção. Com a utilização de ferramentas de análise de código estático, como o *SonarQube*, é possível analisar e destacar pequenos pormenores na escrita do código que se consideram incorretos dependendo das regras definidas. O *SonarQube* possuí de perfis de regras para linguagens de programação que podem ser associados a diversos projetos, adicionalmente podem ser adicionadas regras personalizadas a tais perfis. No que toca à ausência de eventos de *logs* na *framework* ESP-IDF, é particularmente utilizada a estrutura de dados *esp_err_t* para detetar a existência de uma mensagem de *log* após um condicional em erro, dito isto, esta implementação pode ser considera como um *Code Smell*, ou seja, um exemplo de uma má prática (Figura 3.6). Como alternativa, é possível utilizar a ferramenta de análise implementada pela *espressif*, *IDF Clang Tidy*, embora se encontre ainda em fase de desenvolvimento;

- Ausência de erros ortográficos e/ou garantia da percetibilidade da mensagem, para o que se recorreram a alguns *plugins* para localizar inconsistências ortográficas aquando do desenvolvimento do software (Figura 3.7);

- Inexistência de objetos nulos num evento *log*, o que pode ser conseguido testando o objeto antes da sua utilização;

- Conversão correta de objetos, pelo que se devem testar os objetos utilizados à

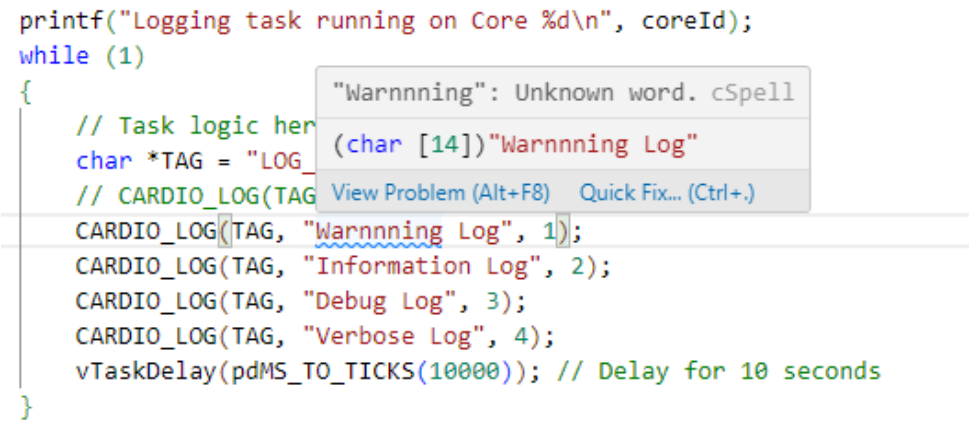


Figura 3.7: Utilização do *plugin Code Spell Checker*

compatibilidade da conversão que se pretende efetuar antes de realizar essa conversão;

- Inclusão dos eventos *logging* como parte do ciclo de vida do projeto, ou seja, se o projeto é gerido em modo *Agile*, os eventos *logs* deverão ser interpretados como parte dos critérios de aceitação da respetiva implementação a que dizem respeito.

3.2.1.3 Quando registar em *Log*?

Relativamente ao momento mais adequado para registar em *log* um novo evento, os princípios seguidos no desenvolvimento do subsistema de *logging* foram os seguintes:

- Para eventos relativos a ocorrências esperadas e informativas devem ser utilizadas declarações *if-else*;
- Para eventos periódicos, como mensagens relativas ao estado da bateria do dispositivo ou uma medida de determinado sensor, utiliza-se uma tarefa dedicada para a declaração desses eventos no *log* com a periodicidade adequada, que para o funcionamento do sistema *CardioWheel* se verificou ser entre 10 a 30 segundos;
- Não criar eventos de *logging* em secções críticas, devendo-se adicionar as mensagens *log* a uma *queueu* para posterior processamento (i.e. após a conclusão da execução do troço de código da secção crítica) numa tarefa dedicada.

3.2.2 Funcionamento do subsistema de *logging*

A solução de *logging* desenvolvida para o sistema CardioWheel é baseada na biblioteca de *logging* da ESP-IDF e tem como objetivo registar em *log* todos os eventos ocorridos

durante uma sessão de condução, i.e. o período de tempo em que um utilizador conduz o veículo automóvel, desde o momento em que o seu motor é ligado até que volte a ser desligado. Assim, o funcionamento do sistema de *logging* foi organizado em três fases; inicialização, sessão de *logging* e terminação.

A fase inicialização é executada com a chamada *CARDIO_LOGGING_INIT(LEVEL)*, onde *LEVEL* corresponde ao nível máximo de evento log permitido. Esta fase é executada durante a etapa final da iniciação do *CardioWheel* e antes do início de cada sessão de condução por forma a capturar todo o percurso de *logging*. Este processo é executado no fluxo primário do programa e é bloqueante, ou seja, o dispositivo não deverá seguir com o fluxo até à inicialização da biblioteca de *logging*. Durante o decorrer desta fase são efetuadas as seguintes configurações:

- *esp_log_level_set()*, para definir o nível máximo dos eventos *log*;
- *MOUNT_SD_CARD()*, para inicializar o cartão de memória e o sistema *File Allocation Table* (FAT) sob a diretoria /sdcard, recorrendo à função nativa da ESP-IDF: *esp_vfs_fat_sdmmc_mount()*;
- *SNTP_INIT()*, para sincronizar o relógio do sistema com um servidor SNTP (são realizadas dez tentativas de ligação ao servidor para obter informação temporal após as quais é utilizada uma data falsa);
- *CREATE_LOG_FILE()*, para a criação e abertura do ficheiro da sessão de *logging* atual, com a nomenclatura **ID-TIMESTAMP.txt**, e configuração deste media para funcionar como canal de saída para a sessão de *logging*, caso o ficheiro tenha sido aberto;
- *xTaskCreatePinnedToCore()*, para criar uma tarefa alocada a um núcleo específico e similar periodicamente eventos *log*.

A ordem de realização destas operações é especialmente importante pois, por exemplo, a criação do ficheiro de sessão necessita não só que exista um cartão de memória com o sistema FAT iniciado, mas também do sincronismo SNTP efetuado. Em caso de qualquer falha nos processos relativos ao cartão de memória, sistema FAT ou criação do ficheiro, então a inicialização do subsistema de *logging* será cancelado e o sistema decorrerá sem persistência de *logs*.

Relativamente à fase de sessão de *logging*, seguiu-se os princípios discutidos na secção 3.1.2 mas adotou-se um formato mais informativo para os eventos de *log*:

[TIMESTAMP] ID LEVEL (TIMER) TAG: MESSAGE

em que,

- *TIMESTAMP*, indica a data e hora do evento;
- *ID*, é o identificador único do dispositivo.

A fase terminação consiste no fecho do ficheiro de sessão e na invocação da função *esp_log_set_vprintf(&vprintf)* para repor o canal de saída original definido na biblioteca de *logging*. De seguida, o ficheiro de sessão é enviado para o servidor na *Cloud*, conforme discutido na secção 3.3 .

3.2.3 Armazenamento de *Logs*

Um meio de armazenamento externo permite separar o ambiente de execução do local de armazenamento. Esta abordagem torna a solução escalável e garante a possibilidade de recuperar todo o conteúdo do armazenamento em caso de falha ou avaria do dispositivo.

Nesta proposta, decidimos persistir a sessão de *logging* num ficheiro dedicado num cartão SD. Numa fase inicial, os eventos de *log* adicionados são primeiro armazenados em cache, na memória interna do dispositivo, e posteriormente sincronizados com o ficheiro de sessão. Dada a importância da integridade dos dados no projeto, foi priorizado um sincronismo pontual aos dados dos *logs*, isto é, após cada evento, em lugar de uma sincronização periódica. Desta forma garante-se que nenhum evento se perde, por outro lado tornando o processo mais exigente.

O ciclo de vida de um ficheiro de sessão começa na fase de inicialização da biblioteca desenvolvida, onde referido ficheiro é criado e preparado para guardar eventos. Por sua vez, termina na fase de finalização onde o ficheiro é fechado, enviado para a *Cloud* e eliminado. Em caso de ocorrência de falhas durante o envio do ficheiro, este persiste em armazenamento até a ocorrência de uma nova sessão de *logging*.

3.2.4 Comunicação

O envio dos ficheiros de sessão de *log* para o servidor na *Cloud* responsável pelo seu processamento requer a utilização de uma tecnologia de comunicação capaz de transmitir quantidades significativas de dados com garantias de integridade e segurança por esses dados serem de natureza crítica. De entre todos os protocolos de comunicação analisados, quatro seguintes destacaram-se como potenciais candidatos a utilizar no sistema de *logging* do *CardioWheel*:

***SSH File Transfer Protocol* (SFTP)** É um protocolo seguro para envio de ficheiros executado sobre o protocolo *Secure Shell* (SSH) [83]. Suporta todas as funcionalidades de autenticação e segurança do protocolo SSH, portanto, resistente a ataques de *sniffing* e *man-in-the-middle*, e a integridade da informação é protegida pela uso de encriptação e funções *hash* criptográficas. A autenticação é efetuada em ambos o cliente e o servidor. A informação é convertida para modo binário, o que a torna impercetível ao olho humano, no entanto, a respetiva transferência entre os dois participantes é lenta comparativamente com outros protocolos.

