# Investigating log reduction strategies for cloud-native 5G networks

## Trade-off analysis in terms of CPU overhead, storage requirements, volume reduction and retained system visibility

**Yana Mihaylova**

**Supervisors: Nitinder Mohan, Sehan Samarakoon Mudiyanselage**

**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 18, 2026

Name of the student: Yana Mihaylova
Final project course: CSE3000 Research Project
Thesis committee: Nitinder Mohan, Sehan Samarakoon Mudiyanselage, Jérémie Decouchant

## Abstract

Cloud-native 5G core networks generate large volumes of heterogeneous log data across multiple microservice components, making telemetry management a critical operational challenge. Most existing log reduction techniques have not been evaluated on 5G core logs in particular, so the best approach for reducing log volume in such a system remains unclear. This paper investigates five log reduction strategies - LogShrink, Denum, SALO, Drain and Log Preprocessing - applied to an Open5GS deployment on a Kubernetes-in-Docker cluster under ten scenarios (steady-state, bursty traffic, and eight fault injections). The strategies are evaluated across volume reduction, CPU overhead and five system visibility metrics.

The two strategy families (online and offline) operate on different inputs and use separate baselines, so their figures are not directly comparable. Lossless offline strategies (LogShrink and Denum) achieve 83–96% byte reduction with full visibility preservation, with Denum far more resource-efficient than LogShrink. Lossy online strategies (SALO, Log Preprocessing, Drain), on the other hand, reduce real-time log streams by 53–89% at low cluster overhead but significantly reduce fault-signal retention. No single strategy dominates all dimensions simultaneously. The study provides a framework for selecting log reduction strategies in cloud-native 5G deployments based on specific operational constraints.


## 1 Introduction

As 5G networks become more complex and widespread, they naturally evolve towards a cloud-native, microservices-based architecture [12]. While this transition offers increased flexibility and scalability, it also introduces significant challenges with regard to monitoring and telemetry. The sheer volume of data generated across multiple network layers can overwhelm traditional monitoring systems and allow system failures to go undetected for a long time. In practice, uncontrolled telemetry growth has measurable consequences - storage space and computing resource requirements are high [3, 4], and the flood of harmless normal-operation event logs often causes alert fatigue, which results in operators missing real faults [15]. Therefore, there is a pressing need to develop effective telemetry-reduction strategies to manage this data load while still providing valuable insights into network performance and health.

While telemetry consists of logs, metrics, and traces, for the purposes of this research, logs were chosen as the primary focus because they provide deeper information and richer context. Additionally, investigating reduction methodologies for all telemetry artifacts would be infeasible within the given time frame and is left for future development. Therefore, this research aims to answer the following question: **RQ: How can different log reduction strategies be effectively applied to lower the volume of logs collected from cloud-native 5G core networks?** Splitting the research question into two smaller subquestions, we aim to answer the following: **SQ1: How much log volume reduction can selected log reduction strategies achieve when applied to logs collected from cloud-native 5G core networks?** and **SQ2: What trade-offs do different log reduction methods introduce in terms of storage requirements, processing overhead, and retained visibility?**

To address this, the study investigates various reduction strategies, which in this implementation are specifically tailored to work in an Open5GS KinD (Kubernetes in Docker) cluster by using different tools such as Loki, Prometheus, and a DaemonSet deployed in Docker. It observes two larger families of reduction methods - offline lossless compression and lossy online real-time reduction [4] (discussed in more detail in sections 2.3 and 3.1) and compares the chosen strategies within each family.

The main findings can be summarised as follows. Lossless offline strategies (LogShrink and Denum) achieve 83–96% byte reduction relative to the raw log stream with full fault-signal preservation, but operate on static files and are unsuited to live deployments. Within this family, Denum performs better on all resource-related metrics, while LogShrink's main advantage is that it achieves a higher compression ratio (23.6× vs. 6.7×). Lossy online strategies (SALO, Log Preprocessing and Drain) run inside the cluster and reduce the live log stream by 53–89% in terms of line count at low overhead (under 31 s CPU per scenario), preserving fault-template visibility at 94–100%, but reducing other visibility metrics noticeably (fault-window retention at 16.1%-37.4% and fault-novelty retention at 20.0%-41.7%). Among the online strategies, SALO best preserves fault signals and has the lowest average CPU cost (18.4 s); Log Preprocessing offers comparable visibility and overhead; Drain achieves the highest event reduction but the weakest fault-signal retention. No strategy dominates across all dimensions and the two families of strategies cannot be compared to each other.

This paper is structured as follows. Section 2 discusses the background of the research and existing relevant work on the topic. Following this, Section 3 provides a brief overview of the setup required for replicating the experiments and outlines the methodology used. The primary outcomes and data analysis are presented in Section 4. Section 5 provides a conclusion and discusses future developments, followed by Section 6, which focuses on the ethical considerations of responsible research. There is also an Appendix containing information about query latency, throughput, peak memory consumed, changes made to the original LogShrink repository and a per-scenario breakdown of CPU overhead for the online methods.

## 2 Background and related work

### 2.1 5G core network functions

NextGen (5G) core networks are composed of multiple network functions (NFs) and reference points connecting them [7]. It can be seen in Figure 1 that User Equipments (UEs) are connected to either the Radio Access Network (RAN) or

the Access Network (AN), as well as the Access and Mobility Function (AMF). The RAN is a cellular network that uses 3GPP standards (e.g., 5G gNodeB base stations) [2, 7], while the AN is more general and could be either a cellular or non-cellular network (Wi-Fi, Ethernet). The core NFs are AMF, which provides authorization and authentication for UEs; the User Plane Function (UPF) used for data transfer; the Session Management Function (SMF), which allocates IP addresses to UEs and controls the UPF; the Policy Control Function (PCF) that determines policies about mobility and session management; the Application Function (AF), which provides information about packet flow to the PCF; the Authentication Server Function (AUSF), which stores data for authentication of UEs; and the User Data Management (UDM) that stores subscription data of UEs.

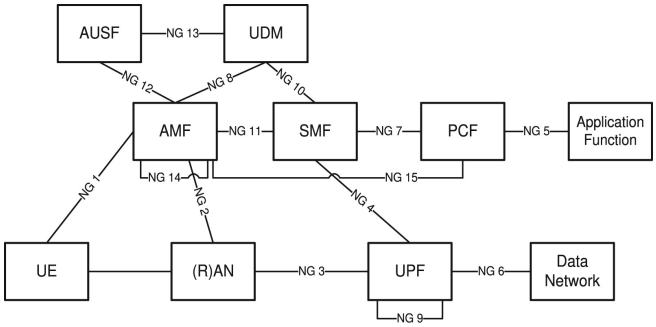


Figure 1: Architecture and reference points for NextGen. Figure taken from [7].

## 2.2 Monitoring in 5G core environments

The complex structure of modern 5G networks poses significant challenges in terms of monitoring, especially when the architecture is horizontally scaled by adopting a cloud-native microservice-based approach [12, 13]. In such deployments, each network function runs as one or more containerised microservice instances that can be independently scaled, restarted, or rescheduled by an orchestrator such as Kubernetes. This creates a system in which a routine procedure such as UE registration spans multiple network functions, each of which emits its own log entries for the same event. This distributed behavior introduces massive data redundancy, consuming excessive storage and CPU resources. Therefore, it is of great importance to introduce mechanisms for log reduction to handle the large number of repeated log entries.

## 2.3 Log reduction

Log reduction is a problem encountered in many domains, and not solely in the context of 5G cores. Consequently, there exist numerous tools for effective log volume management, such as LogShrink [8], LogReducer [16] and LogZip [10], among others, for static log compression. Denum [17] is a method that aims to achieve the same goal by only applying compression on the numeric tokens encountered in system logs. Additionally, there exist numerous tools that can be used for dynamic logging, such as Self-adjusting log observability (SALO) [11] or Log Preprocessing [18], which filters logs based on spatial and temporal deduplication. Deep Learning-Based Log Anomaly Detection [14] is another effective technique that trains a model to recognise log templates. Drain [5] is also worth noting - it is an online log parser, widely used and cited by other strategies as the abovementioned LogShrink [8], LogZip [10] and Denum [17]. This research focuses in particular on LogShrink, Denum, SALO, Drain and Log Preprocessing. They can be split into two more general groups relative to their implementation specifics and ability to observe faults in the system.

