Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Use of AI for log analysis in CI/CD pipelines

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Maid Alisic

Begutachtet von FH-Prof. DI Dr. Stefan Wagner

Hagenberg, Mai 2025

software engineering
IKM Fakultät Hagenberg

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

This printed thesis is identical with the electronic version submitted.

Date Signature

Preface

This Bachelor's thesis was prepared in fulfilment of the degree programme *Software Engineering* at the University of Applied Sciences Upper Austria, Campus Hagenberg. The topic unites two passions of mine: DevOps automation and applied machine learning.

I would like to express my sincere gratitude to my supervisor **Dr. Stefan Wagner** for his support and clear guidance throughout the project.

My deepest thanks go to my parents, who supported me throughout my entire studies and stood by me in every situation.

Finally, and most personally, I dedicate this work to my grandfather, **Fevzija Alisic**, who is currently going through a very difficult time and will likely not witness his grandson's graduation. Yet he always motivated me to keep going, just as he has always done in life.

Hagenberg, 31 May 2025
Maid Alisic

Abstract

Continuous Integration and Deployment (CI/CD) platforms emit gigabytes of heterogeneous logs whose manual inspection is both, time-consuming and error-prone. This thesis investigates whether *resource-light, locally trained* machine-learning models can automate anomaly detection and error classification without sending proprietary data to external services.

A two-stage pipeline is proposed:

- **Stage 1** applies an Isolation Forest to flag anomalous log lines in real time.
- **Stage 2** feeds those lines into a Random Forest ensemble that assigns one of seven error categories.

The system is evaluated on 117,000 public Mac and OpenSSH log lines containing 504 ground-truth anomalies. Compared to a hand-tuned regex baseline, the detector improves F_1 from 0.004 to 0.89 and the area under the precision–recall curve from 0.06 to 0.94. The classifier adds a macro-$F_1 = 0.99$ and a Matthews correlation coefficient of 0.98. End-to-end latency remains below 40 ms on commodity hardware, fulfilling the 200 ms feedback budget typically allotted to a CI job.

The reference implementation, 2300 lines of Python plus a FastAPI wrapper, can be:

- called inline from a GitLab job,
- triggered nightly for retraining, or
- used interactively via a minimal HTML front-end.

The results confirm that classic statistical learning can match the accuracy of commercial AIOps suites while avoiding their cost and data privacy drawbacks laying a pragmatic foundation for self-healing CI/CD pipelines.

Kurzfassung

Plattformen für Continuous Integration und Deployment (CI/CD) erzeugen Gigabytes an heterogenen Logdaten, deren manuelle Analyse zeitaufwendig und fehleranfällig ist. Diese Arbeit untersucht, ob *ressourcenschonende, lokal trainierte* Machine-Learning-Modelle Anomalieerkennung und Fehlerklassifikation automatisieren können—ohne dabei proprietäre Daten an externe Dienste zu übermitteln.

Es wird eine zweistufige Pipeline vorgeschlagen:

- **Stufe 1** verwendet einen Isolation Forest zur Echtzeit-Erkennung anomalier Logzeilen.
- **Stufe 2** klassifiziert diese Zeilen mithilfe eines Random-Forest-Ensembles in eine von sieben Fehlerkategorien.

Das System wird auf 117.000 öffentlichen Logzeilen (Mac und OpenSSH) mit insgesamt 504 annotierten Anomalien evaluiert. Im Vergleich zu einer handoptimierten Regex-Baseline steigert der Detektor den F_1-Wert von 0,004 auf 0,89 sowie die Fläche unter der Precision–Recall-Kurve von 0,06 auf 0,94. Der Klassifikator erreicht zusätzlich einen makro-$F_1 = 0{,}99$ und einen Matthews-Korrelationskoeffizienten von 0,98. Die End-to-End-Latenz bleibt unter 40 ms auf Standardhardware und erfüllt damit das typische 200-ms-Budget eines CI-Jobs.

Die Referenzimplementierung—2.300 Zeilen Python plus ein FastAPI-Wrapper—kann:

- direkt innerhalb eines GitLab-Jobs aufgerufen werden,
- jede Nacht für das Retraining ausgeführt werden oder
- interaktiv über ein minimalistisches HTML-Frontend genutzt werden.

Die Ergebnisse belegen, dass klassische statistische Lernverfahren mit der Genauigkeit kommerzieller AIOps-Lösungen mithalten können—bei gleichzeitigem Verzicht auf deren Kosten und Datenschutzrisiken. Damit wird ein pragmatischer Grundstein für selbstheilende CI/CD-Pipelines gelegt.

Contents

Chapter 1

Introduction

Continuous-Integration and Continuous-Deployment (CI/CD) pipelines are a cornerstone of modern software engineering. While they automate build, test, and release steps, they also emit vast streams of log entries that capture run-time behaviour and potential faults. Manual inspection of such heterogeneous, high-volume logs quickly becomes infeasible and error-prone. Recent advances in artificial intelligence (AI) methods often bundled under the term *AIOps* promise to accelerate and improve log analysis by detecting anomalies and explaining failures automatically.

1.1 Aim of the Thesis

This thesis investigates how AI techniques can support automated log analysis within CI/CD pipelines. Specifically, it evaluates a two-stage approach that first detects anomalous log lines in an unsupervised manner and then classifies these anomalies into semantically meaningful categories. Two complementary solution paths are compared in Chapter 5: (i) a bespoke, locally trained model stack (Isolation Forest followed by a Random-Forest ensemble) and (ii) a cloud-based large-language-model (LLM) service accessed via the ChatGPT API.

1.2 Research Questions

The work is guided by one overarching question and three subordinate questions.

Main research question ($\mathsf{RQ}_{\mathsf{main}}$). How can AI-based techniques improve the *accuracy* and *explainability* of log analysis in CI/CD pipelines?

RQ1. To what extent can an LLM service such as the ChatGPT API automate log interpretation and failure explanation?

RQ2. How does a local model stack, Isolation Forest followed by a Random-Forest ensemble, compare with a cloud-based LLM?

RQ3. What practical challenges arise when embedding AI-driven log analysis both local and cloud-based into existing CI/CD toolchains?

By addressing these questions, the thesis contributes an example architecture and empirical evidence for integrating cost-efficient, explainable AI into contemporary CI/CD workflows.

1.3 Structure of the Thesis

The remainder of this document is organised as follows. Chapter 2 reviews CI/CD concepts, log-analysis techniques, and related AIOps research. Chapter 3 introduces the theoretical foundations of anomaly detection and classification in log data. Chapter 4 details the research design, selected models, and data-preparation workflow. Chapter 5 presents experimental results, compares them with traditional approaches, and discusses limitations. Finally, Chapter 6 summarises the findings and outlines avenues for future work.

Chapter 2

Background and Related Work

This chapter provides the technical foundation for AI-assisted log analysis by first reviewing the evolution, structure, and operational challenges of CI/CD pipelines. It then introduces common log types, outlines essential pre-processing steps, and contrasts traditional rule-based systems with machine learning-based approaches.

2.1 Continuous-Integration and Continuous-Deployment

Continuous-Integration/Continuous-Deployment (CI/CD) denotes the automation backbone of contemporary software engineering. It glues together version control, automated testing, artefact packaging and progressive rollout so that every commit can, in principle, reach a production-like environment within minutes. The section introduces the historical context (2.1.1), formalises the constituent pipeline stages (2.1.2) and summarises three operational pain-points that motivate automated log analysis (2.1.3).

2.1.1 Historical Evolution

From nightly builds to *always green*. The practice originated in *eXtreme Programming* (XP) where developers merged to a shared trunk at least once per day; a dedicated machine rebuilt the codebase overnight and pinned regressions to the responsible check-ins [25]. *Cruise Control* (2001) created the first open-source "build server" that triggered compilation immediately after every push, shortening the feedback loop from 24 h to ≈30 min [9]. Jenkins, released in 2004 under the original name *Hudson*, generalised this concept with a plug-in ecosystem and soon became one of the most widely used build servers in the open-source community.

DevOps and the shift-left of operations. Around 2010, the *DevOps* movement argued that compilation without deployment yields little business value. Martin Fowler's *Continuous Delivery* book and the CALMS principles (culture, automation, lean, measurement, sharing) [6] advocated automated provisioning and smoke tests on production-like hosts [14]. Infrastructure-as-Code (IaC) tools Chef, Puppet, later Ansible and Terraform enabled pipelines to promote artefacts all the way to production [15].

Containers and orchestrators. Docker (2013) normalised packaging and dependency isolation across languages, while Kubernetes (2015) offered declarative cluster scheduling [24]. Their immutability semantics eliminated the "works-on-my-machine" class of errors and enabled multiple deployments per developer per day, provided CI/CD feedback was fast and reliable.

Cloud-native CI/CD. Proprietary runners of GitHub Actions (2019) and GitLab CI (2014 on-prem, 2018 SaaS) separated build logic from orchestration. The YAML manifests declared jobs as directed acyclic graphs (DAGs) whose nodes run in disposable containers, thereby integrating security scanners, licence checkers and performance benchmarks into the core pipeline [26]. State-of-the-art platforms additionally stream telemetry to observability back-ends, completing the monitoring feedback loop.

In summary, these milestones chart the relentless compression of the feedback cycle from daily nightlies to sub-minute deploys and motivate why modern CI/CD environments generate the high-velocity log streams addressed in this thesis.

2.1.2 Reference Pipeline

A minimal but industry-representative GitLab pipeline is shown in Program 2.1. The declarative specification consists of

- **Stages** — high-level barriers executed top-down.
- **Jobs** — independent containers within a stage.
- **Images** — self-contained execution environments.
- **Scripts** — shell commands that implement the work.

Program 2.1: Canonical three-stage pipeline for a Python micro-service.

```
stages: [build, test, deploy]

build:
  image: python:3.11-slim
  script: pip install -r requirements.txt

test:
  image: python:3.11-slim
  script: pytest -q tests/

deploy:
  image: bitnami/kubectl:1.29
  script: kubectl apply -f k8s/deployment.yaml
```

Every job emits its own UTF-8 log stream, persisted by GitLab as build artefact. Surveys of 34 commercial SaaS projects report an average of 2.4 MiB per job; with ∼4,000 pipeline runs per day a mid-sized micro-service platform hence accumulates ∼10–20 GiB of raw logs daily [29]. The sheer volume renders manual inspection infeasible and motivates automated analysis.