***File Transfer Protocol Secure* (FTPS)** É uma extensão do protocolo de transmissão de ficheiros *File Transfer Protocol* (FTP) que suporta *Secure Sockets Layer* (SSL) e o seu sucessor *Transport Layer Security* (TLS), que, por sua vez, utilizam certificados digitais para facilitar o processo de *handshake* e estabelecer uma comunicação encriptada. FTPS adiciona uma camada superficial de segurança ao protocolo de envio de ficheiros, contrariamente ao SFTP que adiciona o envio de ficheiros ao protocolo de segurança. A abstração da componente de segurança via TLS impõe o uso de diversos portos de comunicação, o que dificulta a configuração das *firewalls* e introduz mais pontos vulneráveis na solução. Os dados são enviados em código ASCII, um outro ponto de falha na medida que pode ser interpretada por um utilizador humano treinado.

***Applicability Statement* 2 (AS2)** É uma especificação que indica como dados estruturados de negócio devem ser transportados de uma forma segura na rede [2]. Tal como FTPS, utiliza TLS e possuí de caraterísticas semelhantes ao protocolo SFTP. Umas das funcionalidades que se destacam é a existência de um *Message Disposition Notification* (MDN), um recibo que pode ser devolvido à origem do conteúdo para garantir que a informação não foi adulterada. Esta especificação não se encontra otimizada para envio de ficheiros de grandes dimensões ou em grupos e é tipicamente utilizada para o envio de *Electronic Data Interchange* (EDI) em transmissão ponto a ponto;

***Hyper Text Transfer Protocol Secure* (HTTPS)** É um protocolo de comunicação que, contrariamente ao protocolo *Hyper Text Transfer Protocol* (HTTP), utiliza TLS para garantir segurança na transmissão de dados entre um servidor e um *browser*. Uma vez estabelecida a comunicação, ambas as partes utilizam uma chave partilhada para encriptar a informação transmitida e utiliza também certificados digitais emitidos por autoridades certificadas para garantir a identidade do servidor. Este protocolo é conhecido por facilitar uma boa experiência ao utilizador e acessibilidade, porém, a custo de um ritmo de transmissão de ficheiros mais baixo [21].

Da análise realizada, o protocolo SFTP e a especificação AS2 destacaram-se como potenciais candidatos a adotar na implementação do sistema de *logging*, Contudo, a especificação AS2 requer um processamento relativamente pesado e é pouco explorado na área IoT, é também tipicamente utilizada em âmbitos industriais onde a utilização de transações EDI facilita a automação de pedidos aos fornecedores. Por outro lado, o protocolo SFTP é muitas vezes utilizado em contexto IoT para garantir a segurança e integridade no envio de dados. Por exemplo, em [73] apresenta-se uma implementação deste protocolo em ambiente IoT para envio de informação de *logging* e de eletrocardiogramas a partir de um dispositivo *Android*. A integridade é garantida pela inclusão de um *checksum* no conteúdo enviado, posteriormente decriptado e verificado pelo recetor.

Em todo o caso, é necessário ter alguns cuidados na forma como se introduz o protocolo SFTP num sistema:

- Cada dispositivo deverá ter um par de chave pública/privada dedicado, para que, mesmo que determinado atacante obtenha as credenciais utilizadas por um dispositivo, o risco seja baixo dado que este não as pode aplicar para afetar os restantes dispositivos;
- A autenticação deverá ser efetuada pela combinação de palavra-passe e par de chave pública/privada;
- Um dispositivo autenticado deverá ser considerado como um utilizador com acesso restrito aos recursos do servidor, dado que toda a informação de *logging* está limitada a um local, e esse utilizador deverá apenas ter acesso à referida diretoria de *logging*, sem privilégios administrativos; adicionalmente, para impedir acesso a informação sensível presente nos ficheiros de *logging* já existentes na diretoria, o utilizador deverá apenas estar configurado com acesso para escrita, sem leitura.

No caso da *CardioWheel embedded board*, recorreu-se à biblioteca *CycloneSSH* [7] para introduzir o protocolo SFTP no seu *firmware*. Normalmente, esta biblioteca é utilizada para operar serviços de rede de uma forma segura, como SSH e transferência de ficheiros em redes inseguras, com a utilização de criptografia assimétrica/simétrica como método de autenticação. Desta forma, a *CycloneSSH* garante confidencialidade e integridade no intercâmbio de informação entre o dispositivo e a *Cloud*. Embora as caraterísticas desejáveis que se apresentam, foi necessário adaptar o código fonte da biblioteca de forma a funcionar corretamente com o sistema operativo *freeRTOS* e a *framework* ESP-IDF. As alterações foram efetuadas principalmente a nível do ficheiro

CMAKEList e de variáveis e estruturas estabelecidas, devido a incompatibilidades na nomenclatura.

Uma vez adaptada a biblioteca *CycloneSSH*, foram introduzidas duas implementações com base nas funcionalidades que oferece: configuração da comunicação Wi-Fi e da comunicação SSH. A configuração da comunicação Wi-Fi, dos quais se destaca a inicialização de *hardware* acelerador criptográfico, componentes especializados para melhorar o desempenho e eficiência de operações criptográficas como encriptação, desencriptação, assinaturas digitais, geração de chaves, entre outros. O módulo responsável por esta processo, *CycloneCRYPTO*, oferece um conjunto compreensivo de primitivas criptográficas (funções *hash*, cifras de fluxo e de bloco, chaves públicas) utilizadas para adicionar segurança ao sistema embebido e suporta aceleradores para uma grande parte do mercado de microcontroladores: aceleração criptográfica simétricas e assimétrica. Outro passo a destacar é relativo à inicialização do gerador de números pseudo-aleatórios, o algoritmo utilizado, *Yarrow*, foi desenvolvido para combater ataques práticos e matemáticos. O algoritmo utiliza entropia processada de vários eventos de sistema, *hardware* e de utilizador para gerar as sementes. A comunicação SSH e o respetivo envio de dados, é iniciada pela verificação do endereço do servidor destino e das credenciais utilizadas para a comunicação segura. Uma vez estabelecida a conexão com o servidor, o dispositivo interage com um repositório de ficheiros estático, para onde envia todos os ficheiros pendentes na memória externa. O envio de cada ficheiro decorre sequencialmente da seguinte forma:

- Criação de um ficheiro do lado do servidor com o mesmo nome do ficheiro de sessão de *log* mas com o prefixo "temp-";
- Envio do conteúdo do ficheiro de sessão, linha a linha, para o ficheiro do servidor;
- Remoção do prefixo "temp-" do ficheiro do lado do servidor;
- Remoção do ficheiro de sessão da memória externa do dispositivo.

Dado que, do lado do servidor, o *parser* apenas analisa os ficheiros com uma determinada nomenclatura ("cardioidxxxx.txt"), a adição de um prefixo garante a leitura de apenas ficheiros válidos, isto é, que foram totalmente enviados com sucesso. Posteriormente, uma tarefa é executada do lado do servidor com a finalidade de limpar periodicamente os ficheiros mais antigos. Este atraso possibilita alguma margem de tempo para análise manual dos ficheiros, caso exista essa necessidade.

Do lado do dispositivo, poderão ocorrer erros durante a eliminação dos ficheiros de sessão após o seu envio com sucesso. Para garantir a eliminação desses ficheiros, e

assim evitar a sua duplicação no servidor, foi implementada a seguinte lógica: caso o dispositivo detete um ficheiro com o mesmo nome do lado do servidor então o ficheiro de sessão é eliminado e respetivo envio do ficheiro cancelado, nesta medida, mesmo que a referida tarefa de limpeza elimine o ficheiro no servidor, o *parser* possuí contexto suficiente para determinar se um ficheiro com determinada nomenclatura já se encontra analisado. Por outro lado, pode ocorrer corrupção de ficheiros no lado da *Cloud*. No entanto, dado que o protocolo SSH garante a integridade no envio dos dados, a grande maioria destas ocorrências têm origem no próprio dispositivo ciberfísico antes do envio do ficheiro, pelo que, mesmo que a corrupção seja detetada durante a sessão, a informação já se encontra irrecuperável e um novo ficheiro de sessão terá de ser criado.

3.3 Cloud

Nesta secção são discutidos os três principais elementos que compõem a componente *Cloud* do sistema de *logging* desenvolvido: ferramenta de *parsing*, motor de pesquisa e ferramenta de visualização.

3.3.1 Ferramenta de *Parsing*

Genericamente, uma ferramenta de *parsing* consiste num componente de *software* utilizado para analisar e interpretar a estrutura da informação de acordo com uma sintaxe específica. No contexto deste trabalho, é necessária uma solução capaz de receber, transformar e redirecionar a informação presente nos ficheiros de sessão para um motor de pesquisa. A ferramenta escolhida foi o *Logstash* [10] pelas seguintes razões:

- Possuí uma grande flexibilidade de pontos de entrada e de saída, como ficheiros, base de dados, *queues*, email, comunicação usando o protocolo HTTP, entre outros, para além de oferecer compatibilidade com diversos motores de pesquisa como ponto de saída, como *OpenSearch* [77] e *ElasticSearch* [10];
- Constitui um componente do *Elastic Stack* [10], que é um ecossistema que oferece um conjunto de soluções integradas de pesquisa, análise e visualização e exploração de dados e, portanto, facilita o desenvolvimento do sistema de processamento de dados;
- É uma solução *open source*, com uma comunidade ativa no desenvolvimento de *plugins* que, por sua vez, enriquecem as funcionalidades disponibilizadas;

- Escalabilidade, especialmente com recurso à plataforma *Docker* [8] é possível configurar o *Logstash* a escalar horizontalmente de acordo com a quantidade de dados;
- Oferece garantias de confiabilidade, através de mecanismos de *retry*, *Dead-Letter Queue* (DLQ) e manipulação de erros.

