Investigating log reduction strategies for cloud-native 5G networks

AUTHOR

Yana Mihaylova
y.r.mihaylova@student.tudelft.nl

SUPERVISOR

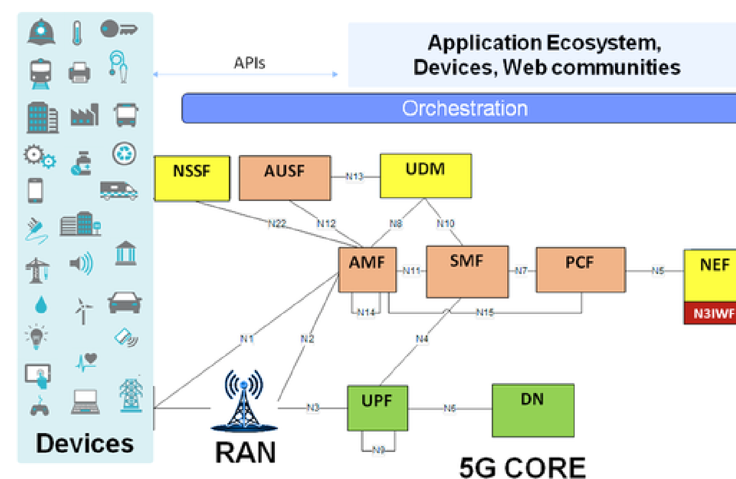
Sehan Samarakoon Mudiyanselage

RESPONSIBLE PROFESSOR

Nitinder Mohan

INTRODUCTION

- Cloud-native 5G core networks consist of different network functions (NFs) occupying Kubernetes pods. They generate high volumes of heterogeneous logs, distributed across multiple layers, which introduces significant challenges related to observability. The load of information can easily overwhelm traditional monitoring systems and cause failures to go unnoticed for a long time.
- Existing log reduction techniques have not been evaluated under realistic 5G network conditions, which leaves the question of whether fault visibility can be preserved during intense reduction unanswered.
- The solution – this research investigates five different log reduction strategies in a KinD Open5GS deployment across steady-state, bursty and fault-injection scenarios, evaluating their trade-offs in terms of volume reduction, retained system visibility, CPU overhead, and storage requirements.



RESEARCH QUESTION

How can different log reduction strategies be effectively applied to lower the volume of logs collected from cloud-native 5G core networks?

- SQ1: How much log volume reduction can selected log reduction strategies achieve when applied to logs collected from cloud-native 5G core networks?
- SQ2: What trade-offs do different log reduction methods introduce in terms of storage requirements, processing overhead, and retained visibility?

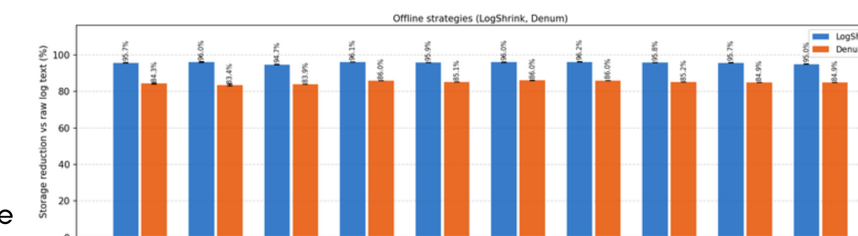
METHODOLOGY

- Strategies: Log reduction splits into two general families of strategies – offline (lossless) and online (lossy). They use different implementations and separate baselines, so they should not be compared with each other.
  - Offline strategies – operate on pre-collected data and preserve all log events (lines).
  - Online strategies – applied during collection and filtering the incoming log stream dynamically. They are prone to discarding valuable diagnostic information. To implement the online strategies, this project uses Kubernetes DaemonSets that tail NF pod logs.