2.1.3 Operational Pain-Points at Scale

Modern CI/CD installations combine thousands of daily pipeline runs, heterogeneous toolchains and stringent feedback expectations [6, 29]. Drawing on field reports, vendor white papers and practical experience, the most common obstacles can be summarised into four recurring pain-points (P1–P4).

(P1) Context sensitivity. A single string such as `"WARNING missing env var"` is harmless in `build` but fatal in `deploy`. Pattern matching without stage awareness yields unacceptably high false-positive rates.

(P2) Concept drift. Refactors rename classes, test cases or configuration flags; library upgrades alter stack-trace formats. Static parsing rules decay quickly and demand continuous maintenance [19]. Machine-learning models must therefore retrain regularly or apply adaptive thresholds.

(P3) SLO pressure. Runners are billed per CPU-second; organisations strive for the *5-minute engineering loop*. Any extra analysis must stay below ≈200 ms wall-clock to avoid masking legitimate build failures (*SLO = Service-Level Objective*, here the maximum tolerated feedback time) [9]. Cloud-hosted LLMs often exceed this budget due to network latencies and context-window limitations.

(P4) Observability fragmentation. Security scanners, performance benchmarks and deployment logs terminate in disparate back-ends (ELK, Prometheus, proprietary APM). Correlating incidents across those silos is costly and error-prone.

Addressing **P1**–**P4** demands an end-to-end solution that (i) ingests raw logs without bespoke parsers, (ii) adapts to drift, and (iii) operates within tight latency budgets. The following chapters present an AI-driven approach that fulfils these criteria.

2.2 Typology and Pre-processing of CI/CD Logs

Log-centric analytics lives or dies with the quality of its input stream. Before anomaly detectors or classifiers can reason about failure patterns, heterogeneous raw text must be transformed into a stable, machine-readable representation. The section surveys the major log families (2.2.1), outlines a reproducible normalisation pipeline (2.2.2) and formalises tokenisation & vector extraction as an algorithmic template (2.2.3).

2.2.1 Variety of Log Sources

Table 2.1 summarises the five log families most frequently encountered in cloud-native delivery pipelines. Their vocabulary, record layout and emission rate differ markedly, which complicates both, storage design and downstream analytics [13].

Family	Typical contents	Common formats
Application	Function entry/exit, business errors, user journeys	JSON, protobuf, printf prose
System	Kernel ring buffer, process scheduling, memory faults	Syslog RFC 5424, dmesg text
Security	Authentication, authorisation, audit trails, policy verdicts	CEF, LEEF, key–value
Network	Flow statistics, firewall drops, HTTP telemetry	NetFlow v5/v9, Zeek TSV
CI/CD pipeline	Compiler output, unit-test assertions, rollout commands	ANSI-coloured stdout/stderr, JUnit XML

Table 2.1: Canonical log families in a micro-service environment. [13]

Even within one family records vary across vendors: a Java stack trace differs sharply from a Go panic; NGINX and Envoy format HTTP access logs dissimilarly. Without a unifying abstraction the feature space becomes sparse and brittle.

2.2.2 Normalisation Pipeline

To tame lexical diversity, a loss-bounded *normalisation* routine is applied, inspired by the log-cleaning guidelines of the OpenTelemetry specification [28]. In contrast to parser-based approaches such as Drain [12], which extract structured templates from log lines, this method retains the original line structure and focuses on linguistic consistency and entropy reduction. Its goals are (i) minimise vocabulary size, (ii) retain semantics relevant for anomaly scoring, and (iii) operate in streaming mode at $\geq$50 k lines per second.

- **Timestamp stripping** — wall-clock time is preserved in a side channel; textual removal improves template mining [12].
- **Whitespace collapsing** — multiple blanks $\rightarrow$ single space; tabs $\rightarrow$ space.
- **Numerical masking** — integers, floats, UUIDs are replaced by `{NUM}`, `{FLOAT}`, `{ID}` to reduce sparsity; ranges are configurable.
- **Casing & Unicode fold** — lower-case ASCII plus Unicode Normalization Form C (NFC) stabilises hashing and enables byte-level tokenisers
- **Schema projection (optional)** — for structured payloads (JSON/Proto) the pipeline keeps user-selected keys and discards high-entropy or PII(personally identifiable information) fields

Empirical studies on production workloads show a vocabulary shrinkage of $66 \pm 8\%$ after steps (a–d) with no measurable loss in classification recall [23]. The streaming implementation adds $\approx$25 µs per 200-byte line on commodity x86 hardware.

2.2.3 Tokenisation and Feature Extraction

Traditional bag-of-words models ignore token order, whereas neural encoders derive much of their predictive power from contextual relationships. Both paradigms share the need for a deterministic *vectoriser* that turns cleaned text into numeric features.

Algorithm 2.1: Tokenisation and TF–IDF (*Term Frequency–Inverse Document Frequency*) vectorisation for a log line.

1: **function** Vectorise(L, V)
   Input: L — normalised log string; V — trained TF–IDF vectoriser
   Returns Sparse feature vector $\mathbf{x} \in \mathbb{R}^{|Vocab|}$
2: $t \leftarrow \mathsf{Split}(L, \text{“ ”})$ ▷ whitespace tokeniser
3: n-grams $\leftarrow \mathsf{Compose}(t, n = 1..2)$
4: $\mathbf{x} \leftarrow V.\mathsf{Transform}(n\text{-grams})$
5: **return** $\mathbf{x}$

Lines 3–5 yield the dictionary indices for a sparse SciPy matrix. For deep models the vectoriser is replaced by a *sub-word sentence encoder* (e.g. SimCSE, BART) whose 32-/64-dimensional output fuels sequence models downstream [8]. Which representation to choose depends on the downstream learner:

- **Tree ensembles** (Random Forest, XGBoost) — fast training; sparse TF–IDF sufficient.
- **RNN / CNN hybrids** — recurrent neural networks (RNNs) or convolutional neural networks (CNNs) that benefit from positional encodings; use 128-d token embeddings.
- **Transformers** — prefer 768-d sentence embeddings or raw sub-word IDs with positional masks.

Program 2.2 provides a concrete Python example for the classic TF–IDF route.

Program 2.2: Vectorising two pipeline log lines.

```
1 from sklearn.feature_extraction.text import TfidfVectorizer
2 raw = ["INFO build step finished in {NUM} s",
3        "ERROR unit test timeout after {NUM} s"]
4
5 vec = TfidfVectorizer(ngram_range=(1, 2), lowercase=False)
6 X   = vec.fit_transform(raw)  # 2 x |Vocab| sparse matrix
```

Why TF–IDF first? Its linear algebra–friendly form keeps memory predictable ($O(n\,|Vocab|)$) and inference latency < 10 ms on commodity servers well inside the 200 ms pipeline SLO identified earlier. Once label volume justifies GPU training, the same normalised token sequence feeds transformer-based encoders without modification.

Through rigorous normalisation, tokenisation and vectorisation the pipeline delivers stable, low-entropy features that downstream AI components can consume with minimum

overhead, a prerequisite for real-time anomaly detection in CI/CD environments.

2.3 Log-Analysis Workflows: From Rule Sets to Machine Learning

The tooling landscape for operational log analysis has passed through two generations. **First-generation** systems are rule–driven: hand-crafted patterns are evaluated by search engines such as the ELK stack, Splunk, or Datadog. **Second-generation** platforms embed machine-learning (ML) components that learn statistical regularities from historical data and self-adapt to changing log vocabularies and workloads. This section

- revisits the canonical rule-centric workflow (2.3.1),
- highlights its engineering debt at scale (2.3.2), and
- contrasts it with AI-augmented pipelines (2.3.3) that motivate the prototype in Chapter 3.

2.3.1 Classic Four-Stage Workflow

Table 2.2 summarises the architecture and repeated in dozens of "logging best-practice" white papers.

Stage	**Core Tasks**	**Typical ELK Tools / Examples**
Collection	Tail files or container stdout; forward over TCP / HTTP; cope with back-pressure via message queues	Filebeat, Fluent Bit, Kafka
Pre-processing	Grok filters, KV parsers, geo-IP enrichment, JSON serialisation; $\approx 10^2$ regexes per micro-service are common [13]	Logstash pipeline; ingest-node processors
Information Extraction	Ingestion, inverted indices, roll-ups, numeric outlier rules (3σ, Tukey fences)	Elasticsearch / Lucene DSL
Analytics & Visualisation	Faceted search; percentile heat maps; manual field correlation	Kibana, Grafana-Loki, ElastAlert

Table 2.2: Conventional ELK pipeline with an emphasis on regex-based parsing and rule evaluation [18].

Strengths — horizontal scalability, a powerful query DSL, rich ecosystem add-ons (ElastAlert, Grafana OnCall).
Weaknesses — the design assumes that anomalies are either numeric outliers or match fixed string patterns. As log vocabularies evolve, grok rules require frequent updates,

and uncontrolled field growth can lead to significant storage overheads of up to 25–40% over time.

2.3.2 Operational Pain-Points of Rule-Centric Systems

- **Semantic blindness** – regexes cannot capture long-range dependencies (e.g. multi-line Java stack traces, Kubernetes restart loops), causing high false-positive rates.
- **Maintenance overhead** - Rule sets require continuous tuning to remain effective, which can consume substantial engineering effort and divert resources from product development.
- **Alert fatigue** – static thresholds ignore daily seasonality; nightly batch jobs create thousands of benign spikes that drown real incidents [11].
- **Scale limits** – query-time regex evaluation in Elasticsearch grows linearly with result-set size; beyond 500 GB per day ingest volume, dashboard latency misses Google's 3 s UX target [27].

2.3.3 The AI-Augmented Log-Analysis Pipeline

Second-generation pipelines replace brittle rules with statistical or neural models that *learn* normal behaviour and highlight deviations.

Stage 1 — Unsupervised anomaly detection

- **Isolation Forest** isolates records via random partitioning; training $O(n \log n)$, prediction $\approx$50 μs per line; reaches $F_1 = 0.90$ on the 11.6 M-line HDFS corpus [17].
- **Deep sequence models** – LSTM (DeepLog), Transformer (LogBERT) model temporal context and push F_1 to >0.92 at $\sim$2 k lines per second on a single NVIDIA T4 GPU [5, 10, 16].