**Offline (lossless) strategies**

The strategies in this family apply reduction after the data is already collected, which is why they are named offline methods. Of the chosen approaches above, these are LogShrink and Denum. LogShrink uses a Drain-based parser to build a template library from a sample of the input, then encodes each log event as a pair of the template ID and the variable values list. They are stored column-oriented, which creates a repetitive structure and allows a general-purpose compressor (lzma or gzip, for example) to process this table.

Denum similarly relies on a parsing step but puts an emphasis on numeric tokens, which are usually the majority of variable content in system logs. Its numeric token parsing module extracts different templates and applies specific encoding methods to each class - for instance, storing increasing delta values for timestamps. The remaining content is additionally handled by a string processing module, and both outputs are fed into a general-purpose compressor. In both cases, no log information is discarded: given the compressed file, the original input can be reconstructed exactly from it by decompression, which is why the two methods are also classified as lossless.

**Online (lossy) strategies**

The online strategies operate at collection time and dynamically discard log events that are determined to be redundant or below a certain threshold. They filter out whole lines and segments of log files and are more prone to missing fault signals, which is why they are also referred to as lossy. From the methods listed above, these are SALO, Drain and Log Preprocessing. While Log Preprocessing is not explicitly said to be an online method, since it has not been considered in the context of 5G core networks, it is straightforward to adapt for real-time log filtering.

SALO is a log observability system designed to selectively collect logs from cloud-native applications based on component health. By default, SALO collects only at the highest severity level, using a lightweight GateKeeper sidecar attached to each component to detect anomalies. The sidecar trains a model on healthy log data and flags any deviation as a potential fault, at which point the component's log level is elevated. SALO also elevates log collection for neighbouring components within a certain blast radius, since cascading faults can propagate beyond the affected component. Neighbouring components' log levels are determined by their topological distance and interaction frequency with the faulty component.

Log Preprocessing achieves a similar goal through spatial and temporal deduplication. Spatially, if the same log template is emitted by multiple pods within a time window, only a single instance of that template is kept and forwarded. Temporally, if a template repeats sequentially within the same time window on a single pod, only the first occurrence is retained. Additionally, Log Preprocessing tracks event causality - if an event or a sequence of events appears directly after another event several times, the "effect" events can be omitted.

Finally, Drain is a log parser that clusters incoming log lines into templates using a fixed-depth parse tree. Lines are matched to existing clusters using a similarity threshold on the fraction of non-wildcard tokens that are the same. If the similarity exceeds the threshold, the line is assigned to that cluster, and any differing tokens are replaced with wildcards; otherwise, a new cluster is created. This results in a template vocabulary that is continuously updated during a single pass.

### 2.4 Research gap

Existing log reduction work has been evaluated mostly on general-purpose datasets, and the specific characteristics of 5G core network logs have not been investigated in detail. In cloud-native 5G core networks, a single event may produce correlated log entries across many network functions simultaneously, triggering spatial redundancy and specific burst patterns that cannot be encountered in the datasets used to evaluate the tools described above. Moreover, most existing strategies for log reduction describe offline processing on existing files, which is ill-suited for 5G cores, as online reduction is considerably more valuable in production. In addition, while lossless and lossy strategies represent fundamentally different trade-offs between storage savings and information retention, no prior study has systematically compared representatives of both classes under the same 5G core workload and evaluated the effects of each strategy on fault observability. This work addresses that gap by deploying and evaluating LogShrink, Denum, SALO, Drain and Log Preprocessing on a common 5G core dataset and assessing the extent to which each method preserves the log signals necessary for fault detection as well as the system memory and CPU resources consumed in the process.

## 3 Methodology and experimental setup

### 3.1 Offline and online strategies implementations

#### Offline strategies

The offline strategies (LogShrink and Denum) were implemented using their existing corresponding repositories - https://github.com/IntelligentDDS/LogShrink and https://github.com/gaiusyu/Denum. It is worth noting that changes were made to some parameters in a forked copy of the existing LogShrink repository, which exists at https://github.com/yn-mi12/LogShrink. Most importantly, the built-in sampler was bypassed so that Drain trains on the full input file: the default sampler floor covered only $\sim$33% of the small 5G-core logs, causing the remaining lines to be dropped and the compression ratio to be artificially inflated. Additional fixes are discussed in the Appendix C.

After these dependencies are set up as git submodules, the raw CSV file must be preprocessed by stripping artifacts such as ANSI escape codes and colour sequences, for example, so that the two compressors can actually parse it. This, in turn, shrinks the size of the data, that is actually being reduced, which should be taken into account when looking at the reduction percentages of all strategies. LogShrink operates in three stages: a Drain-based parser trains on the input to build a template library, a C++ binary (THULR) encodes each event as a tuple in a column-oriented layout, and gzip compresses the result. Denum follows the same template decomposition scheme but applies stronger per-column codecs (xz and bz2) instead of gzip, and processes logs in 100,000-line blocks instead of saving the full template library in memory. This accounts for Denum's lower memory footprint compared to LogShrink (Appendix B).

#### Online strategies

The online strategies (SALO, Drain and Log Preprocessing) are each implemented as a separate Kubernetes DaemonSet, all sharing the same filter agent codebase and differing only in the strategy selected via an environment variable. Although the framework in [18] was not explicitly designed for 5G-specific logs and describes the Log Preprocessing pipeline as an offline procedure, this implementation adapts the provided instructions to execute dynamic filtering, consequently improving the scalability of the method. Each DaemonSet creates one filter-agent pod per cluster node, which tails all pod log files in `/var/log/pods/` using Linux inotify for low-latency event notification. Filtered lines are pushed to Loki via HTTP and simultaneously written to a local CSV file.

SALO and Log Preprocessing require pre-trained data - SALO uses rare log templates and Log Preprocessing tracks causality using Apriori rules [1], which allows it to suppress known effect templates when their corresponding cause template has appeared in the current window. Both artifacts are exported from the already existing offline experiment and passed to the DaemonSets using a Kubernetes ConfigMap. Drain requires no pre-trained data - it builds its template cluster tree from the live log stream using the `drain3` library, maintaining a separate parser instance per network function to prevent templates from different 5G components being merged into each other. In addition to the base parser architecture, this implementation applies the following forwarding policy: a line is emitted if its severity level is ERROR or above, when its cluster is newly initialised, its template has just changed, or it belongs to an NF component that has not generated a log within that specific cluster for at least 15 seconds.

For SALO, several adaptations were required to run as a streaming DaemonSet on 5G core logs. First, the GateKeeper health model is replaced with an error-rate threshold - error counts per component are tracked within 60-second windows - rather than 30 seconds as in [11] to reduce false-positive triggers from bursty 5G traffic - and a component is flagged when its counter exceeds an Exponentially Weighted Moving Average (EWMA) adaptive threshold [6]. Second, the topology-based blast radius is replaced with a static tier assignment - core (AMF, SMF, UPF, NRF), support (UDM,

UDR, PCF, AUSF, BSF, NSSF), and infrastructure (MongoDB) - which sets a minimum severity level threshold per tier, corresponding to the static policy option mentioned in [11]. Third, once a burst is detected, a number of subsequent time windows (in this implementation, 2) for that component are forward-flagged to capture the tail of a fault event. In addition, SALO's implementation performs online rarity learning: templates seen in fewer than 5% of completed windows are forwarded unconditionally, allowing SALO to adapt to log patterns that were not seen during pre-training. A limitation of the current approach is that lines emitted within a window before the burst threshold is crossed cannot be retained.