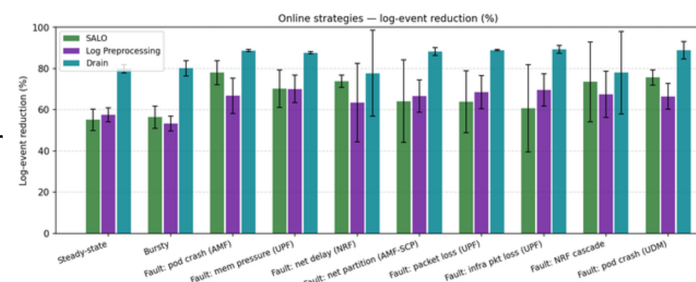
Strategy	Type	Mechanism
LogShrink	Offline	Extract template and variable-values pairs, store them column-oriented and apply a general-purpose compressor
Denum	Offline	Extract numeric tokens (IPs, timestamps, IDs) and apply class-specific encoding, then compress
SALO	Online	Drop low-severity lines from healthy NFs; automatically switch to keep-all mode when an NF's error rate exceeds a given threshold
Log Preprocessing	Online	Suppress duplicate log events - if the same message template was already seen on the same pod (or same NF type) recently, drop it. Also suppress causal events using Apriori rules
Drain	Online	Parse logs into clusters of structurally similar messages and forward only the first occurrence of each cluster per time window. ERROR+ lines and newly initialised clusters are forwarded always

- Scenarios (10): steady-state (50 UEs, 10 min), bursty traffic (UEs alternating between 50 and 5 every 30 s) and 8 different fault injection scenarios (faults are artificially injected to check if they will be detected after reduction)
- Technology stack – Open5GS and UERANSIM on a KinD (Kubernetes in Docker) cluster, with Loki for log collection, Prometheus for resource tracking and Chaos Mesh for fault injection.
- Data collection – 5 passes – 1st – raw data collection and the 2 offline strategies applied post-hoc; 2-4 – each online DaemonSet filter; 5 – visibility measurement against each pass's ground truth. 3 separate runs of all experiments were made.

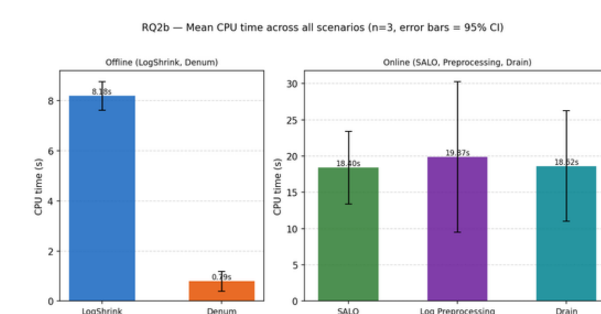
RESULTS



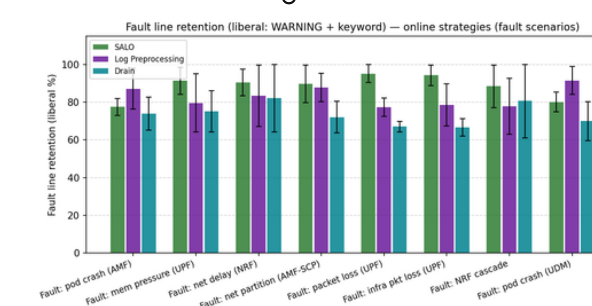
- **Log volume reduction**
  - Lossless strategies achieve 83–96% byte reduction.
  - They operate only on a pre-stripped version of the original CSV because the parser requires a specific format.
  - Online strategies reduce lines by 53–89%. Run-to-run variance is higher than offline, driven by non-deterministic fault severity.



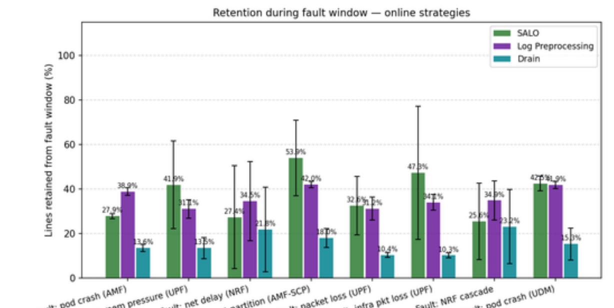
- **CPU overhead**
  - LogShrink averages 8.2s CPU per scenario, while Denum does 0.8 (nearly a 10× gap)
  - Among the online strategies, Log Preprocessing has the highest CPU cost (19.9s). The variance is elevated due to a known anomaly during the 2nd run.