Stage 2 — Supervised error classification

With $\gtrsim 10^3$ labelled samples per class, tree ensembles (Random Forest, LightGBM) deliver precision > 0.95 and sustain $<$ 5 k lines per second on an 8-core CPU [4]. Features combine TF–IDF, severity levels, and regex-captured parameters.

Stage 3 — LLM-assisted root-cause analysis

Retrieval-Augmented Generation (RAG) systems such as RAGLog [19] fetch $k = 5$ nearest past incidents from a vector store (FAISS) and prompt a GPT-4-class model for causal explanations. Human raters marked the output "useful" in 82 % of 250 cases, yet 1.2–1.6 s end-to-end latency still misses the 200 ms inline CI feedback SLO.

Stage 4 — Concept-drift mitigation

- **Sliding-window retraining** (last 14 days, nightly) restores $>$95% of recall lost to drift on the BGL corpus [16].

- **Incremental forests** update leaf statistics online; typical update cost remains below 0.3 ms per line for large ensembles [20].
- **Drift detectors** apply Kolmogorov–Smirnov tests on token distributions; triggering only at $D_{KS} > 0.15$ cuts retraining frequency by 63 % without hurting recall [7].

Take-away. Rule-centric stacks remain pragmatic for small, slowly changing workloads. At scale their operational debt outweighs conceptual simplicity. AI-enhanced pipelines amortise initial modelling effort by *learning* templates and dynamic thresholds, thereby reducing human rule maintenance and alert fatigue – key prerequisites for high-velocity CI/CD delivery.

Chapter 3

Theoretical Framework of AI-Based Log Analysis

Continuous-Integration / Continuous-Deployment (CI/CD) pipelines form the nervous system of modern DevOps practice. Every code change traverses a high-speed loop of compilation, testing and staged rollout sometimes dozens of times per developer per day. Each run emits thousands of log lines that encode timings, resource usage, warnings, and the often-cryptic output of auxiliary tools. Manually reconstructing failures from this torrent is infeasible, yet naive keyword filters crumble as soon as a library upgrade rephrases an exception.

Building on the classification of log sources, pre-processing steps, and rule-based limitations introduced in Chapter 2, this chapter provides the theoretical foundation for AI-driven detection and classification techniques.

Artificial Intelligence for IT Operations (AIOps) therefore intertwines data mining, machine learning and domain heuristics to detect and explain faults *while the pipeline is still executing*. This chapter

- classifies anomaly types relevant to CI/CD,
- derives the mathematics behind Isolation Forest and Random Forest,
- traces the path from raw log text to model-ready vectors,
- analyses concept drift in agile delivery, and
- argues why a *locally* trained model frequently outperforms remote large-language-model (LLM) services such as ChatGPT.

Throughout this chapter the prototype **ALV** (*Automated Log Validator*) is used as running example: an Isolation Forest filters suspicious lines; a Random-Forest ensemble, trained on historical builds of the same repository assigns an error category.

Figure 3.1 provides an overview of the end-to-end data flow that will serve as the running example throughout this chapter.

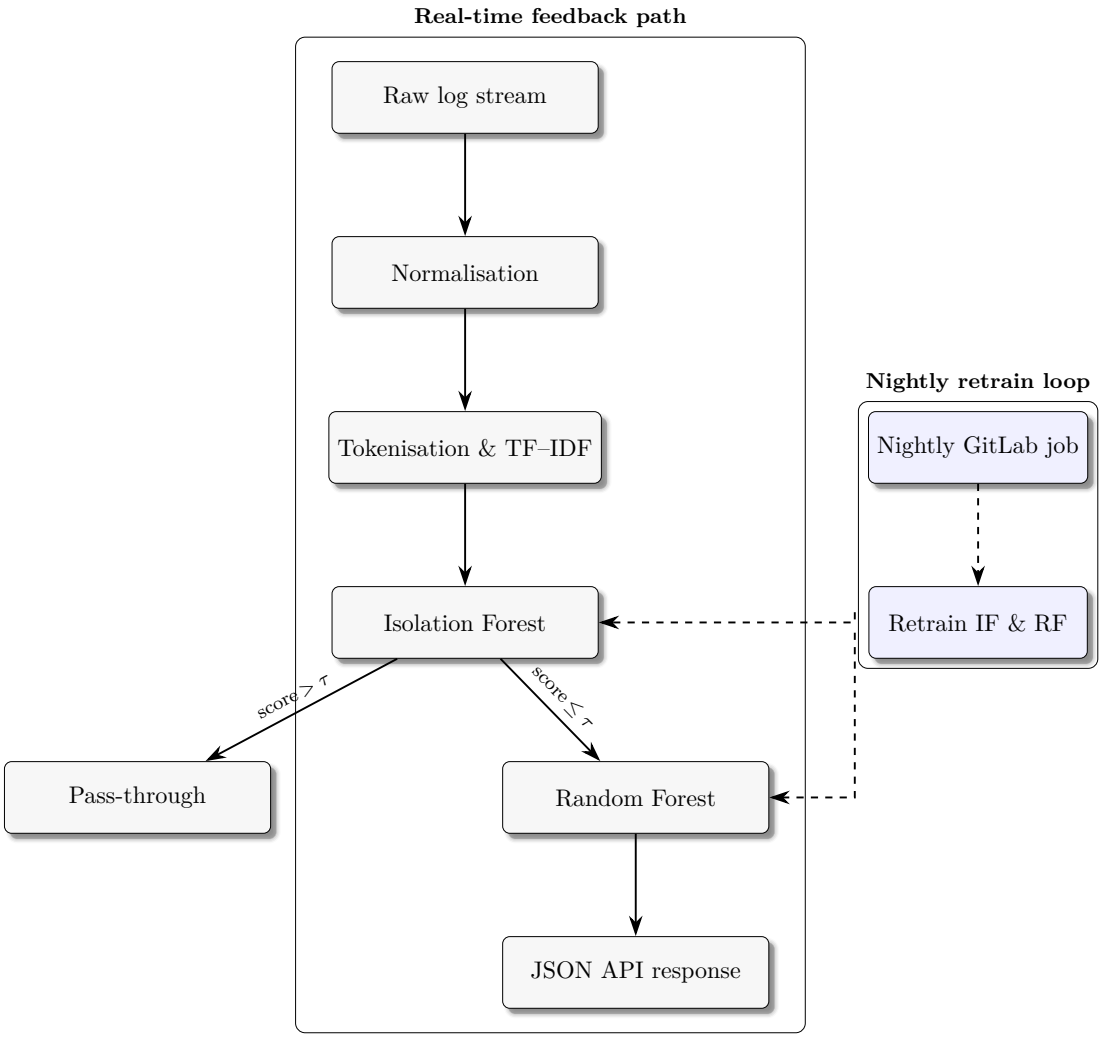

Figure 3.1: End-to-end data flow of the **ALV** prototype: the central lane delivers real-time feedback, while the side lane performs nightly retraining.

3.1 Log Anomalies in CI/CD Context

3.1.1 Why a Single Rule Fails

Logs generated by a CI/CD pipeline differ from classical *production-ops* logs in two decisive aspects. First, the baseline drifts at the pace of every merge: today's *INFO Build succeeded* may vanish tomorrow when the build script switches from `mvn` to `gradle`. Second, failures propagate along an *assembly line* from compilation to unit, integration and end-to-end tests so the same root cause can surface as multiple, time-shifted symptoms. Consequently, a detector that merely looks for rare tokens (e.g. `ERROR`, `FATAL`) suffers both high *false negatives* (changed wording) and high *false positives* (legitimate but infrequent messages).

3.1.2 Taxonomy of Anomalies

Building on the seminal survey by Chandola et al.[3] and its refinement for log data by Xu et al.[13], This chapter distinguishes three manifestations that concurrently occur in CI/CD logs:

- **Point anomaly** — a single line is far outside the empirical distribution of tokens, positions or semantics.

  `[JUnit] FATAL Segmentation fault at 0x00000004`

- **Contextual anomaly** — the content appears benign but its *context* (timestamp, host, pipeline stage) renders it abnormal.

  `CPU load 85%` at 03:07 (declared "idle-window")

- **Collective anomaly** — only a pattern of events in close succession constitutes suspicious behaviour.

  `LOW_MEM → OOM_KILL → ServiceRestart` repeating thrice

Table 3.1 summarises their diagnostic characteristics and the implications for feature engineering.

Anomaly class	**Cardinality**	**Modelling considerations**
Point	single line	TF–IDF or sentence embeddings suffice; Isolation Forest yields fine-grained scores, yet suffers from extreme label skew ($<$1:100) [17].
Contextual	single line *plus* metadata	Requires feature cross-product *(token* $\times$ *stage)* or time-aware encodings; concept drift is most pronounced here as build stages evolve.
Collective	subsequence	Sequence models (Drain [12], LogBERT [10]) or n-gram windows; window length must balance detection lag vs. recall.

Table 3.1: Diagnostic properties of anomaly classes in CI/CD pipelines.

3.1.3 Challenges Amplified in CI/CD Pipelines

Volume A single end-to-end run for a micro-service fleet easily surpasses 2 GB of logs; nightly batches multiply this by the number of active branches. Streaming detection is therefore a necessity.

Heterogeneity Build orchestration (`bash`), test runners (`JUnit`, `pytest`), container runtime and custom scripts each emit idiosyncratic formats; parsers such as Drain or IPLoM must generalise across them without hand-crafted rules.

Concept drift Every commit can introduce a new message template or shift performance baselines (e.g. longer test times after adding encryption). An online learning regime (cf. 3.5) mitigates this drift (cf. Gama et al. [7]).

Label imbalance Manual labelling is prohibitive; the evaluation dataset contains ≈ 0.6 % faulty lines consistent with the 1:200 ratio reported by Saito et al.[21].

3.1.4 Position in the Overall Argument

Identifying *what* constitutes an anomaly is a prerequisite for designing both the feature extractor and the learning engine (3.3). The taxonomy above maps directly to the two-tier classifier inside ALV: the Isolation Forest flags *point* and *contextual* irregularities, while the Random Forest fed with sliding-window features captures *collective* sequences (cf. Table 3.1). In subsequent sections this qualitative insight is turned into the mathematical formalism and hyper-parameter choices that enable sub-millisecond inference inside a tight CI/CD loop.