O modo de execução de *Logstash* é definido por um ficheiro de configuração organizado em três secções, representado na Figura 3.8: *input*, *filter* e *output*. A secção *input* define o ponto de entrada dos dados que, no caso do sistema de *logging* desenvolvido, é qualquer ficheiro com o prefixo "cardioid" (demonstrado através da diretoria *file*). O *Logstash* suporta dois modos de processamento dos ficheiros: *tail*, analisa o ficheiro sempre que novas linhas são adicionadas, e *read*, utilizado para ficheiros estáticos, como é o caso dos ficheiros gerados por este sistema *logging*, onde cada ficheiro é apenas analisado uma vez. Na secção *filter*, que se refere à transformação e interpretação da informação não estruturada em cada linha do ficheiro, é efetuada uma estruturação inicial onde são extraídos os padrões principais do evento (demonstrado na primeira instância do *grok* e *match*). A mensagem do evento, neste caso *content*, é primeiro interpretada como sendo todo o conteúdo do *log* após a *TAG*, neste caso *context*, porém, podem ocorrer padrões secundários nesta mensagem de acordo com regras previamente definidas . Dito isto, caso um dos padrões definidos seja detetado na mensagem do evento, como por exemplo uma palavra seguida de um "=" seguido de uma descrição (*ON* ou *OFF*) ou valor numérico, um segunda estruturação é efetuada. Caso sejam detetados múltiplos padrões na mensagem, o *log* original é separado num número de instâncias igual ao número de padrões detetados. Finalmente, a secção *ouput* especifica o motor de pesquisa *OpenSearch* como ponto de saída da informação transformada, onde é definido o endereço, as credenciais, o índice e algumas instruções relativas ao certificado SSL. O índice constituí um tipo de identificador da instância do *Logstash* que facilita a visualização da informação quando diversas instâncias de *parsing* interagem com a plataforma *OpenSearch*. Opcionalmente, como medida de segurança, é possível configurar o *Logstash* a efetuar uma verificação pelo certificado SSL/TLS após estabelecer uma ligação com o servidor remoto onde se encontra a instância *OpenSearch*.

3.3.2 Motor de Pesquisa

A informação resultante do *parsing* dos *logs* necessita de persistir para poder ser consultada posteriormente. Uma das formas de garantir esta disponibilidade de dados de

```
input {
        file {
                path => "/home/ganilha/kibana/cardioid*"
                mode => "tail"
        }
}

filter {
        grok {
                match => { "message" => [
                        "\[%{TIMESTAMP_ISO8601:timestamp}\] %{NUMBER:id} %{WORD:log_level} \(%{NUMBER:pid}\) %{DATA:context}: %{GREEDYDATA:content}",
                        "%{NUMBER:id} %{WORD:log_level} \(%{NUMBER:pid}\) %{DATA:context}: %{GREEDYDATA:content}"
                        ]
                }
        }

        if [content] =~ /.+=(\d+|\w+)/ {
                # Split the "content" field into an array based on the delimiter " "
                split {
                        field => "content"
                        terminator => " "
                }

                # Use a grok filter to capture multiple occurrences of "parameter=value" or "status=success" pattern for each element in the "content" array
                grok {
                        match => {
                                "content" => [
                                        "%{DATA:parameter}=%{NUMBER:value}",
                                        "%{DATA:parameter}=%{WORD:description}"
                                ]
                        }
                }
        }

        date {
                match => ["timestamp", "YYYY-MM-dd HH:mm:ss"]
                target => "@timestamp"
        }
}

output {
        opensearch {
                hosts => ["https://127.0.0.1:9200"]
                index => "indexforlogstash"
                user => "admin"
                password => "admin"
                ssl => true
                ssl_certificate_verification => false
        }
}
```

Figura 3.8: Configuração da ferramenta *Logstash*

uma forma consistente e segura é recorrendo a um motor de pesquisa. O motor de pesquisa utilizado no sistema de *logging* desenvolvido é o *Open Search* [77], que consiste num motor de pesquisa e de análise distribuído e escalável baseado no motor *Lucene*. Este motor tem a capacidade de realizar as funcionalidades típicas de procura, organização e agregação de dados, ainda que utilizando índices para gerir e filtrar informação proveniente de diferentes origens. A interação entre a plataforma *Open Search* e um pedido externo, proveniente de uma instância *Logstash*, por exemplo, é efetuada através de *Representational State Transfer Application Programming Interface* (REST API), um conjunto de restrições e princípios para projetar e interagir com serviços *web*, com operações *Create, Read, Update, Delete* (CRUD). Este tipo de operações é essencial na gestão e processamento de informação e possibilita efetuar ações complexas na inserção e visualização de dados.

Apesar de esta plataforma ser uma versão *open source* da plataforma *ElasticSearch* ela não oferece uma arquitetura tão robusta como a *ElasticSearch*. Contudo, possibilita o desenvolvimento de *plugins* para enriquecer o ecossistema de funcionamento e, através de ferramentas como o *Logstash*, é ainda possível migrar todo o desenvolvimento e configurações efetuada no *cluster OpenSearch* para um *cluster ElasticSearch* e vice-versa.

Figura 3.9: Estrutura do índice

3.3.2.1 Gestão de Índices

Como referido anteriormente, no *OpenSearch* a informação é organizada em índices, o que significa que é possível definir índices para diferentes necessidades ou diferente aplicações no sistema. Um índice consiste num documento *JavaScript Object Notation* (JSON) utilizado para efetuar o mapeamento da informação transmitida sob o identificador do índice, a nível da nomenclatura e do tipo de informação. Tipicamente, é adicionada informação suplementar ao índice pelo motor de pesquisa e pela ferramenta de *parsing*, como a data de chegada da informação, o utilizador e a diretoria e nome do ficheiro, de forma a contextualizar a informação recebida. A Figura 3.9 mostra a estrutura do índice configurado para interpretar a informação enviada pela instância *Logstash* referente ao sistema de *logging* da *CardioWheel*.

De notar que cada nó do *cluster OpenSearch* tem um limite de espaço em que se pode alocar informação referente a um índice. Consequentemente, a plataforma divide os índices em fragmentos para uma distribuição equilibrada entre os nós. Para garantir disponibilidade em caso de falha de um nó, de cada fragmento primário podem ser criadas réplicas, posteriormente distribuídas por nós distintos do fragmento primário a que correspondem. Esta dinâmica é também utilizada para melhorar o desempenho do *cluster* no processamento dos pedidos. A Figura 3.10 demonstra a visualização dos parâmetros principais de um índice na plataforma.

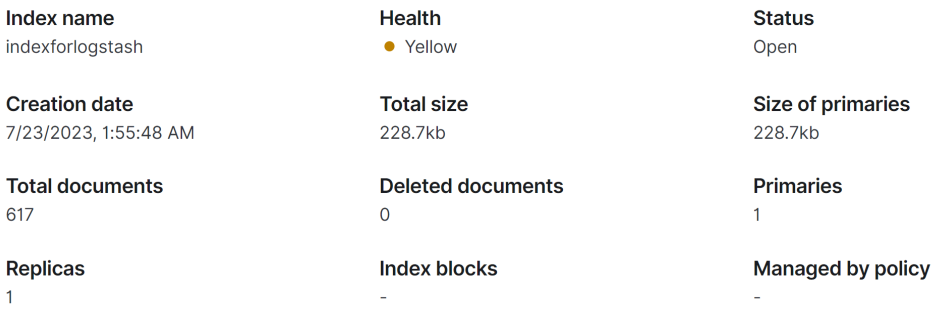


Figura 3.10: Visualização do índice

O *OpenSearch* permite que diversas ações sejam efetuadas em cada índice, das quais se destacam:

- fechar o índice, que fica indisponível para visualização e adição de informação;
- migrar o conteúdo de um índice para outro índice com o mesmo ou menor número de fragmentos primário;
- divisão do índice em diversos índices, em que cada fragmento primário é dividido em múltiplos fragmento primários que são associados a um novo índice;
- alterar o intervalo de atualização;
- associar um bloco, que limita o tipo de operações disponíveis, por exemplo de escrita ou de leitura;
- associar uma política, tipicamente utilizada para priorizar índices mais recente através de operações periódicas em índices mais antigos para reduzir o número de réplicas, o eliminar ou alterar os blocos associados.

3.3.2.2 Deteção de Anomalias

Umas das funcionalidades desejadas na plataforma *OpenSearch* consiste na deteção de anomalias nos dispositivos pela análise da informação presente nos *logs*. O algoritmo utilizado para esta deteção, *Robust Random Cut Forest* (RRCF) [82], é baseado no algoritmo *Isolation Forest* [23] que foi desenvolvido para detetar anomalias pelo isolamento de eventos (de *log* por exemplo) sem recorrer a medidas de densidade ou de distância. Utiliza, assim, duas propriedades quantitativas das anomalias: as características dos eventos constituem uma minoria no conjunto de dados e possuem atributos/valores

muitos distintos de eventos considerados normais. O isolamento dos eventos é representado através de uma árvore binária, onde uma pontuação anómala é atribuída a um evento de acordo com o valor de profundidade na árvore necessário para chegar ao nível a que corresponde. Desta forma, o algoritmo é utilizado para construir um grupo de árvores de dimensão pré-definida, denominadas *ITrees*, a partir de um conjunto de dados e, a cada árvore, é atribuída um sub-conjunto aleatório de dados do conteúdo original. A pontuação final de determinado evento é calculada pela agregação de todas as medidas de profundidade das *ITrees* [23]. O RRCF foi implementado de forma a minimizar falsos positivos pela substituição da seleção uniforme aleatória por uma seleção aleatória com base na variedade geométrica dos dados [82].