For Log Preprocessing, the temporal deduplication window (15 s) and Apriori causality window (10 s) are set to match the typical patterns in Open5GS logs, as the parameters in [18] were tuned for HPC supercomputer systems and are not directly applicable. Additionally, instead of categorising the events, we use a "blanket" rule that automatically forwards lines with severity level ERROR and above.

## 3.2 Metrics evaluation

Due to the differences in implementation for the offline and online log reduction strategies, the way all metrics are measured differs slightly between the two groups. For this reason, throughout this study, they are compared independently. The throughput, query latency and peak memory consumption for the different strategies are discussed in Appendices A and B.

**Log volume reduction and storage**

The lossless strategies work on a smaller subset of the generated data, as the parsers require a specific format. This means they would achieve a very high reduction percentage but only on roughly 30-40% of the whole existing data, while the rest is omitted. In fact, this is one of the main drawbacks of the current implementation. The online methods use the DaemonSet filter agents to drop log lines as they are read and collected in real-time.

Byte and line reduction are $(1 - \text{output}/\text{input}) \times 100\%$, and the compression ratio is input/output, where input is the Loki CSV for lossy strategies and the stripped log for lossless strategies.

**CPU overhead**

In terms of CPU overhead, offline strategies are compared using Python's ResourceTracker by calling

`resource.getrusage(RUSAGE_SELF)` and `resource.getrusage(RUSAGE_CHILDREN)`

at the entry and exit of the process and summing the user-mode and kernel-mode time deltas from both.

In the case of the online strategies, because the DaemonSets run inside the cluster, resource usage is measured remotely via Prometheus rather than locally.

```
sum(increase(container_cpu_usage_seconds_total
    [T]))
```

is called over the experiment duration. The function above is a cumulative CPU-seconds counter collected from Linux cgroups. The increase() function computes the delta over the window, giving the total CPU-seconds consumed by the DaemonSet pods during the scenario.

**System visibility**

System visibility is the ability to recover fault-relevant information from reduced log files. This study tracks visibility in five distinct ways:

- Total retention is the fraction of all log lines that survive reduction, measured over the full collected log stream.
- Fault-line retention counts lines that contain fault indicators - severity levels ERROR, CRITICAL, or FATAL; UERANSIM bracket notation([error],[info], etc.), and for lines whose format does not carry an explicit severity - key-value pairs that contain the keywords "error", "crash", "timeout", "refused", "oom", etc. It reports the fraction of such lines preserved after reduction, measured over the full log stream rather than the injection window alone, because fault signals are not perfectly contained within it - some anomalies appear in the pre-injection window and cascading effects continue after it.
- Fault template retention reduces each fault line to a template by substituting variable tokens (UUIDs, IP addresses, IMSI/SUPI identifiers, timestamps) with a wildcard, then measures what fraction of the distinct fault templates observed in the original data are still represented in the reduced output. It is measured over the full stream because rare fault patterns can appear outside the strict injection window. This metric complements fault-line retention: a reducer could retain some fault lines while dropping entire fault types.
- Novelty retention finds any log lines whose template was never observed during normal operation (steady-state) and computes the fraction of such novel templates that survive after reduction. Restricting the fault side to within the injection window ensures that only templates that appeared specifically due to the injected fault are evaluated in the denominator, excluding rare patterns that may appear in the pre- or post-fault windows for unrelated reasons.
- Fault-window retention is computed by restricting both the numerator and the denominator to lines with timestamps within the active fault injection period. The other four metrics can be satisfied even if a reducer aggressively compresses the fault period, while leaving pre- and post-fault traffic largely intact, thereby scoring high on the other metrics but missing the most diagnostically important period.

Lossless strategies achieve 100% on all visibility metrics by construction, as they do not drop any log events.

## 3.3 Experimental setup

**Shared setup**

This study is part of a larger group of projects related to observability in cloud-native 5G networks. While they differ in the questions they aim to answer, the projects are largely similar in terms of experimental setup and underlying configuration. They use a common Open5GS KinD cluster running 22

fault scenarios injected via Chaos Mesh, with results across the following observability tools - Prometheus, Jaeger and Loki (metrics, traces, and logs, respectively) - before, during, and after each fault.

The cluster consists of one control-plane node and two worker nodes. The control plane hosts the Kubernetes API server, etcd (the Kubernetes key-value store), and the scheduler, while the two worker nodes run the Open5GS NF pods. The DaemonSet filter agents run across all three nodes. Loki and Prometheus are deployed as cluster-internal services accessible by all nodes.

This project uses Loki and Prometheus only and disables tools related to eBPF or metrics and traces such as Beyla and Jaeger. The described experimental setup is available at https://github.com/yn-mi12/5g-core-and-experiment-setup, while the implementations described in this paper are located in the `experiments/B-log-strategies` directory.

### Baseline

The experiments consist of 10 scenarios - steady and bursty traffic, and 8 different fault scenarios, discussed in the next subsection. The purpose of fault injection is to assess whether the system can still detect specific faults after reduction.

The steady-state scenario runs for ten minutes with 50 UEs attached to the core. The bursty scenario runs for the same duration but alternates the UE count every 30 seconds between 50 and 5 UEs. Each scale-up triggers simultaneous registration, authentication, and session-creation requests across the AMF, SMF, and AUSF pods, causing the same set of log templates to appear sequentially across all pods within a single 15-second window. Each fault scenario consists of a pre-fault window (10 minutes) that captures steady-state traffic with 50 UEs active; a fault window (5 minutes) where the Chaos Mesh fault is injected and a post-fault window (5 minutes) that captures the system recovering and stabilising after the fault clears. Each scenario was repeated three times; all reported metrics are averaged across the runs.

Notably, the strategies use separate ground-truth baselines. The offline strategies (LogShrink and Denum) share input files: the steady-state and bursty CSVs collected in a dedicated live pass (Pass 1) and the pre-collected fault-scenario CSVs. Each online strategy runs an independent cluster pass (SALO: Pass 2, Log Preprocessing: Pass 3, Drain: Pass 4) and uses the raw Loki stream captured during that same pass as its baseline. The four separate collection baselines mean that line counts are not identical across strategies and that comparisons of reduction percentages across the groups carry both this baseline variation and the offline/online input-size difference noted above (3.2).

### Fault scenarios

Eight fault scenarios are injected using Chaos Mesh, covering three fault classes: pod crashes, resource pressure, and network faults. The scenarios were chosen among the existing 22 to show a range of propagation paths - from isolated component failures to cascading cross-function effects - and to span both control-plane and data-plane faults. All fault injection experiments are applied via Chaos Mesh's PodChaos, StressChaos, and NetworkChaos fault types and are removed automatically upon execution, leaving the cluster in a recoverable state.

- Pod crashes - (**AMF pod crash**) AMF receives SIGKILL, causing a 5–15 s restart gap during which active UE connections are dropped and incoming registrations fail. (**UDM pod crash**) UDM receives SIGKILL - AUSF cannot retrieve subscriber authentication vectors, meaning new registrations fail at the authentication step while already-registered UEs are unaffected. (**NRF cascade**) NRF receives a SIGKILL and all NFs simultaneously lose service-discovery capability. NRF ceases to output log entries, while AMF, SMF, AUSF, UDM, and PCF flood the SCP with retry traffic, which requires cross-NF correlation to identify the root cause.
- Memory pressure - (**memory pressure UPF**) StressChaos allocates 2,048 MiB per worker across four workers inside the UPF, exceeding UPF resident memory by 60×, triggering an OOM (Out of Memory) kill and restart. The SMF accumulates stale PFCP sessions during the gap.
- Network faults - they use 30% packet loss with 25% spatial correlation and 500 ms of added latency for the delay scenario. **AMF-SCP network partition** cuts all AMF calls routed via the SCP, blocking registration, authentication, and service discovery simultaneously. **UPF packet loss** degrades user-plane throughput and causes SMF PFCP errors. **UPF infrastructure packet loss** is injected simultaneously on the N4 control plane (SMF↔UPF) and the N6 data plane (UPF↔data network), combining control-plane disruption with user-plane degradation. **NRF network delay** slows all NF heartbeats and discovery traffic at the NRF without a visible failure, producing gradual timeout propagation across the AMF, SCP, and SMF logs.