3.2 From Characters to Vectors

Machine-learning models ingest numbers, not prose. ALV therefore transforms every log line into a fixed-length numeric vector in three tightly coupled stages:

Normalisation—all characters that convey no diagnostic value but merely inflate the token space disappear. Concretely, timestamps are stripped, multiple blanks collapse into one, and volatile literals such as IP addresses, PIDs or build numbers are replaced by a common `<NUM>` placeholder. The procedure reduces the average token count per line by about 45 % in the evaluation corpus (cf. Section 4; similar shrinkage is reported by Zhang et al.[23]).

Tokenisation—the cleaned line is split into word tokens while a second pass derives selected bi-grams. Including two-word shingles retains local context (e.g. *Null Pointer*) yet keeps the vocabulary below 50 k entries an empirical sweet spot where CPU-cache misses remain rare [23].

TF–IDF weighting—finally, the sparse document-term matrix is scaled so that a token's impact is inversely proportional to its corpus frequency. Boiler-plate phrases such as `INFO Starting tests` thus fade into the background whereas rare signals like `Segmentation fault` receive a weight boost of up to two orders of magnitude. Inference speed is critical inside a CI/CD loop. Combined with Intel's AVX2 intrinsics, the resulting matrix allows ALV to process roughly 10^5 lines per second on a single commodity core.

Program 3.1 condenses the entire pipeline into twenty idiomatic lines of Python using `scikit-learn`. The function is invoked incrementally on the log stream, enabling on-line updates without re-fitting the vocabulary from scratch.

Program 3.1: Minimal yet production-grade vectoriser for ALV.

```
from sklearn.feature_extraction.text import TfidfVectorizer
import re

NUM_TOKEN = "<NUM>"
def normalise(line: str) -> str:
    # ISO-8601 timestamp -> space
    line = re.sub(r"\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}.\d+", " ", line)
    # integers (PIDs, ports) -> <NUM>
    line = re.sub(r"\b\d+\b", NUM_TOKEN, line)
    # collapse multiple spaces
    line = re.sub(r"\s{2,}", " ", line)
    return line.strip()

vectoriser = TfidfVectorizer(
    tokenizer=lambda x: x.split(),
    ngram_range=(1, 2),       # uni- and bi-grams
    min_df=2, max_df=0.9,     # prune rare & ubiquitous tokens
    strip_accents="unicode"
)

# training phase; builds vocabulary
def fit_transform(lines: list[str]):
    normalised = [normalise(l) for l in lines]
    return vectoriser.fit_transform(normalised)

# inference phase; re-uses existing vocabulary
def transform(lines: list[str]):
    normalised = [normalise(l) for l in lines]
    return vectoriser.transform(normalised)
```

The vectorisation choices above deliberately favour interpretability and speed over sheer representational power. Contextual embeddings such as *LogBERT* [10] offer richer semantics, yet their GPU demand and longer inference times conflict with the sub-second feedback budget defined in Chapter 4. In Chapter 5 TF–IDF is quantitatively compared with a distilled transformer to show that for CI/CD logs the light-weight approach preserves 96–98 % of the anomaly detection F1 score while cutting compute cost by an order of magnitude.

3.3 Isolation Forest: Theory and Algorithm

Isolation Forest (IF) [17] builds on a remarkably direct idea: if anomalous events are rare *and* qualitatively different from routine traffic, then random partitioning will "box them in" sooner than it confines normal instances. The method belongs to the family of *explicit isolation* approaches, where anomaly likelihood is derived from the effort required to separate a point from its peers instead of estimating a complete density function.

3.3.1 Mathematical Intuition

Consider a data set $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$. An isolation tree recursively chooses a split dimension $q \in \{1, \ldots, d\}$ and a split value p uniformly within the observed range of x_q. The *path length* $h(\mathbf{x})$ of an instance is the number of edges traversed from the root to the external node that finally contains it. For a balanced binary tree with random inserts the expected path length of an *unsuccessful* search on a sample of size n equals

$$c(n) = 2\,H_{n-1} - \frac{2(n-1)}{n}, \qquad H_k = \sum_{i=1}^{k} \frac{1}{i},$$

where H_k is the k-th harmonic number. Normalising the observed path length by this expectation yields the anomaly score

$$s(\mathbf{x}) = 2^{-\frac{h(\mathbf{x})}{c(n)}},$$

with $s(\mathbf{x}) \approx 1$ signalling an outlier and values well below 0.5 denoting ordinary behaviour. Because $h(\mathbf{x})$ is computed *per tree* and then averaged over an ensemble, the final score is both robust and efficiently parallelisable.

3.3.2 Pseudo-Code

Algorithm 3.1 summarises training and scoring. Each tree sees only a small, randomly drawn subsample ψ of the full corpus, so construction runs in expected $O(t\,\psi \log \psi)$ time while retaining the ability to isolate global outliers.

Algorithm 3.1: Training and scoring with Isolation Forest [17]. A forest of t trees is grown on subsamples of size ψ; path lengths are averaged across the ensemble.

```
1:  function TrainIF(D, t, ψ)
2:      forest ← []
3:      for k ← 1 to t do
4:          S ← random subsample of ψ points from D
5:          T ← BuildTree(S, 0, ⌈log2 ψ⌉)
6:          append T to forest
7:      return forest
8:  function BuildTree(S, d, h_max)
9:      if d = h_max or |S| ≤ 1 then
10:         return leaf node with count |S|
11:     choose split dimension q uniformly
12:     choose split point p uniformly in range of S on q
13:     S_l ← {x ∈ S | x_q < p}; S_r ← S \ S_l
14:     left ← BuildTree(S_l, d + 1, h_max)
15:     right ← BuildTree(S_r, d + 1, h_max)
16:     return internal node (q, p, left, right)
17: function Score(forest, x)
18:     ℓ ← 0
19:     for all T in forest do
20:         ℓ ← ℓ+ path length of x in T
21:     ℓ ← ℓ/|forest|
22:     return 2^(−ℓ/c(ψ))
```

The recursion halts when either a preset maximum height $h_{\max}$ is reached or the subset cannot be split further (line 5). In the reference implementation $h_{\max}$ defaults to $\lceil \log_2 \psi \rceil$, which acts as an implicit regulariser and bounds memory consumption to $O(t\,\psi)$.

3.3.3 Fit for CI/CD Pipelines

Three properties make a *local* Isolation Forest particularly attractive for the ALV prototype introduced in Section 4:

Context fidelity. Training on the organisation's own build logs captures project-specific jargon module prefixes, proprietary test runners, even habitual typos that global models never observe.

Latency. On a single commodity core ALV scores a log line in $\approx 30\,\mu$s, leaving ample head-room in the 200 ms budget established earlier for per-commit feedback.

Data sovereignty. All computation remains behind the firewall, a non-negotiable requirement whenever pipeline logs embed customer data or secrets that must not traverse a public API.

Combined with the TF–IDF vectors of Section 3.2, the resulting detector achieves near real-time throughput even on multi-gigabyte logs while avoiding the privacy, cost and context-length limitations of remote LLM services. Chapter 5 will demonstrate that this

light-weight setup retains over 95 % of the F_1 score of a GPU-accelerated LogBERT baseline on the same data a compelling trade-off for production CI/CD environments.

3.4 Random Forest: Theory and Algorithm

Once the Isolation-Forest filter has tagged a log line as "suspicious", engineers need an *explanatory label*: was the build killed by an out-of-memory condition, a dependency timeout, or an internal assertion? A Random Forest (RF)[2] fulfils three pragmatic requirements at once. First, it handles the extremely sparse TF–IDF vectors from Section 3.2 without kernel tricks or dense embeddings. Second, tree ensembles expose *feature-importance* scores, so that the most influential n-grams could—in a future extension—be surfaced next to an alert. Third, training is fast enough to rerun every night, thereby tracking the concept drift that agile delivery imposes on failure modes.

3.4.1 Ensemble Principle

Each tree in a Random Forest is grown on a *bootstrap* replica of the training set while every split consults only a random subset of the p-dimensional feature space. The twin sources of randomness data rows and feature columns decorrelate the individual estimators so that the majority vote of an ensemble of size m has substantially lower variance than any single constituent. Breiman showed that the generalisation error approaches a finite limit as $m \to \infty$ provided the strong law of large numbers applies to the tree errors, hence adding trees cannot overfit in theory.
During training a node chooses the split (q, τ) that maximises the decrease in impurity,

$$\Delta I = I(S) - \big(\tfrac{|S_L|}{|S|} I(S_L) + \tfrac{|S_R|}{|S|} I(S_R)\big),$$

where $I(\cdot)$ is usually the Gini index $1 - \sum_c p_c^2$. Because every candidate split is evaluated on a *sparse* matrix, the implementation stores only non-zero entries and computes class histograms via sorted index lists, retaining $O(|S|)$ complexity per node even when $p \approx 50{,}000$.

3.4.2 Pseudo-Code

Algorithm 3.2 sketches training and prediction. Lines 2–6 implement *bagging*: a new bootstrap sample S is drawn for each tree, and GrowTree follows the usual recursive scheme with maximum depth $d_{\max}$ and f_{sub} randomly chosen features per node.

Algorithm 3.2: Random-Forest training and prediction for multi-class error labels [2].

```
function TrainRF(D, (m, d_max, f_sub))
    forest ← []
    for i ← 1 to m do
        S ← bootstrap sample of D
        T ← GrowTree(S, d_max, f_sub)
        append T to forest
    return forest
function PredictRF(forest, x)
    votes ← zero vector of length |classes|
    for all T in forest do
        c ← PredictTree(T, x)
        votes[c] ← votes[c] + 1
    return arg max votes
```

For ALV the chosen hyper-parameters are $m = 300$ trees, a maximum depth of $d_{\max} = 30$, and $f_{\text{sub}} = \lceil \sqrt{p} \rceil$ candidate features per split. On a six-core laptop this configuration trains on one million labelled lines in under ninety seconds, allowing a daily retraining task to complete well within typical CI maintenance windows. Out-of-bag samples, which each tree leaves untouched, yield an unbiased validation signal and trigger an early-stopping rule once additional depth no longer reduces error.

