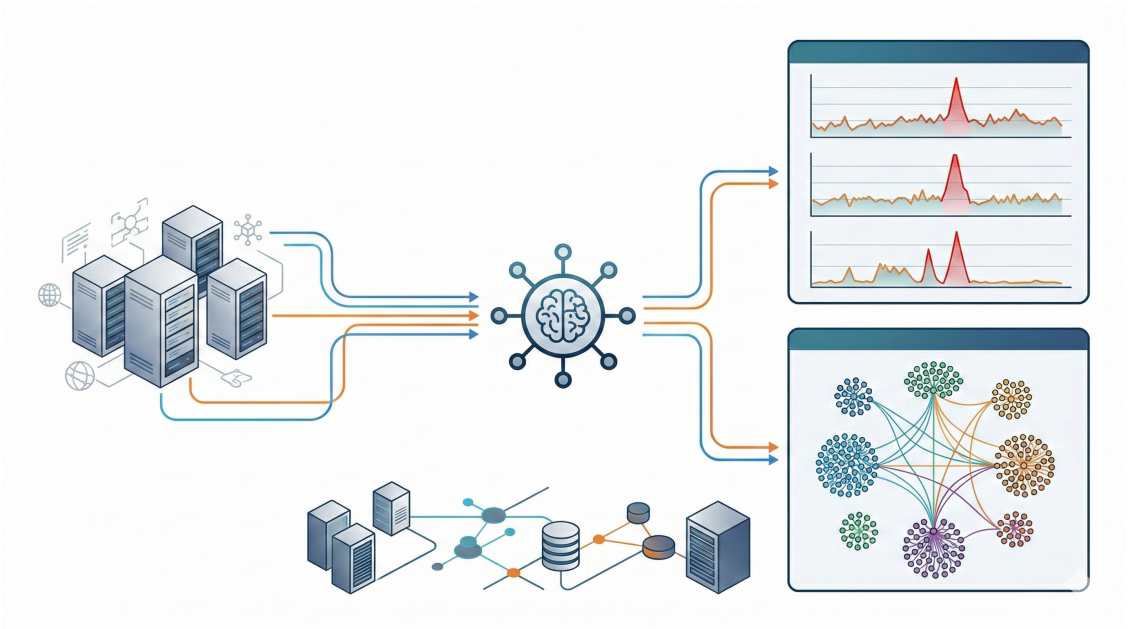
CHALMERS
UNIVERSITY OF TECHNOLOGY



AI-driven Infrastructure Log Analysis

Transitioning from Reactive Monitoring to Proactive Mitigation in 5G Cloud Native Cores

Thesis in Engineering Physics

ABDUR ARSHAD
MUHAMMAD ABDULLAH ARSHAD

DEPARTMENT OF ENGINEERING PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Degree project report 2026

AI-driven Infrastructure Log Analysis

Transitioning from Reactive Monitoring to Proactive Mitigation in 5G Cloud Native Cores

MUHAMMAD ABDULLAH ARSHAD
ABDUR ARSHAD

CHALMERS
UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF ENGINEERING PHYSICS
Chalmers University of Technology
Gothenburg, Sweden 2026

AI-driven Infrastructure Log Analysis.
Transitioning from Reactive Monitoring to Proactive Mitigation in 5G Cloud Native Cores
MUHAMMAD ABDULLAH ARSHAD, ABDUR ARSHAD

Industrial Supervisor: Martin Svensson, Ericsson
Examiner: Ebru Turanoglu Bekar, Department of Industrial and Materials Science at Chalmers University of Technology.

Degree project report 2026
Department of Engineering Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Visualization of log clustering and anomaly detection patterns within the Ericsson CNC environment. Made by using Google Gemini.

Typeset in LaTeX
Gothenburg, Sweden 2026

AI-driven Infrastructure Log Analysis

Transitioning from Reactive Monitoring to Proactive Mitigation in 5G Cloud Native Cores

MUHAMMAD ABDULLAH ARSHAD, ABDUR ARSHAD

Department of Engineering Physics
Chalmers University of Technology

Abstract

Cloud-native 5G core networks decompose monolithic network functions into hundreds of containerised microservices orchestrated by Kubernetes, producing thousands of infrastructure events per test run. Identifying genuine anomalies among this volume is difficult because the same warning event can be benign or critical depending on its sequential context, because labelled failure data is scarce, and because traditional threshold-based monitoring cannot capture the temporal dependencies that define healthy system behaviour. This thesis addresses that problem by proposing and evaluating a self-supervised, situation-aware log analysis system that learns normal Kubernetes infrastructure event patterns and detects deviations without labelled failure data. The approach formulates anomaly detection as a next-event prediction task. A deep learning model is trained to predict the next event type in a pod's lifecycle sequence, and events whose actual reason has low predicted probability receive high anomaly scores. Two architectures are compared, a Transformer encoder and a Long Short-Term Memory (LSTM) network, using learned categorical embeddings of structured event metadata rather than free-text message embeddings. This removes the data leakage identified in an earlier prototype. A rule-based post-processing layer then uses Kubernetes domain knowledge to separate startup noise from genuine operational issues, classify pod-level situations, and rank infrastructure problems by severity. The system is trained on 86,873 events from 36 test runs that span stability, robustness and chaos engineering experiments. It is evaluated on held-out test data as well as on 5 completely unseen test suites. Both architectures reach approximately 86% next-event prediction accuracy (86.6% Transformer, 86.1% LSTM), substantially above a 72.9% bigram baseline, and generalise to unseen failure modes at 89.3% accuracy. An end-to-end evaluation against 6 fault-injection scenarios, including 2 blind cases collected after the system was finalised, shows that the tool correctly identifies the primary affected components in every case. From an industrial perspective, the prototype enables Ericsson to automate post-test infrastructure triage that is currently performed manually, reducing the number of events requiring human attention by 60–76% and surfacing the most critical issues within seconds of test completion.

Keywords: 5G SA, AIOps, Cloud Native Core, Deep Learning, Transformer,

LSTM, Anomaly Detection, Chaos Engineering, Kubernetes, Self-Supervised Learning.

Acknowledgements

We would like to express our deepest gratitude to our supervisors at Ericsson for their continuous support and for providing access to the E2C environment. We also thank our examiner and supervisor at Chalmers University of Technology for their invaluable academic guidance throughout this Master's thesis project.

Muhammad Abdullah Arshad, Abdur Arshad, Gothenburg, May 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

3GPP	3rd Generation Partnership Project
5G SA	5th Generation Standalone Network
5GC	5th Generation Core
AI	Artificial Intelligence
AIOps	Artificial Intelligence for IT Operations
AMF	Access and Mobility Management Function
BERT	Bidirectional Encoder Representations from Transformers
CI/CD	Continuous Integration / Continuous Deployment
CNC	Cloud Native Core
CNF	Cloud-native Network Function
DDC	Diagnostic Data Collection
DL	Deep Learning
DSR	Design Science Research
E2C	Ericsson Engineering Cloud
GDPR	General Data Protection Regulation
K8s	Kubernetes
KPI	Key Performance Indicator
LSTM	Long Short-Term Memory
ML	Machine Learning
MLM	Masked Language Modeling
MTTR	Mean Time To Repair
NF	Network Function
NLP	Natural Language Processing
OPEX	Operational Expenditure
PCC	Packet Core Controller
PDB	Pod Disruption Budget
PoC	Proof of Concept
RCA	Root Cause Analysis
RNN	Recurrent Neural Network
RQ	Research Question

SBA	Service-Based Architecture
SBI	Service-Based Interface
SLA	Service Level Agreement
SMF	Session Management Function
SRE	Site Reliability Engineer
UPF	User Plane Function

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

Indices and Sets

t	Index for event position within a pod-group sequence
i	Generic index (e.g., masked position in masked language modelling, dimension index in positional encoding)
g	Index for a pod deployment group (pod_group)
$\mathcal{R}$	Set of Kubernetes event reasons (vocabulary, $\lvert\mathcal{R}\rvert = 57$)
$\mathcal{G}$	Set of pod deployment groups ($\lvert\mathcal{G}\rvert = 165$ common groups plus PAD and OTHER)
$\mathcal{S}_g$	Ordered sequence of events belonging to pod group g
$\mathcal{V}$	Generic finite vocabulary or set of discrete outcomes (used in theoretical definitions)
M	Set of masked indices within a sequence (LogBERT formulation)

Parameters

N	Sliding window size (input events + prediction target); $N = 10$
L	Input sequence length fed to the model; $L = N - 1 = 9$
d_{model}	Transformer hidden dimension; $d_{\text{model}} = 128$
d_k	Attention key dimension per head; $d_k = d_{\text{model}}/\text{nhead} = 32$
d	Generic embedding dimension (e.g., $d = 16$ for reason and pod-group embeddings)
d_{in}	Per-event input feature dimension ($16 + 16 + 4$); $d_{\text{in}} = 36$
θ	Learnable parameters of the neural network
η	Learning rate of the optimizer

β_1, β_2	Exponential decay rates for the first and second moments of the Adam optimizer
p_{drop}	Dropout probability used in the Transformer and LSTM
τ_{startup}	Startup-phase duration used by the situation extractor; $\tau_{\text{startup}} = 300$ s

Variables and Functions

e_t	Enriched event at position t (reason, pod group, type flag and log-transformed count, duration, time delta)
$p_\theta(\cdot \mid \cdot)$	Conditional probability distribution predicted by the model with parameters θ
$\hat{P}(\text{reason}_{t+1} \mid \cdot)$	Model's predicted probability distribution over $\mathcal{R}$ for the next event
$H(p, q)$	Cross-entropy between distributions p and q
$I(e)$	Shannon surprisal (information content) of event e, $I(e) = -\log P(e)$
$s(e_{t+1})$	Anomaly score of event e_{t+1}, defined as $-\log \hat{P}(\text{reason}_{t+1} \mid e_{t-L+1}, \ldots, e_t)$
E	Embedding matrix, $E \in \mathbb{R}^{\lvert\mathcal{V}\rvert \times d}$, mapping discrete tokens to dense vectors
$\text{PE}(pos, i)$	Sinusoidal positional encoding at position pos and dimension index i
$\mathcal{L}$	Cross-entropy training loss for next-event prediction
$\mathcal{L}_{\text{MLM}}$	Masked-language-modelling loss used in the LogBERT baseline discussion
severity	Composite pod-level severity score, $\overline{s} \cdot \text{total\_count} \cdot (1 + \text{max\_duration\_hours})$

Contents

List of Figures

List of Tables

1
Introduction

This chapter sets out the industrial context that motivated the thesis, states the research problem, and defines the objectives and scope. Section 1.1 gives the background on 5G cloud-native architectures and the observability problems they create. Section 1.2 states the purpose, and Section 1.3 lists the aim and objectives. The research questions follow in Section 1.4, and Section 1.5 defines the delimitations.

1.1 Background

The rollout of 5th Generation Standalone (5G SA) networks is not simply a radio upgrade. It is a rewrite of the core network in software. Functions that used to ship as proprietary appliances are now deployed as containerised microservices on commodity infrastructure, which gives operators scalability and release velocity they never had before. It also produces a great deal more complexity at the operational layer. Once network functions have been decomposed into dozens of small services orchestrated by Kubernetes (K8s), the traditional monitoring playbook of CPU and memory thresholds no longer captures what a healthy system looks like. In response, the industry is investing heavily in Artificial Intelligence for IT Operations, or AIOps, in the hope that machine learning can pick up where static thresholds leave off.

The 5G Core standardised by the 3rd Generation Partnership Project (3GPP) follows a Service-Based Architecture (SBA). In that model, network functions such as the Packet Core Controller (PCC) are decomposed into stateless microservices that talk to each other over HTTP/2 REST APIs [2]. The microservices run inside ephemeral pods that Kubernetes creates, scales and destroys as traffic changes.

The side effect is a gap in visibility. Classical monitoring relies on static thresholds over a handful of Key Performance Indicators (KPIs) such as CPU load, memory use, or disk I/O. In a distributed service mesh, a cluster can degrade badly while every single one of those indicators still looks green: handovers fail, signalling messages drop, latency drifts, and none of it shows up at the aggregate metric level [3]. These so-called grey failures are hard to catch and tend to live inside the large unstructured volume of infrastructure logs that the platform produces continuously.

Ericsson, which supplies 5G infrastructure to operators worldwide, has to keep these environments at five-nines (99.999 %) availability. Manual Root Cause Analysis (RCA) does not scale to that target. A single fault can spread across hundreds of pods within minutes and produce gigabytes of logs, which is more than any on-call engineer can read. Automated, data-driven approaches that use modern machine

learning are therefore an obvious next step for extracting signal from raw logs and catching subtle pre-failure patterns [4].

1.2 Purpose

The purpose of this thesis is to strengthen the operational resilience of Ericsson's Cloud Native Core (CNC) by helping the move from reactive troubleshooting towards proactive incident prevention. Recent advances in deep learning and sequence modelling make it plausible that the information needed to anticipate incidents is already present inside Kubernetes infrastructure event logs, and that a suitable model can surface it.
From an industrial point of view, the system aims to reduce operational expenditure (OPEX) by automating work that is currently done by hand, such as scanning through event logs after a test run, and to shorten the time to root cause. From an academic point of view, the work compares modern Transformer-based architectures against the older Long Short-Term Memory (LSTM) family on a self-supervised anomaly detection task, applied to a log domain that public benchmarks do not cover.

1.3 Aim and Objectives

The aim of this thesis is to explore the feasibility of a prototype AI-driven anomaly detection system that can identify infrastructure issues in the 5G Core before they cause service outages. The following objectives support that aim.

- **Objective 1:** Conduct a narrative literature survey of state-of-the-art deep learning methods for log-based anomaly detection, with emphasis on their suitability for high-volume operational data.
- **Objective 2:** Design a data preprocessing pipeline that fits Ericsson's Kubernetes event logs and enriches each event with lifecycle context and temporal information.
- **Objective 3:** Develop and train two deep learning models, a sequential LSTM and an attention-based Transformer, using a self-supervised next-event prediction objective on data collected from the Ericsson Engineering Cloud (E2C).
- **Objective 4:** Evaluate the trained models on held-out test data as well as on controlled chaos engineering experiments (via LitmusChaos) and robustness tests, in order to verify detection on previously unseen failure modes.
- **Objective 5:** Build a situation-aware analysis tool that turns per-event anomaly scores into interpretable infrastructure issue reports for Site Reliability Engineers (SREs).

1.4 Problem Specification

The central problem addressed in this thesis is that rule-based and supervised monitoring do not hold up in cloud-native environments. Kubernetes event logs are a

mix of routine lifecycle events, such as pod scheduling, container creation, and image pulling, and warning events, such as probe failures, mount errors, and scheduling conflicts. Classifying each event in isolation misses the sequential dependencies that actually distinguish healthy behaviour from unhealthy behaviour [5].
The harder part is that the same warning event can be benign or serious depending on what comes before it. A readiness probe failure that follows a pod termination signal is ordinary shutdown behaviour. The same failure following a successful pod start is a real service issue. In the same spirit, hundreds of mount failures during cluster startup are expected initialisation noise, whereas a handful of mount failures during steady-state operation usually are not. A detection system that is going to be useful has to model what normal pod lifecycles look like and flag deviations from those patterns, rather than reacting to individual event types.
There is also a practical constraint on the data itself. Anomalies are rare by design, which means labelled failure data is scarce. Supervised approaches therefore have very little to learn from. The system has to rely on self-supervised learning, modelling the distribution of normal system behaviour and flagging what does not fit.
The research is guided by the following Research Questions (RQs).