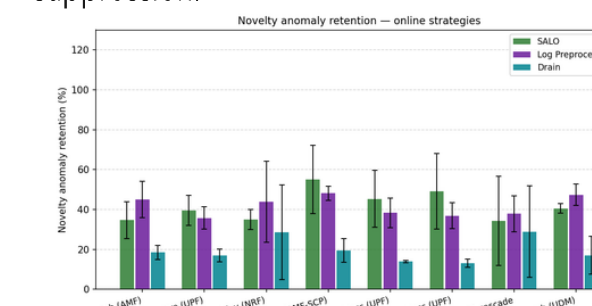


- **System visibility**
  - Fault-template visibility (distinct fault event types visible after reduction) – Log Preprocessing reaches 100% for all fault scenarios, SALO and Drain miss some templates but achieve averages of 98.8% and 98.6%.



  - Fault-line retention (the % of kept lines with severity WARNING+) – SALO performs best, averaging 88.4%, as its burst detector switches to keep-all mode during error spikes. Drain is weakest at 73.6% – its 15-second suppression window discards repeated occurrences of a cluster.
  - Fault-window retention (the % of kept lines from inside the fault-injection window) – routine traffic co-occurs with the fault and is aggressively reduced by all 3 strategies. Drain's significantly lower value again reflects its temporal suppression.





  - Novelty template retention (the % of kept lines whose template was never seen in the steady-state baseline) – the most critical metric for fault diagnosis, as novel templates are the strongest signal of a new fault type. The scores (41.7% on average for SALO and Log Preprocessing and 20% for Drain) are low, which should be concerning.

- **Trade-off analysis**

RQ2d — Online strategy comparison (SALO, Preprocessing, Drain)

	Event reduction (%)	Compression ratio	Fault template visibility (%)	Fault window retention (%)	Fault line retention (liberal %)	ERROR+ retention (%)	Total retention (%)	Novelty retention (%)	Novel fault templates (during %)	CPU time (s)	CPU throughput (MB/CPU-s)
SALO	67.2%	3.3×	98.8%	37.4%	88.4%	100.0%	34.6%	41.7%	100.0%	18.40s	0.22
Log Preprocessing	65.0%	2.9×	100.0%	36.1%	83.0%	100.0%	37.3%	41.7%	100.0%	19.87s	0.21
Drain	84.7%	11.0×	98.6%	16.1%	73.6%	100.0%	16.0%	20.0%	100.0%	18.62s	0.21

RQ2d — Offline strategy comparison (LogShrink, Denum)

	Byte reduction (%)	Compression ratio	CPU time (s)	Peak mem (MiB)	Query latency (s)	CPU throughput (MB/CPU-s)
LogShrink	95.7%	23.6×	8.18s	1567M	0.04s	0.40
Denum	85.0%	6.7×	0.79s	74M	0.01s	5.24

  - SALO achieves the lowest CPU and highest visibility retention. Log Preprocessing is a simpler rule-based alternative with comparable visibility at slightly higher CPU. Drain achieves the highest event reduction but retains only 20% of novel templates and 16.1% of fault-window lines.
  - Denum leads in every resource-related metric – 10× lower CPU, 19× lower peak memory and faster query latency. LogShrink's sole advantage is a higher compression ratio.

CONCLUSION

- All five strategies achieve measurable reduction, while no single one dominates all dimensions simultaneously. The two strategy families must be evaluated independently.
- Within the offline group, Denum is the clearly preferable method. Within the online group, SALO offers the best balance between fault-signal preservation and event reduction.
- Future work: extend reduction to metrics and traces, develop a unified evaluation baseline.