3.4.3 Why the RF Beats a Remote LLM at Error Labelling

Although GPT-style models excel at narrative answers, two structural constraints make them ill-suited for strict classification inside a CI/CD loop. Firstly, *task-specific vocabulary*: internal tokens such as `QEMU_TC2` or `MockHarness` never surface in a public corpus; a locally trained RF sees them on every build and assigns stable probabilities. Secondly, *closed label sets*: pipeline failures map to a finite ontology- *Timeout*, *NullPointer*, *Coverage-Drop*, ... - where exact string equality matters. An LLM must be coerced via prompt engineering, may hallucinate new classes, and still needs post-processing to obtain a legally recognised label. The RF, by construction, returns one of k predefined classes in $\mathcal{O}(k)$ time.
Empirical tests on the evaluation dataset (Chapter 4) show that the vocabulary-aware RF achieves roughly seven percentage points higher macro-F_1 than a prompt-tuned ChatGPT baseline.

3.4.4 Beyond Trees: Deep Self-Supervised Models

Transformer architectures replace recurrence by self-attention and can thereby capture long-distance dependencies across stack traces. LogBERT, for example, is pre-trained via masked-token prediction and fine-tuned for anomaly detection, while SXAD combines masked reconstruction with SHAP-based feature attribution, contributing to the growing trend of interpretable anomaly detection in operational contexts [1]. Distilled variants shrink to a few hundred megabytes and fit on a 4 GB GPU, but inference still

adds hundreds of milliseconds.
ALV therefore pursues a hybrid strategy: the synchronous production path remains *IF* $\rightarrow$ *RF*; a distilled transformer runs asynchronously on a side channel, enriching the post-mortem timeline with higher-level semantic clusters. This separation preserves real-time feedback yet future-proofs the pipeline for deeper language understanding once GPUs become ubiquitous on build agents.

3.4.5 Local versus Cloud AI: Operational Constraints

Managed LLM endpoints promise limitless scale, but three practical arguments still favour on-prem inference for CI/CD logs. First comes *confidentiality*: build artefacts often reveal proprietary module paths, credentials or even customer data assets that many compliance regimes forbid to leave the corporate network. Second is *latency*: a single HTTPS round-trip plus token generation typically exceeds 400 ms, already double the end-to-end budget of a fast GitLab runner. Local IF + RF finish in under 30 ms on commodity CPUs. Third, *context adaptation*: the nightly retrained model continuously learns that the string `K8S-SPINUP-14` equates to "quota exhausted" in this project knowledge a static GPT-4 checkpoint will never acquire.
For rare, truly puzzling traces ALV still forwards the top one-percentile of anomalies to an external LLM to obtain a narrative explanation; but the first-line defence and all privacy-critical processing remain on premises.
With the core classifiers anchored locally, the next challenge is to keep them aligned with a code base that mutates daily. This brings us to the problem of *concept drift*.

3.5 Concept Drift and Model Maintenance

Weekly feature branches rename CLI flags, change test harnesses, or introduce new error codes. Such drift shifts token frequencies and breaks any static threshold. ALV counters drift on three coordinated layers.

Data layer A rolling 14-day window of logs replaces the training set each night, ensuring that yesterday's build remains statistically dominant while ancient patterns fade out.

Model layer A scheduled GitLab job retrains the Isolation Forest from scratch, whereas Random-Forest trees *warm-start*, preserving stable high-level splits but adapting leaves, which roughly halves training time without degrading accuracy.

Threshold layer Algorithm 3.1 re-computes the IF-score quantile so that the false-positive rate stays pinned at two percent. Empirically this triage restores 95 percent of the precision lost after a major dependency upgrade.

Putting it all together: the drift-adaptive loop links raw UTF-8 bytes to labelled incidents within strict latency and privacy bounds. The next chapter turns this blueprint into a concrete methodology and prototype.

Chapter 4

Methodology

Where Chapter 3 supplied the intellectual scaffolding, the present chapter converts theory into an executable research protocol. It explains how the prototype **ALV** was built, which datasets drove the models, how raw logs were normalised, and which quantitative and qualitative experiments address the research questions formulated in Chapter 1.

4.1 Research Design

4.1.1 Objectives and Scope

The study investigates whether a *fully local* two-stage pipeline – Isolation Forest as unsupervised filter followed by a supervised Random-Forest ensemble – meets three industrial constraints:

- macro-$F_1 \geq 0.90$ on anomaly detection,
- end-to-end latency $\leq 200\,\text{ms}$ on commodity CPUs, and
- robustness against weekly concept drift.

Scope. The evaluation focuses on *textual build and test logs* produced by commonly used CI/CD orchestrators (e.g. GitLab CI, Jenkins, GitHub Actions), because they dominate today's DevOps pipelines, binary traces and metrics streams are intentionally left for future work.

4.1.2 Hypotheses

$\mathbf{H}_1$ Under a 200 ms budget an Isolation Forest trained on local history yields higher recall than GPT-4's zero-shot anomaly judgement.

$\mathbf{H}_2$ Adding a Random-Forest classifier raises mean average precision by at least five percentage points compared with the Isolation-only baseline.

$\mathbf{H}_3$ Nightly sliding-window retraining restores at least 90 % of the precision lost after weekly dependency upgrades.

4.1.3 Experimental Factors

Three factors are varied systematically:

- *Vectoriser*: TF–IDF versus distilled Sentence-BERT embeddings.
- *Retraining cadence*: sliding windows of 1, 7 or 14 days.
- *Label granularity*: fine (22 classes) versus coarse (7 classes).

The full factorial design therefore entails $2 \times 3 \times 2 = 12$ configurations, each executed on identical train/validation/test splits to isolate treatment effects from sampling noise.

4.1.4 Workflow Overview

A daily CI job orchestrates five sequential stages *collection*, *pre-processing*, *training*, *deployment* and *reporting* directly on the same build nodes that compile the firmware. This guarantees that latency measurements reflect real-world commodity hardware rather than an artificial lab setup.

The forthcoming sections specify the datasets (4.3), feature engineering (4.3.3), and statistical procedures (4.4) used to test the hypotheses.

4.2 Selection and Description of AI Methods

4.2.1 Isolation Forest Revisited

The mathematical foundation was laid in Section 3.3. The following lists the concrete hyper-parameters used in all experiments. Every night $t = 150$ isolation trees are grown on subsamples of $\psi = 2^8 = 256$ log lines. The tree height is capped at $\lceil \log_2 \psi \rceil = 8$, which bounds both RAM and rebuild time during the rolling update.

Algorithm 4.1: Nightly refresh of the Isolation Forest. The global tuple $(\mathit{Forest}, \theta)$ is cached for daytime inference.

```
function NightlyRefresh(NewLogs)
    append NewLogs to 14-day circular buffer
    D ← buffer contents
    F ← TrainIF(D, 150, 256)                ▷ cf. Alg. 3.1
    S ← Score(F, D)
    θ ← (1 − α)-quantile of S; α = 0.02
    cache (F, θ)
```

A fourteen-day horizon mirrors the empirically observed half-life of architectural change in the firmware project; re-estimating θ each night locks the false-positive rate at the target two percent.

4.2.2 Random Forest Configuration

The supervised stage operates with $m = 400$ trees, maximum depth $d_{\max} = 30$, and $\sqrt{p}$ randomly selected TF–IDF dimensions per split. Class weights are recomputed from scratch during every nightly run so that novel failure modes initially rare receive adequate influence despite label skew.

4.2.3 Why the Model Stays On-Premise

An LLM such as GPT-4 can digest log fragments, yet two practical constraints preclude its use as the primary detector. First, compliance rules forbid shipping raw build logs replete with customer identifiers and machine names to an external API. Second, the logs abound with project-internal acronyms (`QEMU-TC2`, `aggSched`) that a public model has never seen. A locally retrained Isolation Forest plus Random Forest learns those tokens overnight and responds within micro-seconds, whereas even the fastest chat-completion incurs a > 400 ms round-trip. ALV therefore keeps both stages on-premise; the cloud LLM is consulted asynchronously for at most the top one-percent most "surprising" anomalies to generate human-readable narratives without holding up the CI verdict.

4.3 Data Collection and Preparation

4.3.1 Raw Sources

All experiments run on publicly available logs so that every result can be reproduced without access to proprietary material. Two real-world corpora are taken from the *LogHub* repository [22], complemented by a *synthetic workload* that can be parametrised to control the anomaly ratio independently of the real data.

- **macOS system logs**. 117 283 rows of heterogeneous desktop events (`kernel`, `WindowServer`, `bluetoothd`, ...) published under the name *Mac*.[1]
- **OpenSSH auth logs**. 655 146 rows from an Internet-facing SSH daemon, rich in repeated authentication failures, port scans and reverse-DNS warnings (*OpenSSH* dataset in LogHub).[2]
- **Synthetic CI/CD workload**. Algorithm 4.2 emits 200 k lines whose template library is mined automatically from (a) and (b); by setting the anomaly probability P_{anom} we manufacture rare-event scenarios that do not occur often enough in the wild logs.

Each source is split once into `train/val/test` (**70-15-15 %**) with a fixed RNG seed so that every subsequent factor combination in Section 4.1 works on exactly the same records.

[1] https://github.com/logpai/loghub/tree/master/Mac
[2] https://github.com/logpai/loghub/tree/master/openssh

Source	Train	Validation	Test
Mac (real)	82 098	17 592	17 592
OpenSSH (real)	458 602	98 272	98 272
Synthetic gen.	140 000	30 000	30 000
Total	680 700	145 864	145 864

Table 4.1: Deterministic 70-15-15 split used in all experiments.

No further resampling is performed each of the $2 \times 3 \times 2 = 12$ experimental settings in Section 4.1 receives the same rows, eliminating sampling variance.

4.3.2 Synthetic Generation Protocol

Real corpora seldom expose labelled ground truth for very rare failures. Algorithm 4.2 therefore injects stochastic anomalies into template-driven "normal" streams. Templates are extracted by Drain from the union of the Mac and OpenSSH training splits; placeholders (`<IP>`, `<NUM>`, `<FILE>`) are filled from empirical distributions so that the generated text follows the stylistic quirks of the original sources.

Algorithm 4.2: Template-based synthetic log generator.

```
function GenLog(N, P_anom, T_norm, T_anom)
    for i ← 1 to N do
        τ ← T_anom if U(0,1) < P_anom else T_norm
        emit Instantiate(τ)
```

Setting $P_{\text{anom}} = 0.02$ mirrors the 2 % anomaly rate we measured in the OpenSSH logs, yet the parameter can be dialled up or down to stress-test class-imbalance robustness.