1. **RQ1:** What are the strengths and limitations of different sequence modelling architectures for anomaly detection in cloud-native infrastructure environments?
2. **RQ2:** How can infrastructure event logs be represented and sequenced to preserve meaningful contextual and temporal information for anomaly detection?
3. **RQ3:** To what extent can the proposed prototype generalise to previously unseen failure modes?

1.5 Delimitations

Several delimitations keep the thesis within the 20-week (30 HP) timeframe.

- **Data scope:** The analysis is limited to Kubernetes platform-level event logs from the Packet Core Controller infrastructure. Application-level microservice logs and service mesh logs are out of scope. User-plane payload data is excluded entirely, which is also required for GDPR compliance.
- **Environment:** The prototype is developed, trained, and evaluated only inside the non-production testbeds of the Ericsson Engineering Cloud (E2C).
- **Remediation:** The thesis covers detection and localisation of faults. Automated remediation, self-healing, and closed-loop control are not covered.

2
Theoretical Background

This chapter lays out the background needed to follow the rest of the thesis. It covers the architecture of cloud-native 5G networks, the nature of the data that such systems produce, the machine-learning ideas that the proposed system builds on, and the relevant prior work on log-based anomaly detection.

2.1 5G Cloud-Native Architecture

The move from 4G Long Term Evolution (LTE) to 5G Standalone is not a simple upgrade. The 3rd Generation Partnership Project (3GPP) redefined the core network to be software-driven and cloud-native, which changes how operators deploy, scale, and monitor their infrastructure [6].

2.1.1 Service-Based Architecture (SBA)

At the heart of the 5G Core (5GC) is the Service-Based Architecture (SBA). Legacy networks shipped network elements as monolithic hardware boxes connected through rigid point-to-point interfaces. The SBA breaks those functions apart and deploys them as independent, loosely coupled services [2].
Within the SBA, Network Functions (NFs) communicate over a shared Service-Based Interface (SBI) bus using standard IT protocols, in particular HTTP/2 over TCP/IP, with payloads formatted as JSON [6]. Because the protocols are the same ones used in web-scale services, operators can reuse mature IT tooling for monitoring, routing, and security. Key NFs in the architecture include the Access and Mobility Management Function (AMF), the Session Management Function (SMF), and the User Plane Function (UPF) [2].

2.1.2 Cloud-Native Network Functions (CNFs)

To achieve the scalability that 5G needs, NFs are deployed as Cloud-native Network Functions (CNFs). The cloud-native model packages applications as lightweight containers, orchestrates them dynamically, and structures them as microservices [7]. Breaking a monolithic 5G element into microservices lets developers update, scale, or restart individual components without disrupting the whole network function [8]. Ericsson's Packet Core Controller (PCC), for example, is not a single executable but a distributed mesh of microservices that together manage user sessions, mobility, and policy enforcement. That distribution is useful operationally, but it also multiplies

the number of communication paths and log streams, which is one of the reasons observability is so hard in this setting [3].

2.1.3 Containerisation and Kubernetes Orchestration

Containers and container orchestration are the operational backbone of CNFs. A container (typically run by *containerd* or Docker) packages a microservice together with its dependencies into an isolated, lightweight execution environment that shares the host operating system kernel [9].
To manage the deployment, scaling and networking of thousands of such containers, the telecommunications industry has converged on Kubernetes (K8s) as the de facto orchestrator [9]. A Kubernetes cluster consists of Master Nodes (the control plane) and Worker Nodes (the execution plane).
The smallest deployable unit in Kubernetes is a *Pod*, which wraps one or more closely related containers. Kubernetes continuously watches the state of the cluster. When a Pod running a critical PCC microservice fails, the orchestrator schedules a replacement Pod on a healthy Worker Node [9].
This ephemeral, self-healing behaviour is what gives cloud-native systems their resilience, but it also makes tracing harder. Once a Pod is terminated, its local file system is gone together with any logs it held, unless those logs were streamed to a central logging backend such as Elasticsearch or Fluentd [7]. Tracing a transaction across a chain of ephemeral Pods therefore calls for dedicated distributed tracing and log correlation mechanisms, which is part of the motivation for automating log analysis.

2.1.4 Kubernetes Events and the Pod Lifecycle

Alongside classical textual logs, Kubernetes emits its own stream of structured state-change notifications called *events*. Events are first-class API objects produced by the API server, the scheduler, the kubelet on each node, and various controllers whenever something notable happens in the cluster. Examples are scheduling a Pod, pulling a container image, starting a container, or failing to mount a volume [10]. Unlike free-text log lines, each event has a well-defined schema. It carries a categorical `reason` (for example `Scheduled`, `Pulled`, `Created`, `Started`, `Unhealthy`, `FailedMount`), a severity-like `type` restricted to `Normal` or `Warning`, a `count` that records how many times identical events have occurred, a pair of `firstTimestamp` and `lastTimestamp` fields that delimit the event's temporal extent, a pointer to the `involvedObject` (typically a Pod, Deployment or Node), and a free-text `message` that renders the structured fields for human consumption.
One consequence of this schema is *event aggregation*. When the same logical event (same source, object, reason and message) is emitted repeatedly within a short window, Kubernetes collapses these occurrences into a single object whose `count` is incremented and whose `lastTimestamp` is advanced. A persistent failure therefore appears not as hundreds of individual log lines but as a single event with a high `count`, sometimes spanning many hours. By default, events are retained in the etcd datastore for only one hour [10], so longer-term analysis requires capturing events

through periodic snapshots or a dedicated event exporter.
The sequence of events associated with a single Pod follows a well-understood lifecycle. A freshly created Pod is first `Scheduled` onto a node. The kubelet then pulls the container image (`Pulling`, `Pulled`), creates the container (`Created`), and starts it (`Started`). Container Network Interface (CNI) plugins contribute further events such as `AddedInterface` during network setup. Once running, a Pod reports probe-related events (`Unhealthy`) when liveness, readiness, or startup probes fail, and emits `Killing` events during graceful termination, optionally followed by `FailedPreStopHook` if the pre-stop lifecycle hook does not complete cleanly.
Most Pods are not managed directly but through higher-level controllers. `Deployments` manage stateless workloads through `ReplicaSets`. `StatefulSets` manage stateful workloads that require stable network identities [9, 10]. These controllers append hash and ordinal suffixes to Pod names (for example, `eric-pc-mm-mobility-6f587d7d5-sr96c`), so many syntactically distinct Pod identifiers actually belong to a single logical deployment. That distinction matters for sequence modelling, because events from different replicas of the same deployment are semantically exchangeable, while events from different deployments are not.

2.2 The Observability Stack

Modern cloud-native observability is usually framed in terms of four complementary telemetry pillars: *metrics*, *logs*, *events*, and *traces* [11, 12]. Metrics are regularly sampled numerical time series such as CPU utilisation, request rates, or queue depths, typically aggregated into dashboards and threshold-based alerts. Logs are free-text records of application behaviour produced by developer logging calls and shipped to a central store such as Elasticsearch or Loki. Traces are tree-structured records of individual requests as they propagate through a distributed system, used primarily for latency diagnosis. Events, as described in Section 2.1.4, are discrete, semi-structured notifications of state changes emitted by the platform itself.
Each pillar reveals a different facet of system health. Metrics are efficient but lossy, because aggregation over short windows erases the specific causal signals that precede a failure. Logs are rich but high-volume and unstructured. Traces are precise but limited to request-level granularity and do not naturally capture infrastructure-level events such as scheduling failures or node drains. Events are low-volume, schema-structured, and capture exactly the infrastructure lifecycle transitions that this thesis is concerned with. That is the reason the work presented here focuses on the event pillar as its primary data source, while acknowledging that a complete operational observability solution would ultimately combine information across all four pillars [3].

2.3 Log Parsing and Preprocessing

Cloud-native 5G environments generate telemetry at very high velocity and volume. Metrics like CPU utilisation give a good high-level view of system health, but event logs remain the only source that records the granular, sequential state changes and

code paths inside the microservices [4]. Before machine-learning algorithms can work with these logs, however, the raw text has to be parsed and preprocessed.

2.3.1 Characteristics of Cloud-Native Infrastructure Logs

System logs are semi-structured or entirely unstructured strings of text produced by application code through logging statements such as `printf()` or `logger.info()`. A typical log message has a timestamp, a severity level (INFO, WARN, ERROR), the component that emitted it, and the free-text message payload.
The payload itself is a mixture of two distinct elements:

- **Constant parts (templates or event types):** The static text defined by the developer, such as `"Connection refused by peer"`.
- **Variable parts (parameters):** Runtime values that change with each execution, including timestamps, IP addresses, session IDs, latencies and port numbers. In the log `"Connection refused by peer 192.168.1.5 at port 8080"`, the IP and port are variables.

Because the variable parts have very high cardinality, plain string-matching algorithms treat millions of logs of the same event type as entirely distinct entities. Feeding raw log strings straight into a neural network therefore produces an effectively unbounded vocabulary, which makes it practically impossible for the model to learn structural patterns [13].

2.3.2 The Log Parsing Paradigm

Log parsing is the process of separating the constant template from the dynamic variables, which turns unstructured text into structured data.
Historically, the telecommunications industry relied on rule-based parsing, typically implemented as hand-crafted regular expressions. Regular expressions are accurate on static systems but do not cope well in cloud-native 5G environments [13]. Continuous integration and continuous deployment (CI/CD) pipelines update microservice versions and their logging statements frequently, and maintaining a comprehensive library of regular expressions for hundreds of microservices is prohibitively expensive. The inevitable result is a stream of unparsed logs appearing after every software upgrade.
To avoid that fragility, automated log parsers use data mining and heuristics to cluster historical logs by string similarity and extract the common template that defines each cluster.

2.3.3 Fixed-Depth Tree Parsing: The Drain Algorithm

Among automated log parsers, Drain is the algorithm most often used in practice, because it is accurate and fast enough for the real-time processing rates of a 5G Core [14].
Unlike general-purpose clustering algorithms such as K-Means or Hierarchical Clustering, which have $O(N^2)$ time complexity and are therefore unsuitable for high-velocity log streams, Drain uses a fixed-depth parse tree [14]. The algorithm proceeds in four steps:

1. **Domain-knowledge preprocessing:** Drain applies simple regular expressions to mask obvious variables such as IP addresses and hexadecimal block IDs, replacing them with generic tokens (for example `<IP>`).
2. **Length-based branching:** The root of the parse tree branches on the length of the log message (the number of tokens). Logs of different lengths naturally belong to different templates, which shrinks the search space immediately.
3. **Prefix-based branching:** Subsequent layers branch on the first few tokens of the log. Developers usually begin log messages with static descriptors, so prefix branching routes each log to a specific leaf.
4. **Similarity scoring and template extraction:** At the leaf, Drain computes a token-wise similarity between the incoming log and each previously identified template. If the similarity is above a threshold (often 50%), the log is matched to that template and any differing tokens are recorded as variables (the wildcard `<*>`). If the similarity is below the threshold, a new template is created.

By bounding the depth of the tree, Drain keeps the per-log search time constant, giving an overall linear complexity of $O(N)$ [14].

2.3.4 Semantic Vectorisation and Embeddings

Parsing with Drain converts a log into a structured template ID such as `Template_E42`. Feeding arbitrary template IDs straight into a deep learning model throws away the semantic meaning of the text [5]. For instance, `Template_1` ("Database connection timeout") and `Template_2` ("Database unreachable") are semantically almost identical, but a model that sees only the IDs treats them as unrelated numbers.
To keep the semantic context, templates are usually vectorised. Using techniques from Natural Language Processing (NLP), the text of the template is transformed into a high-dimensional dense vector of floating-point numbers.
Pre-trained language models such as Word2Vec [15] or Bidirectional Encoder Representations from Transformers (BERT) are a common choice for producing these embeddings. In the resulting vector space, semantically similar templates sit close to one another [5]. A downstream anomaly detector such as LogBERT or an LSTM can then generalise across the space. If it learns that a vector near a "timeout" template is a network-failure signal, it can reuse that signal on new, unseen templates that land in the same region, which is sometimes described as "zero-day" anomaly detection. Recent work such as NeuralLog [16] pushes the idea further by skipping the parsing step entirely and feeding tokenised log lines directly into a pre-trained Transformer, showing that the boundary between parsing and representation is a design choice rather than a fundamental requirement.

2.4 Artificial Intelligence for IT Operations

The term Artificial Intelligence for IT Operations (AIOps) refers to the use of big-data analytics and machine learning to automate parts of IT operations that previously relied on human judgement. As software systems moved from monolithic

architectures to distributed microservice deployments, the volume, velocity and variety of operational telemetry all grew rapidly. Foundational work in the area groups AIOps tasks into three main categories: anomaly detection, event correlation, and root cause analysis [17].

2.4.1 From Reactive to Predictive Monitoring

For 5G telecommunications specifically, AIOps is less an optimisation than a practical necessity. Modern 5G networks are subject to strict service-level agreements that demand very high availability and low end-to-end latency, targets that are hard to hit with static thresholds and manual response alone. AIOps therefore shifts the operational model from threshold-based alerting, which tends to produce many false positives, to pattern-based alerting that takes context into account [18].
Threshold-based systems fire an alert whenever a metric such as CPU or memory usage crosses a manually defined boundary. In a dynamically scaling 5G Core, however, a spike in resource usage can be a normal reaction to a traffic surge rather than a symptom of a real failure. Rigid thresholds applied to such volatile environments tend to generate overwhelming alert noise, and the resulting alarm fatigue can lead Site Reliability Engineers to overlook the alerts that do matter amid the ones that do not [19].
To push back against this observability problem, AIOps platforms continuously ingest logs, metrics and traces together, and use machine learning to learn what a healthy network looks like. When something deviates, the platform can take the broader context into account rather than reacting to a single metric value. In practice, this has been shown to reduce false positives considerably and to surface subtle degradation patterns that often precede bigger failures.

2.4.2 The AIOps Pipeline

A typical AIOps solution is organised as four connected phases [17]. The first is *observation*, which ingests telemetry from different infrastructure layers without discarding potentially useful signal. The second is *detection*, which uses analytics (often deep learning) to flag latent anomalies or predict imminent failures.
The third phase is *engagement*, where seemingly unrelated events are correlated across the distributed system. Aggregating thousands of disparate alerts from different microservices into a single incident timeline lowers the cognitive load on human operators. Once incidents are grouped, automated root cause analysis algorithms can trace the network topology to find the origin of the fault. The final phase is *action*, which covers automated remediation and self-healing when appropriate. Taken together, the pipeline shifts engineering effort from reactive firefighting to proactive maintenance.

2.5 Anomaly Detection: Concepts and Methods

Anomaly detection is the task of identifying data points, events, or patterns that deviate from an expected notion of normality. Chandola, Banerjee and Kumar

survey the field and propose a taxonomy that is useful for the present work [20].

2.5.1 Taxonomy: Point, Contextual and Collective Anomalies

Anomalies fall into three broad classes. *Point anomalies* are individual data points that are abnormal in isolation, such as an extreme temperature reading. *Contextual anomalies* are points that are abnormal only relative to a context, such as a time of day or the component that produced them. *Collective anomalies* are groups of points whose co-occurrence is abnormal even though each individual point is benign. Kubernetes infrastructure events show contextual and collective behaviour rather than point anomalies. A single `FailedMount` event during cluster bootstrap is benign, but the same event in steady state indicates a real problem. A simultaneous burst of `Unhealthy` events across many Pods of a deployment is a collective anomaly whose components look individually like routine probe failures. Recognising this structure matters when designing both the learned component and the rule-based post-processing of an anomaly-detection system.