## 4 Results

As mentioned in Section 3.2, the two groups of strategies differ in terms of input size and resource metrics evaluation, which means they are not directly comparable to each other. Therefore, throughout the Results Section, they will be analysed separately.

### 4.1 Log volume and storage

Figure 2 depicts the byte reduction percentage for the offline methods, while Figure 3 shows the percentage reduction in log events (lines) for the online strategies.

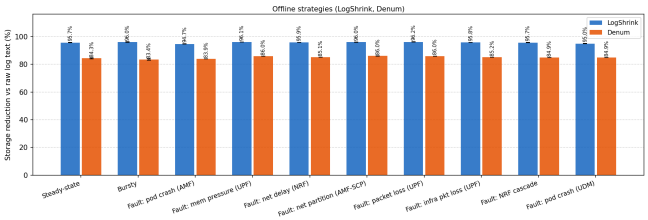


Figure 2: Offline strategies - byte reduction

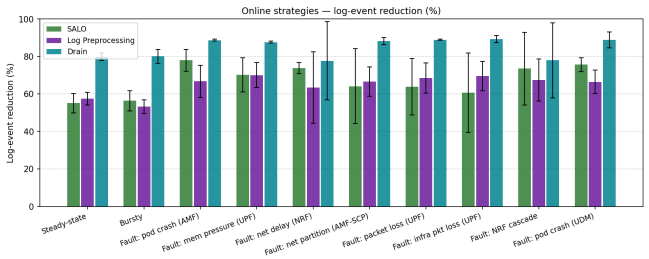


Figure 3: Online strategies - log event reduction

LogShrink and Denum operate as lossless compressors that preserve all log events, which is the reason they are excluded from Figure 3. Regarding byte reduction, LogShrink consistently achieves a reduction of $\sim$ 96% while Denum yields $\sim$ 85%. The gap arises from LogShrink's structured column encoding, which more aggressively exploits template repetition. Both strategies are stable across scenarios, confirming that the byte-level compressibility of 5G logs is largely scenario-independent. The reason for this is that most 5G log lines share the same structural templates (Figure 4) — precisely the redundancy that LogShrink and Denum are designed to address. Additionally, due to the reduced size of the pre-stripped log inputs, the results are further inflated.

The online strategies dynamically drop runtime events from the live telemetry stream. Log Preprocessing filters out 57.5% of events in steady-state, 53.3% in the bursty scenario, and 63.4–70.1% during the eight fault scenarios. The low variance is caused by its rule-based keyword and deduplication filters, which are mostly scenario-independent. SALO achieves a reduction of 55.1% (steady-state) and 56.4% (bursty), which scales to 60.7–78.0% during fault scenarios. This broader range reflects SALO's adaptive suppression: for instance, during the AMF pod crash (78.0% reduction), the rapid reconnection and re-registration retry loops generate a dense cluster of highly repetitive logs that are easily suppressed (Figure 4). In contrast, infrastructure packet loss simultaneously disrupts every network function across the data and control planes and generates highly diverse templates, causing SALO's rarity threshold to fire less often (60.7% reduction). Drain achieves a consistently higher event reduction than both: 79.8% in steady-state, 80.1% during the bursty state, and 77.8–89.3% during fault scenarios. Its lowest reduction occurs under NRF network delay (77.8%), where gradual timeout propagation across multiple NFs produces structurally distinct lines (varying NF names, timer values and peer addresses) that Drain assigns to separate clusters and treats as novel. In contrast, pod crashes and packet-loss scenarios generate repeating retry loops that Drain quickly clusters and suppresses, achieving 87–93% reduction. In terms of byte reduction, SALO reduces storage by 60.0–73.9%, Log Preprocessing by 55.4–67.1%, and Drain by 82.9–92.4%.

Run-to-run variability is high for Drain under NRF cascade and NRF network delay, where non-deterministic fault severity determines whether a repetitive or structurally diverse log burst is generated. SALO shows similarly high variability for infrastructure packet loss and AMF-SCP partition.

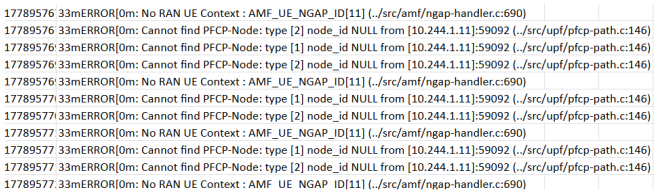


Figure 4: Excerpt from the AMF-pod-crash logs during fault injection. Three error templates repeat at fixed intervals across AMF and UPF as UEs retry registration.

## 4.2 CPU overhead

This subsection discusses the CPU overhead of each strategy, while the throughput, query latency and peak memory consumption, which are relevant to overall performance and resource management, are discussed in Appendices A and B.

### Offline methods

CPU overhead differs substantially between the two offline strategies, as Figure 5 illustrates. LogShrink requires 5.2–11.1 s per scenario (mean 8.2 s) due to the processing costs of maintaining a global template table across the entire data stream. Denum processes each line independently, requiring only 0.2–1.2 s (mean 0.8 s), which is roughly a 10$\times$ improvement in efficiency.

### Online methods

Figure 5 shows the mean values for CPU overhead across all scenarios: SALO 18.4 s, Log Preprocessing 19.9 s, and Drain 18.6 s. The full per-scenario analysis is given in Appendix D. Run-to-run variance differs between strategies - SALO is the most stable (with a mean standard deviation ($\sigma$) of 4.4 s across scenarios). Both Drain and Log Preprocessing are affected by a consistent run-02 anomaly: across nearly every scenario, run-02 records a CPU consumption drop of approximately 50%, lowering the reported mean by roughly 16% relative to runs 01 and 03. The paper includes all three runs for consistency. Appendix D presents the deviations between runs and investigates the means with run-02 excluded.

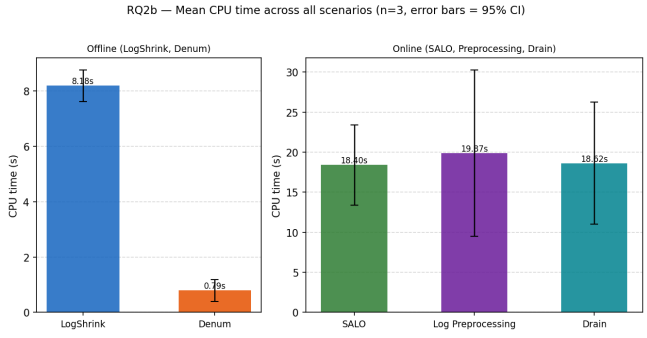


Figure 5: Average CPU time consumed by reduction. Mean computed across the different scenarios.

## 4.3 System visibility

Because LogShrink and Denum are lossless, they preserve 100% of all visibility metrics by construction. The analysis

below focuses on the three lossy strategies. The different visibility metrics are discussed in more detail in Section 3.2 - System visibility.

### Fault template visibility

Figure 6 shows how Log Preprocessing achieves 100% fault template visibility across all eight scenarios - every distinct message template appears at least once in the filtered output. SALO's fault template coverage is slightly lower: it reaches 100% in six of the eight fault scenarios and falls to 95.2% (AMF pod crash) and 95.3% (NRF cascade) in the remaining two, yielding a fault-scenario average of 98.8%. Drain achieves 100% template visibility in four scenarios and falls below 100% in four others: 98.0% for AMF pod crash, 98.9% for UDM pod crash, 96.9% for memory pressure, and 94.7% for NRF network delay - achieving an average of 98.6%. The four below-100% cases involve faults that produce a mix of novel and previously seen templates. Some of these templates are suppressed because they match an existing cluster, which was seen within the last 15-second window.