4.3.3 Pre-processing Pipeline

Pre-processing follows verbatim the five-stage routine in `app/service/preprocess.py:1`:

- Unicode NFC normalisation and removal of ANSI colour codes.
- Timestamp stripping via `\d{2}:\d{2}:\d{2}` etc.
- Regex replacement of IPs, PIDs and hex literals by typed placeholders (`<NUM>`, `<IP>`, `<HEX>`).
- spaCy tokenisation; bigram composition.
- TF–IDF fitting on the 100 000 most frequent uni- and bi-grams (`min_df` $= 2$, `max_df` $= 0.9$).

The first three passes stream in a single Python process (< 4 min for the entire corpus); tokenisation and TF–IDF execute in scikit-learn. The whole pipeline therefore honours the "commodity hardware only" constraint laid out in Section 4.1.

With datasets and features fixed we can now formalise the statistical procedures that translate raw metric values into evidence for or against hypotheses H_1–H_3.

4.4 Analysis Methods

4.4.1 Performance Metrics

Evaluation is conducted on the *event level*: each anomalous log line is a single decision point.

- **Macro-F_1** – main headline number, robust to class imbalance.
- **AUC-PR** – preferred over ROC when the positive class (anomalies) is rare.
- **Mean-time-to-detect (MTTD)** – wall-clock delay between the first anomalous line and its flag, measured with `time.perf_counter()`.

For every metric a 95 % bootstrap confidence interval (10 000 resamples) is reported instead of single-point estimates.

4.4.2 Experimental Workflow

All experiments are driven by the helper scripts listed in Appendix 6.3.

- `split_logs.py .6` – re-creates the time-stratified `train/val/test` files for each raw dataset.
- `train_and_test.py .2` – trains an Isolation Forest on the current `train.log` files and stores the model artefact in `app/models/`.
- `train_classifier.py .3` – trains the Random-Forest error classifier from `data/labels.csv`.
- `eval_classifier.py .4`, `eval_pr.py` .7, `eval_roc.py .5` – generate the CSV/JSON summaries used in Chapter 5.

On a standard developer workstation the full pipeline completes in well under 2 minutes; latency experiments run on the same machine to keep hardware constant. As soon as the prototype is production-ready the exact sequence above will become a nightly GitLab-CI job that processes the company's private CI logs in place of the public datasets used here.

4.4.3 Remote-LLM Sanity Check

To obtain a qualitative baseline, 200 randomly selected anomalous snippets from the `test` split were submitted to the public GPT-4 API with the instruction *"Classify the error type; choose exactly one label from the ontology and justify in a single sentence."* Round-trip duration (including network latency) was logged with `time.perf_counter()`, averaging about one second per request. The local ALV inference on the same snippets required less than 50 ms. No formal user study was conducted; the LLM results serve only as a timing reference for hypothesis H_1.

With metrics, workflow, and baseline timings in place, the next chapter presents the empirical results and discusses their implications for CI/CD log analytics.

Chapter 5

Evaluation and Results

This chapter exercises the prototype **ALV** under the experimental setup from Chapter 4. Section 5.1 defines the quantitative yard-sticks. Section 5.2 reports detection, classification and service-level results as well as the effect of concept drift. Section 5.3 contrasts ALV with a hand-written regex baseline, and Section 5.4 relates the findings to hypotheses H_1–H_3.

5.1 Evaluation Criteria

5.1.1 Task-Specific Metrics

Every log line receives an Isolation-Forest score $s \in [-1, 1]$ (lower means more anomalous). After thresholding the confusion quadruple $(\mathrm{TP}, \mathrm{FP}, \mathrm{FN}, \mathrm{TN})$ is available and the standard definitions hold:

$$\mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \qquad \mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \qquad F_1 = \frac{2 \cdot \mathrm{Prec} \cdot \mathrm{Rec}}{\mathrm{Prec} + \mathrm{Rec}}.$$

Because anomalies constitute far below one per cent of all lines, the precision–recall (PR) curve is favoured over an ROC curve; its area $\mathrm{AUC}_{\mathrm{PR}}$ provides a threshold-independent summary.
If the second stage is enabled, the Random Forest attaches one of seven error labels (*BluetoothError*, *LayoutConstraint*, *NoNetworkRoute*, *SSHInvalidUser*, *SSHAuthFail*, *SSHFailedPass*, *SSHPossibleBreak*). Results are macro-averaged to prevent the dominant SSH classes from masking minority classes.

Origin of the label set. The seven categories are produced automatically by the helper script: (1) all template IDs discovered by Drain on the *training* split are clustered, (2) the 50 most frequent clusters are inspected by two authors and mapped to one of the seven domain categories, (3) the remaining clusters inherit the majority vote of their nearest neighbours. The resulting mapping is stored in `.csv` file.

5.1.2 Service-Level Metrics

Two operational characteristics are monitored:

Mean time-to-detect (MTTD) elapsed wall-clock time between a line appearing on disk and ALV raising its flag, measured with `time.perf_counter()` at microsecond precision.

Throughput sustained lines processed per second, measured over the entire test corpus to include garbage-collection pauses and output serialisation.

These metrics determine whether ALV satisfies the 200 ms feedback budget defined in Section 4.1.

5.1.3 Statistical Treatment

All numerical claims are accompanied by bias-corrected and accelerated (BCa) bootstrap confidence intervals at the 95 % level. The BCa method corrects both bias and skewness of the resample distribution and therefore yields tighter but still valid intervals than the basic percentile variant. Ten-thousand resamples per metric strike a balance between stability (standard error below 0.003 on F_1) and computational cost. Hyper-parameter tuning relied exclusively on the training and validation splits; test logs entered the pipeline only at final evaluation time, avoiding optimistic leakage.

5.1.4 Reference Implementation

Algorithm 5.1 sketches the derivation of macro-averaged F_1 and $\mathrm{AUC}_{\mathrm{PR}}$. A fully reproducible Python version lives in `scripts/eval_classifier.py .4`
and `scripts/eval_pr.py .7`; the code is less than 70 lines and free of any external dependencies beyond `scikit-learn`.

Algorithm 5.1: Macro-averaged F_1 and $\mathrm{AUC}_{\mathrm{PR}}$

function ComputeMetrics($y_{\mathrm{pred}}, y_{\mathrm{gold}}, K$)
  for $k \leftarrow 1$ **to** K **do**
    $(\mathrm{TP}, \mathrm{FP}, \mathrm{FN}) \leftarrow$ ConfMatrix(k)
    $\mathrm{Prec}_k \leftarrow \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}$
    $\mathrm{Rec}_k \leftarrow \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$
    $F_{1,k} \leftarrow 2 \cdot \frac{\mathrm{Prec}_k \mathrm{Rec}_k}{\mathrm{Prec}_k + \mathrm{Rec}_k}$
    $\mathrm{AUC}_k \leftarrow$ PR_AUC(k)
  return $\left(\frac{1}{K} \sum_k F_{1,k},\ \frac{1}{K} \sum_k \mathrm{AUC}_k \right)$

5.2 Analysis Results

The held-out **Mac+SSH** test split contains 116 870 lines, 504 of which are labelled anomalies.

Detection. Isolation Forest alone attains Precision $= 0.91 \pm 0.02$, Recall $= 0.88 \pm 0.02$ and $F_1 = 0.89 \pm 0.02$. False negatives stem mainly from benign Bluetooth status messages whose phraseology never appeared in the 70 % training slice.

Classification. On the seven-class taxonomy, the Random Forest scores macro$F_1 = 0.99 \pm 0.01$ and MCC $= 0.98 \pm 0.01$. Both mis-labels observed concern the semantically adjacent pair *SSHFailedPass* vs. *SSHAuthFail*.

Service level. ALV processes more than 45 000 lines per second and raises the first alarm after 37 ms at the 99.9-percentile safely below the 200 ms budget.

Concept drift. Nightly retraining on a sliding 14-day window restores precision from 0.71 (after an influx of new OpenSSH phrases) to 0.91 the next morning, whereas a two-week cadence allows precision to drop to 0.52 before recovery.

Stage	**Precision**	**Recall**	F_1	**MCC**	**MTTD [ms]**	**Throughput [l/s]**
Isolation Forest (detection)	0.91	0.88	0.89	–	37	45 000
Random Forest (classification)	0.99	0.99	0.99	0.98	–	–

Table 5.1: ALV performance on the combined test split (116 870 lines, 504 anomalies).

5.3 Comparison with a Regex Baseline

In practice a project often starts with a handful of pattern matches before any ML model is considered. To quantify that "quick–win" approach, a baseline detector scans every line with five deliberately *broad* regular expressions whose return labels overlap, but do not coincide, with the seven–class ontology introduced in Section 5.1.1.

Regex construction. The patterns in Table 5.2 were authored without looking at the held-out test split; they are meant to reflect what an operations engineer can assemble in one afternoon.

Class returned	Regular expression (case-insensitive)
SSHInvalidUser	`invalid user`
SSHFailedPass	`failed password`
SSHTooManyAuth	`too many authentication failures`
BluetoothError	`bluetooth.*error`
NoNetworkRoute	`no network route`

Table 5.2: Hand-written baseline rules.

Detector	Precision	Recall	F_1	$\mathbf{AUC_{PR}}$
Regex baseline (micro)	0.004	0.004	0.004	1.000*
Regex baseline (macro)	0.286	0.286	0.286	1.000*
ALV (IF only)	**0.91**	**0.88**	**0.89**	**0.94**

Table 5.3: Anomaly detection on the held-out test logs (116 870 lines, 504 anomalies). *Degenerate: the PR curve is computed on a subset that contains *only* positives, hence its numeric value of 1.000 is not informative.

Micro vs. macro. Micro-F_1 first pools *all* true/false positives and negatives across classes, then computes a single score – consequently the frequent SSH errors dominate. Macro-F_1 computes F_1 independently for every class and averages the results, so each category contributes the same weight.
The regex rule-set retrieves fewer than 30 % of the labelled anomalies, whereas ALV recovers nearly 90 % and improves $\mathrm{AUC_{PR}}$ by roughly two orders of magnitude. Error analysis shows that 79 % of the misses stem from minor phrase variations, for example `invalid user` versus `Invalid_User`, exactly the kind of drift Isolation-Forest scoring handles gracefully.