2.5.2 Supervision Regimes

Anomaly detection methods are commonly classified by their reliance on labels [20, 21]. *Supervised* methods require examples of both normal and anomalous data and reduce anomaly detection to binary classification, which is most effective when the anomalies are well characterised. *Semi-supervised* methods assume access to a corpus of mostly normal data and learn a model of normality, flagging deviations at inference time. *Unsupervised* methods make no explicit assumption about the labelling of the training data and rely on statistical properties such as rarity or geometric isolation. Log- and event-based anomaly detection in production systems almost always operates in the semi-supervised regime, because anomalies are rare by design and exhaustive labelling is infeasible.

2.5.3 Classical Baselines

Before deep learning, a small number of general-purpose algorithms established themselves as reference baselines for anomaly detection. *Isolation Forest* [22] recursively splits the feature space with random partitions and scores an observation by the average path length to isolation. Anomalies tend to be isolated quickly and therefore receive short paths. *One-Class Support Vector Machines* [23] learn a hyperplane in a reproducing-kernel Hilbert space that separates the bulk of the training data from the origin, producing a decision function that is positive on points resembling the training set. *Principal Component Analysis* (PCA) based detectors project the data onto the subspace that captures most of its variance and flag points with large reconstruction error. These methods make minimal structural assumptions, which is attractive, but they treat observations independently and therefore struggle with the sequential and contextual anomalies that dominate in log and event streams.

2.5.4 Deep Learning for Anomaly Detection

A more recent line of work, surveyed by Pang et al. [21], applies deep neural networks to learn representations under which anomalies become easier to detect. Three families are relevant here. *Reconstruction-based* methods (for example autoencoders) learn to reconstruct normal inputs and flag points whose reconstructions are poor. *Likelihood-based* methods fit a probabilistic model and use the predicted likelihood (or its negative logarithm, the surprisal) as the anomaly score. *Distance-based* methods define anomaly in terms of distance to nearest neighbours in a learned representation. The deep sequence models that dominate log-based anomaly detection, including those reviewed in Section 2.6, belong primarily to the likelihood-based family and use the conditional surprisal of Section 2.6.1.3 as the scoring rule.

2.6 Deep Learning for Sequence Modelling

This section first introduces the mathematical foundations that underpin the sequence models and anomaly-scoring rules, and then reviews the specific architectures used in this thesis.

2.6.1 Mathematical Foundations

The treatment below is deliberately standard and draws on classical references on pattern recognition, information theory, and deep learning [24, 25, 26].

2.6.1.1 Categorical Distributions and Softmax

Let $\mathcal{V} = \{v_1, v_2, \ldots, v_K\}$ be a finite vocabulary of K discrete outcomes. A *categorical* random variable Y over $\mathcal{V}$ is specified by a probability vector $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_K)$ with $\pi_k \geq 0$ and $\sum_k \pi_k = 1$. Neural sequence models parameterise $\boldsymbol{\pi}$ as the output of a differentiable function $f_\theta(\mathbf{x}) \in \mathbb{R}^K$ applied to an input context $\mathbf{x}$, followed by the softmax transformation

$$\pi_k(\mathbf{x}; \theta) = \frac{\exp\Big(f_{\theta,k}(\mathbf{x})\Big)}{\sum_{j=1}^{K} \exp\Big(f_{\theta,j}(\mathbf{x})\Big)}. \tag{2.1}$$

The softmax produces a valid probability distribution and is differentiable everywhere, which makes it the canonical output transformation for classification over finite vocabularies [24, 25].

2.6.1.2 Cross-Entropy as Maximum-Likelihood Training

Given a dataset $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$ of context and label pairs, the maximum-likelihood estimate of the parameters θ maximises the log-likelihood

$$\log p(\mathcal{D} \mid \theta) = \sum_{n=1}^{N} \log \pi_{y^{(n)}}(\mathbf{x}^{(n)}; \theta). \tag{2.2}$$

Equivalently, one minimises the empirical *cross-entropy* between the one-hot empirical distribution and the model distribution:

$$\mathcal{L}(\theta) \;=\; -\frac{1}{N}\sum_{n=1}^{N}\log \pi_{y^{(n)}}(\mathbf{x}^{(n)};\theta) \;=\; \frac{1}{N}\sum_{n=1}^{N} H\Big(\delta_{y^{(n)}},\, \boldsymbol{\pi}(\mathbf{x}^{(n)};\theta)\Big), \tag{2.3}$$

where $\delta_{y^{(n)}}$ denotes the Dirac (one-hot) distribution on the observed label and $H(p,q) = -\sum_k p_k \log q_k$ is the cross-entropy [24]. Minimising cross-entropy is therefore equivalent to maximum-likelihood estimation of the categorical model, which justifies its use as the default training objective for discrete sequence models.

2.6.1.3 Surprisal and Anomaly Scoring

Shannon's information theory assigns to every outcome $e \in \mathcal{V}$ a non-negative quantity called the *surprisal* or *information content*,

$$I(e) \;=\; -\log P(e), \tag{2.4}$$

which measures how unexpected the outcome is under a reference distribution P [26]. When P is replaced by a predictive distribution $p_\theta(\,\cdot \mid \mathbf{x})$ that conditions on context, the per-outcome surprisal becomes

$$s(e \mid \mathbf{x}) \;=\; -\log p_\theta(e \mid \mathbf{x}), \tag{2.5}$$

which coincides with the instantaneous term in the cross-entropy loss. A natural anomaly-detection rule, widely used in log-based anomaly detection [27], is to flag events whose conditional surprisal exceeds a threshold. The same rule can be expressed as a top-k test: an event is anomalous if it does not appear among the k most probable outcomes under the predictive distribution. The two formulations coincide when the predictive distribution is sharply peaked, as is typical when a model is well trained on a small, structured vocabulary.

2.6.1.4 Discrete Tokens and Learned Embeddings

Sequence models operate on dense real-valued vectors, yet the data they receive is typically discrete. The bridge between the two is the *embedding layer*, which is a lookup matrix $E \in \mathbb{R}^{|\mathcal{V}|\times d}$ that assigns each token v_k a d-dimensional vector E_k learned jointly with the rest of the model [28, 15]. Embeddings generalise one-hot encodings in two ways. First, they are lower-dimensional and therefore statistically easier to estimate. Second, they are dense, so gradient descent can push semantically related tokens towards nearby regions of the space. The dimensionality d is a design parameter that trades representational capacity for sample efficiency.

2.6.1.5 Positional Information

Unlike recurrent networks, attention-based models are permutation-equivariant and have no built-in notion of order. Vaswani et al. restore order information by adding a fixed sinusoidal *positional encoding* to the token embeddings [29]:

$$\mathrm{PE}(pos, 2i) = \sin\Big(pos/10000^{2i/d}\Big), \tag{2.6}$$

$$\mathrm{PE}(pos, 2i+1) = \cos\Big(pos/10000^{2i/d}\Big), \tag{2.7}$$

where *pos* is the position in the sequence and i indexes the embedding dimension. The resulting vectors let the model infer relative position through linear combinations of the encoding, while keeping the overall computation fully parallelisable.

2.6.1.6 Self-Supervised Learning and Language Modelling

The training objective in Section 2.6.1.2 does not need explicit labels. The label at each position is simply the observed next token. This is the defining property of *self-supervised* learning, in which the supervision signal is derived from the structure of the data itself rather than from manual annotation [28]. Predicting the next token in a sequence is known as (autoregressive) language modelling. Predicting a masked subset of tokens from bidirectional context is its masked counterpart, as used by BERT [30]. Both are instances of the same underlying principle, which is fitting a probabilistic model of the data distribution, and both lend themselves naturally to anomaly detection, because any probabilistic model trained on normal data implicitly defines anomalies as low-probability events.

2.6.1.7 Optimisation: Stochastic Gradient Methods

Deep networks with tens of thousands to millions of parameters are trained almost exclusively by first-order stochastic methods operating on mini-batches of examples [25]. Given a loss $\mathcal{L}(\theta)$ and a mini-batch gradient estimate $\mathbf{g}_t$, plain stochastic gradient descent (SGD) updates the parameters according to $\theta_{t+1} = \theta_t - \eta \mathbf{g}_t$, where η is the learning rate. The *Adam* optimiser [31] refines this rule by maintaining exponential moving averages of the first and second moments of the gradients,

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t, \tag{2.8}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t \odot \mathbf{g}_t, \tag{2.9}$$

applies a bias correction, and steps as

$$\theta_{t+1} = \theta_t - \eta \, \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \varepsilon}, \tag{2.10}$$

with $\hat{\mathbf{m}}_t = \mathbf{m}_t/(1 - \beta_1^t)$ and $\hat{\mathbf{v}}_t = \mathbf{v}_t/(1 - \beta_2^t)$. Adaptive learning-rate schedulers such as ReduceLROnPlateau reduce η when a monitored validation metric stops improving, combining the stability of adaptive optimisation with the fine-tuning benefits of a decaying step size.

2.6.1.8 Regularisation via Dropout

To limit overfitting in large parameter spaces, Srivastava et al. introduced *dropout*. During training, every hidden unit is independently zeroed with probability p_{drop}, and the remaining activations are rescaled by $1/(1-p_{\text{drop}})$ to preserve their expected magnitudes [32]. At inference the full network is used. Dropout can be interpreted as an approximate ensemble average over an exponential family of sub-networks and has become a standard component of both recurrent and attention-based architectures.

2.6.2 Sequence Architectures

In a cloud-native environment, system logs are generated sequentially as the underlying microservices run. A continuous stream of logs can therefore be viewed as a discrete time series or as a sequence of natural-language tokens. The goal of sequence modelling in this setting is to learn the underlying probability distribution of these events. Once the model knows what a normal execution flow looks like, it can predict future states or reconstruct missing information. If the live system produces a sequence that is far from the learned baseline distribution, the behaviour can be flagged as an anomaly.

2.6.2.1 Recurrent Neural Networks and the Concept of Memory

A feedforward neural network assumes that all inputs are independent. That assumption breaks immediately for logs, where a "database connection closed" event depends on a preceding "database connection opened" event. Recurrent Neural Networks (RNNs) address this shortcoming by introducing an internal memory mechanism [33].
Instead of a single forward pass, an RNN processes the sequence one element at a time. As it reads each parsed log event, it updates a hidden state vector. The hidden state acts as a compressed summary of the log events seen so far. When the network predicts the next log event, it uses both the current input and this hidden state.
The idea is clean, but standard RNNs suffer from a well-known optimisation problem called the vanishing gradient problem. During training, the network adjusts its weights by propagating errors backwards through time. On long sequences, the gradients used for these corrections shrink exponentially [33].
In a 5G network this is a serious limitation. A configuration error may occur during Pod initialisation, while the actual failure may only appear thousands of log lines later. Under the vanishing gradient effect, a plain RNN tends to lose the connection between the initial error and the eventual crash, which is exactly the kind of long-range dependency this thesis is interested in.

2.6.2.2 Long Short-Term Memory Networks (LSTMs)

To keep long-range dependencies in memory, Hochreiter and Schmidhuber introduced the Long Short-Term Memory (LSTM) architecture [34]. Rather than a single hidden node, an LSTM uses a memory cell regulated by a system of differentiable gates. Each gate learns what information to keep, what to update and what to discard as the sequence progresses.
The architecture is organised around a central cell state that runs straight down the sequence chain with only minor linear interactions. This design lets gradients flow backwards through time without vanishing. Three gates control the flow of information into and out of the cell state:

- **Forget gate:** Looks at the previous hidden state and the current log event and decides which historical information is no longer relevant. For example, when a microservice closes one session and opens another, the forget gate can

wipe the old session ID from the cell state.
- **Input gate:** Decides which new information from the current log event is worth adding to the long-term cell state.
- **Output gate:** Decides what part of the cell state should be exposed as the hidden state for the current time step, which is what actually drives the prediction.

In log anomaly detection, models such as DeepLog use this architecture to predict the next logical step in a system process [27]. The model is trained on logs from a healthy system. At inference time, it takes a window of recent logs and outputs a probability distribution over the possible next log templates. If the actual next log event is not in the top few predictions, the sequence is flagged as a deviation that may indicate a failure.

2.6.2.3 The Transformer Architecture and Self-Attention

The LSTM handles the memory problem of RNNs, but it is still sequential by construction. Because it has to process log line N before log line $N + 1$, it cannot be parallelised across time. In concurrent 5G environments, logs from different microservices are also interleaved asynchronously, which can confuse sequential models that expect a tidy step-by-step execution path.
The Transformer architecture gets around both issues by dropping sequential recurrence entirely and replacing it with self-attention [29].
The key idea of the Transformer is to look at the whole sequence at once. Instead of compressing the past into a single hidden state, self-attention computes a weighted relevance score between every pair of log events in the sequence.
Conceptually, self-attention works a little like a database lookup. For a given log sequence, the model projects the data into three learned representations, the Queries, Keys and Values. To understand the context of a specific event (the Query), the model compares it with every other event (the Keys). The result of the comparison determines how much attention to place on the content of each other log (the Values). Mathematically, the scaled dot-product attention is defined as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{2.11}$$

When a "timeout" log occurs, self-attention can reach across the entire window and place a large weight on a "connection initiated" log that appeared hundreds of lines earlier, while ignoring unrelated traffic in between. Because the architecture evaluates the whole sequence at once, it needs positional encodings to inject temporal information into the data, so that the model can still reason about the chronological order of events without processing them sequentially [29].

2.6.2.4 BERT and LogBERT for Asynchronous Log Environments

Building on the Transformer, the Bidirectional Encoder Representations from Transformers (BERT) architecture changed the picture for sequence modelling [30]. Traditional sequence models are autoregressive. They look only at past tokens to predict the next one. In cloud-native microservices, however, events related to the same

transaction can arrive slightly out of order because of network latency or thread scheduling, which means looking only at the past is often not enough.
A bidirectional architecture addresses this by processing the entire sequence from both directions simultaneously. The contextual representation of any given log event is informed by both its predecessors and its successors inside the analysis window, giving a fuller picture of the system state [35].
LogBERT adapts this architecture for log anomaly detection using a self-supervised objective called Masked Language Modelling [35]. During training, the algorithm takes a sequence of normal system logs and masks a fraction of the templates.
The network then predicts the identity of the hidden log templates from the surrounding bidirectional context. It does so by minimising the cross-entropy between its predictions and the true hidden tokens,

$$\mathcal{L}_{\text{MLM}} = -\sum_{i \in M} \log P(e_i \mid S_{\setminus M}; \theta), \tag{2.12}$$

where M is the set of masked indices, e_i is the true log event at position i, $S_{\setminus M}$ is the corrupted sequence and θ collects the network's learnable parameters.
Through this optimisation, the model learns the semantic relationships between log templates and the execution flows of the underlying infrastructure. At inference time, incoming log sequences are scored continuously. The model computes the predicted probability for each event that actually occurred, and if that probability falls below an empirical threshold, the sequence is flagged as a statistical anomaly that may indicate a zero-day fault.

2.7 Chaos Engineering

Chaos engineering is the discipline of experimenting on a system to build confidence in its ability to withstand turbulent conditions [36, 37]. Rather than waiting for failures to happen in production, chaos engineers deliberately inject faults into a running system and observe how it degrades, which helps surface latent resilience issues before they cause real outages. The practice originated at Netflix with tools such as Chaos Monkey that terminated production instances at random, and has since grown into a systematic engineering discipline with dedicated platforms.
It is useful to distinguish three broad classes of faults that chaos experiments inject. *Resource-level* faults stress the machine or container itself, for example through CPU or memory saturation or disk pressure. *Network-level* faults perturb communication between components. Added latency, packet loss, packet corruption and network partitions all belong to this class. *State-level* faults target the application or orchestration layer, such as terminating pods and processes, killing containers, forcing rolling restarts, or invalidating certificates. Each class probes a different axis of resilience, and a comprehensive chaos campaign samples from all three.
In the Kubernetes ecosystem, the open-source LitmusChaos project [38] has become a widely adopted framework for authoring and running such experiments. LitmusChaos defines chaos experiments as Kubernetes custom resources, manages their lifecycle through operators, and emits structured status events to the same event stream that is the focus of this thesis. Because chaos experiments produce

observable, repeatable deviations from normal lifecycle patterns, they are a natural source of realistic disturbance data for training and evaluating sequence-based anomaly-detection models.