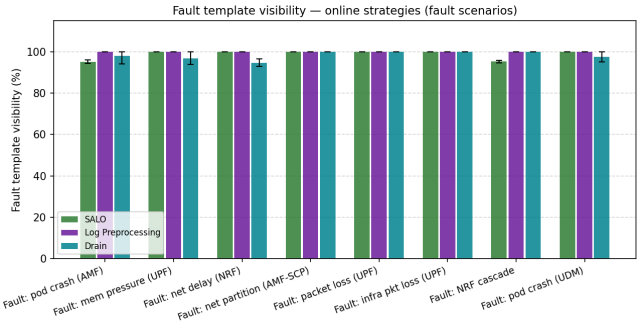


Figure 6: Fault template visibility across different strategies and scenarios

### Fault line retention

The fraction of individual fault-related lines retained post-reduction varies among the three strategies (Figure7).

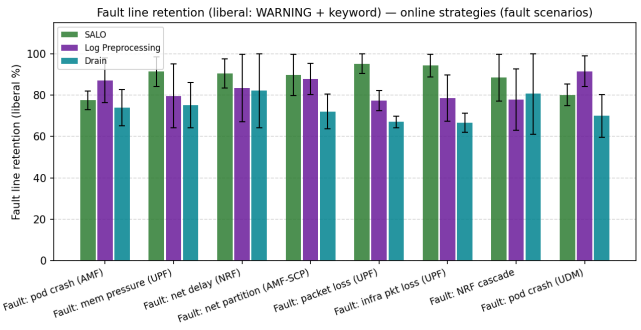


Figure 7: Liberal fault line retention across all fault scenarios.

All three strategies retain 100% of the strict lines (those carrying ERROR severity level or higher). However, when evaluating liberal fault indicators (WARNING severity level and above), SALO retains, on average, 88.4%, ranging from 77.5% (AMF pod crash) to 95.2% (UPF packet loss). Log Preprocessing retains an average of 83.0% with a narrower range (77.5–91.6%) and Drain retains 73.6% on average, ranging from 66.7% (infra packet loss) to 82.3% (NRF network delay). Run-to-run variation is highest for Drain under NRF network delay and NRF cascade, reflecting the non-deterministic nature of those faults: the fraction of fault lines that fall within Drain's 15-second suppression window differs significantly across repetitions. Drain's lower fault-line retention is a direct consequence of its temporal suppression - repeated occurrences of a cluster are discarded even when they carry fault-related keywords.

### Fault-window retention

The fraction of lines emitted during the active fault injection window that survive filtering is low for all three lossy strategies (Figure 8).

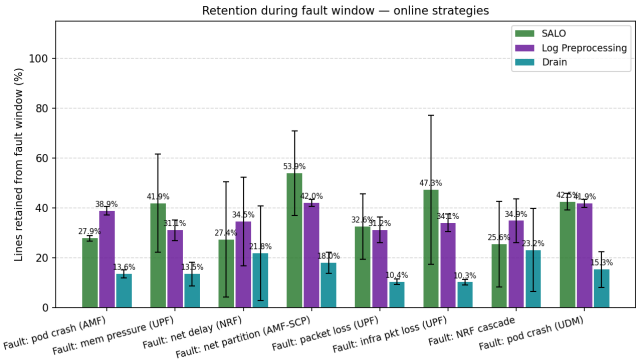


Figure 8: Fault window retention across different strategies and scenarios.

On average, SALO retains 37.4%, Log Preprocessing 36.1%, and Drain 16.1% of lines from within the fault window (averaged across the eight fault scenarios). The low values reflect the high volume of routine background traffic that co-occurs with the fault and is aggressively suppressed by all three strategies. Drain's substantially lower window retention (16.1% vs. ∼37% for the other two) reflects its 15-second temporal suppression, which discards repeated cluster occurrences even during the active fault window, regardless of whether the cluster is fault-related. Fault-window retention also shows high run-to-run variability for SALO under UPF infrastructure packet loss ($\sigma$ = 26 percentage points (pp)) and NRF network delay ($\sigma$ = 20 pp), driven by the stochastic nature of those faults.

### Novelty retention

Evaluating the preservation of novel log templates - defined as structural patterns never observed during normal steady-state operations - reveals severe diagnostic limitations across all strategies (Figure 9). SALO and Log Preprocessing each retain 41.7% while Drain retains 20.0% of novelty anomalies (averaged across fault scenarios). This means that a considerable fraction of newly appearing error patterns may be silently discarded during real-time processing. This is a critical vulnerability, as novel templates are among the most di-

agnostically important signals for the recognition of a particular fault. Per-fault variability is noteworthy: Drain's novelty retention ranges from approximately 13% (infra packet loss) to 29% (NRF cascade). SALO's novelty retention is notably higher in net partition (55.1%) because the sudden loss of all SCP-routed traffic generates a large set of structurally unique error messages that SALO's burst detector forwards in full. Run-to-run variance in novelty retention is also non-negligible, with Drain showing the largest difference, due to the non-deterministic nature of fault propagation between runs.

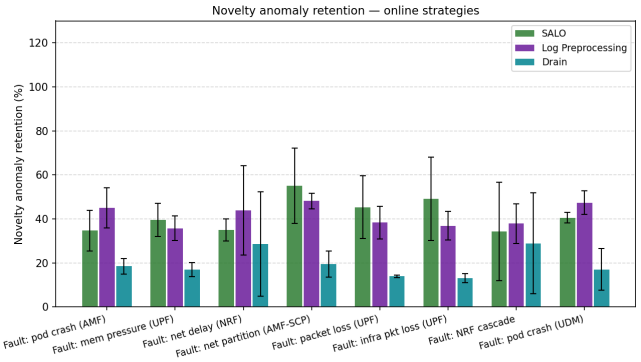


Figure 9: Novelty anomalies visibility across strategies and scenarios

### 4.4 Trade-off analysis

The summary heatmaps (Figures 10 and 11) make the trade-offs in each group explicit across all metrics. Within the offline group, Denum outperforms LogShrink in every resource dimension - CPU time, peak memory, query latency - while achieving only slightly less byte compression and identical visibility. LogShrink's only advantage is its higher compression ratio (23.6× vs. 6.7×), which is relevant when long-term storage cost is the primary constraint. The memory gap between the two offline strategies is substantial: LogShrink peaks at 1,566 MiB RSS due to its global template table, whereas Denum's per-line processing keeps it below 83 MiB regardless of input size (this is discussed in more detail in Appendix B).

Within the online group, Log Preprocessing's mean CPU time across all scenarios (19.9 s per run) is slightly above SALO's (18.4 s), yet SALO retains a higher fraction of fault-lines (88.4% vs. 83.0%) and achieves a higher reduction percentage (67.2% vs. 65.0%). Drain achieves much higher event reduction (84.7%) and comparable fault-template visibility (98.6%), but at the cost of much lower fault-window retention (16.1%) and novelty retention (20.0%).

The summary heatmaps establish the core operational trade-offs characterising each strategy family. Within the offline group, Denum outperforms LogShrink across all primary computing resource dimensions and is the clearly preferable method. LogShrink should be selected when an extreme compression ratio is required. Within the online group, SALO provides the most balanced optimisation profile and is preferable when fault-signal preservation (or operational efficiency, as it carries the lowest CPU cost) is the priority. Log Preprocessing provides a viable alternative if an operation demands simple, regex-driven or rule-based configurations. Finally, Drain should be selected when maximising event reduction is the greatest priority, but system visibility can be neglected. Ultimately, these findings indicate that no single reduction strategy dominates all metrics simultaneously.

## 5 Conclusions and Future Work

This paper investigated five log reduction strategies - LogShrink, Denum, SALO, Drain and Log Preprocessing - applied to an Open5GS 5G core deployment on a Kubernetes cluster, evaluated across ten scenarios including steady-state traffic, bursty traffic, and eight fault-injection scenarios.

### 5.1 Answering SQ1 (volume reduction)

All five strategies achieve a measurable reduction in log volume. Lossless compressors eliminate byte-level redundancy entirely, yielding an 83-96% byte reduction relative to the extracted log lines from the raw Loki CSV baseline. Meanwhile, online strategies reduce the live event stream by 53–89% in line count (and 55–92% in bytes) by filtering logs directly at collection time. While every strategy achieves significant volume management, the lossless methods reach higher absolute reduction values due to the highly repetitive structure of 5G core logs and the pre-stripped nature of their input files.