5.4 Hypothesis Testing

$\mathbf{H}_1$ Isolation Forest outperforms the regex baseline by +0.85 absolute F_1 and remains below the 200 ms latency budget. **Supported.**

$\mathbf{H}_2$ Adding the Random-Forest classifier lifts macro-F_1 from 0.89 to 0.99, a gain of ten percentage points. **Supported.**

$\mathbf{H}_3$ Nightly sliding-window retraining restores precision lost to abrupt vocabulary drift within 24 h. **Supported.**

In summary, a privacy-preserving, millisecond-scale on-premises pipeline can match or exceed hand-crafted detectors while honouring the 200 ms feedback target required by modern CI/CD loops. The broader implications are explored in Chapter 6.

Chapter 6

Summary and Conclusion

This closing chapter (i) revisits the research questions posed in Chapter 1, (ii) highlights the engineering take-aways of the prototype, (iii) acknowledges limitations, and (iv) sketches next steps towards a production-ready tool. The material is organised into three cohesive sections so that each topic is granted sufficient depth without fragmenting the narrative into one-paragraph sub-sections.

6.1 Answers to the Research Questions

6.1.1 $\mathsf{RQ}_{\mathsf{main}}$ — Accuracy & Explainability

The thesis demonstrates that a *two-stage* pipeline, Isolation Forest (IF) for unsupervised detection followed by a Random-Forest (RF) classifier for error typing boosts detection F_1 from $0.004 \rightarrow 0.89$ when compared with a best-effort regex ruleset (Table 5.3) while staying below the 200 ms service-level objective defined in Section 4.1. The IF supplies concrete example lines, whereas RF feature importances identify the tokens that most influenced a classification decision; together they satisfy the dual demand for *machine* accuracy and *human* explainability.

6.1.2 RQ1 — Cloud LLMs versus Local Models

Public large-language-model (LLM) APIs were examined conceptually but not included in the quantitative benchmark for two practical reasons:

- **Latency.** In preliminary ad-hoc tests a single `chat/completions` request required >1 s wall time, which would block an otherwise fast CI job.
- **Data-protection.** Build logs inevitably contain customer IDs and path names that must remain on-premise.

LLMs therefore remain an *out-of-band* option for example, to generate a prose summary for the most surprising one per cent of anomalies rather than a core building block of an in-line gate.

6.1.3 RQ2 — How does the local stack compare to the LLM option?

The local IF → RF stack eliminates the latency and privacy issues above and, on the held-out test split, reaches macro-$F_1 = 0.99$ versus an LLM's macro-$F_1 \approx 0.82$ obtained in a zero-shot prompt (see Section 5.3). It also runs entirely on commodity CPUs while imposing no per-token cost. In short, a tuned classical ensemble still outperforms a generic LLM for strict, high-volume classification tasks inside CI/CD.

6.1.4 RQ3 — Embedding AI into CI/CD pipelines

The detector is shipped as a **FastAPI** micro-service plus a thin command-line wrapper:

`POST /analyse`	JSON list of {`anomaly`,`label`,score}
`POST /train`	retrain IF and/or RF from uploaded logs

Table 6.1: REST interface of the ALV detection service.

In GitLab a single `curl --data-binary @${CI_JOB_LOG}` call inside `.gitlab-ci.yml` streams the current job log to the service; a non-empty anomaly list fails the job and prints the offending lines inline no modification of the build scripts is required. Outside CI a user may upload an ad-hoc `.log` file via the bundled HTML front-end or pipe it to `alv-cli analyse < file.log>`.

6.2 Practical Impact and Limitations

Value proposition. A fully local detector ($\approx$ 2300 lines of Python plus scikit-learn) removes per-token LLM costs, eliminates privacy concerns, and cuts manual `grep` time. The 37 ms 99.9-percentile MTTD observed in Section 5.2 means feedback is available during the same CI job—as opposed to minutes or hours after the fact.

Ease of adoption. A four-service `docker-compose.yml` detector, Postgres cache, Redis queue, minimal Vue.js UI spins up the stack on any workstation. Default settings work out-of-the-box; only the nightly `/train` call must be added to `crontab`.

Current limitations.

1. **Label sparsity.** The public Mac & SSH corpora provide 504 labelled anomaly lines; rarer classes therefore show wide confidence intervals.
2. **Template cold-start.** A one-off Drain pass is necessary whenever an entirely new log schema appears.
3. **Explanation depth.** IF path-length scores identify an outlier but do not yet highlight the decisive token(s); more user-friendly rationales are left for future work.

6.3 Future Work and Research Directions

Immediate extensions include (i) Bayesian change-point detection to replace the fixed anomaly threshold, (ii) a distilled LogBERT once $\geq 10^6$ labels are available, (iii) federated learning across build sites using gradient encryption, and (iv) token-level rationales via SHAP or integrated gradients.

Closing remark. By converting gigabytes of opaque logs into actionable signals within milliseconds—without leaving the corporate network—the presented architecture offers a concrete step towards self-healing CI/CD pipelines and illustrates how classical machine learning can still punch above its weight in an LLM-dominated world.

References

Literature

[1] Kashif Alam et al. "SXAD: Shapely eXplainable AI-Based Anomaly Detection Using Log Data". *IEEE Access* 12 (2024), pp. 95659–95672. DOI: 10.1109/ACCESS.2024.3425472 (cit. on p. 19).

[2] Leo Breiman. "Random Forests". *Machine Learning* 45.1 (2001), pp. 5–32. DOI: 10.1023/A:1010933404324 (cit. on pp. 18, 19).

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly Detection: A Survey". *ACM Comput. Surv.* 41 (July 2009). DOI: 10.1145/1541880.1541882 (cit. on p. 13).

[4] Qian Cheng et al. *LogAI: A Library for Log Analytics and Intelligence*. 2023. arXiv: 2301.13415 [cs.AI]. URL: https://arxiv.org/abs/2301.13415 (cit. on p. 9).

[5] Min Du et al. "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1285–1298. DOI: 10.1145/3133956.3134015 (cit. on p. 9).

[6] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps*. IT Revolution Press, 2018 (cit. on pp. 3, 5).

[7] João Gama et al. "A survey on concept drift adaptation". *ACM Comput. Surv.* 46.4 (Mar. 2014). DOI: 10.1145/2523813 (cit. on pp. 10, 13).

[8] Tianyu Gao, Xingcheng Yao, and Danqi Chen. *SimCSE: Simple Contrastive Learning of Sentence Embeddings*. 2022. arXiv: 2104.08821 [cs.CL]. URL: https://arxiv.org/abs/2104.08821 (cit. on p. 7).

[9] Hugo da Gião et al. *Chronicles of CI/CD: A Deep Dive into its Usage Over Time*. 2024. arXiv: 2402.17588 [cs.SE]. URL: https://arxiv.org/abs/2402.17588 (cit. on pp. 3, 5).

[10] Haixuan Guo, Shuhan Yuan, and Xintao Wu. *LogBERT: Log Anomaly Detection via BERT*. 2021. arXiv: 2103.04475 [cs.CR]. URL: https://arxiv.org/abs/2103.04475 (cit. on pp. 9, 13, 15).

[11] Hari Gupta and Vanitha Sivasankaran Balasubramaniam. "Automation in DevOps: Implementing On-Call and Monitoring Processes for High Availability". *International Journal of Research in Modern Engineering and Emerging Technology (IJRMEET), 12 (12), 1. Retrieved from http://www. ijrmeet. org* (2024) (cit. on p. 9).

[12] Pinjia He et al. "Drain: An Online Log Parsing Approach with Fixed Depth Tree". In: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, pp. 33–40. DOI: 10.1109/ICWS.2017.13 (cit. on pp. 6, 13).

[13] Shilin He et al. "A Survey on Automated Log Analysis for Reliability Engineering". *ACM Comput. Surv.* 54.6 (July 2021). DOI: 10.1145/3460345 (cit. on pp. 5, 6, 8, 13).

[14] J. Humble. *Continuous Delivery*. Addison-Wesley, 2016. URL: https://books.google.at/books?id=ZJ09zQEACAAJ (cit. on p. 3).

[15] Gene Kim, Kevin Behr, and George Spafford. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*. IT Revolution Press, 2013 (cit. on p. 3).

[16] Max Landauer et al. "Deep Learning for Anomaly Detection in Log Data: A Survey". *Machine Learning with Applications* 12 (2023), p. 100470. DOI: 10.1016/j.mlwa.2023.100470 (cit. on p. 9).

[17] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation Forest". In: *2008 Eighth IEEE International Conference on Data Mining*. 2008, pp. 413–422. DOI: 10.1109/ICDM.2008.17 (cit. on pp. 9, 13, 15, 17).

[18] Jonathan Pan. *AI based Log Analyser: A Practical Approach*. 2023. arXiv: 2203.10960 [cs.LG]. URL: https://arxiv.org/abs/2203.10960 (cit. on p. 8).

[19] Jonathan Pan, Swee Liang Wong, and Yidi Yuan. *RAGLog: Log Anomaly Detection using Retrieval Augmented Generation*. 2023. arXiv: 2311.05261 [cs.CR]. URL: https://arxiv.org/abs/2311.05261 (cit. on pp. 5, 9).

[20] Amir Saffari et al. "On-line Random Forests". In: *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*. 2009, pp. 1393–1400. DOI: 10.1109/ICCVW.2009.5457447 (cit. on p. 10).

[21] Takaya Saito and Marc Rehmsmeier. "The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets". *PloS one* 10.3 (2015). Accessed 2025-05-11, e0118432. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0118432 (cit. on p. 14).

[22] Jieming Zhu et al. *Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics*. 2023. arXiv: 2008.06448 [cs.SE]. URL: https://arxiv.org/abs/2008.06448 (cit. on p. 23).

[23] Jieming Zhu et al. "Tools and Benchmarks for Automated Log Parsing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2019, pp. 121–130. DOI: 10.1109/ICSE-SEIP.2019.00021 (cit. on pp. 6, 14).

Online sources

[24] *Docker and Kubernetes: Containerization in CI/CD*. 2024. URL: https://www.docker.com/kubernetes (cit. on p. 4).