Os resultados do algoritmo RRCF na plataforma *OpenSearch* são interpretados recorrendo a duas métricas: grau de anormalidade e grau de confiança. O grau de anormalidade corresponde a indica o nível quantitativo da anormalidade associada ao conteúdo da informação. Por sua vez, o grau de confiança corresponde à probabilidade de um grau de anormalidade ser confiável, tendo tendência para aumentar com a quantidade de informação observada pelo modelo. Um detetor terá de ser configurado para descobrir anomalias em um ou mais parâmetros definidos no índice, através de um método de agregação para cada parâmetro, ou seja, a anomalia é descoberta com base na média, contagem, somatório, valor mínimo ou valor máximo de cada parâmetro. É necessário algum cuidado no desenvolvimento de um modelo multi-parâmetro dado que, embora correlacione as anomalias entre parâmetros, é menos provável a deteção de pequenas anomalias (*Curse of Dimensionality*) e pode afetar a precisão e o *recall* (a capacidade de um modelo de encontrar informação relevante num grupo de dados) do modelo. A existência de uma grande proporção de ruído pode ainda amplificar este impacto negativo. A seleção dos parâmetros é, portanto, um processo iterativo e é recomendado utilizar um máximo de cinco parâmetros por detetor [77] , embora este valor máximo seja configurável. A janela de deteção é definida pelo número de intervalos de agregação no grupo de dados (*shingle*) que terá ser ajustado de acordo com as necessidades da deteção. Um menor número de intervalos pode aumentar o *recall* mas também falsos positivos; por outro lado, um maior valor é tipicamente útil para ignorar ruído. O modelo permite um alcance de um a 60 intervalos, sendo que o valor um é utilizado apenas para um modelo multi-parâmetro. O detetor é executado periodicamente para agrupar informação e gerar anomalias. Quanto menor este intervalo mais frequentemente são atualizados os resultados e mais recursos são requeridos pelo detetor. É necessário que este intervalo esteja ajustado à frequência a que nova informação é adicionada. Para acomodar o tempo de processamento extra necessário, é

possível definir um período de atraso dedicado à fase de agrupamento de dados. Devido aos requerimentos de funcionamento do modelo e à dificuldade em replicar um caso de estudo relevante, pouco se pode concluir relativamente à eficiência do modelo na deteção de anomalias para ocorrências esperadas em ambiente de produção porém, para os testes delimitados, o modelo demonstrou a capacidade necessária de detetar anomalias para grupos de dados limitados.

3.3.2.3 Alertas e Notificações

O *OpenSearch* disponibiliza a possibilidade de gerar alertas e notificações através de monitores para complementar a funcionalidade de deteção de anomalias. Para monitorizar anomalias, o monitor deverá estar associado ao detetor em causa e configurado com um *trigger* para acionar um alerta sob as condições desejadas. Tipicamente, o *trigger* é preparado para ativar quando, em caso de anomalia, o grau de anormalidade e de confiança ultrapassam um determinado valor limite. Um monitor pode possuir múltiplos *triggers*, cada um com um nível de severidade associado, entre 1 e 5. Por sua vez, a notificação do alerta pode ser enviada por diversos canais disponibilizados pela plataforma:

- *Slack* [85] ou *Chime* [5], que são serviços de mensagens instantâneas, semelhante ao *Teams* [87], desenvolvidos para comunicação profissional e organizacional e *Cloud*, disponibilizando endereços únicos, denominados *Webhooks*, pelos quais a plataforma *OpenSearch* poderá passar um pedido em formato JSON;
- *Custom Webhook*, refere-se a qualquer outro tipo de *Webhook* que disponibilize um endereço e com a capacidade de interpretar o formato da mensagem enviada, que tipicamente possuem um formato específico e incluem informações relativas ao detetor utilizado, ao *trigger* utilizado, à severidade da anomalia, o período da anomalias, entre outros;
- Email;
- *Amazon SNS*, um serviço de notificação disponibilizado como parte da *Amazon Web Services*;

3.3.3 Interface de Utilizador

Uma vez processada e guardada, a informação relativa às sessões de *log* necessita de ser visualizada e interpretada pelos utilizadores. O *Opensearch* possuí a interface *Opensearch Dashboards* [77] para complementar o seu *stack*, onde é possível interagir com

as diferentes funcionalidades, para gerir índices, configurar modelos de deteção de anomalias, configurar alertas e notificações, entre outras. Adicionalmente, é possível desenvolver *dashboards* para um ou vários índices, facilitando a visualização da informação. Na Figura 3.11 apresenta-se o *dashboard* desenvolvido no contexto do índice utilizado para a solução desenvolvida e que possuí as seguintes visualizações:

- Lista de parâmetros extraídos de padrões, uma lista extraída de padrões detetados na mensagem do evento *log* (por exemplo, na ocorrência "SENSOR=ON", o parâmetro extraído é "SENSOR");
- Lista de descrições associadas aos parâmetros extraídos, uma lista que resulta da extrações de descrições associados aos parâmetros em padrões detetados (por exemplo, na ocorrência "SENSOR=ON", a descrição é "ON");
- Evolução do valor numérico associado ao parâmetro extraído, demonstra a evolução numérica associada ao parâmetro (por exemplo a ocorrência "SENSOR=14"). A visualização apresenta a evolução do valor máximo, mínimo e médio;
- Lista de contextos *log*;
- Distribuição Temporal de *logs*;
- Lista de níveis *log*;
- Tabela de valores, que, neste caso, se divide em três colunas: contexto, mensagem e número de ocorrências;
- Lista de identificadores dos dispositivos;
- Contagem de *logs*;
- *Timestamp* mais recente.

3.4 Conclusão

O sistema de *logging* proposto foi definido para resolver a problemática de onde e que tipo de dados registar em *log* sobre o sistema *CardioWheel*. Dada a natureza deste sistema, foi de grande importância declarar eventos de *logging* para registar e seguir o fluxo do programa executado na *CardioWheel embedded board* na medição de sinais vitais do utilizador sem comprometer as componentes de *hardware* conhecidas como

sendo limitativas tal como o espaço de armazenamento, a capacidade da bateria ou capacidade de processamento do sistema. Por outro lado, foi definido como registar os dados de *logging*, na medida que a composição textual da mensagem tem de ser consistente entre eventos e amigável aos processos posteriores de *parsing*. Outra componente importante, que se insere no "como" apresentar a informação, respeita ao nível de detalhe associado aos eventos que, por sua vez, têm de ser definidos a respeito do tipo de evento que se deseja apresentar. Ainda sobre este nível de detalhe, foi também focal possibilitar diferentes configurações (como *thresholds*) para delinear o comportamento do sistema de *logging* entre os dois tipos de ambientes em que o sistema *CardioWheel* pode funcionar, i.e. desenvolvimento e produção. Tendo alinhado estes três tópicos, foi identificado o melhor tipo de armazenamento a introduzir na *CardioWheel embedded board* e o modo de comunicação com as restantes componentes do sistema, executadas na *Cloud*.

Para a segunda componente do sistema, foi utilizada a plataforma *OpenSearch* para auxiliar o processamento e a visualização posterior dos dados. Esta plataforma introduz algumas funcionalidades desejadas para o sistema *logging*, como deteção de anomalias e análise de origem de causas. O *OpenSearch* utiliza encriptação, autenticação e controlo de acesso para salvaguardar dados e monitorizar a atividade do *cluster*. Possuí ainda compatibilidade com diversos ambientes *Cloud*, com soluções transparentes à replicação, e é capaz de cumprir com os requerimentos analíticos definidos para uma boa monitorização remota de todo o sistema de *logging*.

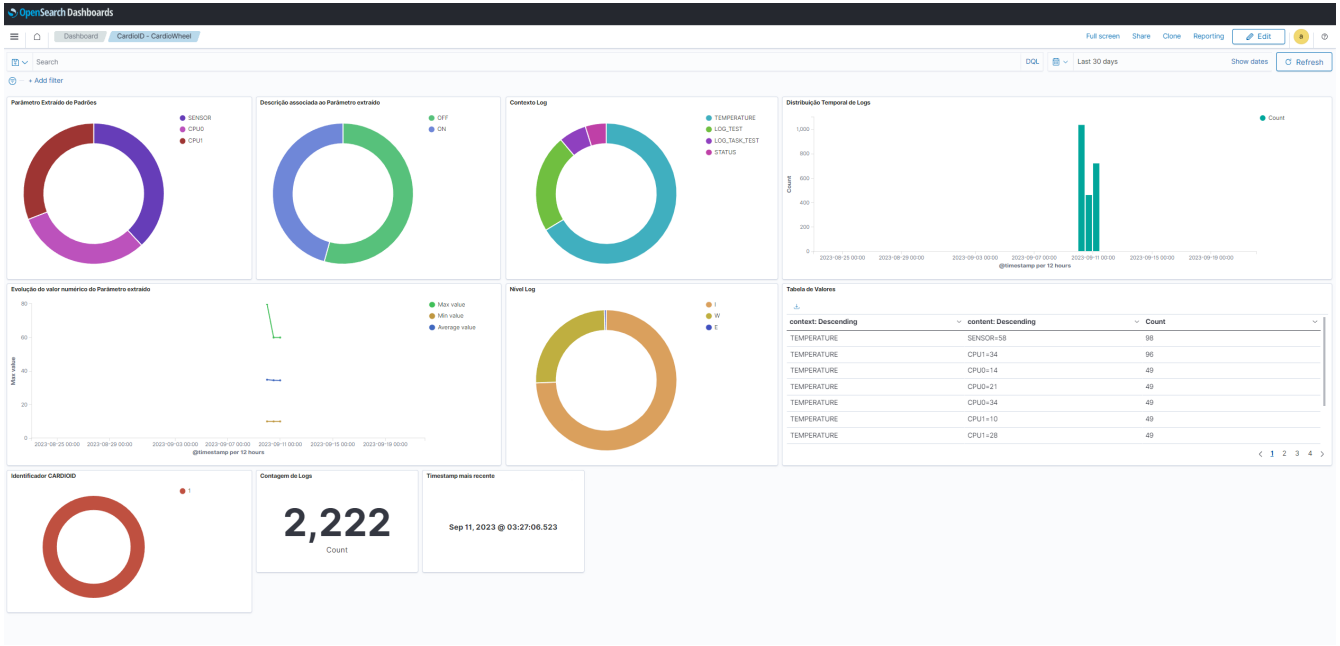


Figura 3.11: Exemplo da *Opensearch Dashboards*

4

Resultados Experimentais

De acordo com as necessidades da *CardioID*, espera-se uma sessão de *logging* de cerca de três horas sem interrupção. Na imagem seguinte demonstra-se que, utilizando a solução proposta, foi possível efetuar uma sessão de quatro horas de funcionamento, com um ritmo de quatro eventos *log* por segundo. Este processo não gerou qualquer falha do lado do dispositivo ou qualquer incoerência na informação introduzida nos ficheiros.