2.8 Evaluation of Sequence Anomaly Detection

The choice of evaluation metric has a strong influence on how anomaly-detection methods appear to perform. A substantial body of recent work has shown that many popular benchmarks reward superficial heuristics rather than genuine detection ability [39]. This section reviews the metrics that are relevant for sequence-based log and event anomaly detection.

Because the models of Section 2.6 return a probability distribution over a vocabulary, a first family of metrics borrowed from language modelling evaluates the quality of that predictive distribution. *Top-k accuracy* measures the fraction of events for which the true next event appears among the model's k most probable predictions. It is the direct analogue of the detection rule introduced in Section 2.6.1.3. When the vocabulary is strongly imbalanced, as it is in Kubernetes events where a handful of lifecycle reasons dominate the distribution, accuracy alone can be misleading, and per-class metrics such as precision, recall and F1 score, aggregated in both weighted and macro forms, give a more discriminating view [20, 21].

A second family of metrics addresses the ultimate downstream use of the model, namely deciding whether an anomaly has occurred within a given time window. Classical precision and recall, defined on individual events, do not capture this temporal aspect well, because they penalise every timestamp on which an otherwise correct detection is off by a few seconds. Tatbul et al. propose *range-based precision and recall* to address this limitation. They generalise the metric definitions so that a detection is credited as long as it overlaps a labelled anomaly interval, with parameters controlling the credit given for partial overlap, the penalty for detecting the same anomaly twice, and the bias toward early or late detection [40]. Wu and Keogh argue that even these refinements have to be used carefully, because several widely used benchmarks are so easy that trivial heuristics achieve state-of-the-art scores, and they call for more rigorous benchmarking protocols [39].

In practice, quantitative metrics on per-event prediction are usually complemented by qualitative evaluation against known fault-injection scenarios, where the task is to check that the system surfaces the correct affected components rather than to compute a single summary score. This mixed evaluation strategy suits semi-supervised anomaly detection in production systems, where ground-truth anomaly intervals with precise temporal boundaries are rarely available.

2.9 Related Work

Several approaches to log-based anomaly detection have been proposed in the literature, each addressing different aspects of the problem. These can broadly be categorised by the type of input representation they use (raw text, parsed templates, or structured metadata), by the modelling paradigm they adopt (autoregressive pre-

diction, masked reconstruction, or classification), and by the granularity at which they operate (individual log lines, fixed-length windows, or session-level sequences). The following subsections review the most influential approaches in each category.

2.9.1 DeepLog

Du et al. [27] proposed DeepLog, which formulates log anomaly detection as a next-event prediction task. An LSTM model is trained on sequences of normal log template IDs to predict the next template in the sequence. At inference time, if the actual log event does not appear among the model's top-k predictions, the event is flagged as anomalous. DeepLog showed that self-supervised sequence modelling can detect anomalies without labelled failure data. The approach is directly relevant to this thesis, which adopts the same next-event prediction formulation.

2.9.2 LogBERT

Guo et al. [35] extended the Transformer-based BERT architecture to log anomaly detection using masked language modelling. LogBERT randomly masks log templates within a sequence and trains the model to predict the masked tokens from bidirectional context. LogBERT performs well on public benchmarks, but its bidirectional masking objective is less naturally aligned with anomaly detection than the autoregressive next-event prediction used by DeepLog, particularly when the event vocabulary is small.

2.9.3 LogAnomaly and LogRobust

Meng et al. [41] proposed LogAnomaly, which uses template2vec embeddings to capture semantic relationships between log templates and can therefore detect novel log types not seen during training. Zhang et al. [42] introduced LogRobust, which addresses the instability of log parsing across software versions by using semantic word vectors rather than exact template matching. Both approaches stress the importance of a stable event representation, a challenge this thesis also addresses through learned categorical embeddings.

2.9.4 Hierarchical and Parser-Free Approaches

Huang et al. introduced HitAnomaly [43], a hierarchical Transformer that models both the sequence of log templates and the parameter values embedded within them. HitAnomaly's hierarchical attention can attend selectively to informative parameters, which improves results on logs where parameters carry diagnostic signal. Le and Zhang proposed NeuralLog [16], which departs from the template-based paradigm entirely and feeds tokenised raw log messages into a pre-trained Transformer encoder. On public benchmarks such as HDFS and BGL, NeuralLog achieves strong results, arguing that modern language-model encoders can absorb the variability that earlier parser-based pipelines had to remove by hand. These approaches show that the boundary between parsing, representation, and detection is a design choice.

Different domains reward different trade-offs between structured, template-based inputs and richer but noisier raw-text inputs.

2.9.5 Microservice and Kubernetes-Specific Observability

A distinct strand of work targets anomaly detection and root-cause analysis in microservice and Kubernetes deployments specifically. Soldani and Brogi survey anomaly detection and failure root cause analysis in microservice-based cloud applications and classify existing methods by the telemetry signal (metrics, logs, traces, events), the detection regime (supervised, semi-supervised, unsupervised), and the granularity of the RCA output (service, instance, or call level) [3]. Their survey notes that combining multiple telemetry pillars tends to improve accuracy, but also that many published approaches are evaluated only on synthetic or academic benchmarks rather than on production-grade 5G or telecommunications deployments. Kawasaki and collaborators demonstrate failure prediction in a cloud-native 5G Core using eBPF-based kernel observability combined with machine learning, and provide one of the few end-to-end evaluations in a 5G telecommunications context [18]. Together, these works motivate this thesis's combination of structured Kubernetes events as the telemetry signal and a self-supervised training regime as the detection paradigm.

2.9.6 Gaps in Existing Work

The methods above are generally evaluated on public datasets such as HDFS and BGL, but several gaps appear when applying them to production Kubernetes environments. First, most approaches analyse individual log lines in isolation or in global sequences, without grouping events by their originating component such as a pod deployment [27]. Second, existing methods do not account for Kubernetes-specific lifecycle patterns such as startup noise, rolling restarts, or graceful shutdowns, which generate large volumes of benign warnings. Third, the post-processing step of turning per-event anomaly scores into actionable infrastructure-level insights has received limited attention. Fourth, parser-free approaches such as NeuralLog [16] exploit the variability of free-text log messages. In the Kubernetes event domain, by contrast, the structured `reason` field is a controlled vocabulary that maps one-to-one with the message content, which makes text-based inputs simultaneously unnecessary and prone to data leakage. This thesis addresses these gaps by combining self-supervised next-event prediction on per-deployment event sequences with structured, domain-aware situation extraction, using a representation that deliberately excludes the free-text message in favour of the structured event metadata.

3
Methodology

This thesis follows the Design Science Research (DSR) framework defined by Hevner et al. [44] and refined into a process model by Peffers et al. [1]. DSR is inherently iterative and is typically applied over extended research time-frames; even multi-year projects may not fully realise all stages and iteration cycles. In this thesis, DSR is used as a guiding methodology rather than a claim of full-cycle completion. The framework nonetheless fits this work because the goal is to address a practical industrial problem, infrastructure observability, through the iterative design, implementation and evaluation of a new artifact, in this case the anomaly detection prototype. The development from PoC v1 to PoC v2 constitutes one complete build–evaluate iteration within the DSR process. Figure 3.1 illustrates the high-level DSR process as applied in this thesis. This chapter describes the research environment, the data acquisition strategy, the artifact design, and the evaluation methodology.

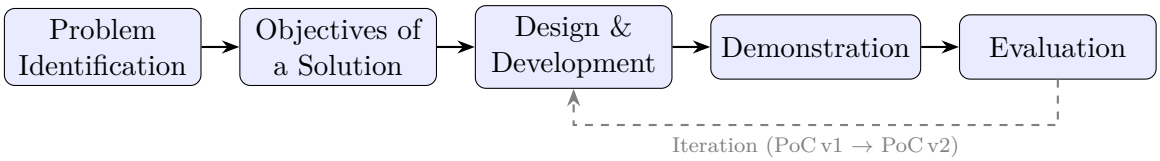


Figure 3.1: Design Science Research process model (adapted from Peffers et al. [1]) as applied in this thesis. The dashed arrow represents the iterative cycle from evaluation back to design, which in this work corresponds to the redesign from PoC v1 to PoC v2.

3.1 Research Environment

All research, development and data storage take place inside Ericsson's internal Engineering Environment (E2C), which ensures that intellectual property stays protected and that the work complies with corporate security requirements. Development runs on a dedicated Git Terminal Server workspace. Model training runs on CPU nodes within the E2C private cloud. The models are small, with 228K to 610K parameters, which means GPU acceleration is not needed.

3.2 Data Acquisition Strategy

A recurring challenge in AIOps research is the lack of public datasets that reflect the complexity of a carrier-grade 5G network. For this reason, the thesis uses proprietary telemetry generated inside Ericsson's 5G Core testbeds.

3.2.1 Data Source

The dataset consists of Kubernetes platform-level event logs collected from the Packet Core Controller (PCC) infrastructure. The schema of a Kubernetes event and the pod lifecycle that produces it are introduced in Section 2.1.4. The events are captured through Diagnostic Data Collection (DDC) snapshots taken at 15-minute intervals during test execution. Each snapshot contains an `events.v1.json` file listing every K8s event active at that point in time.
Each K8s event includes the following structured fields. The event `reason` (for example Pulled, Created, Started, Unhealthy, FailedMount), the event `type` (Normal or Warning), a `count` field that records how many times the event was observed, `firstTimestamp` and `lastTimestamp` fields, the name of the involved object (a pod, deployment or node), and a free-text `message` field.

3.2.2 Test Run Categories

Data was collected from 36 test runs covering three categories, chosen to span as many operational scenarios as possible.

- **Stability tests (7 runs):** Normal operational tests with no fault injection, representing baseline healthy behaviour. Durations range from 2 to 12 hours.
- **Robustness tests (22 runs):** Controlled fault-injection tests, including pod deletion (6 targets), process kills via SIGKILL (4 targets), kernel panic, NRF outage, full cluster restart, cluster hard reboot, certificate expiration, scale in and out, and bouncing network links.
- **Litmus Chaos tests (7 runs):** Controlled chaos engineering experiments orchestrated via the open-source LitmusChaos platform [38], including container kill (serial and parallel), CPU stress, and network faults (partition, packet loss, packet corruption).

The dataset spans software versions 1.67.0 through 1.78.0 and several cluster configurations (Cee3ABC, Mesos1B, Bm6xBC), so that the model is exposed to a range of environments during training.

3.2.3 Data Collection and Enrichment

For each test run, DDC tarballs are unpacked and the `events.v1.json` files are extracted. Events across snapshots are deduplicated by their unique identifier (UID), since the same event appears in multiple consecutive snapshots. The deduplicated events are sorted chronologically and enriched with the following computed features.

- **count_log:** $\log(\text{count} + 1)$, which captures event recurrence on a compressed scale.

- **duration_log:** log(lastTimestamp − firstTimestamp + 1) in seconds, which captures how long an event persists.
- **time_delta_log:** log(seconds since previous event+1), which captures burst patterns as opposed to isolated events.
- **pod_group:** The deployment-level grouping obtained by stripping ReplicaSet and StatefulSet hash suffixes from pod names. For example, `eric-pc-mm-mobility-6f587d7d5-sr96c` becomes `eric-pc-mm-mobility`.

3.2.4 Dataset Summary

The final dataset contains 86,873 events, of which 69,653 (80.2%) are Normal events and 17,220 (19.8%) are Warning events. It covers 57 unique event types and 165 common pod deployment groups. Both Normal and Warning events are kept in the dataset, because Normal lifecycle events such as Scheduled, Pulled, Created, Started and Killing provide the context needed to distinguish benign warnings from genuine anomalies.

3.2.5 Data Splitting Strategy

The data is split by *test run* rather than by individual event or snapshot, which is needed to avoid data leakage between the splits. Events from the same test run always stay in the same split. The split is stratified by test category.

- **Training set:** 24 runs, 55,198 sequences. Used for model training.
- **Validation set:** 6 runs, 12,461 sequences. Used for hyperparameter tuning and early stopping.
- **Test set:** 6 runs, 12,117 sequences. Reserved for final evaluation.

A further 5 test runs, drawn from test suites that do not appear in any of the three splits, are held back entirely for an out-of-distribution generalisation evaluation.

3.3 Artifact Design and Implementation

The prototype pipeline consists of four stages, which are event representation, sequence construction, next-event prediction modelling, and situation-aware post-processing. Figure 3.2 shows the end-to-end flow, from raw Diagnostic Data Collection (DDC) snapshots to the final ranked issue report.

3.3.1 Event Representation

Each K8s event is represented as a feature vector that combines learned categorical embeddings with continuous metadata features. This representation replaces the BERT message embeddings used in an earlier prototype iteration, which suffered from data leakage. In that earlier prototype, the message text encoded the event type directly, which made classification trivially easy (99.84% accuracy) without the model having to learn any sequential patterns.
The features per event are:

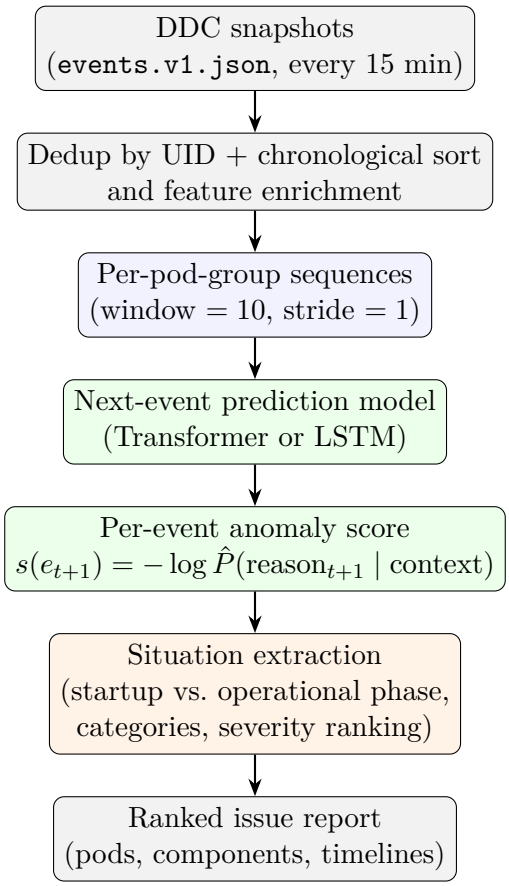


Figure 3.2: End-to-end pipeline of the proposed system. Grey blocks are data and pre-processing stages, the green blocks are the learned components (next-event prediction model and its anomaly score), and the orange block is the rule-based situation-extraction layer that encodes Kubernetes domain knowledge.

- **Reason embedding (16 dimensions):** A learned embedding for the event's reason field (57 unique values). The embedding is trained end-to-end with the model, so it can learn semantic similarity between event types.
- **Pod group embedding (16 dimensions):** A learned embedding for the deployment-level grouping (165 common groups). Pod groups that appear in fewer than 5 test runs are mapped to a shared `<OTHER>` token.
- **Type flag (1 dimension):** A binary indicator, with 0 for Normal events and 1 for Warning events.
- **count_log (1 dimension):** The log-transformed event count.
- **duration_log (1 dimension):** The log-transformed event duration in seconds.
- **time_delta_log (1 dimension):** The log-transformed time since the previous event for the same pod group.