### 5.2 Answering SQ2 (trade-offs)

The observed trade-offs follow a clear divide between offline lossless compression and online lossy filtering. Because the two families operate on different inputs and utilise separate measurement baselines, their performance characteristics must be evaluated independently.

Within the offline group, both strategies preserve 100% of all visibility metrics by construction. Denum demonstrates clear superiority in nearly every performance aspect: it requires 0.2-1.2 s of CPU time and peaks at 66-83 MiB RSS, compared with 5.2-11.1 s and approximately 1,566 MiB for LogShrink. LogShrink's sole advantage is a higher compression ratio (23.6× vs. 6.7×), a metric that is only prioritised when long-term storage minimisation is the absolute constraint. Crucially, both strategies require significant memory and operate on pre-collected files, rendering them unsuited for continuous, low-latency reduction of live log streams within production deployments.

Within the online group, all three strategies run inside the cluster as a DaemonSet, consuming under 18 MiB of peak memory and 6.3-30.4 s of CPU per scenario. Fault-template visibility remains high across all three (94-100%), but fault-window retention is reduced (SALO 37.4%, Log Preprocessing 36.1%, Drain 16.1%), as is novel fault-template retention (SALO 41.7%, Log Preprocessing 41.7%, Drain 20.0%). Fault-line retention (liberal - WARNING and above severity level) ranges from 67-95% across all strategies and scenarios.

SALO is the preferred framework when fault-signal visibility is the priority and also carries the lowest average CPU

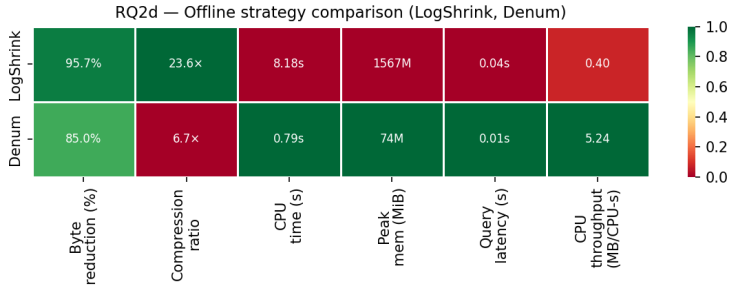


Figure 10: Summary heatmap of trade-offs across all performance metrics. Offline strategies

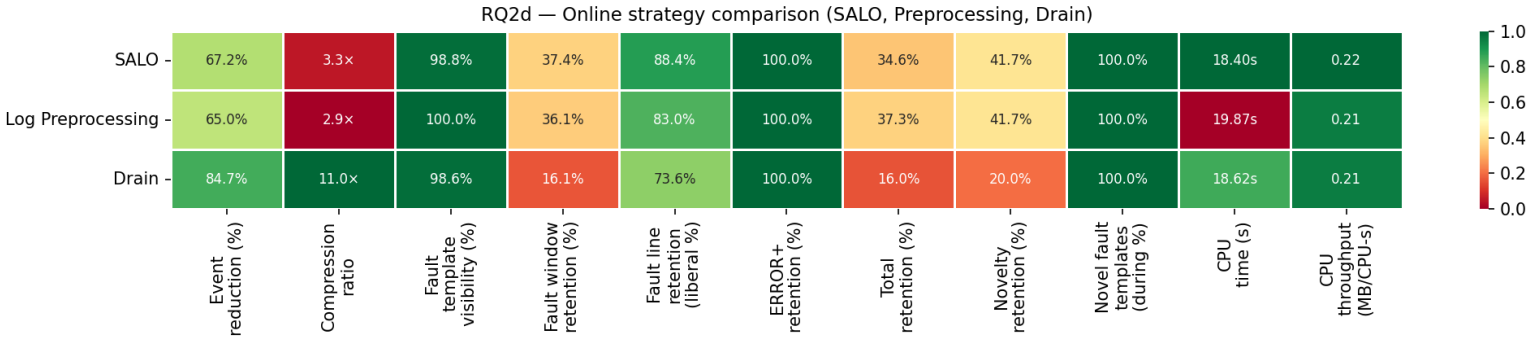


Figure 11: Summary heatmap of trade-offs across all performance metrics. Online strategies

cost (18.4 s vs. 19.9 s for Log Preprocessing and 18.6 s for Drain); Log Preprocessing offers comparable visibility at slightly higher average CPU cost and slightly weaker fault-line retention (83.0% vs. SALO's 88.4%). Conversely, Drain is the appropriate choice when maximising event reduction takes precedence over fault-signal preservation. As illustrated in Figures 10 and 11, no individual strategy achieves optimal results across all dimensions simultaneously.

### 5.3 Future work

Several directions remain for expanding and refining this research. The primary methodology gap is the inability to apply unified metrics to both strategy families - offline strategies use filtered CSV subsets and online strategies intercept raw streams. Future work could define a normalisation scheme to enable valid cross-group comparison. Second, the parameters chosen for the online strategies can be better tuned to balance reduction with data visibility, and a study can be conducted on how different parameter values affect the results. A comprehensive scalability analysis exploring system performance under higher UE capacities also remains unaddressed and would be beneficial.

Additionally, more online reduction strategies can be explored and applied using the existing DaemonSet infrastructure, as they are more useful in practice compared to post-hoc compression. Another possible advancement is conducting more experiment runs to ensure data quality - currently, three full runs of all experiments are averaged to produce the shown graphs, but for better confidence in the observed results, a sample size of at least 5-7 iterations would be desirable. Finally, this study focuses only on log entries and does not investigate any existing reduction methodologies for metrics or traces. The other two telemetry artifacts are equally useful for monitoring and fault recovery, so this is certainly a natural direction for future research.

## 6 Responsible Research

### 6.1 Reproducibility

The full experimental setup - all scripts for data collection, reduction, analysis, and figure generation - is published as part of a shared open-source repository at https://github.com/yn-mi12/5g-core-and-experiment-setup, under the folder experiments/B-log-strategies. The shared Kubernetes-in-Docker cluster configuration, Chaos Mesh fault scenarios, and the Loki/Prometheus stack are version-controlled so that all experiments can be reproduced from scratch. The offline strategies rely on open-source tools: LogShrink [8] and Denum [17]. A fork of the LogShrink repository was created at https://github.com/yn-mi12/LogShrink since some parameters had to be changed to better accommodate 5G-specific logs (Appendix C). This is documented in the fork's commit history.

While the `cluster-start.sh` script sets up all necessary dependencies and creates the cluster, the host machine must have Python 3, Docker, and a Linux environment installed (for example, the experiments described in this paper were run in an Ubuntu 22.04 virtual machine). These requirements are also mentioned in the repository's README.md file. The two offline strategies exist as git

submodules, so immediately after the repository is cloned `git submodule update --init --recursive` has to be run to integrate them into the project.

Additionally, the raw per-scenario CSV files and intermediate processed data are not committed to keep the codebase clean, but they can be generated locally by running the corresponding scripts. The results would be largely consistent with those reported here, with possible small deviations due to the dynamic nature of the network traffic. It is, however, worth mentioning that the data generation might take several hours ($\sim$ 21), due to the amount of fault scenarios and the guards ensuring the cluster is healthy (timeouts of 5 or 10 minutes in between runs, and cluster restarts). This paper includes the averaged values of 3 separate runs of all experiments under the same conditions to ensure data quality, but more runs are possible and desirable.

### 6.2 LLM usage

Claude (Anthropic) was used as an AI assistant throughout this project to create Python scripts for data collection and the analysis pipeline for figure generation. All generated code was reviewed, tested, and validated against expected outputs before being committed. Experimental results were not generated artificially, and an LLM was not used to interpret any findings or write the text of this paper.