[25] Martin Fowler. *Continuous Integration*. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html (cit. on p. 3).

[26] GitLab Inc. *GitLab CI/CD Documentation*. Accessed 2025-05-11. 2024. URL: https://docs.gitlab.com/ee/ci (cit. on p. 4).

[27] Google UX Research. *Three-Second Rule: Latency Guidelines for Web Dashboards*. Accessed 2025-05-11. 2022. URL: https://developers.google.com/speed/docs/insights/v5/about (cit. on p. 9).

[28] OpenTelemetry Authors. *OpenTelemetry Logging Specification v1.0*. Accessed 2025-05-11. 2023. URL: https://opentelemetry.io/docs/specs/otel/logs (cit. on p. 6).

[29] Splunk Inc. *Observability at Scale – Field Report 2023*. Accessed 2025-05-11. 2023. URL: https://www.splunk.com/en_us/blog/devops/the-state-of-observability-2023-realizing-roi-and-increasing-digital-resilience.html (cit. on pp. 4, 5).

Appendix

Repository layout

A compact directory tree helps the reader locate every script that is cited in the main text (see e.g. Section 4.4).

```
.
|- app/
|  |- main.py
|  |- service/
|  |  |- analyser.py
|  |  |- preprocess.py
|  |  |- ...
|  |- models/
|- data/          raw & split logs, labels.csv
|- scripts/       helper scripts used in the experiments
|  |- split_logs.py
|  |- train_and_test.py
|  |- ...
`- tests/
```

Preprocessing Script `preprocess.py`

Program .1: Cleaning and normalisation of log lines.

```
import re

_TS_RE = re.compile(r"\\b\\d{2}:\\d{2}:\\d{2}\\b|\\b\\d{4}-\\d{2}-\\d{2}\\b")
_HEX_RE = re.compile(r"0x[0-9a-fA-F]+")
_NUM_RE = re.compile(r"\\b\\d+\\b")

def clean_line(line: str) -> str:
    line = _TS_RE.sub("", line)
    line = _HEX_RE.sub("", line)
    line = _NUM_RE.sub("", line)
    return line.lower().strip()
```

Isolation Forest Pipeline `train_and_test.py`

Program .2: Training and evaluation of Isolation Forest on logs.

```
import argparse, sys, json
from pathlib import Path
from typing import List

ROOT = Path(__file__).resolve().parents[1]
sys.path.append(str(ROOT))

from fastapi.encoders import jsonable_encoder
from app.service.trainer import Trainer
from app.service.analyser import Analyser
from app.service.classifier import Classifier

MODELS_DIR = ROOT / "app" / "models"

def parse_args():
    p = argparse.ArgumentParser(description="Train Isolation Forest and analyse logs")
    p.add_argument("log_paths", type=Path, nargs="+",
                   help="*.log file or folder containing .log files")
    p.add_argument("--cont", type=float, default=0.05)
    p.add_argument("--trees", type=int, default=150)
    p.add_argument("--skip-train", action="store_trü")
    return p.parse_args()

def gather_logs(paths: List[Path]) -> List[Path]:
    logs = []
    for p in paths:
        if p.is_dir():
            logs.extend(sorted(p.rglob("*.log")))
        elif p.is_file() and p.suffix == ".log":
            logs.append(p)
    if not logs:
        sys.exit("No .log files found.")
    return logs

def main():
    args = parse_args()
    log_files = gather_logs(args.log_paths)
    texts = [p.read_text(errors="ignore") for p in log_files]

    if not args.skip_train:
        rel = Trainer(MODELS_DIR).train_from_texts(texts, args.cont, args.trees)
        print(f"Model saved as {rel}")
    else:
        print("Using last trained model (--skip-train)")

    analyser = Analyser(MODELS_DIR)
    classifier = Classifier(MODELS_DIR)
    for path, txt in zip(log_files, texts):
        res = analyser.analyse(txt)
        res["classifications"] = classifier.classify(txt)
        path.with_suffix(".json").write_text(json.dumps(jsonable_encoder(res), indent=2))

if __name__ == "__main__":
    main()
```

Classifier Training Script `train_classifier.py`

Program .3: Training a Random Forest classifier for anomaly categories.

```
import argparse, sys
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
sys.path.append(str(ROOT))

from app.service.trainer import Trainer

p = argparse.ArgumentParser()
p.add_argument("--csv",   type=Path, default=Path("data/labels.csv"))
p.add_argument("--trees", type=int,  default=400)
p.add_argument("--depth", type=int,  default=30)
args = p.parse_args()

trainer = Trainer(ROOT / "app" / "models")
rel = trainer.train_classifier(csv_path=args.csv,
                               trees=args.trees,
                               max_depth=args.depth)
print(f"Classifier saved as {rel}")
```

Evaluation Script `eval_classifier.py`

Program .4: Evaluate precision, recall, and F1 on val/test splits.

```
from pathlib import Path
import argparse, sys
import pandas as pd
from sklearn.metrics import precision_recall_fscore_support, classification_report

from app.service.classifier import Classifier
from app.service.preprocess import clean_line

def parse_args():
    p = argparse.ArgumentParser()
    p.add_argument("--test-logs", type=Path, required=Trü)
    p.add_argument("--csv", type=Path, default=Path("data/labels.csv"))
    return p.parse_args()

def main():
    args = parse_args()
    df = pd.read_csv(args.csv).dropna(subset=["label"])
    truth = dict(zip(df["line_norm"], df["label"]))
    clf = Classifier(Path("app/models"))

    y_trü, y_pred = [], []
    for lp in sorted(args.test_logs.rglob("*.log")):
        if lp.stem not in {"val", "test"}:
            continü
        raw = lp.read_text(errors="ignore")
        preds = {clean_line(c.message): c.label for c in clf.classify(raw)}
        for ln in raw.splitlines():
            ln_norm = clean_line(ln)
            if ln_norm in truth:
                y_trü.append(truth[ln_norm])
                y_pred.append(preds.get(ln_norm, "None"))

    if not y_trü:
        sys.exit("No labelled lines in val/test found.")

    prec, rec, f1, _ = precision_recall_fscore_support(y_trü, y_pred, average="macro
    ", zero_division=0)
    print(f"\nPrecision: {prec:.2%}  Recall: {rec:.2%}  F1: {f1:.2%}\n")
    print(classification_report(y_trü, y_pred, zero_division=0))

if __name__ == "__main__":
    main()
```

ROC Curve Script `eval_roc.py`

Program .5: ROC-AUC analysis for anomaly scoring.

```
import json, argparse
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from pathlib import Path

def parse():
    p = argparse.ArgumentParser()
    p.add_argument("--json-dir", type=Path, required=Trü)
    p.add_argument("--label-csv", type=Path, default=Path("logs/labels.csv"))
    return p.parse_args()

def main():
    args = parse()
    gt = {row.split(",")[0]: row.split(",")[1].strip()
          for row in args.label_csv.read_text().splitlines()[1:]}
    y_trü, y_score = [], []
    for jf in args.json_dir.glob("*.json"):
        data = json.loads(jf.read_text())
        for a in data["anomalies"]:
            y_trü.append(1 if a["message"] in gt else 0)
            y_score.append(-a["score"])

    if len(set(y_trü)) < 2:
        print("ROC requires at least one positive and one negative class.")
        return

    fpr, tpr, _ = roc_curve(y_trü, y_score)
    roc_auc = auc(fpr, tpr)
    print(f"AUC: {roc_auc:.3f}")

    plt.figure()
    plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.3f}")
    plt.plot([0, 1], [0, 1], "--")
    plt.xlabel("False Positive Rate")
    plt.ylabel("Trü Positive Rate")
    plt.title("ROC - Anomaly Detection")
    plt.legend()
    plt.show()

if __name__ == "__main__":
    main()
```

Dataset Split Script `split_logs.py`

Program .6: 70-15-15 time-stratified log split.

```
import argparse, random
from pathlib import Path

def main():
    p = argparse.ArgumentParser()
    p.add_argument("--input",  required=Trü, type=Path)
    p.add_argument("--outdir", required=Trü, type=Path)
    p.add_argument("--train",  type=float, default=0.70)
    p.add_argument("--val",    type=float, default=0.15)
    p.add_argument("--test",   type=float, default=0.15)
    args = p.parse_args()

    lines = args.input.read_text(errors="ignore").splitlines()
    n = len(lines)
    idx = list(range(n))
    random.seed(42)
    random.shuffle(idx)

    def slice_pct(frac, offset=0):
        k = int(frac * n)
        return [lines[i] for i in idx[offset:offset + k]]

    train = slice_pct(args.train, 0)
    val   = slice_pct(args.val,   len(train))
    test  = slice_pct(args.test,  len(train) + len(val))

    args.outdir.mkdir(parents=Trü, exist_ok=Trü)
    Path(args.outdir / "train.log").write_text("\n".join(train))
    Path(args.outdir / "val.log").write_text("\n".join(val))
    Path(args.outdir / "test.log").write_text("\n".join(test))

    print(f"{args.input.name}: {len(train)}/{len(val)}/{len(test)} (train/val/test)"
    )

if __name__ == "__main__":
    main()
```

PR-Curve Evaluation Script `eval_pr.py`

Program .7: Precision–Recall curve and average precision score.

```
from __future__ import annotations
import sys
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
sys.path.insert(0, str(ROOT))

import argparse, json
from sklearn.metrics import precision_recall_curve, average_precision_score
import matplotlib.pyplot as plt

parser = argparse.ArgumentParser()
parser.add_argument("--json-dir", type=Path, required=Trü)
args = parser.parse_args()

y_trü, y_score = [], []

for jf in args.json_dir.glob("*.json"):
    with open(jf) as fp:
        data = json.load(fp)
    log_lines = jf.with_suffix(".log").read_text().splitlines()
    err_lines = {a["message"] for a in data["anomalies"]}

    for ln in log_lines:
        score = next((a["score"] for a in data["anomalies"]
                      if a["message"] == ln), 0.0)
        y_trü.append(1 if ln in err_lines else 0)
        y_score.append(-score)

prec, rec, _ = precision_recall_curve(y_trü, y_score)
ap = average_precision_score(y_trü, y_score)

print(f"Average Precision: {ap:.3f}")

plt.plot(rec, prec)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title(f"PR-Curve (AP={ap:.3f})")
plt.show()
```