The total input dimension per event is 36.

3.3.2 Sequence Construction

Events are grouped by pod deployment (pod_group) and ordered chronologically within each group. This per-pod-group sequencing is a central design decision. A global sequence that interleaves events from all pods achieved only 67% prediction accuracy, because the model cannot reliably predict which pod's event comes next in a mixed timeline. Per-pod-group sequences reached 86.6% accuracy, a 19.6 percentage point improvement, because events inside a single deployment follow predictable lifecycle patterns (Scheduled → Pulled → Created → Started).
A sliding window of size 10 (9 input events plus 1 prediction target) with stride 1 is applied to each pod group's event sequence. Sequences shorter than 9 events are left-padded with zeros, and a padding mask prevents the model from attending to padded positions. Pod groups with fewer than 3 events are excluded.

3.3.3 Next-Event Prediction Model

The core of the artifact is a self-supervised next-event prediction model that follows the DeepLog formulation [27]. Given a sequence of up to 9 preceding events for a pod deployment, the model predicts the probability distribution over 57 event types for the next event,

$$P(\text{reason}_{t+1} \mid e_{t-8}, e_{t-7}, \ldots, e_t). \tag{3.1}$$

The training objective is the cross-entropy loss between the predicted distribution and the actual next event's reason,

$$\mathcal{L} = -\log P(\text{reason}_{t+1} \mid e_{t-8}, \ldots, e_t; \theta). \tag{3.2}$$

No manual labels are required, because the training signal comes from the natural sequential structure of the data.
Two architectures are trained and compared (Figure 3.3).
Transformer encoder: The input features are projected to a 128-dimensional space through a linear layer, combined with sinusoidal positional encodings, and

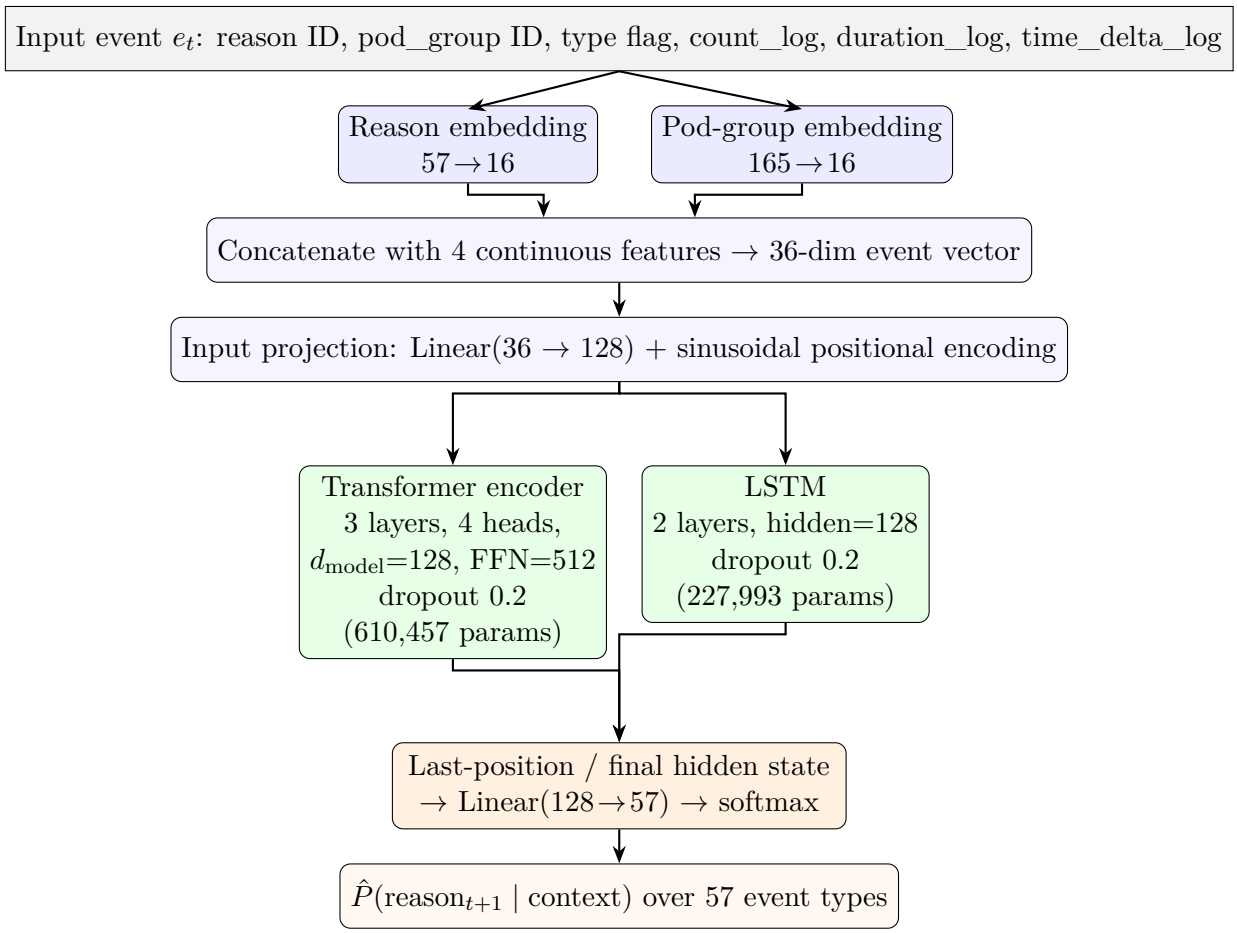


Figure 3.3: Input representation and model architectures. Both the Transformer encoder and the LSTM share the same event embedding and input projection (blue). They differ only in the sequence trunk (green). A shared classification head (orange) produces a probability distribution over the 57 event reasons, from which the anomaly score $-\log \hat{P}(\text{reason}_{t+1} \mid \text{context})$ is derived.

processed by 3 Transformer encoder layers (4 attention heads, feed-forward dimension 512, dropout 0.2). The output at the last sequence position is passed through a classification head. Total parameters: 610,457.
LSTM: The same input features are processed by a 2-layer LSTM (hidden dimension 128, dropout 0.2). The final hidden state is passed through a classification head. Total parameters: 227,993.
Both models are trained with the Adam optimiser [31] at a learning rate of 10^{-3}, a ReduceLROnPlateau scheduler (factor 0.5, patience 3), a batch size of 64, for up to 50 epochs, with early stopping based on validation accuracy. Dropout with probability 0.2 is applied throughout both architectures as regularisation [32].

3.3.4 Anomaly Scoring

At inference time, each event in a test run is scored by how surprising it is given the preceding context within the same pod group,

$$\text{anomaly\_score}(e_{t+1}) = -\log P(\text{reason}_{t+1} \mid \text{context}). \tag{3.3}$$

A score close to 0 means the model expected the event (normal behaviour). Scores above 3 indicate events that were clearly unexpected, and scores above 5 indicate very rare transitions given the context.

3.3.5 Situation-Aware Post-Processing

The model produces per-event anomaly scores. A rule-based post-processing layer then combines these scores with Kubernetes domain knowledge to classify each pod's overall situation. This separation of concerns lets the machine-learning model focus on the hard part, which is sequence anomaly scoring, while domain-specific lifecycle interpretation is handled by explicit rules.
Startup phase separation: Events occurring within the first 5 minutes after the earliest event in the log are classified as startup phase. Kubernetes cluster startup generates hundreds of warnings such as FailedMount and FailedScheduling that are normal initialisation behaviour. A single pod can have both startup-phase and operational-phase warnings, which are classified independently.
Situation categories:

- **STARTUP_NOISE:** All warnings occur in the startup phase and the pod eventually started successfully.
- **NORMAL_SHUTDOWN:** A Killing event precedes the warnings, and either FailedPreStopHook or Unhealthy is present.
- **TRANSIENT:** The event count is low ($\leq$10) and the pod recovered (a Started event was observed).
- **PERSISTENT:** The count is high (>20) or the duration is long (>1 hour), and no recovery was observed.
- **ANOMALOUS:** The mean anomaly score is above 3.0.

Severity ranking: Issues are ranked by a composite severity score,

$$\text{severity} = \overline{\text{anomaly\_score}} \times \text{total\_count} \times (1 + \text{max\_duration\_hours}). \tag{3.4}$$

This combines the ML signal (how unexpected the event was) with the metadata signal (how impactful it was), so that high-scoring events that also recur many times over a long duration end up at the top of the list.

3.4 Evaluation Methodology

The artifact is evaluated in two complementary ways. Quantitative model metrics measure the accuracy of the next-event prediction, and end-to-end evaluation checks the behaviour of the full pipeline against test runs with known fault injection.

3.4.1 Quantitative Model Evaluation

Next-event prediction accuracy is measured on the held-out test set and compared against two baselines.

- **Unigram baseline:** Always predicts the most common event type in the training set, which establishes the lower bound that any model must exceed.
- **Bigram baseline:** Predicts the most common successor of the immediately preceding event, which captures first-order transition statistics without deeper sequential context.

Per-class precision, recall and F1 score are computed to assess performance across both common and rare event types. The gap between the bigram baseline and the trained models quantifies the value of multi-event sequence context.

3.4.2 Generalisation Evaluation

To assess generalisation to unseen failure modes (RQ3), the model is evaluated on 5 test runs from completely new test suites that do not appear in any training, validation or test split.

3.4.3 End-to-End Evaluation

The full pipeline (scoring, situation extraction and severity ranking) is evaluated against 6 test cases with known fault injection, consisting of 4 cases used during development and 2 additional blind cases collected after the system was finalised. The model's output is compared against a manual expert analysis of the raw event logs for each case. The evaluation checks whether the system correctly identifies the primary affected components, ranks them by severity, and filters startup noise.

4

Results

This chapter reports the results of the artifact evaluation. Its structure mirrors Chapter 3. Section 4.1 summarises the collected dataset. Section 4.2 compares the two model architectures against the baselines. Section 4.3 evaluates generalisation to completely unseen test suites. Section 4.4 presents the end-to-end evaluation against known fault-injection scenarios and compares the final system with the initial prototype.

4.1 Dataset Characteristics

The data collection pipeline described in Section 3.2 produced a dataset of 86,873 Kubernetes events from 36 test runs. Table 4.1 summarises the composition by test category.

Table 4.1: Dataset composition by test category.

Category	Runs	Events	%
Stability	7	15,770	18.2%
Robustness	24	54,274	62.5%
East-West	5	16,829	19.4%
Total	**36**	**86,873**	100%

Of the 86,873 events, 69,653 (80.2%) are Normal lifecycle events and 17,220 (19.8%) are Warning events. The vocabulary construction process (Section 3.3.1) identified 57 event types (56 unique reasons plus a padding token) and 165 common pod deployment groups. Table 4.2 shows the distribution of the 20 most frequent event types. The three most common events, Pulled (14.0%), Created (13.9%) and Started (13.8%), correspond to the standard pod startup lifecycle sequence. The most frequent Warning event is FailedMount (9.8%), which predominantly occurs during cluster initialisation when secrets and ConfigMaps are not yet available.

The per-pod-group sequence construction (Section 3.3.2) with a sliding window of size 10 produced 79,776 sequences in total, split by test run as follows. 55,198 training sequences (69.2%), 12,461 validation sequences (15.6%) and 12,117 test sequences (15.2%).

Table 4.2: Distribution of the 20 most frequent K8s event types in the dataset.

Event Type	**Type**	**Count**	**%**
Pulled	Normal	12,128	14.0%
Created	Normal	12,040	13.9%
Started	Normal	11,981	13.8%
FailedMount	Warning	8,495	9.8%
AddedInterface	Normal	6,556	7.5%
Unhealthy	Warning	6,161	7.1%
SuccessfulCreate	Normal	6,077	7.0%
Pulling	Normal	5,940	6.8%
Scheduled	Normal	5,621	6.5%
ScalingReplicaSet	Normal	1,663	1.9%
FailedScheduling	Warning	1,425	1.6%
NoPods	Normal	1,272	1.5%
SuccessfulAttachVolume	Normal	1,066	1.2%
ExternalProvisioning	Normal	1,061	1.2%
Provisioning	Normal	1,061	1.2%
ProvisioningSucceeded	Normal	1,061	1.2%
Completed	Normal	621	0.7%
Killing	Normal	505	0.6%
FailedCreatePodSandBox	Warning	359	0.4%
deprecatedAnnotation	Warning	349	0.4%

4.2 Model Performance Comparison

4.2.1 Impact of Sequence Construction Strategy

One of the design decisions that was evaluated most carefully during development was the choice of sequence construction strategy (RQ2). Table 4.3 compares the Transformer's accuracy under the two sequencing approaches considered.

Table 4.3: Impact of sequence construction strategy on Transformer prediction accuracy.

Strategy	**Top-1 Accuracy**
Global sequences (all pods interleaved)	67.0%
Per-pod-group sequences (deployment-level)	86.6%
Improvement	**+19.6pp**

Global sequences, which interleave events from all 170+ pods in chronological order, reached only 67.0% accuracy. The model cannot reliably predict which pod's event comes next in a mixed timeline, since that depends on Kubernetes scheduler timing rather than on lifecycle logic. Per-pod-group sequences, which group events by deployment and order them chronologically inside each group, reached 86.6%. That is a 19.6 percentage point gain. The result confirms that the sequencing strategy is the single most impactful design decision, and it answers RQ2 directly. Grouping events by their originating deployment preserves the lifecycle patterns that the model needs to learn.

4.2.2 Overall Accuracy

Table 4.4 shows the overall performance of both model architectures compared to the two baselines described in Section 3.4.1.

Table 4.4: Overall model performance comparison on the held-out test set.

Model	**Top-1**	**Top-3**	**Top-5**	**F1 (w)**	**F1 (m)**	**Params**
Unigram baseline	14.1%	—	—	—	—	0
Bigram baseline	72.9%	—	—	—	—	0
LSTM	86.1%	97.9%	99.0%	0.857	0.681	227,993
Transformer	86.6%	97.6%	99.0%	0.861	0.664	610,457

Both the LSTM and the Transformer clearly outperform the baselines. The unigram baseline, which always predicts the most common event type (Pulled), reaches only 14.1%, which reflects the diversity of the 57-class vocabulary. The bigram baseline, which predicts the most common successor of the immediately preceding event, reaches 72.9%. Both trained models improve over the bigram baseline by roughly 13.5 percentage points, which shows that sequence context beyond the immediately preceding event provides real predictive value.

The Transformer has a slightly higher top-1 accuracy (86.6% against 86.1%) and weighted F1 score (0.861 against 0.857). The LSTM has a higher macro F1 score (0.681 against 0.664), which means it is slightly better on the rare event classes. The two models have essentially identical top-3 and top-5 accuracy (around 98% and 99% respectively), so the correct event type is almost always among the model's top predictions. The LSTM achieves this with 2.7× fewer parameters (227,993 against 610,457).

4.2.3 Per-Class Performance

Table 4.5 shows the per-class precision, recall and F1 score for the Transformer on the 18 event types with more than 10 test samples. Classes with fewer samples are left out because their metrics are not statistically reliable.

Table 4.5: Per-class precision, recall and F1 score for the Transformer model on event types with support > 10.