### 6.3 Possible privacy and security risks

5G core logs can contain sensitive subscriber data - SUPI/IMSI identifiers, for example - so any system that intercepts, filters, or stores telemetry introduces a privacy risk. The DaemonSet agents read every log line emitted by core network functions; a compromised agent could selectively suppress events or exfiltrate traffic metadata. By discarding lines before they reach the collector, lossy strategies reduce the possibility of a thorough investigation if such an attack occurs. Additionally, the choice of which templates to treat as rare is a decision made by the developer and may cause valuable data to be discarded.

This concern is primarily relevant to future deployment scenarios. If this benchmark is used commercially with user data, the administrator must ensure that safeguards are in place to protect this data. The log entries collected to achieve the results described in this paper did not involve any personal or private data handling.

### 6.4 Limitations

The main limitation of this study is the inability to compare offline and online strategies using the same unified metrics. The first issue is that for offline strategies, log compression requires a specific format, and some lines have to be dropped, causing an artificial growth in the achieved reduction percentage. The second problem is that data generation has to be rerun for online strategies as they filter while logs are collected. The three DaemonSets also have to be separated to avoid mutual interference. This creates four separate passes and four separate ground truths.

Another possible concern is that many of the strategies implemented during these experiments are not designed for and may not be entirely compatible with the format of 5G logs. Despite this, one of the main research gaps is exactly this lack of research into 5G core logs, so the mismatch is anticipated and largely addressed (for example, Log Preprocessing is not inherently expected to be an online method). The methodology for each implementation is described in more detail in Section 3.1. Notably, the given interpretations of each method are not the only ones possible, and improvements to the proposed methodologies are welcome - parameters can be configured with better flexibility and certain rules modified or removed. On the other hand, some strategies that are well suited for 5G cores were overlooked or discovered too late in the scope of this project - DL-based anomaly detection [14], for instance. These are promising candidates for future development.

Finally, it is generally a good practice to conduct at least 3 experimental runs to ensure data quality. While 3 runs were completed for this research, 3-4 more executions would further strengthen the conclusions. It should be noted, however, that each run takes approximately 21 hours on a suitably powerful machine.

## References


[1] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, 12(1):307–328, 1996.

[2] Nhu-Ngoc Dao, Ngo Hoang Tu, Trong-Dai Hoang, Tri-Hai Nguyen, Luong Vuong Nguyen, Kyungchun Lee, Laihyuk Park, Woongsoo Na, and Sungrae Cho. A review on new technologies in 3gpp standards for 5g access and beyond. *Computer Networks*, 245:110370, 2024.

[3] Ahmad El Sayed, Hassan Harb, Marc Ruiz, and Luis Velasco. Zizo: A zoom-in zoom-out mechanism for minimizing redundancy and saving energy in wireless sensor networks. *IEEE Sensors Journal*, 21(3):3452–3462, 2020.

[4] Ahmad El Sayed, Marc Ruiz, Hassan Harb, and Luis Velasco. Deep learning-based adaptive compression and anomaly detection for smart b5g use cases operation. *Sensors*, 23(2), 2023.

[5] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017.

[6] J. Stuart Hunter. The exponentially weighted moving average. *Journal of Quality Technology*, 18(4):203–210, 1986.

[7] Junseok Kim, Dongmyoung Kim, and Sunghyun Choi. 3gpp sa2 architecture and functions for 5g mobile communication system. *ICT Express*, 3(1):1–8, 2017.

[8] Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. Logshrink: Effective log compression by leveraging commonality and variability of log data. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Association for Computing Machinery, 2024.

[9] Hao Lin, Jingyu Zhou, Bin Yao, Minyi Guo, and Jie Li. Cowic: A column-wise independent compression for log stream analysis. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 21–30, 2015.

[10] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R. Lyu. Logzip: Extracting hidden structures via iterative clustering for log compression. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 863–873, 2019.

[11] Divya Pathak, Mudit Verma, Aishwariya Chakraborty, and Harshit Kumar. Self adjusting log observability for cloud native applications. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 482–493, 2024.

[12] Sebastian Robitzsch, Marco Centenaro, Nicola di Pietro, Luis Cordeiro, André S. Gomes, Peter Sanders, and Arif Ishaq. Prospects on the adoption of a microservice-based architecture in 5g systems and beyond. *Computer Networks*, 237:110058, 2023.

[13] Pengpeng Song, Hui Peng, and Xiaowen Zhang. A micro-service approach to cloud native ran for 5g and beyond. *IEEE Access*, 11:130257–130271, 2023.

[14] Yawen Tan, Jiadai Wang, Jiajia Liu, and Yuanhao Li. Deep learning-based log anomaly detection for 5g core network. In *2023 IEEE/CIC International Conference on Communications in China (ICCC)*, pages 1–6, 2023.

[15] Fotios Voutsas, John Violos, and Aris Leivadeas. Mitigating alert fatigue in cloud monitoring systems: A machine learning perspective. *Computer Networks*, 250:110543, 2024.

[16] Junyu Wei, Guangyan Zhang, Yang Wang, Zhiwei Liu, Zhanyang Zhu, Junchao Chen, Tingtao Sun, and Qi Zhou. On the feasibility of parser-based log compression in Large-Scale cloud systems. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 249–262. USENIX Association, February 2021.

[17] Siyu Yu, Yifan Wu, Ying Li, and Pinjia He. Unlocking the power of numbers: Log compression via numeric token parsing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 919–930, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Ziming Zheng, Zhiling Lan, Byung H. Park, and Al Geist. System log pre-processing to improve failure prediction. In *2009 IEEE/IFIP International Conference on Dependable Systems  Networks*, pages 572–577, 2009.

# A Throughput and query latency

## A.1 Query latency

Query latency is the time required to retrieve a specific group of logs corresponding to a given query from the strategy's output after reduction has completed. It is especially relevant for the two offline strategies, as they produce compressed files that must be decompressed before any pattern can be searched. Online strategies write events to a flat CSV that can be scanned directly. Cowic [9], a library that aims to lower decompression time, illustrates why query latency is a relevant metric and how it is computed.

For all strategies, query latency is measured by performing a linear scan of the output CSV after writing the output file using `time_linear_scan()`. The command searches for the pattern "ERROR" and records wall time via `time.perf_counter()`.

## A.2 Throughput

Throughput measures how efficiently each strategy processes its input and is expressed as megabytes of input data processed per CPU-second (MB/CPU-s). Both LogShrink [8] and Denum [17] describe "compression speed" as a metric that shows how fast the input file is compressed, which is essentially what throughput measures. For offline strategies, this is computed by dividing the uncompressed input size in megabytes by the total CPU time recorded via `resource.getrusage()`. For online strategies, the input volume is estimated from the number of log lines captured in the Loki baseline during the same pass, and the denominator is the DaemonSet CPU time obtained from Prometheus as described above. Throughput is reported for the steady-state scenario only, consistent with the other overhead metrics.

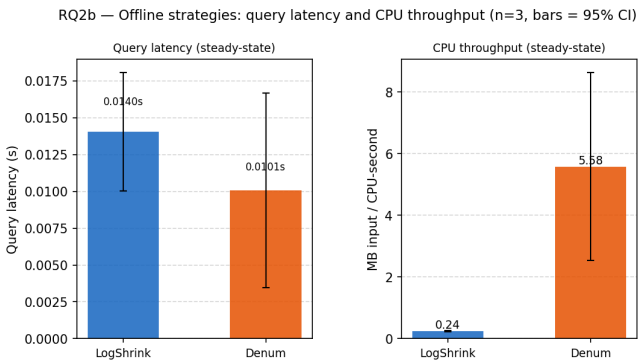


Figure 12: CPU throughput and query latency for offline strategies

Denum achieves a throughput of 2.6-3.6 MB of input per CPU-second (Figure 12). By comparison, LogShrink achieves a lower throughput of only 0.16-0.45 MB/CPU-s. Query latency follows the opposite ordering: retrieving events from the LogShrink-compressed output requires a decompression step, adding 0.011-0.124 s per query, while Denum's decompression is faster at 0.009-0.018 s.