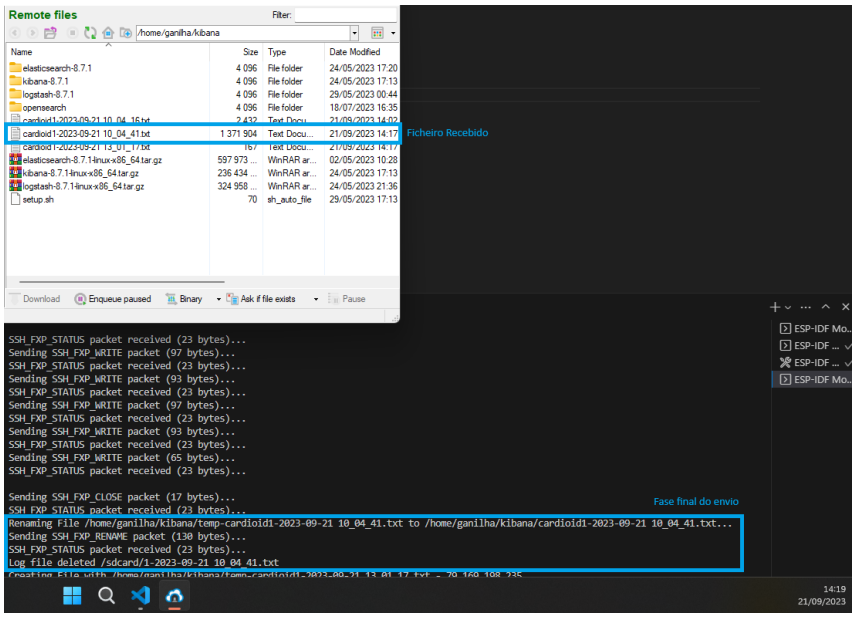


Figura 4.1: Simulação de uma sessão

Esta solução utiliza, em parte, funcionalidades robustas já disponibilizadas pela *framework* ESP-IDF e espera-se que possua a capacidade de efetuar sessões de maior duração sem degradação de qualidade. Como se verifica na imagem, o ficheiro de sessão possuí um tamanho de 1.4Gbs e a biblioteca *CycloneSSH* adaptada não mostrou qualquer dificuldade em transmitir todo o conteúdo guardado, com a inclusão de dois ficheiros de sessão pendentes. Neste ambiente de teste, o sistema terminou a tarefa, desde o inicio do processo de envio até a remoção do ficheiro em memória, em cerca de 15 minutos.

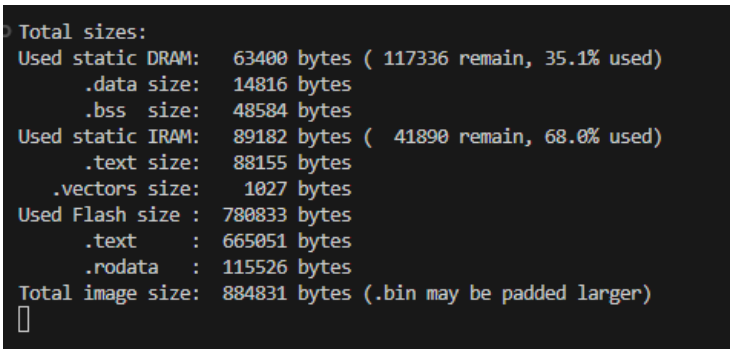
É importante identificar o tempo de processamento dedicado às ações introduzidas pela biblioteca *logging*, pelo que se espera uma duração média de 6 segundos para a fase de inicialização, dependendo do tempo de resposta por parte do servidor SNTP, e um período relativamente instantâneo, na ordem máxima de 1-2 ms, para o processamento de uma declaração *log*, um resultado equivalente em declarações *log* geradas sem as modificações efetuadas. No contexto crítico em que se encontra o projeto, estes valores correspondem a períodos aceitáveis no decorrer de uma sessão na medida que o funcionamento do dispositivo não é particularmente afetado.

No que diz respeito ao tamanho das declarações *log*, a componente estática do evento pode-se resumir a 31 bytes para o valor mínimo que o *TIMER* pode assumir e 40 bytes para o valor máximo, sendo este 4,294,967,295 ms (aproximadamente 50 dias). A soma total do tamanho da declaração poderá ser posteriormente calculada pela adição da *TAG*, *ID* e *MESSAGE*. No decorrer de uma sessão *logging* espera-se, como pior caso, um evento de dimensão de 105 bytes, que se refere à informação de inicialização da biblioteca ("SNTP:NOK MEM:NOK ACEL:NOK ECG:NOK WIFI:NOK BLE:NOK CPU:NOK"), contexto *INIT*, o número máximo que o identificador pode tomar (3 bytes) e componente estática para um valor máximo do *TIMER*.

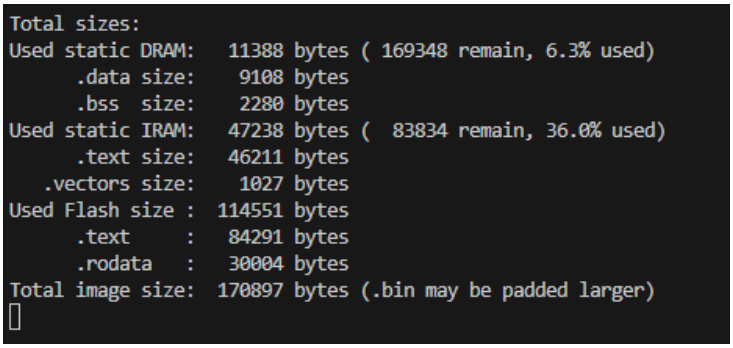
É necessário dimensionar a velocidade de escrita de um cartão de memória capaz de suportar a escrita do evento de maior dimensão no período mínimo previsto. Com recurso à função *cpu_hal_get_cycle_count()* disponibilizada pela plataforma ESP-IDF obteve-se um contagem de ciclos do processador mínima de 41000, entre duas sincronizações com o ficheiro. Tendo em conta a frequência de funcionamento do processador de 160 Mhz obteve-se um tempo estimado de 0.26 ms via a equação:

$$f[Hz] = \frac{cycles}{t[s]}$$

Assumindo 104 bytes de dimensão máxima de um evento *log*, é necessário um cartão de memória com um ritmo mínimo de 0.4 Mb/s de escrita.



(a)



(b)

Figura 4.2: (a) Compilação com a solução. (b) Compilação sem a solução.

A quantidade de código que é possível introduzir é limitada em dispositivos IoT e depende da disponibilidade da *Dynamic Random-Access Memory* (DRAM) e da *Internal Random-Access Memory* (IRAM). A biblioteca *logging* efetua funções não essenciais para o contexto da CardioWheel, pelo que, a sua introdução não deverá ocupar recursos de uma forma excessiva. Na imagem que se segue, verifica-se um aumento de 28.8% na utilização de DRAM e de 32% na utilização de IRAM após a introdução da biblioteca desenvolvida. Este aumento deve-se principalmente á introdução da biblioteca *CycloneSSH*, dada a complexidade associada a uma solução capaz de garantir segurança e integridade durante o envio remoto de informação. Desta forma, resta uma capacidade de 64.9% de DRAM e de 32% de IRAM para uso nas restantes funcionalidades.

É, ainda, de grande importância determinar o impacto energético que a solução coloca sob os dispositivos IoT. A figura demonstra o ritmo de acumulação energético, em mWh, do dispositivo, com e sem a solução introduzida, no decorrer de um sessão de *logging* de 30 minutos, com um ritmo de 4 declarações *log* por segundo. Em ambos casos foram efetuadas 5 sessões de *logging* para cada cenário, pelo que os resultados obtidos referem a média resultante. As evoluções observadas, denotam-se duas fases de consumo: uma fase inicial de maior contribuição energética, com uma duração aproximada de 2 minutos e onde a corrente elétrica varia entre 0.050 A e 0.055 A

com a solução e 0.043 A e 0.044 A sem a solução, e uma fase posterior que varia entre 0.047 A e 0.050 A com a solução e 0.041 A e 0.043 A sem a solução. Considerando um ritmo constante nesta ultima fase, espera-se um aumento de aproximadamente 14% no consumo energético de uma sessão de *logging* com duração de 3 horas, não considerando o envio do ficheiro de sessão. Por outro lado, uma sessão de *logging*, com duração de 5 minutos e dedicada apenas ao envio de informação, contribuí em média 50 mWh, sendo que a corrente elétrica varia entre 0.135 A e 0.200 A durante os primeiros 2 minutos e aproximadamente constante posteriormente, com uma média de 0.110 A. Estima-se que os valores superiores de corrente observados, no inicio de cada sessão, estejam relacionados com processos de inicialização do *ESP32* conjugados com as restantes funcionalidade implementadas.

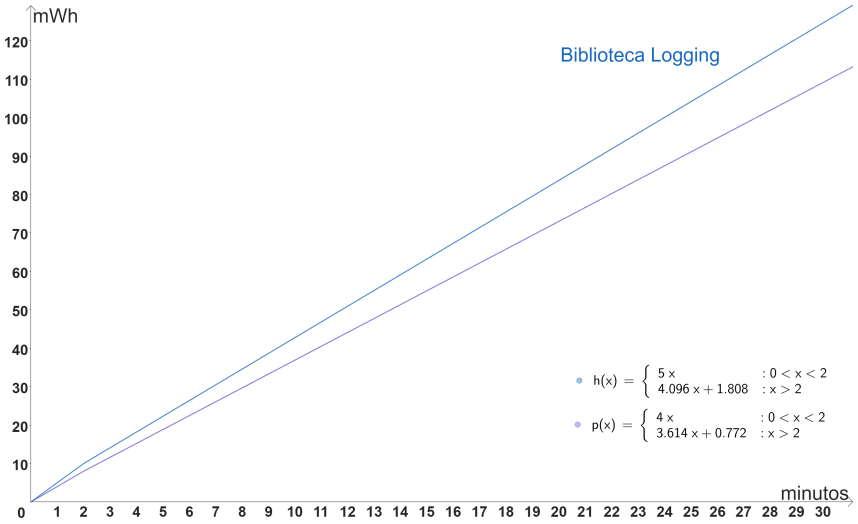


Figura 4.3: Consumo

4.1 Ambiente Experimental

Para efetuar a comunicação entre o dispositivo e as entidades externas, neste caso o túnel SSH e o servidor SNTP, foi utilizado um *hotspot* com um ritmo de *upload* superior a 1Mbps. As instâncias da componente *Cloud* foram executadas num computador *Intel Corporation Xeon E3-1200 v3/4th Gen Core Processor Integrated Graphics Controller* com 8Gbs de memória *RAM*. O sistema operativo utilizado corresponde ao *Ubuntu* 20.04.3 LTS, uma distribuição do *Linux*. O computador disponibiliza um serviço SSH que, por sua vez, é exposto no porto 226 via *port forwarding* num GIGA ROUTER NOS. Adicionalmente, encontram-se instanciadas as seguintes ferramentas, em modo *standalone*:

Instância	Versão	RAM	CPU	Armazenamento
Logstash	8.6.1	700Mbs	1-10%	800Mbs
Opensearch	2.9.0	1.2Gbs	10-30%	1.16Gbs
Opensearch Dashboards	2.9.0	200Mbs	1-30%	1.34Gbs
MySQL	5	500Mbs	1-10%	1.1Gbs
Tomcat	9	500Mbs	1-10%	500Mbs
Docker	20.10.24	-	-	500Mbs

Tabela 4.1: Especificações das ferramentas

4.2 Trabalho Futuro

Dada as funcionalidades do sistema desenvolvido, a adição do *timestamp* na declaração *log* identifica um ponto de redundância na implementação da solução *logging*, na medida que a declaração *log* gerada pela *framework* ESP-IDF possuí um temporizador de execução do sistema (*TIMER*) que permite contextualizar, com facilidade, o *timestamp* de 3 horas de uma sessão *logging*. O *TIMER* pode ainda ser calculado pelo ciclo do *hardware* (referido no capítulo anterior) para persistir mesmo em caso de falhas inesperadas que forçam o dispositivo a reiniciar dando, assim, continuidade à sessão de *logging*. De forma a aderir à remoção do *timestamp* como constituinte da declaração *log*, poderá ser introduzido, como um primeiro evento do ficheiro de sessão, o *timestamp* referente ao inicio da sessão de *logging*. A ferramenta de *parsing*, neste caso o *Logstash*, poderá, posteriormente, utilizar este primeiro *timestamp* em conjunto com o temporizador de execução do sistema para diferir os *timestamps* de cada evento subsequente. Tendo em conta o tamanho da componente estática, com a remoção do *timestamp* são retirados 22 bytes de comprimento, obtendo 9 bytes para o valor mínimo do *TIMER* e 18 bytes para o máximo. Pela comparação da componente estática de uma declaração *log* constituída apenas pelo *timestamp* com uma declaração *log* constituída apenas pelo *TIMER*, sendo este o maior valor que pode tomar, existe, também, uma redução de 9 bytes de comprimento. O identificar do dispositivo (*ID*) é um outro parâmetro alvo de redundância dado que, tipicamente, se encontra constante durante uma sessão de *logging* e, portanto, poderá ser introduzido no inicio do ficheiro de sessão para posterior processamento na fase de *parsing* da informação. A remoção do identificador resulta numa redução de $x + 1$ bytes, sendo x o tamanho do identificador em bytes.

Uma outra problemática encontra-se direcionada à utilização DRAM e IRAM, que se encontra relacionada com as instruções de código introduzidas na solução. A questão surge da impossibilidade de comparar os recursos alocados pelas funcionalidades essenciais da CardioWheel com os recursos disponíveis após a introdução da biblioteca

desenvolvida. Caso os recursos disponibilizados sejam insuficientes, poderá ser necessário utilizar uma alternativa à biblioteca *CycloneSSH* ou refatorar a abordagem para reduzir a utilização DRAM e IRAM.

A sincronização da informação *log* na solução é efetuada através do comando *fsync()*, que envia as alterações no ficheiro presentes em memória internar para a memória externa. A utilização deste comando garante a integridade da informação do ficheiro de sessão, no entanto, dado que componentes de memória (como o cartão SD ou *Solid-State Drive* (SSD)) se distribuem em segmentos com um número limite de ciclos remoção, não garante uma distribuição equilibrada por todos os segmentos da memória que, por sua vez, pode diminuir o tempo de vida da componente. Uma das soluções que se sugere para prolongar a durabilidade da memória externa é a adição do *Flash-Friendly File System* (F2FS) na *framework* ESP-IDF, um sistema de ficheiros desenvolvido com a longevidade da memória em mente. Como alternativa, deverá ser introduzido um mecanismo de equilíbrio para os sistemas de ficheiros disponíveis na biblioteca.

5

Conclusões e Trabalho Futuro

Logs são universalmente utilizados para monitorizar e diagnosticar problemas e falhas em sistemas informáticos, em aplicações funcionais, tanto em ambiente de produção como em ambiente de teste, nomeadamente quando se trata de áreas críticas, como a da saúde. A introdução de um sistema de *logging* em dispositivos IoT é uma área pouco explorada e continua a ser alvo de discussão face à natureza limitativa do ecossistema.

A presente dissertação propõe umas das primeiras abordagens a um sistema de *logging* completo dedicado aos ecossistemas IoT. A proposta foi desenvolvida pela conjugação de diversas ferramentas disponibilizadas no mercado, desde ao sistema operativo FreeRTOS com a *framework* ESP-IDF, que disponibiliza um núcleo pelo qual desenvolver a componente embebida e efetuar as adições necessárias para cumprir com os requisitos limitativos dos dispositivos, ao ambiente *Cloud*, que proporciona uma solução escalável e desacoplada, para onde migrar e efetuar o processamento e visualização previamente indisponível na componente embebida.

Para que o sistema cumpra com os objetivos delimitados pela *CardioID* ou por qualquer outra entidade futura, é de grande importância cumprir as práticas definidas nas secções que abordam *Where to Log?*, *What to Log?* e *How to Log?*, principalmente a respeito dos padrões utilizados na mensagem do evento, que devem estar sincronizados com o esperado pela ferramenta de *parsing*. Em ambiente *Cloud*, as tecnologias utilizadas não devem ser executadas em modo *standalone*, para tornar a solução escálavel e resistente a oscilações de *payloads* sem comprometer desempenho. O modelo de deteção de anomalias terá, também, de ser bem estudado e preparado de acordo com o parâmetro, ou

parâmetros, necessários para determinado tipo de anomalias. Estes alertas são transversais a qualquer contexto aplicacional e devem ser respeitados de forma a garantir um comportamento ideal no sistema de *logging*.

Por outro lado, o sistema desenhado não constitui uma solução perfeita, o aumento de tempo de processamento do evento de *logging*, devido á obtenção do *timestamp*, é alarmante e indesejável. Em sistemas que requerem um ritmo de *logging* superior, a probabilidade de degradação e ocupação do processador é maior e pode afetar a execução de outras tarefas mais importantes, algo que deverá ser evitado, especialmente num contexto mais crítico, como o da *CardioWheel*. De uma forma semelhante ao desacoplamento efetuado para executar tarefas mais intensivas em ambiente *Cloud*, como *parsing*, o *timestamp* poderá ser diferido no momento do *parsing*, pela contextualização do temporizador de execução do sistema com um *timestamp* inicial enviado no inicio do ficheiro de sessão. Esta alteração resulta, também, numa redução de 25% do tamanho da declaração *log*. O *Webhook* desenvolvido para receber as notificações enviadas pela plataforma *Opensearch* representa um possível ponto de melhoria na solução apresentada, um aplicativo móvel poderá ser utilizado para consultar ativamente a tabela de notificações, ou, como alternativa, implementar uma infraestrutura para envio e receção de emails. Uma outra melhoria seria dedicada a migrar a plataforma *Opensearch* para a plataforma *ElasticSearch* que, embora se encontre sob uma subscrição paga, oferece uma solução mais robusta com a adição de mecanismos para análise de origem de causas e previsão de falhas, porém, o desenvolvimento de *plugins* para a plataforma *Opensearch* também apresenta uma opção viável.

Referências

[1] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs", em *Proceedings of the 3rd IEEE Workshop on IP Operations Management (IPOM 2003) (IEEE Cat. No.03EX764)*, 2003, páginas 119–126. DOI: 10.1109/IPOM.2003.1251233.

[2] D. Moberg, "MIME-Based Secure Peer-to-Peer Business Data Interchange Using HTTP, Applicability Statement 2 (AS2)", jan. de 2005.

[3] Jacob Benesty, Jingdong Chen, Yiteng Huang & Israel Cohen, "Pearson Correlation Coefficient", em *Noise Reduction in Speech Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, páginas 1–4. DOI: 10.1007/978-3-642-00296-0_5.

[4] António Cerca, Andre Lourenco & Artur Ferreira, "A CardioWheel-based Fatigue and Drowsiness Detection System", vol. 8, páginas –, set. de 2022. DOI: 10.34629/ipl.isel.i-ETC.85.

[5] *Chime*. URL: https://www.chime.com.

[6] *Cisco AppDynamics*. URL: https://docs.appdynamics.com.

[7] *CycloneSSH*. URL: https://oryx-embedded.com/products/CycloneSSH.html.

[8] *Docker*. URL: https://docs.docker.com.

[9] *Dynatrace*. URL: https://www.dynatrace.com/support/help/get-started/what-is-dynatrace.

[10] *Elastic Stack*. URL: https://www.elastic.co/guide/index.html.

[11] Ding Yuan, Soyeon Park & Yuanyuan Zhou, "Characterizing logging practices in open-source software", em *2012 34th International Conference on Software Engineering (ICSE)*, 2012, páginas 102–112. DOI: 10.1109/ICSE.2012.6227202.

[12] Boyuan Chen & Zhen Jiang, "Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation", *Empirical Software Engineering*, vol. 22, fev. de 2017. DOI: 10.1007/s10664-016-9429-5.

[13] J.C. Munson & S.G. Elbaum, "Code churn: a measure for estimating the impact of code change", em *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998, páginas 24–31. DOI: 10.1109/ICSM.1998.738486.

[14] Yi Zeng, Jinfu Chen, Weiyi Shang & Tse-Hsun (Peter) Chen, "Studying the characteristics of logging practices in mobile apps: a case study on F-Droid", *Empirical Software Engineering*, vol. 24, n.º 6, páginas 3394–3434, 2019. DOI: 10.1007/s10664-019-09687-9.

[15] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza & Domenico Cotroneo, "Industry Practices and Event Logging: Assessment of a Critical Software Development Process", em *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, páginas 169–178. DOI: 10.1109/ICSE.2015.145.

[16] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu & Tao Xie, "An Exploratory Study of Logging Configuration Practice in Java", em *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, páginas 459–469. DOI: 10.1109/ICSME.2019.00079.

[17] Weiyi Shang, Meiyappan Nagappan & Ahmed E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software", *Empirical Software Engineering*, vol. 20, n.º 1, páginas 1–27, 2015. DOI: 10.1007/s10664-013-9274-8.

[18] *EspressIF Biblioteca Logging*. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/log.html.

[19] Joao A. Duraes & Henrique S. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach", *IEEE Transactions on Software Engineering*, vol. 32, n.º 11, páginas 849–867, 2006. DOI: 10.1109/TSE.2006.113.

[20] Ivanilton Polato, Reginaldo Ré, Alfredo Goldman & Fabio Kon, "A comprehensive view of Hadoop research—A systematic literature review", *Journal of Network and Computer Applications*, vol. 46, páginas 1–25, 2014, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2014.07.022.

[21] George Danezis, *Traffic Analysis of the HTTP Protocol over TLS*, 2009.

[22] Jeanderson Cândido, *Log-based software monitoring: a systematic mapping study*. IEEE, 2021. URL: https://peerj.com/articles/cs-489/.

[23] Fei Tony Liu, Kai Ming Ting & Zhi-Hua Zhou, "Isolation-Based Anomaly Detection", vol. 6, n.º 1, 2012. DOI: 10.1145/2133360.2133363.

[24] *Kafka*. URL: https://kafka.apache.org/documentation/.

[25] Adam Oliner & Jon Stearley, "What Supercomputers Say: A Study of Five System Logs", em *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, 2007, páginas 575–584. DOI: 10.1109/DSN.2007.103.

[26] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang & Xuewei Chen, "Log Clustering Based Problem Identification for Online Service Systems", em *Proceedings of the 38th International Conference on Software Engineering Companion*, New York, NY, USA: Association for Computing Machinery, 2016, 102–111, ISBN: 9781450342056. DOI: 10.1145/2889160.2889232. URL: https://doi.org/10.1145/2889160.2889232.

[27] Wei Xu, Ling Huang, Armando Fox, David Patterson & Michael I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs", em *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, New York, NY, USA: Association for Computing Machinery, 2009, 117–132, ISBN: 9781605587523. DOI: 10.1145/1629575.1629587. URL: https://doi.org/10.1145/1629575.1629587.

[28] Min Du, Feifei Li, Guineng Zheng & Vivek Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning", em *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, 1285–1298. DOI: 10.1145/3133956.3134015.

[29] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun & Rong Zhou, "LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs", em *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, International Joint Conferences on Artificial Intelligence Organization, jul. de 2019, páginas 4739–4745. DOI: 10.24963/ijcai.2019/658.

[30] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen & Dongmei Zhang, "Robust Log-Based Anomaly Detection on Unstable Log Data", em *Proceedings*

of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, 807–817. DOI: 10.1145/3338906.3338931.

[31] Siyang Lu, BingBing Rao, Xiang Wei, Byungchul Tak, Long Wang & Liqiang Wang, "Log-based Abnormal Task Detection and Root Cause Analysis for Spark", em *2017 IEEE International Conference on Web Services (ICWS)*, 2017, páginas 389–396. DOI: 10.1109/ICWS.2017.135.

[32] Chinghway Lim, Navjot Singh & Shalini Yajnik, "A log mining approach to failure analysis of enterprise telephony systems", em *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, páginas 398–403. DOI: 10.1109/DSN.2008.4630109.

[33] Christophe Bertero, Matthieu Roy, Carla Sauvanaud & Gilles Tredan, "Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection", em *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, páginas 351–360. DOI: 10.1109/ISSRE.2017.43.

[34] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber & John Grundy, "Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis", em *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, páginas 24–34. DOI: 10.1109/ISSRE.2015.7381796.

[35] Shilin He, Jieming Zhu, Pinjia He & Michael R. Lyu, "Experience Report: System Log Analysis for Anomaly Detection", em *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, páginas 207–218. DOI: 10.1109/ISSRE.2016.21.

[36] Marcello Cinque, Domenico Cotroneo, Roberto Natella & Antonio Pecchia, "Assessing and improving the effectiveness of logs for the analysis of software faults", em *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, páginas 457–466. DOI: 10.1109/DSN.2010.5544279.

[37] Marcello Cinque, Domenico Cotroneo & Antonio Pecchia, "Event Logs for the Analysis of Software Failures: A Rule-Based Approach", *IEEE Transactions on Software Engineering*, vol. 39, n.º 6, páginas 806–821, 2013. DOI: 10.1109/TSE.2012.67.

[38] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou & Stefan Savage, "Improving Software Diagnosability via Log Enhancement", vol. 30, n.º 1, 2012. DOI: 10.1145/2110356.2110360.

[39] J. Wang, C. Li, S. Han, S. Sarkar & X. Zhou, "Predictive maintenance based on event-log analysis: A case study", *IBM Journal of Research and Development*, vol. 61, n.º 1, 11:121–11:132, 2017. DOI: 10.1147/JRD.2017.2648298.

[40] Mehran Hassani, Weiyi Shang, Emad Shihab & Nikolaos Tsantalis, "Studying and Detecting Log-Related Issues", vol. 23, n.º 6, 2018. DOI: 10.1007/s10664-018-9603-z.

[41] Shanshan Li, Xu Niu, Zhouyang Jia, Xiangke Liao, Ji Wang & Tao Li, "Guiding log revisions by learning from software evolution history", *Empirical Software Engineering*, vol. 25, n.º 3, páginas 2302–2340, 2020. DOI: 10.1007/s10664-019-09757-y.

[42] Boyuan Chen & Zhen Ming Jiang, "Characterizing and Detecting Anti-Patterns in the Logging Code", em *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, páginas 71–81. DOI: 10.1109/ICSE.2017.15.

[43] Zhenhao Li, Tse-Hsun Chen, Jinqiu Yang & Weiyi Shang, "DLFinder: Characterizing and Detecting Duplicate Logging Code Smells", em *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, páginas 152–163. DOI: 10.1109/ICSE.2019.00032.

[44] Pinjia He, Zhuangbin Chen, Shilin He & Michael R. Lyu, "Characterizing the Natural Language Descriptions in Software Logging Statements", em *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, páginas 178–189. DOI: 10.1145/3238147.3238193.

[45] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan & Shanping Li, "Which Variables Should I Log?", *IEEE Transactions on Software Engineering*, vol. 47, n.º 9, páginas 2012–2031, 2021. DOI: 10.1109/TSE.2019.2941943.

[46] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang & Tao Xie, "Where Do Developers Log? An Empirical Study on Logging Practices in Industry", em *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, 24–33. DOI: 10.1145/2591062.2591175.

[47] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu & Dongmei Zhang, "Learning to Log: Helping Developers Make Informed Logging Decisions", em *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, páginas 415–425. DOI: 10.1109/ICSE.2015.60.

[48] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang & Ahmed E. Hassan, "Studying software logging using topic models", *Empirical Software Engineering*, vol. 23, n.º 5, páginas 2655–2694, 2018. DOI: 10.1007/s10664-018-9595-8.

[49] Heng Li, Weiyi Shang & Ahmed E. Hassan, "Which log level should developers choose for a new logging statement?", *Empirical Software Engineering*, vol. 22, n.º 4, páginas 1684–1716, 2017. DOI: 10.1007/s10664-016-9456-2.

[50] Han Anu, Jie Chen, Wenchang Shi, Jianwei Hou, Bin Liang & Bo Qin, "An Approach to Recommendation of Verbosity Log Levels Based on Logging Intention", em *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, páginas 125–134. DOI: 10.1109/ICSME.2019.00022.

[51] Michal Aharon, Gilad Barash, Ira Cohen & Eli Mordechai, "One Graph Is Worth a Thousand Logs: Uncovering Hidden Structures in Massive System Event Logs", em *Machine Learning and Knowledge Discovery in Databases*, 2009, páginas 227–243. DOI: 10.1007/978-3-642-04180-8_32.

[52] Adetokunbo A.O. Makanju, A. Nur Zincir-Heywood & Evangelos E. Milios, "Clustering Event Logs Using Iterative Partitioning", 2009, 1255–1264. DOI: 10.1145/1557019.1557154.

[53] Yinglung Liang, Yanyong Zhang, Hui Xiong & Ramendra Sahoo, "An Adaptive Semantic Filter for Blue Gene/L Failure Log Analysis", em *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, páginas 1–8. DOI: 10.1109/IPDPS.2007.370635.

[54] Ana Gainaru, Franck Cappello, Stefan Trausan-Matu & Bill Kramer, "Event Log Mining Tool for Large Scale HPC Systems", em *Euro-Par 2011 Parallel Processing*, 2011, páginas 52–64. DOI: 10.1007/978-3-642-23400-2_6.

[55] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang & Abdullah Mueen, "LogMine: Fast Pattern Recognition for Log Analytics", em *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, 1573–1582. DOI: 10.1145/2983323.2983358.

[56] Pin Zhou, Binny Gill, Wendy Belluomini & Avani Wildani, "GAUL: Gestalt Analysis of Unstructured Logs for Diagnosing Recurring Problems in Large Enterprise Storage Systems", em *2010 29th IEEE Symposium on Reliable Distributed Systems*, 2010, páginas 148–159. DOI: 10.1109/SRDS.2010.25.

[57] Liang Tang & Tao Li, "LogTree: A Framework for Generating System Events from Raw Textual Logs", em *2010 IEEE International Conference on Data Mining*, 2010, páginas 491–500. DOI: 10.1109/ICDM.2010.76.

[58] Pinjia He, Jieming Zhu, Shilin He, Jian Li & Michael R. Lyu, "An Evaluation Study on Log Parsing and Its Use in Log Mining", em *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, páginas 654–661. DOI: 10.1109/DSN.2016.66.

[59] Pinjia He, Jieming Zhu, Shilin He, Jian Li & Michael R. Lyu, "Towards Automated Log Parsing for Large-Scale Log Data Analysis", *IEEE Transactions on Dependable and Secure Computing*, vol. 15, n.º 6, páginas 931–944, 2018. DOI: 10.1109/TDSC.2017.2762673.

[60] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H. Chin & Sumayah Alrwais, "Detection of Early-Stage Enterprise Infection by Mining Large-Scale Log Data", em *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, páginas 45–56. DOI: 10.1109/DSN.2015.14.

[61] Jie Chu, Zihui Ge, Richard Huber, Ping Ji, Jennifer Yates & Yung-Chao Yu, "ALERT-ID: Analyze Logs of the Network Element in Real Time for Intrusion Detection", vol. 7462, set. de 2012, páginas 294–313, ISBN: 978-3-642-33337-8. DOI: 10.1007/978-3-642-33338-5_15.

[62] Eunjung Yoon & Anna Squicciarini, "Toward Detecting Compromised MapReduce Workers through Log Analysis", em *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, páginas 41–50. DOI: 10.1109/CCGrid.2014.120.

[63] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels & Engin Kirda, "Beehive: Large-Scale Log Analysis for Detecting Suspicious Activity in Enterprise Networks", em *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, 199–208. DOI: 10.1145/2523649.2523670.

[64] E.L. Barse & E. Jonsson, "Extracting attack manifestations to determine log data requirements for intrusion detection", em *20th Annual Computer Security Applications Conference*, 2004, páginas 158–167. DOI: 10.1109/CSAC.2004.20.

[65] C. Abad, J. Taylor, C. Sengul, W. Yurcik, Y. Zhou & K. Rowe, "Log correlation for intrusion detection: a proof of concept", em *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, 2003, páginas 255–264. DOI: 10.1109/CSAC.2003.1254330.

[66] Tatsuaki Kimura, Keisuke Ishibashi, Tatsuya Mori, Hiroshi Sawada, Tsuyoshi Toyono, Ken Nishimatsu, Akio Watanabe, Akihiro Shimoda & Kohei Shiomoto, "Spatio-temporal factorization of log data for understanding network events",

em *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, 2014, páginas 610–618. DOI: 10.1109/INFOCOM.2014.6847986.

[67] Martin Ester, Hans-Peter Kriegel, Jörg Sander & Xiaowei Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise", em *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, AAAI Press, 1996, 226–231.

[68] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang & Tao Xie, "Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing", em *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, páginas 527–538. DOI: 10.1109/ASE.2019.00056.

[69] Aidi Pi, Wei Chen, Xiaobo Zhou & Mike Ji, "Profiling Distributed Systems in Lightweight Virtualized Environments with Logs and Resource Metrics", em *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, New York, NY, USA: Association for Computing Machinery, 2018, 168–179, ISBN: 9781450357852. DOI: 10.1145/3208040.3208044. URL: https://doi.org/10.1145/3208040.3208044.

[70] Edward Chuah, Arshad Jhumka, Sai Narasimhamurthy, John Hammond, James C. Browne & Bill Barth, "Linking Resource Usage Anomalies with System Failures from Cluster Log Data", em *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, 2013, páginas 111–120. DOI: 10.1109/SRDS.2013.20.

[71] Edward Chuah, Gary Lee, William-Chandra Tjhi, Shyh-Hao Kuo, Terence Hung, John Hammond, Tommy Minyard & James C. Browne, "Establishing Hypothesis for Recurrent System Failures from Cluster Log Files", em *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, 2011, páginas 15–22. DOI: 10.1109/DASC.2011.27.

[72] Ziming Zheng, Li Yu, Wei Tang, Zhiling Lan, Rinku Gupta, Narayan Desai, Susan Coghlan & Daniel Buettner, "Co-analysis of RAS Log and Job Log on Blue Gene/P", em *2011 IEEE International Parallel and Distributed Processing Symposium*, 2011, páginas 840–851. DOI: 10.1109/IPDPS.2011.83.

[73] Junaid Mohammed, Chung-Horng Lung, Adrian Ocneanu, Abhinav Thakral, Colin Jones & Andy Adler, "Internet of Things: Remote Patient Monitoring Using Web Services and Cloud Computing", em *2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom)*, 2014, páginas 256–263. DOI: 10.1109/iThings.2014.45.

[74] Ilias Mavridis & Helen Karatza, "Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark", *Journal of Systems and Software*, vol. 125, páginas 133–151, 2017. DOI: 10.1016/j.jss.2016.11.037.

[75] Hao Lin, Jingyu Zhou, Bin Yao, Minyi Guo & Jie Li, "Cowic: A Column-Wise Independent Compression for Log Stream Analysis", em *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, páginas 21–30. DOI: 10.1109/CCGrid.2015.45.

[76] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng & Michael R. Lyu, "Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression", em *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, páginas 863–873. DOI: 10.1109/ASE.2019.00085.

[77] *OpenSearch*. URL: https://dattell.com/data-architecture-blog/opensearch-vs-elasticsearch/.

[78] *OpenStack*. URL: https://docs.openstack.org/2023.2/?_ga=2.137590584.909952709.1702222272-626289166.1702222272.

[79] *OpenTelemetry*. URL: https://opentelemetry.io/docs/.

[80] *Open Zipkin*. URL: https://github.com/openzipkin/zipkin.

[81] *OverOps*. URL: https://doc.overops.com/docs.

[82] Sudipto Guha, Nina Mishra, Gourav Roy & Okke Schrijvers, "Robust Random Cut Forest Based Anomaly Detection on Streams", em *International Conference on Machine Learning*, 2016. URL: https://api.semanticscholar.org/CorpusID:927435.

[83] SSH File Transfer Protocol, "Secure Shell Working Group J. Galbraith Internet-Draft VanDyke Software Expires: January 11, 2007 O. Saarenmaa F-Secure July 10, 2006", 2006.

[84] *ShiViz*. URL: https://bestchai.bitbucket.io/shiviz/.

[85] *Slack*. URL: https://slack.com.

[86] Alexandre da Silva Veith & Marcos Dias de Assuncao, "Apache Spark", em *Encyclopedia of Big Data Technologies*, Sherif Sakr & Albert Y. Zomaya, eds. Cham: Springer International Publishing, 2019, páginas 77–81, ISBN: 978-3-319-77525-8. DOI: 10.1007/978-3-319-77525-8_37.

[87] *Teams*. URL: https://www.microsoft.com/pt-pt/microsoft-teams/log-in.

[88] *TICK Stack*. URL: https://wiki.archlinux.org/title/TICK_stack.

[89] *Trace Compass*. URL: https://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/Overview.html#Overview.