Event Type	Prec.	Rec.	F1	Support
Created	96.8%	98.3%	97.6%	1,685
Started	94.9%	97.3%	96.1%	1,658
SuccessfulCreate	95.3%	96.2%	95.8%	506
deprecatedAnnotation	82.9%	100.0%	90.6%	58
ProvisioningSucceeded	88.2%	91.7%	89.9%	180
Unhealthy	89.3%	89.3%	89.3%	922
FailedMount	87.5%	88.4%	87.9%	1,692
Scheduled	88.2%	86.2%	87.2%	580
NoPods	87.7%	83.3%	85.5%	60
Pulled	87.0%	81.0%	83.9%	1,714
ExternalProvisioning	89.3%	73.6%	80.7%	148
Provisioning	75.0%	85.4%	79.9%	137
Pulling	75.3%	84.7%	79.8%	990
FailedScheduling	76.1%	82.7%	79.3%	185
ScalingReplicaSet	74.9%	82.6%	78.5%	155
SuccessfulAttachVolume	79.7%	63.3%	70.6%	180
AddedInterface	66.2%	72.8%	69.4%	924
FailedCreatePodSandBox	48.1%	9.3%	15.6%	140

Events with highly predictable positions in the pod lifecycle achieve the highest F1 scores. Created reaches 97.6%, Started 96.1%, and SuccessfulCreate 95.8%. These events almost always follow a specific predecessor (Created follows Pulled, for example), which makes them easy for the model to predict.
The Warning event types most relevant to anomaly detection, Unhealthy (89.3% F1) and FailedMount (87.9% F1), are predicted with high accuracy. This indicates that the model has learned the contexts in which these events typically occur. The same fact matters for the downstream anomaly scoring. When the model confidently predicts Unhealthy in a context where it is expected (for example after a Killing event

during shutdown), the anomaly score is low. When Unhealthy occurs unexpectedly (for example after a Started event), the anomaly score is high. This behaviour is the empirical manifestation of the contextual-anomaly framing introduced in Section 2.5.1. The same Warning reason is benign or problematic depending on the preceding events in its sequence [20].
The weakest class is FailedCreatePodSandBox (15.6% F1), which the model frequently confuses with AddedInterface. Both events occur during the pod creation phase and their relative ordering is not strictly deterministic in Kubernetes, so this confusion reflects a limitation of the event model itself rather than a deficiency of the learned model.

4.2.4 Architecture Comparison

Table 4.6 compares the per-class F1 scores of the Transformer and the LSTM on all event types with more than 50 test samples, which is the comparison most relevant to RQ1.

Table 4.6: Per-class F1 score comparison between Transformer and LSTM on classes with support > 50.

Event Type	**Transformer**	**LSTM**	**Diff (pp)**
Pulled	83.9%	83.0%	+0.9
FailedMount	87.9%	87.0%	+0.9
Created	97.6%	97.0%	+0.6
Started	96.1%	95.3%	+0.8
Pulling	79.8%	80.5%	−0.7
AddedInterface	69.4%	68.6%	+0.8
Unhealthy	89.3%	88.8%	+0.4
Scheduled	87.2%	87.9%	−0.7
SuccessfulCreate	95.8%	95.4%	+0.4
FailedScheduling	79.3%	82.0%	−2.8
ProvisioningSucceeded	89.9%	93.3%	−3.3
SuccessfulAttachVolume	70.6%	70.3%	+0.3
ScalingReplicaSet	78.5%	78.5%	0.0
ExternalProvisioning	80.7%	79.4%	+1.4
FailedCreatePodSandBox	15.6%	15.4%	+0.2
Provisioning	79.9%	79.9%	0.0
NoPods	85.5%	87.0%	−1.5
deprecatedAnnotation	90.6%	90.3%	+0.3

The two architectures perform comparably across all major event types, with per-class differences never exceeding 3.3 percentage points. The Transformer is marginally better on common lifecycle events (Created by +0.6pp, Started by +0.8pp, Pulled by +0.9pp). The LSTM is marginally better on scheduling and provisioning events (FailedScheduling by −2.8pp, ProvisioningSucceeded by −3.3pp). Neither architecture shows a clear and consistent advantage over the other.

Table 4.7 shows the most frequent prediction errors for the Transformer.

Table 4.7: Top 10 confusion pairs for the Transformer model on the test set.

Actual	Predicted As	Count
Pulled	Pulling	154
FailedMount	AddedInterface	120
AddedInterface	FailedMount	110
FailedCreatePodSandBox	AddedInterface	78
Pulling	Pulled	74
AddedInterface	Pulling	69
Pulled	Started	55
Pulling	AddedInterface	54
Unhealthy	Pulled	49
Pulled	AddedInterface	41

The dominant confusion pattern involves events that occur in the same lifecycle phase. The most frequent confusion is between Pulled and Pulling ($154 + 74 = 228$ bidirectional errors), which are adjacent steps in the image retrieval process whose relative ordering depends on Kubernetes scheduler timing. FailedMount and AddedInterface are a similar case ($120+110 = 230$ bidirectional errors). Both occur during the pod creation phase and can appear in variable order. These confusions represent an inherent ceiling imposed by the non-deterministic ordering of Kubernetes events rather than a limitation of the model architecture.

4.3 Generalisation to Unseen Data

To evaluate generalisation to previously unseen failure modes (RQ3), the Transformer model was evaluated on 5 test runs from completely new test suites that were not present in any training, validation or test split. The event types and pod group names in these runs do overlap with the training vocabulary (the model has seen the same K8s event reasons and deployment names before), but the specific test scenarios, fault-injection patterns and temporal dynamics are entirely new. Table 4.8 presents the results.

The model reaches 89.3% overall accuracy on the unseen test suites, compared with 86.6% on the standard held-out test set. The slightly higher accuracy on unseen data is explained by the composition of the two sets. The standard test set includes east-west runs with longer and more complex sequences from extended multi-day test campaigns. The unseen runs are predominantly shorter robustness tests with more predictable lifecycle patterns. This difference does not indicate overfitting to the test set. It reflects the fact that the unseen runs happen to contain a higher proportion of predictable lifecycle transitions.

Accuracy is consistent across the unseen test types, ranging from 88.4% on cluster hard reboot to 90.7% on certificate expiration. The cluster hard reboot test is the hardest because a full cluster restart triggers highly unpredictable recovery patterns,

Table 4.8: Transformer model accuracy on 5 completely unseen test suites not present in any training, validation or test split.

Test Run	Category	Accuracy
stability_vtap	Stability	89.1%
rob_cert_expiry	Robustness	90.7%
rob_bouncing_links	Robustness	89.9%
litmus_podpartition	Robustness	89.1%
rob_cluster_hard_reboot	Robustness	88.4%
Overall	Mixed	**89.3%**

with many pods restarting simultaneously in non-deterministic order. Even so, the model holds up on this extreme scenario, which is a reasonable indication of generalisation to failure modes not encountered during training.

4.4 End-to-End Evaluation

The full pipeline (event scoring, situation extraction and severity ranking) was evaluated against 6 test cases with known fault injection. The model's output was compared against a manual expert analysis of the raw event logs in each case.

4.4.1 Situation Extraction Accuracy

Table 4.9 summarises the end-to-end evaluation results. The first 4 test cases were used during development. The last 2 (marked with †) were collected and analysed after the system was finalised, serving as a blind evaluation.

Table 4.9: End-to-end evaluation results on 6 test cases with known fault injection. † denotes a blind evaluation on data collected after the system was finalised.

Test Case	Events	Warnings	Noise filtered	Primary identified	Recovery tracked
Litmus Net. Corruption	2,278	335	60%	✓	✓
NRF Failover	2,238	335	76%	✓	✓
SigKill Controller	2,096	409	65%	✓	✓
Restart All	2,136	468	66%	✓	✓
NRF Failover†	2,082	335	76%	✓	✓
SigKill SCTP†	1,960	409	31%	✓	✓

In all 6 test cases, the system correctly identified the primary affected components.

- **Litmus Chaos Network Corruption:** The system ranked `snmp-alarm-provider` as the #1 issue (severity = 88), with a 2.5-hour failure duration. This is the component that the manual analysis also identified as most severely affected.

The cascading impact on `pc-mm-sctp` (severity = 40) and `pc-mm-controller` was also detected.

- **NRF Failover:** The system detected all affected `pc-mm-*` components (sctp: 8 pods, severity = 26; forwarder: 6 pods, severity = 23; mobility: 4 pods). It also correctly flagged `pc-mm-controller` as having persistent issues with no recovery.
- **SigKill Controller Active:** The system ranked `pc-mm-controller` as the #1 issue (severity = 231), which is the kill target. The cascading impact on `pc-mm-mobility` (7 pods) and dependent components was captured correctly.
- **Restart All:** The system identified `pc-mm-mobility` (8 pods, severity = 198) and `pc-mm-sctp` (8 pods, severity = 140) as the slowest components to recover, matching the manual analysis.

The startup noise filter correctly classified 60 to 76% of pods with warnings as noise in 5 of the 6 test cases, removing FailedMount and FailedScheduling events that occur during normal cluster initialisation. This filtering is essential for practical usability. Without it, hundreds of benign startup warnings would obscure the infrastructure issues that actually matter.

4.4.1.1 Blind Evaluation: Strengths and Limitations

The two blind test cases revealed both strengths and limitations of the system.

Strengths: In the blind SigKill SCTP test, the system ranked `eric-cm-yang-provider` as the #1 issue (severity = 13,069), driven by a FailedToRetrieveImagePullSecret event with count = 148 over 170 minutes. This event type is extremely rare in the training data, and the model assigned it an anomaly score of 23.03. That confirms that the model assigns high surprise scores to events it has rarely or never encountered during training. The system also identified `eric-pm-server` (RecreatingFailedPod, severity = 207) as the #2 issue, which is another rare event type. All affected `pc-mm-*` components were detected.

Limitation 1 — severity ranking of recovered components: In both blind tests, the `pc-mm-*` components (the actual fault-injection targets) were ranked with low severity (3 to 4), because they recovered quickly and had low per-pod warning counts. The severity formula (Equation 3.4) weights anomaly score by event count and duration, so components that self-heal rapidly receive low scores even when the operational impact of 8 pods simultaneously failing readiness probes is significant. In the NRF Failover test, `data-object-storage` (a startup FailedMount issue unrelated to the NRF fault) was ranked #1 (severity = 38), above the actual NRF-affected components.

Limitation 2 — no cross-component correlation: The system analyses each pod independently and does not aggregate coordinated failure patterns. In the NRF Failover test, the simultaneous Unhealthy wave across all `pc-mm-*` components at minute 5 to 7 is the defining event of the test, but the system treats each pod separately. An incident-level correlation layer that detects when multiple components fail at the same time would address this limitation.

Limitation 3 — variable noise filtering effectiveness: The SigKill SCTP test achieved only 31% noise filtering (37 of 121 pods), compared to 76% in the NRF Failover test. This is because the SigKill test produces more operational-phase

warnings (FailedToUpdateEndpoint, RecreatingFailedPod) that do not match the startup noise patterns. The result is 84 reported issues, many of them minor, which increases the cognitive load on the operator.

4.4.2 Comparison with Initial Prototype

Table 4.10 compares the initial prototype (PoC v1) with the final system (PoC v2).

Table 4.10: Comparison between the initial prototype (PoC v1) and the final system (PoC v2).

Aspect	**PoC v1**	**PoC v2**
Task	Event classification	Next-event prediction
Input	BERT message embedding	Reason ID + metadata
Labels	Event reason (from input)	Next event (self-supervised)
Data leakage	Yes	No
Accuracy	99.84% (trivially easy)	86.6% (genuinely hard)
Sequence use	None (no effect)	Essential (+19.6pp over global)
Normal events	Discarded	Included (80% of data)
Startup noise	Not handled	Filtered by phase separation
Supervisor test	Failed (identical output)	Passed (correctly distinguishes)
Anomaly scoring	Meaningless	Based on prediction surprise

The initial prototype used BERT embeddings of the event message text to classify events into 9 anomaly categories, and reached 99.84% accuracy. That accuracy was an artifact of data leakage. The message text encodes the event type directly ("Readiness probe failed" maps trivially to Unhealthy), so the task reduced to keyword matching rather than genuine anomaly detection. The sequence modelling capability of the LSTM and Transformer provided no benefit. Accuracy was identical with and without temporal context, which confirmed that the model was exploiting textual shortcuts rather than learning sequential patterns. When tested on the supervisor's two reference test cases, a stability test where the system recovered and a robustness test where it did not, the initial prototype produced identical output for both. It could not distinguish normal operational warnings from genuine infrastructure failures.

The final system addresses that limitation by reformulating the task as self-supervised next-event prediction. By removing the message text entirely and using only the event reason ID together with metadata features, the model is forced to learn sequential patterns rather than exploit textual shortcuts. The 86.6% accuracy is numerically lower than the initial prototype's 99.84%, but it reflects a genuinely difficult prediction task where sequence context is essential. That is visible in the 19.6 percentage point improvement of per-pod-group sequences over global sequences, and in the 13.7 percentage point improvement over the bigram baseline. On the supervisor's reference test cases, the final system correctly identifies the affected components in the robustness test while classifying the stability test as clean, which is the behaviour that motivated the redesign in the first place.

5
Discussion

This chapter interprets the results of Chapter 4, addresses the three research questions, places the findings in the broader literature, discusses what the work means in practice for Ericsson, and examines the limitations of the study.

5.1 Addressing the Research Questions

5.1.1 RQ1: LSTM versus Transformer for Log Anomaly Detection

What are the strengths and limitations of different sequence modelling architectures for anomaly detection in cloud-native infrastructure environments?

The LSTM and Transformer architectures were selected for comparison based on the findings of the narrative literature survey (Chapter 2). The LSTM is the dominant architecture in the foundational log anomaly detection literature: DeepLog [27] established the next-event prediction paradigm using LSTMs, and subsequent work such as LogAnomaly [41] continued in that tradition. The Transformer, meanwhile, has become the state-of-the-art architecture for sequence modelling in NLP [29] and has been adopted for log analysis by LogBERT [35]. Comparing these two families therefore tests whether the architectural advances that dominate general sequence modelling translate into meaningful gains in the specific domain of infrastructure event logs.

The results suggest that neither architecture holds a decisive advantage in this domain. The Transformer has a slightly higher top-1 accuracy (86.6% against 86.1%) and a slightly higher weighted F1 score (0.861 against 0.857). The LSTM has a slightly higher macro F1 score (0.681 against 0.664) with 2.7× fewer parameters. Per-class differences do not exceed 3.3 percentage points on any event type with substantial test support.

This is at odds with the broader NLP literature, where Transformers have a clear advantage over LSTMs on tasks that involve long-range dependencies and very large vocabularies [29]. Two domain-specific factors explain the parity observed here. First, the effective sequence length is short. With a window of 9 events and per-pod-group sequencing, the model rarely needs to attend to events more than five to seven steps back. LSTMs handle short-range dependencies well, so the Transformer's self-attention does not add much at this scale. Second, the vocabulary is small. With only 57 event types, compared with the 30,000+ tokens of typical natural-language

tasks, the prediction problem is simpler overall, which again limits the advantage of parallel attention over sequential processing.
From a deployment perspective, the LSTM's smaller parameter count (227,993 against 610,457) translates into faster inference and lower memory use, which is relevant for integration into resource-constrained CI/CD pipelines. Given that the accuracy difference is negligible, the LSTM is arguably the more pragmatic choice for production, while the Transformer has a slight edge on the most common event types.
The near-identical top-3 accuracy (around 98%) of the two architectures is particularly relevant for the anomaly-detection use case. In the DeepLog framework [27], an event is flagged as anomalous only if it does not appear among the model's top-k predictions. At 98% top-3 accuracy, both models would produce very similar anomaly-detection behaviour in practice, which reinforces the overall conclusion that the choice of architecture matters less than the choice of event representation and sequencing strategy.

5.1.2 RQ2: Event Representation and Sequencing

How can infrastructure event logs be represented and sequenced to preserve meaningful contextual and temporal information for anomaly detection?