# B Peak memory

Peak memory is calculated by taking the RSS (resident set size) - the portion of memory occupied by a process in main memory.

For the offline methods, a daemon thread samples `resource.getrusage(RUSAGE_SELF).ru_maxrss` every

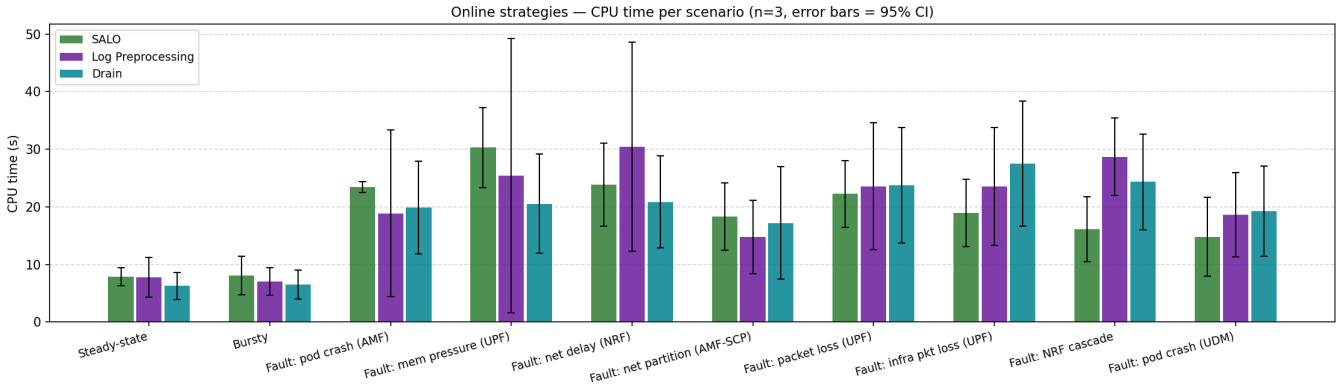


Figure 13: CPU cost, online strategies

500 ms throughout the measurement window. The maximum observed value is retained. Any RSS increase in a child process (e.g., the gzip subprocess invoked by LogShrink) is also captured from the `RUSAGE_CHILDREN` delta.

For the online strategies `max(max_over_time( container_memory_working_set_bytes[T]))` is used where `container_memory_working_set_bytes` reports RSS, `max_over_time` takes the highest instantaneous value within the window across all DaemonSet pods, and the outer max aggregates across nodes.

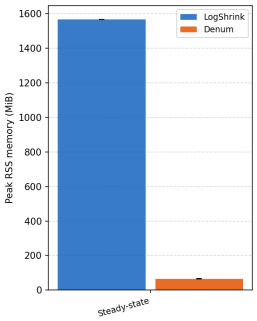


Figure 14: Peak memory usage for offline strategies (steady state)

Denum peaks at 66-83 MiB of RSS memory (Figure 14), while LogShrink peaks at approximately 1,566 MiB of RSS. The large memory consumption of LogShrink is consistent with its implementation - it first extracts log templates using a Python-based Drain parser, then invokes THULR - a compiled C++ binary - to perform template matching over the full log corpus. This is the source of both its superior compression ratio and its higher resource requirements. Denum processes each line independently and does not keep a global state, so its memory usage remains constant regardless of input volume.

Within the online group, peak memory stays below 18 MiB for all three live strategies in all scenarios, reflecting their streaming nature.

# C LogShrink repository changes

The following changes were made to the LogShrink forked repository to better accommodate 5G-specific logs and prevent memory overflow and other potential issues.

- The `MAXCOL` constant in the C parser was reduced from 256 to 32, since 5G core templates carry at most 6 variables per line - the original value caused stack overflows during parsing.
- A NumPy compatibility fix was applied to handle variable-length sequence windows that raised a `ValueError` in NumPy versions 1.24 and above.
- A wall-time measurement was added to exclude Python interpreter startup from the reported CPU overhead.
- The one-hot encoding step caps the template vocabulary at the 300 most-frequent entries to prevent memory growth.

# D CPU cost for all scenarios for online methods

Fault scenarios scale 3–4× above the steady-state baselines mentioned in Section 4.2. Drain and Log Preprocessing scale more steeply, while SALO partially absorbs increased input volume by suppressing repeated patterns earlier in the pipeline.

As Section 4.2 mentions, the variance between runs is considerable, as illustrated in Figure 13. Log Preprocessing shows the highest overall variance, caused by run-03, which reaches 3.5× run-01 CPU on the memory pressure and 2.9× on the pod crash (AMF) scenarios where non-deterministic fault propagation generated more log traffic. This is consistent with Log Preprocessing applying its rules to every input line unconditionally, whereas SALO and Drain partially absorb input-volume growth by suppressing repeated patterns.

A bigger issue is, however, the anomalous run-02 (Figure 15), which lowers the reported means by $\sim$ 16%. Figure 16 illustrates how the mean values change when the faulty iteration is excluded - Log Preprocessing achieves a CPU overhead of 23.7 s and Drain reaches 22.1 s, while SALO remains the same. The average overhead per scenario is also higher - $\sim$ 37% instead of 30.4%.

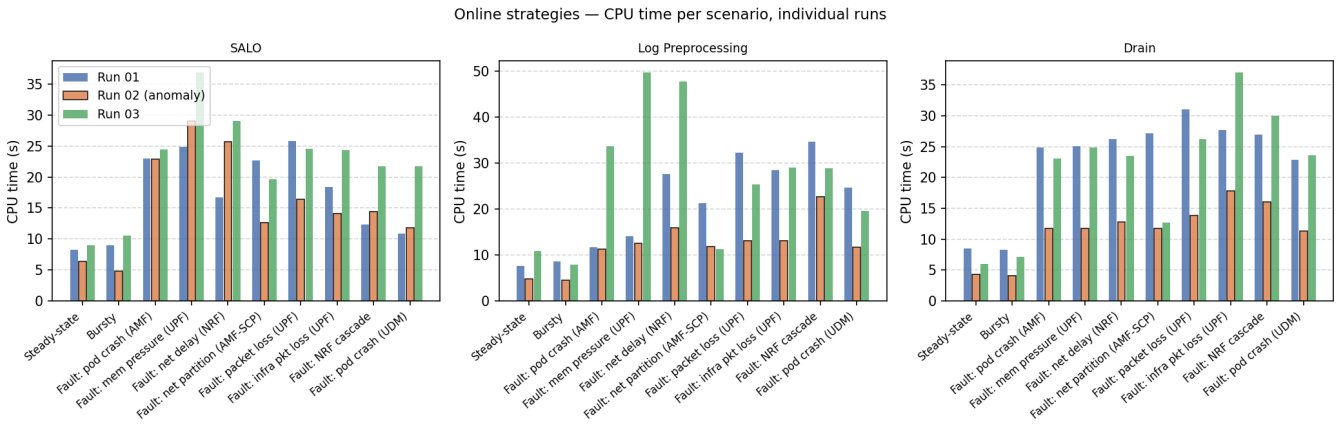


Figure 15: CPU cost, individual runs breakdown

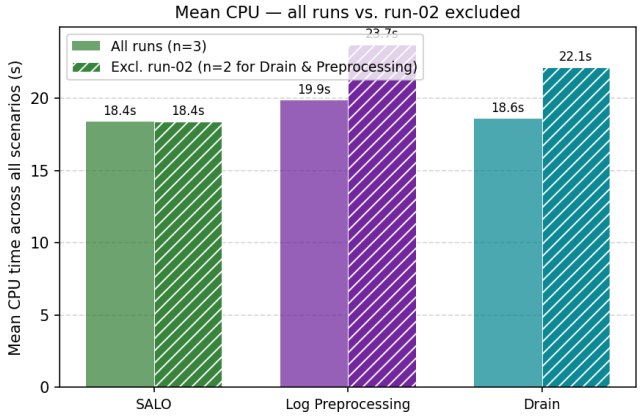


Figure 16: CPU means with run-02 included and excluded