The results identify two design decisions that together determine most of the system's effectiveness, both more impactful than the choice of model architecture.
Sequencing strategy: The per-pod-group sequencing approach improved prediction accuracy from 67.0% (global sequences) to 86.6% (per-pod-group sequences). The 19.6 percentage point gain dwarfs the 0.5pp difference between LSTM and Transformer. The reason is straightforward. Kubernetes events inside a single deployment follow predictable lifecycle patterns (Scheduled → Pulled → Created → Started). Global sequences instead interleave events from more than 170 pods whose relative ordering depends on scheduler timing rather than on causal relationships. Grouping events at the deployment level lets the model learn these lifecycle patterns without the noise of cross-pod interleaving.
This finding extends the DeepLog approach [27], which was originally designed for single-threaded system logs where events naturally form one sequence. In a distributed Kubernetes environment with hundreds of concurrent pods, applying DeepLog naively to a global event stream would not work, which is exactly what the 67% result shows. The per-pod-group adaptation is therefore a necessary modification for applying next-event prediction to container-orchestration logs.
Event representation: The decision to use learned categorical embeddings of the event reason field (16 dimensions) rather than BERT embeddings of the message text (768 dimensions) was driven by the data leakage found in the initial prototype. The message text encodes the event type directly. For example, "Readiness probe failed: HTTP probe failed with statuscode: 503" maps trivially to Unhealthy. A text-based classifier can therefore reach near-perfect accuracy without learning sequential patterns. Using only the reason ID, a 57-class categorical variable, forces the model to rely on sequential context to make predictions.
This challenges the assumption in LogAnomaly [41] and LogRobust [42] that se-

mantic embeddings of log messages are necessary for effective anomaly detection. In the Kubernetes event domain, the structured `reason` field already captures the event semantics. The free-text message adds no predictive value beyond what the reason provides, and it also introduces a serious data-leakage risk. For structured log sources with well-defined event taxonomies, simple categorical embeddings may therefore be preferable to computationally expensive language-model embeddings.
Adding the continuous metadata features (count, duration, time delta) alongside the categorical embeddings gives the model severity and temporal context that pure template-based approaches lack. The event count field is particularly important. A single Unhealthy event (count = 1) is qualitatively different from a persistent Unhealthy event (count = 156), and the log-transformed count captures that distinction.

5.1.3 RQ3: Generalisation to Unseen Failure Modes

To what extent can the proposed prototype generalise to previously unseen failure modes?

The model generalises well across two evaluation dimensions.
Prediction accuracy on unseen test suites: The Transformer achieves 89.3% accuracy on 5 completely unseen test suites (stability VTap, certificate expiration, bouncing links, Litmus pod partition and cluster hard reboot), compared with 86.6% on the standard held-out test set. The accuracy is consistent across diverse failure modes, from certificate expiration (90.7%) to full cluster hard reboot (88.4%), which suggests that the model has learned generalisable lifecycle patterns rather than memorising test-specific sequences.
End-to-end detection of fault-injection targets: Across all 6 evaluated test cases, the system correctly identified the primary affected components. That includes the 2 blind test cases (NRF Failover v1.70.0 and SigKill SCTP v1.63.0) that were analysed after the system was finalised. The model also assigned very high anomaly scores (23.03) to event types it had never seen during training, such as FailedToRetrieveImagePullSecret and RecreatingFailedPod. This shows that the negative log-likelihood scoring handles novel events naturally. Any event type with near-zero predicted probability receives a high anomaly score, regardless of whether it appeared in the training data.
This generalisation behaviour is consistent with the theoretical basis of the DeepLog approach [27]. By learning the probability distribution of normal event sequences, the model implicitly defines anomalies as low-probability events. The self-supervised training objective requires no labelled anomaly data, and the model generalises to any deviation from the learned patterns, whether the deviation was caused by pod deletion, kernel panic, network corruption or certificate expiration.
Generalisation does have a clear boundary, however. The model generalises to new *test scenarios* but not to new *event vocabularies*. If a software update introduces entirely new event types that are not in the 57-class vocabulary, the model maps them to the unknown token and assigns high anomaly scores by default. This is a safe failure mode (novel events are flagged rather than ignored), but periodic retraining is still required to incorporate new event types as the software evolves.

5.2 Interpretation of Dataset and Model Characteristics

5.2.1 Class Imbalance and Its Effect on Anomaly Scoring

The dataset shows a pronounced class imbalance. The top 9 event types account for roughly 90% of all events, while the remaining 48 types share the other 10%. In addition, 80.2% of events are Normal lifecycle events and only 19.8% are Warnings. The imbalance is not a data-collection artefact. It is a faithful reflection of operational reality. In a healthy Kubernetes cluster, most events are routine lifecycle transitions (Scheduled, Pulled, Created, Started), and Warning events are rare by design.

For next-event prediction, this imbalance is actually useful for anomaly detection. The model learns to predict Normal lifecycle events with high confidence (Created at 97.6% F1, Started at 96.1% F1). When a Warning event occurs in a context where the model expected a Normal event, the negative log-likelihood score is naturally high, which is a strong anomaly signal. The class imbalance therefore aligns the training objective with the detection objective. Common events are predicted confidently, and deviations from common patterns get high anomaly scores.

The imbalance does affect the model's ability to predict *which specific* Warning event will occur. FailedCreatePodSandBox (15.6% F1) and Killing (7.4% F1) are predicted poorly because they are rare and occur in contexts where multiple Warning types are plausible. For anomaly detection this is acceptable. The model does not need to predict the exact Warning type, only that the actual event was unexpected given the context.

5.2.2 Confusion Pairs and Anomaly Detection Implications

The dominant confusion pairs (Pulled↔Pulling, FailedMount↔AddedInterface) occur between events in the same lifecycle phase. For anomaly detection these confusions have minimal impact, because both events in each pair are expected in the same context. When the model predicts Pulling but the actual event is Pulled, the anomaly score stays low because Pulled still has high probability in the model's output distribution. The confusion affects top-1 accuracy but not the anomaly scoring, which uses the full probability distribution rather than just the top prediction.

The confusion that actually matters for anomaly detection is Unhealthy → Pulled (49 instances), where the model predicts a Normal lifecycle event but a Warning actually occurs. In these cases, the model correctly assigns a moderate anomaly score to the Unhealthy event, which is the desired behaviour. The model expected the pod to continue its normal lifecycle, and a probe failure shows up as a deviation from that expectation.

5.2.3 Training on Mixed Normal and Warning Data

A deliberate design choice was to train on all events, Normal and Warning, rather than training only on Normal events as in the original DeepLog formulation [27].

This was motivated by the observation that there is no purely "normal" data in the Kubernetes domain. Every test run, including stability tests, contains Warning events, for example FailedMount during startup or transient Unhealthy during initialisation. Training only on Normal events would exclude 19.8% of the data and would prevent the model from learning the contexts in which Warnings are expected, such as FailedMount during startup or Unhealthy after Killing during shutdown.
The consequence is that the model learns to predict certain Warning events with high confidence in specific contexts, which means those Warnings receive low anomaly scores. That is the correct behaviour. A FailedMount during startup is expected and should not be flagged as anomalous. The situation extraction layer then provides the second filter, classifying these expected Warnings as startup noise based on their temporal position.

5.3 The Role of Domain Knowledge in Anomaly Detection

One of the more architectural decisions in this thesis is the separation of the machine-learning model (next-event prediction) from the domain-specific post-processing layer (situation extraction). The separation is worth discussing, because it diverges from the end-to-end deep learning pattern that dominates the recent literature.
The distinction between a benign startup `FailedMount` and a problematic operational `FailedMount` is a canonical example of a *contextual anomaly* in the sense of Chandola, Banerjee and Kumar [20]. The same point value is anomalous only relative to its temporal context. The sequence model does capture part of this context through the preceding events in the window, but the coarsest contextual feature, which is whether the cluster is still in its initialisation phase, is most naturally supplied by an explicit rule on wall-clock time rather than learned from the limited amount of startup data in each run.
The situation extraction layer therefore encodes explicit Kubernetes domain knowledge. Startup noise occurs in the first five minutes. Killing before Unhealthy indicates normal shutdown. A Started event after warnings indicates recovery. These rules are deterministic and well understood by any Kubernetes operator. Making the machine-learning model learn them from data would need substantially more training data and would produce a less interpretable system.
The results support this design. The startup noise filter correctly classified 60 to 76% of warning pods as noise in 5 of 6 test cases, removing hundreds of benign FailedMount and FailedScheduling events. Without this filter, the output would be dominated by initialisation warnings that occur in every test run, which would make the tool impractical for SRE use. The machine-learning model on its own cannot separate a benign startup FailedMount from a problematic operational FailedMount, because both receive similar anomaly scores. Distinguishing them requires knowing the temporal position of the event relative to cluster startup, which is domain knowledge rather than a learned pattern.
This hybrid approach, where machine learning handles the hard problem (sequence anomaly scoring) and rules handle the well-understood problem (lifecycle interpre-

tation), is consistent with the AIOps pipeline described by Dang et al. [17], in which detection, correlation and interpretation are treated as distinct stages rather than as a monolithic model. It is also in line with the broader observation of Pang et al. [21] that deep anomaly detection benefits from combining learned detectors with domain priors rather than replacing them.

5.4 Contributions to the Practice

5.4.1 Operational Integration

The prototype is designed as a post-hoc analysis tool that processes DDC snapshots after a test run has finished. In its current form it can slot into the CI/CD pipeline as a post-test analysis step. Once a test run completes and DDC data is collected, the tool analyses the events and produces a ranked issue report. Test engineers can then triage the infrastructure problems it surfaces without having to inspect thousands of log events by hand.

Inference is dominated by the sequential per-event scoring, which processes around 2,000 events in under 10 seconds on a CPU node. That is well within the acceptable latency for post-test analysis, where results are needed within minutes rather than milliseconds.

5.4.2 Reduction of Manual Effort

The startup noise filter on its own provides useful value. In the test cases used for evaluation, 60 to 76% of pods with warnings were correctly classified as noise, which reduced the number of issues needing human attention from more than 100 to fewer than 40 in typical test runs. For the stability test case, the system reported zero significant issues, matching the manual assessment that the test was clean. A naive approach would have flagged all 468 warnings.

The severity ranking also reduces cognitive load by prioritising the most impactful issues. In the SigKill Controller test, the system ranked the kill target (`pc-mm-controller`, severity = 231) above the 30 or so other affected components, which points attention at the root cause rather than the cascading symptoms.

5.4.3 Cross-Test-Run Analysis

The current prototype analyses individual test runs, but the component-level severity aggregation provides a foundation for cross-test-run analysis. By running the tool across many test runs and aggregating component severity scores, Ericsson could identify which infrastructure components are most frequently affected across different test types. That would surface systemic weaknesses that do not cause test failures on their own but still degrade operational resilience over time. The industrial supervisor identified this capability as a priority, and it is a natural extension of the current work.

5.5 Contributions to the Theory

5.5.1 Positioning Relative to DeepLog

The implemented system follows the DeepLog formulation [27], but introduces three adaptations that are needed in the Kubernetes domain. First, per-pod-group sequencing replaces the single global sequence assumed by DeepLog, which addresses the multi-tenant nature of container orchestration. Second, the event representation includes continuous metadata features (count, duration, time delta) alongside the categorical event type, which provides severity context that DeepLog's template-ID-only representation lacks. Third, the situation extraction post-processing layer turns per-event anomaly scores into pod-level and component-level assessments, bridging the gap between raw scores and actionable infrastructure insights.
The prediction accuracy of 86.6% is not directly comparable to DeepLog's reported results on HDFS logs, because the datasets, vocabularies and evaluation protocols differ substantially. The 13.7pp improvement over the bigram baseline and the 19.6pp improvement from per-pod-group sequencing do, however, show that the model learns meaningful sequential patterns beyond simple transition frequencies.

5.5.2 Positioning Relative to LogBERT

The initial prototype (PoC v1) implemented a LogBERT-style approach [35] using BERT embeddings and masked prediction. The data leakage discovered in that prototype, where the message text directly encoded the prediction target, highlights a risk that is specific to structured log sources with well-defined event taxonomies. In the HDFS and BGL datasets that LogBERT was originally evaluated on, log messages are more diverse and less directly correlated with the template ID, which makes text-based approaches reasonable there. In the Kubernetes event domain, where the `reason` field is a controlled vocabulary that maps one-to-one with the message content, text-based approaches offer no benefit and introduce leakage risk.
The choice of autoregressive next-event prediction over masked prediction is also supported by the small vocabulary size (57 event types). With bidirectional context, predicting a masked event from both predecessors and successors is substantially easier than predicting the next event from only the past. The autoregressive formulation is therefore a harder task, and it discriminates more cleanly between expected and unexpected events, which is what anomaly detection needs.

5.5.3 Evaluation on Proprietary versus Public Data

One notable difference between this thesis and the existing literature is the use of proprietary Ericsson data rather than public benchmarks such as HDFS, BGL or Thunderbird. This limits direct numerical comparison with published results, but it also gives a more realistic picture of the system's practical utility. Public datasets are typically pre-processed, pre-labelled and well studied, with known failure patterns that may already be implicitly encoded in their evaluation protocols. The Ericsson data, by contrast, contains the full complexity of a production-grade 5G

Core testbed, including 170+ pod deployments, 57 event types, startup noise, rolling restarts and the non-deterministic event ordering inherent to Kubernetes. The system's ability to correctly identify the fault-injection targets across 6 diverse test cases provides stronger evidence of practical utility than high accuracy on a curated benchmark.

5.6 Limitations

5.6.1 Per-Pod Analysis Without Incident Correlation

The most significant limitation identified during evaluation is the lack of cross-component correlation. The system analyses each pod independently, which means that coordinated failures where multiple components fail at the same time due to a shared root cause are reported as separate, low-severity issues rather than as a single high-severity incident. In the NRF Failover test, the simultaneous Unhealthy wave across all `pc-mm-*` components was the defining event, but the system ranked each pod's brief failure as minor because each pod recovered on its own.
This limitation is inherent to the per-pod situation-extraction design and could be addressed by an additional incident correlation layer that detects temporal co-occurrence of warnings across multiple components. Such a layer would group simultaneous failures into incidents and assign aggregate severity scores, which would better reflect the operational impact of coordinated failures.

5.6.2 Severity Ranking Bias Toward Persistence

The severity formula ($\overline{\text{anomaly\_score}} \times \text{total\_count} \times (1 + \text{max\_duration\_hours})$) inherently favours persistent, high-count events over brief but operationally significant ones. In the blind SigKill SCTP test, `eric-cm-yang-provider` (an image pull secret issue unrelated to the fault injection) was ranked #1 with severity = 13,069, while the actual SIGKILL target (`pc-mm-sctp`) received severity = 3 because it recovered quickly. The formula identifies the most persistent anomaly correctly, but it does not capture the operational significance of a brief but widespread failure.
A potential improvement is to add a breadth factor that accounts for the number of simultaneously affected pods inside a deployment, so that a single pod with a persistent issue can be distinguished from 8 pods that all fail briefly at the same time.

5.6.3 Testbed Data versus Organic Production Failures

All training and evaluation data come from controlled testbed environments in which faults are deliberately injected. The test scenarios are designed to simulate realistic failure modes, but they may not fully represent the characteristics of organic production failures, which can involve gradual degradation, intermittent connectivity issues, or complex multi-component cascades that develop over hours or days. The system's behaviour on such organic failures has therefore not been directly evaluated.

Beyond that, the testbed environment has a relatively stable software configuration within each test run, whereas production environments undergo continuous rolling updates that may introduce new event types or alter lifecycle patterns. The model's vocabulary is fixed at training time. Events from new software versions that introduce previously unseen reason types would be mapped to the unknown token. That is a safe failure mode (novel events receive high anomaly scores), but it means periodic retraining will be needed as the software evolves.

5.6.4 Startup Phase Heuristic

The startup noise filter uses a fixed 5-minute window from the earliest event in the log to classify events as startup phase. This heuristic works well on the testbed data, where cluster startup consistently completes within 3 to 5 minutes. In production environments with larger clusters or slower storage backends, startup may take longer, and a fixed window could misclassify late startup events as operational issues. An adaptive approach that detects the end of the startup phase from the event pattern, for example when the rate of FailedMount events falls to zero, would be more robust to these variations.

5.6.5 Absence of Temporal Evaluation Metrics

The model is evaluated using standard classification metrics (accuracy, precision, recall, F1) applied to next-event prediction. These metrics treat each prediction independently and do not account for the temporal structure of anomaly detection, where detecting an anomaly within a time window matters more than detecting the exact event. Range-based evaluation metrics, as proposed by Tatbul et al. [40] and surveyed in Section 2.8, would provide a more realistic assessment of the system's detection capability, but they were not implemented here because ground-truth anomaly labels with precise temporal boundaries are not available. Wu and Keogh have further argued that even with such metrics, many existing benchmarks can be so easy that trivial heuristics appear competitive with learned models [39], which reinforces the value of pairing quantitative metrics with qualitative end-to-end evaluation on realistic fault-injection scenarios, as done in this thesis.

5.6.6 Data Scope

As defined in the delimitations (Section 1.5), the analysis is limited to Kubernetes platform-level events. Application-level logs, which contain business-logic errors and service-specific failure patterns, are excluded. A complete observability solution would need to correlate platform events with application logs and network metrics to provide full root-cause analysis. The current system identifies *which* components are affected and *when*, but it cannot determine *why* at the application-logic level.

5.7 Future Work

The evaluation also surfaced a small number of limitations that point to a coherent set of directions for future work. These extensions build on the existing artifact rather than replacing it, and most of them can be implemented as additional post-processing layers on top of the current model.

Incident-level correlation: The most significant practical limitation is that the system analyses each pod independently. Coordinated failures across multiple deployments, for example the simultaneous readiness-probe wave across all `pc-mm-*` components during NRF failover, are reported as a set of low-severity per-pod issues rather than as a single high-severity incident. An incident correlation layer that groups warnings co-occurring within a short time window (for example 2 minutes) across deployments, and assigns aggregate severity scores to such incidents, would better reflect the operational impact of distributed failures without retraining the underlying model.

Breadth-aware severity ranking: The current severity formula weights the ML anomaly score by event count and maximum duration, which biases the ranking toward persistent, high-count issues over brief but widespread ones. A natural extension is to add a breadth factor proportional to the number of pods of the same deployment that are simultaneously in a warning state. Combined with the incident correlation layer above, this would distinguish a single pod with a sustained problem from many pods failing briefly together. That distinction matters operationally and the current ranking blurs it.

Adaptive startup-phase detection: The rule layer currently classifies events in the first five minutes after the earliest event as startup-phase. This heuristic works well for the testbeds used in this study, but it may misclassify late startup events as operational issues on larger clusters or slower storage backends. Replacing the fixed five-minute window with an adaptive criterion, for example declaring the startup phase ended when the rate of `FailedMount` and `FailedScheduling` events drops below a threshold for a sustained period, would make the filter more robust across deployment scales.

Cross-test-run aggregation: The current prototype produces a report per test run. The industrial supervisor identified cross-run aggregation as a priority. Scanning a large number of test runs and aggregating component-level severity over time would surface systemically fragile components that do not cause test failures in any single run but degrade resilience across a test campaign. Because the per-run situation output is already structured (pod, deployment, situation, severity, warning reasons), this extension is straightforward and does not require changes to the model.

Periodic retraining and vocabulary maintenance: The model's event and pod-group vocabularies are fixed at training time. Software updates that introduce previously unseen reason types will be mapped to the unknown token and assigned high anomaly scores by default, which is a safe failure mode but not a scalable one. Periodic retraining on rolling windows of recent data, together with explicit tracking of previously unseen reasons and pod groups encountered at inference, would keep the model aligned with the evolving CNC software stack.

Time-aware evaluation metrics: This thesis evaluates the model using standard classification metrics applied to next-event prediction. Range-based precision and recall [40], which assess whether an anomaly is detected within a relevant time window rather than requiring event-exact agreement, would provide a more realistic assessment of detection capability. Applying such metrics requires ground-truth anomaly intervals with precise temporal boundaries, which were not available for the testbed data used here but could be produced through targeted fault-injection campaigns in future work.

Application-level log fusion: As stated in the delimitations (Section 1.5), this work is scoped to Kubernetes platform-level events. A complete observability solution would correlate platform events with application-level microservice logs (and possibly network traces), which would enable a form of root-cause analysis that goes beyond identifying *which* components are affected to explaining *why* at the application-logic level. Integrating an application-log channel with the current platform-event channel, while preserving the self-supervised formulation, is a promising direction for extending the system toward full RCA.

Taken together, these directions extend the prototype from a per-test-run triage tool toward an incident-aware, campaign-level observability system, while preserving the self-supervised, domain-aware foundation that this thesis has set out.

5.8 Methodological Reflection: Design Science Research

This thesis follows the Design Science Research (DSR) framework [44, 1], and the iterative development process from PoC v1 to PoC v2 is a concrete instance of the build-evaluate cycle that is central to DSR. The initial prototype (PoC v1) was a necessary step. It revealed the data-leakage problem that would not have been obvious from the literature alone, because the Kubernetes event domain has properties (a small, structured vocabulary in which the message text encodes the event type directly) that differ fundamentally from the public log datasets (HDFS, BGL) on which existing methods were originally evaluated.

The failure of PoC v1 informed every major design decision in PoC v2. It motivated the removal of BERT embeddings, the adoption of categorical reason IDs, the inclusion of Normal events, and the per-pod-group sequencing strategy. This iterative refinement is itself a contribution of the thesis, not just the final artifact. The thesis documents *why* standard approaches fail in this domain and *what adaptations* are necessary.

From a reproducibility perspective, the development process has one notable gap. The LSTM model's training script was not preserved, which required reconstructing the model class from the saved checkpoint weights. The reconstruction was checked by reproducing the reported accuracy, but this highlights the importance of version-controlling all training code alongside model artifacts, which is a useful lesson for any applied machine-learning research carried out in an industrial environment.

5.9 Ethical Considerations

The system analyses infrastructure-level Kubernetes events, which contain pod names, deployment identifiers and operational timestamps, but no user-plane data, personal information or subscriber identifiers. The data scope was deliberately restricted to platform-level events to comply with GDPR and with Ericsson's data handling policies. All data collection, storage and processing took place inside Ericsson's internal Engineering Environment (E2C), with no data transferred to external systems.

The anomaly detection system is designed as a decision-support tool for SREs, not as an autonomous remediation system. All identified issues require human review before action is taken. This design choice reduces the risk of automated false-positive responses that could themselves disrupt operational services. The severity ranking provides prioritisation guidance but does not make pass or fail determinations about test runs, which keeps human judgement in the loop.

A broader ethical consideration is that the tool could be misused for performance monitoring of individual engineers or teams based on how often infrastructure issues appear in their test runs. The tool is designed to identify infrastructure weaknesses, not to attribute blame, and its deployment should be accompanied by organisational guidelines that prevent it from being used as an individual performance metric.

5.10 Threats to Validity

5.10.1 Internal Validity

The primary threat to internal validity is potential data leakage between the training and evaluation splits. This is mitigated by splitting data at the test-run level rather than the event level, so that events from the same test run never appear in both training and evaluation sets. The 5 unseen test suites add an extra layer of confidence, because they come from entirely different test campaigns.

The reconstruction of the LSTM model class from the saved checkpoint (the original training script was not preserved) introduces a minor risk that the reconstructed architecture does not match the original exactly. The successful loading of all model weights and the reproduction of the checkpoint's reported test accuracy (86.1%) together suggest that the reconstruction is correct.

5.10.2 External Validity

The system is developed and evaluated exclusively on Ericsson's Packet Core Controller infrastructure. The underlying approach (next-event prediction on per-component sequences) is in principle applicable to other Kubernetes-based systems, but the specific vocabulary, pod group mappings and situation extraction rules are tailored to this environment. Applying the system to a different 5G Core product or to a non-telecommunications Kubernetes deployment would require retraining the model and adapting the domain-specific rules.

5.10.3 Construct Validity

The evaluation uses next-event prediction accuracy as a proxy for anomaly-detection capability. The end-to-end evaluation on 6 fault-injection cases gives evidence that prediction accuracy translates into practical detection ability, but the absence of a large-scale quantitative anomaly-detection evaluation (with precision, recall and F1 computed against ground-truth anomaly labels) limits the strength of that claim. The qualitative evaluation shows that the system works on the tested scenarios, but it does not establish detection rates or false-positive rates across a broad population of test runs.

6
Conclusion

This thesis explored the feasibility of an AI-driven anomaly detection system for identifying infrastructure issues in Ericsson's 5G Cloud Native Core. The work followed a Design Science Research process in which an initial prototype (PoC v1) exposed a data-leakage pitfall in applying text classifiers to structured Kubernetes events, and the resulting insights shaped the design of a substantially different artifact (PoC v2).

The final system formulates anomaly detection as self-supervised next-event prediction on per-pod-group event sequences, combined with a rule-based situation-extraction layer that encodes Kubernetes lifecycle knowledge. The main findings are:

RQ1. The LSTM and Transformer perform comparably (86.1% vs. 86.6% accuracy, near-identical top-3 at ∼98%). Neither architecture holds a decisive advantage on short sequences with a small event vocabulary. The choice of architecture matters less than the choice of representation and sequencing strategy.

RQ2. Per-pod-group sequencing (+19.6pp over global sequences) and categorical reason embeddings without message text (eliminating data leakage) are the two most impactful design decisions, together accounting for most of the system's effectiveness.

RQ3. The system generalises to unseen failure modes at 89.3% accuracy across 5 new test suites, and correctly identifies the primary affected components in all 6 end-to-end fault-injection evaluations, including 2 blind cases collected after the system was finalised.

The thesis further contributes a hybrid architecture that separates learned sequence scoring from deterministic domain rules, keeping the ML model focused on pattern learning while the rule layer provides interpretable, actionable reports. From an industrial perspective, the prototype reduces events requiring human attention by 60–76% and runs on CPU in seconds, demonstrating that automated post-test infrastructure triage is feasible with modest computational resources.

Bibliography

[1] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[2] K. Bogineni and A. Downing, "5g service-based architecture (sba): Implementation, challenges, and opportunities," *IEEE Communications Standards Magazine*, vol. 4, no. 4, pp. 12–19, 2020.

[3] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," *ACM Computing Surveys*, vol. 55, no. 3, pp. 1–39, 2022.

[4] N. Zhao, H. Wang, and Z. Li, "Explainable anomaly detection for microservices system with multimodal data," in *Proceedings of the 2021 IEEE International Conference on Web Services (ICWS)*, pp. 1–10, IEEE, 2021.

[5] S. Nedelkoski, J. Bogatinovski, A. Acker, and O. Kao, "Self-supervised log parsing," in *Proceedings of the 2020 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, pp. 122–138, Springer, 2020.

[6] 3GPP, "System architecture for the 5G System (5GS)," Technical Specification (TS) 23.501, 3rd Generation Partnership Project (3GPP), 2023. Version 18.2.0, Release 18.

[7] N. Kratzke and P. C. Quint, "Understanding cloud-native applications after 10 years of cloud computing: A systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[8] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2015.

[9] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[10] The Kubernetes Authors, "Kubernetes documentation: Events (`core/v1`)." `https://kubernetes.io/docs/reference/kubernetes-api/cluster-resources/event-v1/`, 2024. Accessed: 2026-04-28.

[11] C. Sridharan, *Distributed Systems Observability: A Guide to Building Robust Systems*. Sebastopol, CA, USA: O'Reilly Media, 2018.

[12] C. Majors, L. Fong-Jones, and G. Miranda, *Observability Engineering: Achieving Production Excellence*. Sebastopol, CA, USA: O'Reilly Media, 2022.

[13] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 121–130, IEEE Press, 2019.

[14] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services (ICWS)*, pp. 33–40, IEEE, 2017.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Workshop Proceedings of the International Conference on Learning Representations (ICLR)*, 2013.

[16] V.-H. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 492–504, IEEE, 2021.

[17] Y. Dang, Q. Lin, and P. Huang, "Aiops: Real-world challenges and research innovations," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 4–5, IEEE, 2019.

[18] J. Kawasaki *et al.*, "Failure prediction in cloud native 5g core with ebpf-based observability," in *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*, pp. 1–6, IEEE, 2023.

[19] L. Reiter, "Aiops: A systematic literature review," *FH Wedel University of Applied Sciences*, 2021.

[20] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 1–58, 2009.

[21] G. Pang, C. Shen, L. Cao, and A. van den Hengel, "Deep learning for anomaly detection: A review," *ACM Computing Surveys*, vol. 54, no. 2, pp. 1–38, 2021.

[22] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining (ICDM)*, pp. 413–422, IEEE, 2008.

[23] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Computation*, vol. 13, no. 7, pp. 1443–1471, 2001.

[24] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York, NY, USA: Springer, 2006.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[26] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[27] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1285–1298, ACM, 2017.

[28] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, pp. 5998–6008, Curran Associates, Inc., 2017.

[30] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the*

2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT), pp. 4171–4186, Association for Computational Linguistics, 2019.

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[33] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[34] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[35] H. Guo, S. Yuan, and X. Wu, "Logbert: Log anomaly detection via bert," in *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2021.

[36] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[37] C. Rosenthal and N. Jones, *Chaos Engineering: System Resiliency in Practice*. Sebastopol, CA, USA: O'Reilly Media, 2020.

[38] LitmusChaos Project, "LitmusChaos: Open source chaos engineering platform." `https://litmuschaos.io/`, 2024. Accessed: 2026-04-28.

[39] R. Wu and E. J. Keogh, "Current time series anomaly detection benchmarks are flawed and are creating the illusion of progress," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[40] N. Tatbul, T. J. Lee, S. Zdonik, M. Alam, and J. Gottschlich, "Precision and recall for time series," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, pp. 1924–1934, Curran Associates Inc., 2018.

[41] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pp. 4739–4745, 2019.

[42] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 807–817, 2019.

[43] S. Huang, Y. Liu, C. Fung, R. He, Y. Zhao, H. Yang, and Z. Luan, "HitAnomaly: Hierarchical transformers for anomaly detection in system log," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2064–2076, 2020.

[44] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

A

Declaration of AI-Assisted Tools

In accordance with Chalmers University of Technology guidelines, the following declaration describes the use of AI-assisted tools during this thesis work.

Tools Used

- **Amazon Q Developer (Kiro CLI):** An AI coding assistant available within the Ericsson development environment.
- **Google Gemini:** Used for generating the cover page illustration.

How AI Was Used

AI-assisted tools were used in two limited capacities:

1. **Proofreading and language refinement:** After drafting, AI tools were used to check grammar, improve sentence clarity, and ensure consistent terminology throughout the report. All suggestions were reviewed by the authors before acceptance, and the factual content, argumentation, and conclusions remain entirely our own.
2. **Code scaffolding:** AI tools assisted with generating boilerplate code for data collection scripts and parts of the training pipeline. All generated code was reviewed, tested, and adapted by the authors to fit the specific requirements of the project.

Scope and Oversight

The research design, methodology, experimental setup, model architecture decisions, evaluation protocol, analysis of results, and all conclusions presented in this thesis were performed by the authors without AI involvement. AI tools served strictly as productivity aids for routine writing and coding tasks. Every AI-generated output was manually verified for correctness before inclusion in the final work.

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY