TASK Quarterly 30 (2) 2026

Functionality comparison of open-source PaaS monitoring systems

Filip Sołtys [1,2*], **Henryk Krawczyk** [1,2]

[1] Centre of Informatics TASK, Gdańsk University of Technology, Narutowicza 11/12, 80-233 Gdańsk, Poland
[2] Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, Narutowicza 11/12, 80-233 Gdańsk, Poland

*filip.soltys@pg.edu.pl

https://doi.org/10.34808/tq2026/30.2/c

Abstract

For Platform-as-a-Service (PaaS) environments characterized by elasticity, multi-tenancy, and layered dependencies, this paper proposes a structured method for selecting a vendor-neutral open-source observability stack. The proposed method is based on an analysis of commercial monitoring solutions and covers five functional categories: metrics monitoring, visualization, log monitoring, distributed tracing, and alert routing/incident response. Selected open-source tools are evaluated using technical and operational criteria, including integration, scalability, high availability, and maintainability, with a 1–5 Mean Opinion Score (MOS)-style rubric. Based on the results, the proposed toolchain consists of Prometheus, Grafana, Loki, Jaeger, and Alertmanager. It can serve as a practical monitoring framework for different PaaS environments.

Keywords:

Platform-as-a-Service (PaaS), Cloud monitoring, Open-source solutions, Toolchain proposition

1. Introduction

Platform as a Service (PaaS) shifts a large portion of operational responsibility from application teams to the platform layer, while simultaneously increasing the number of interdependent operational components, such as runtimes, controllers, service meshes, collectors, storage backends, and alerting services that can affect service reliability. According to the National Institute of Standards and Technology (NIST) definition [1], PaaS users control their deployed applications (and sometimes application-hosting configuration), but do not manage the underlying cloud infrastructure-making platform-level telemetry and diagnostics essential for both operators and tenants [2]. Tomarchio et al. emphasize that cloud platforms increasingly rely on orchestration across heterogeneous and multi-cloud environments, where applications, platform services, and infrastructure components are distributed across different administrative and technological domains. This supports the motivation of the present paper; in such environments, monitoring cannot depend only on a single provider-specific toolset, but should rely on interoperable and vendor-neutral observability mechanisms. Therefore, their work provides a conceptual background for treating portability, integration, and cross-layer visibility as important criteria in the evaluation of open-source PaaS monitoring systems.

In practice, monitoring for PaaS is less about observing static infrastructure "health" and more about maintaining confidence in service behavior under continuous change. Monitoring must support trend analysis, fast debugging, and actionable alerting, while enabling different stakeholders (platform/Site Reliability Engineering (SRE) teams and application teams) to share a consistent operational picture. These goals align with reliability engineering guidance that emphasizes monitoring as a foundation for alerting, debugging, and decision-making, rather than a passive reporting mechanism.

The importance of cloud monitoring has also been emphasized in previous research. Aceto et al. provide a broad survey of cloud-monitoring definitions, motivations, platforms, and open challenges, while Syed et al. classify cloud-monitoring approaches through a taxonomy covering architectural and operational aspects. These works confirm that monitoring is a central requirement for operating complex cloud environments [3].

A PaaS environment is typically elastic by design (workloads scale up/down, instances are short-lived), multi-tenant (shared control planes and shared infrastructure), and layered (applications run atop platform services, which run atop orchestration and infrastructure). This creates a diagnostic gap; when something goes wrong, symptoms may surface at the application level even if the root cause is in platform automation, networking, or shared dependencies. Effective PaaS monitoring therefore must make system behavior observable end-to-end and across layers so that platform operators can protect the platform's service-level objectives (SLOs), and application teams can protect user-facing SLOs with minimal friction.

Modern observability practice converges on combining multiple telemetry "signals" because no single signal is sufficient in dynamic distributed systems:

- Metrics capture quantitative behavior and are typically the first line for detection and Service-Level Objective (SLO)/Service-Level Agreement (SLA) tracking.
- Logs provide discrete events and rich context for incident investigation and audit.
- Traces reveal request paths across microservices and platform components, enabling root-cause analysis for latency and dependency problems.

Recent studies on cloud-native observability confirm that metrics, logs, and traces should be treated as complementary signals rather than isolated monitoring outputs. Kosińska et al. present a systematic mapping study of observability in cloud-native applications, while Usman et al. discuss observability requirements in distributed edge and container-based microservice environments [4].

To make these signals operationally useful in PaaS, correlation becomes a first-class need; engineers should be able to pivot from a metric anomaly to the relevant logs and traces (and back) using shared context such as service identity, labels/tags, timestamps, and trace identifiers. Interoperable context propagation is especially important when services and intermediaries are heterogeneous; the World Wide Web Consortium (W3C) Trace Context specification formalizes how trace identifiers can be propagated across boundaries to avoid "broken traces" in multi-component transactions.

Finally, PaaS monitoring is incomplete without an incident-response layer. Alerting must translate telemetry into prioritized, routed, and deduplicated notifications that reduce noise and accelerate recovery. In a PaaS setting, alerting quality directly influences platform trust; excessive noise results in alert fatigue (Table 1), while insufficient sensitivity delays detection and increases blast radius. Consequently, selecting an observability stack for PaaS is inherently a hard decision; it must support reliable operations, enable shared understanding ("single pane of glass"), and remain sustainable under growth and change.

Section 2 presents professional monitoring patterns for cloud and PaaS-oriented applications, using native monitoring ecosystems of AWS, Azure, and Google Cloud as reference points. It discusses how commercial platforms organize telemetry collection, storage, correlation,

visualization, alerting, and governance, and identifies recurring architectural patterns such as centralized monitoring, agentless default telemetry, multi-signal correlation, topology-aware diagnosis, adaptive alerting, and observability-as-code.

Section 3 introduces the evaluation method for open-source monitoring systems. The section defines five functional categories required for a complete PaaS observability stack: metrics monitoring, visualization, log monitoring, distributed tracing, and alerting/incident response. For each category, it presents selection criteria, describes the evaluated tools, and applies a 1–5 MOS-style scoring rubric to compare their suitability for scalable and maintainable PaaS monitoring environments.

The evaluation identified a best-in-class open-source tool in each functional category. The average score is the arithmetic mean of criterion-level 1–5 MOS ratings, where 1 denotes weak or partially incomplete solutions and 5 denotes mature, production-ready solutions. The highest-scoring tools per category were: Prometheus (avg. 4.40; runner-up VictoriaMetrics 4.20) for metrics, Grafana (4.50; runner-up Zabbix 2.80) for visualization, Loki (4.20; runner-up Zabbix 2.50) for log monitoring, Jaeger (4.40; runner-up OpenTelemetry 3.70) for distributed tracing, and Alertmanager (4.30; runner-ups Grafana 3.90 and Zabbix 3.10) for alert routing and noise reduction. Quantitative MOS results are reported in Tables 1–5 (Section 3). No inferential statistics are provided, as the evaluation is a structured rubric-based comparison.

Section 4 consolidates the evaluation results into a general monitoring framework. It first discusses cross-cutting criteria such as scalability, high availability, interoperability, configuration maintainability, resource overhead, documentation quality, and community maturity. Then, based on the highest-scoring tools in each category, it proposes an integrated open-source toolchain composed of Prometheus, Grafana, Loki, Jaeger, and Alertmanager, and presents a simplified architecture showing how metrics, logs, traces, visualization, and alert routing can be combined into a practical PaaS monitoring system.

The final section summarizes the main findings, discusses practical implications of the proposed toolchain, and outlines possible directions for further validation and development of the framework.

To improve readability and avoid ambiguity, Table 1 summarizes selected terms and phrases used throughout the paper. These terms are common in monitoring and observability practice, but some of them may be interpreted differently depending on the cloud provider, tool ecosystem, or operational context. The table therefore provides concise working definitions used consistently in this publication.

Table 1: Definitions of selected monitoring and observability terms

Term / phrase	Definition
MOS score	A 1–5 Mean Opinion Score-style rating used to assess how well a tool satisfies a given evaluation criterion, where 1 means weak or missing support and 5 means mature, production-ready support.
Average score	The arithmetic mean of criterion-level MOS scores within a given functional category.
Moving parts	Interdependent operational components of a monitoring environment, such as runtimes, controllers, service meshes, collectors, storage backends, and alerting services.
Low setup	Low initial configuration effort for managed services, where baseline telemetry is collected automatically.
Vendor lock-in	Increased migration cost caused by provider-specific telemetry formats, APIs, dashboards, alerting rules, and access-control models.
Burn-rate indicator	A metric showing how quickly a service consumes its allowed error budget; for example, a burn rate above 1 means the service is consuming the budget faster than planned.
Scraped samples	Individual metric data points collected periodically by Prometheus from exposed `/metrics` endpoints.
RBAC	Role-based access control; a permission model in which access rights are assigned according to user roles.
Log bucket	A logical storage container for log entries, with its own retention and access-control settings.
At-rest encryption	Encryption of stored telemetry data, as opposed to data being transmitted over the network.
Alert fatigue	Reduced responsiveness caused by excessive low-value or noisy alerts.
Silencing	Temporary suppression of known or expected alerts.
Maintenance window	A planned period during which alerts may be suppressed because maintenance work is expected to affect normal system behavior.
War-room tooling	Collaborative incident-response tools, such as chat rooms, incident channels, or conference bridges, used during operational incidents.
Agent sprawl	The uncontrolled growth of multiple telemetry agents, collectors, or sidecars across services and hosts, increasing configuration and maintenance burden.

2. Professional monitoring systems patterns for cloud applications

2.1. Conceptual foundations

Professional monitoring in cloud-native PaaS contexts aims to provide timely, decision-relevant evidence about the runtime state of applications, the managed platform components they depend on, and the user-visible outcomes of failures or degradations. This chapter focuses exclusively on commercial cloud provider monitoring services—specifically the native offerings of the three major providers operated by *Microsoft Azure (Azure)*, *Amazon Web Services (AWS)* and *Google Cloud Platform (GCP)* and the professional monitoring patterns embodied in these ecosystems.

A widely used analytical lens for "observability" (as reflected across provider-documentation) is the separation of telemetry into metrics, logs, and traces, extended in practice by "change/audit" records. Azure Monitor documentation uses this multi-type perspective explicitly—monitoring data is collected, stored, correlated, and acted on across these data forms [5]. In this chapter, the terms are used in the following operational senses:

Metrics are numeric time series describing system behavior at points in time (e.g., latency percentiles, error counts, CPU utilization). Azure Monitor defines metrics as numeric data collected into a time-series database [6]. Logs are timestamped event records (structured or unstructured) emitted by runtime components; GCP Cloud Logging defines its scope as storing, searching, analyzing, monitoring, and alerting on logging data [7]. Traces are end-to-end request lifecycle records across components; AWS X-Ray and Cloud Trace are representative provider-native tracing systems [8]. "Change/audit" telemetry captures control-plane activity and configuration drift (e.g., API calls, policy changes), exemplified by AWS CloudTrail and Google Cloud Audit Logs [9].

Service-level reliability engineering practice often structures monitoring goals through service-level indicators (SLIs) and service-level objectives (SLOs). Google's Cloud Monitoring "service monitoring" materials define SLOs as targets over SLIs and treat them as first-class configuration objects [10]. AWS CloudWatch similarly supports creating SLOs (via Application Signals) for key services and tracking them on SLO dashboards [11]. Microsoft's Well-Architected guidance defines SLOs as specific, measurable reliability/quality targets, tying them conceptually to customer expectations [6].

The use of commercial cloud platforms as reference models is consistent with earlier research on cloud monitoring tools. Alhamazani et al. discuss research dimensions and design issues in commercial cloud monitoring systems, including dynamic resource tracking, QoS monitoring, SLA violation detection, and operational configuration changes [12].

2.2. Provider-native monitoring portfolios and architectural roles

This subsection enumerates the core native services used for monitoring PaaS environments across AWS, Azure, and GCP, and describes provider-native components required to form end-to-end monitoring architecture (collection, ingestion, storage, correlation, visualization, alerting, and governance).

2.2.1 AWS portfolio and roles

Amazon CloudWatch is positioned as the central monitoring hub; it monitors AWS resources and applications "in real time" and provides observability tooling for performance and operational health [14, 15]. CloudWatch's collection layer is hybrid by design; besides default service telemetry, the CloudWatch agent collects metrics, logs, and traces from virtual machines, on-premises servers, and containerized applications [16]. For serverless PaaS use cases, AWS Lambda automatically publishes invocation metrics to CloudWatch without additional role permissions, supporting an "agentless-by-default" pattern for function-level metrics [17].

Ingestion/storage behavior is a core design constraint shaping monitoring patterns. CloudWatch retains metric data with tiered granularity windows (minutes to months), which encourages downsampling and aggregation strategies for long-horizon analytics [18]. CloudWatch Logs centralizes logs from services and applications; log groups have indefinite retention by default, but retention can be explicitly set (from short to multi-year periods), enabling cost and compliance control through retention policy design [19]. CloudWatch Logs data is always encrypted at rest and can be configured to use customer-managed keys via Key Management Service (KMS), aligning monitoring storage with regulated data handling [20].

Distributed tracing is implemented primarily through AWS X-Ray, which collects request data and supports filtering/insight into downstream calls [8]. X-Ray's service-graph model merges nodes from all services participating in a trace ID into a service graph (service map), enabling topology-aware analysis of latency and dependencies [21]. X-Ray retains trace data for 30 days, and the service uses sampling controls to manage volume—a retention/sampling coupling that materially influences tracing patterns in cost-sensitive PaaS

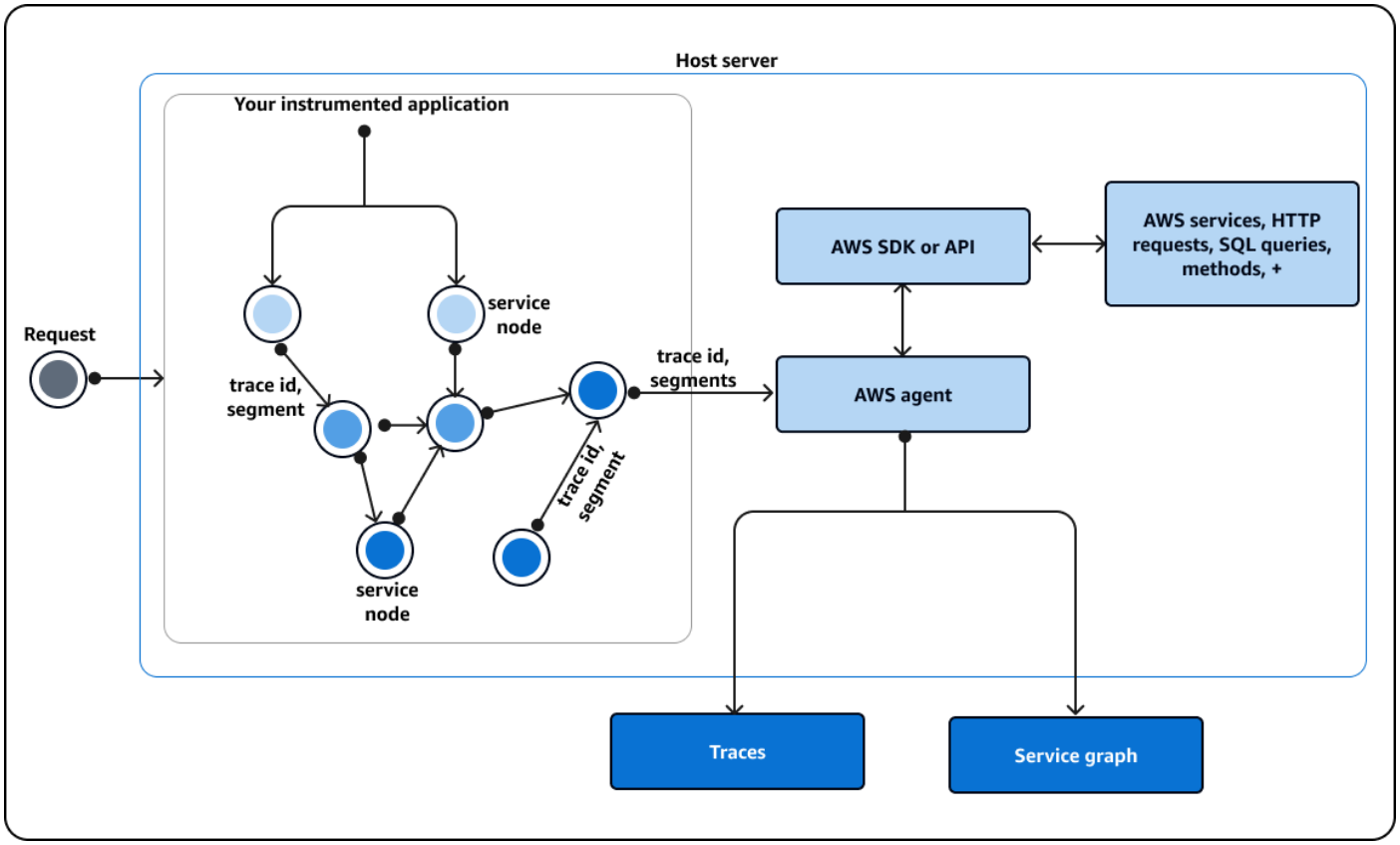

Figure 1: Representation of how AWS X-Ray works [13]

monitoring [21]. Figure 1 presents the AWS X-Ray tracing model, in which is shown instrumented application that produces trace segments describing requests, service calls, and downstream dependencies. These segments are collected by the AWS X-Ray agent and aggregated into complete traces and a service graph. This enables topology-aware analysis of distributed applications by showing both request paths and dependencies between participating services.

For "change/audit" monitoring and compliance evidence, AWS CloudTrail provides event history (90 days) by default and can be extended via trails and storage configuration for longer retention [9]. CloudTrail can also deliver events into CloudWatch Logs for near-real-time security/ops monitoring patterns, enabling metric filters and alerting on audit events [22]. AWS Config, hereafter referred to as Config, complements CloudTrail by recording configuration state and relationships of AWS resources over time; it periodically uploads configuration history files to an Simple Storage Service (S3) bucket [22]. Config supports explicit retention configuration for configuration items (minimum 30 days, up to 7 years), an important compliance/forensics control [23].

Processing/correlation features in AWS illustrate "professional" monitoring patterns that go beyond thresholds. CloudWatch outlier detection applies machine learning to historical metric data to model expected values (including seasonal patterns) and can drive anomaly-based alarms [24]. CloudTrail Insights compares current API call volume and API error frequency against historical baseline patterns. If the current activity deviates significantly from the expected behavior, it generates an insight event, which helps detect abnormal operational or security-related activity. [25]. CloudWatch also supports log anomaly detection that scans incoming log events to surface anomalous patterns, representing an ingestion-time analytics capability [26].

Visualization and topology features include CloudWatch dashboards, cross-account/cross-Region dashboards, and application maps. CloudWatch integrates traces, metrics, and logs into a combined trace-map experience—correlating telemetry to diagnose end-to-end behavior [27].

Alerting and incident response are integrated professionally: CloudWatch alarms can trigger notifications and also create OpsItems in Systems Manager OpsCenter or create incidents in Systems Manager Incident Manager, linking detection to operational workflows [28]. Cost models are usage-based across telemetry types; CloudWatch pricing dimensions include metrics, logs ingestion/storage, and anomaly detection, reinforcing cost-control patterns such as selective custom metrics, restrictive log retention, and anomaly scope design. CloudTrail and Config likewise price by inges-

tion/evaluations, motivating scoping decisions (which events/resources to record and for how long).

2.2.2 Azure portfolio and roles

Azure Monitor is positioned as a comprehensive monitoring solution for collecting, analyzing, and responding to monitoring data across cloud and on-premises environments. Its architecture is explicitly data-platform oriented: metrics, logs, and distributed traces are collected and stored so that common tooling can correlate and analyze them [29, 30].

Metrics in Azure Monitor are numeric time-series data; platform and custom metrics are stored for 93 days (with user experience constraints in portal charting windows), which shapes practices such as exporting or aggregating for long-horizon analysis [6]. Logs are handled by Azure Monitor Logs in Log Analytics workspaces—central repositories designed for query-driven analysis and variable retention by data type [31]. Distributed trace data from instrumented applications are stored within Azure Monitor Logs workspaces, aligning trace retention with workspace retention and access control [29].

Application performance monitoring (APM) is provided through Azure Monitor Application Insights. The official Application Insights guidance emphasizes telemetry collection and storage in Log Analytics workspaces and applies to applications hosted both in Azure and elsewhere [32]. Application Insights telemetry spans requests, dependencies, exceptions, and traces/logs, supporting APM-style correlation within a workspace. It also distinguishes between preaggregated ("standard") and log-based metrics derived from stored telemetry, enabling precision vs. cost trade-offs [33].

In PaaS integration patterns, Azure emphasizes native hookups for managed platforms. Azure App Service monitoring integrates with Azure Monitor and application monitoring workflows, supporting telemetry visualization and alerting for web workloads in a managed hosting context [34]. Azure Functions offers built-in integration with Application Insights for execution monitoring, while host configuration controls logging volume—reflecting a deliberate design where telemetry volume must be governed at source to manage cost and noise [35]. In container PaaS scenarios Azure Kubernetes Service (AKS), Azure Monitor provides container workload visibility via "Container Insights," which uses a containerized agent approach and supports real-time viewing of logs/events/metrics ("Live Data") [35].

For platform and network adjacent monitoring, Network Watcher provides network diagnostic tooling (explicitly framed as infrastructure as a service (IaaS)/network health rather than PaaS monitoring), while Azure Advisor provides best-practice recommendations across cost, performance, reliability, and security [36]. Azure Service Health provides a personalized view of service issues, planned maintenance, and advisories and supports health alerting, which is often treated as a "meta-monitoring" input for platform risk management [37].

Alerting is structured around alert rules and action groups: action groups can notify and trigger automation (including runbooks and functions), and Azure provides dynamic thresholds (historical-pattern-based) and smart detection alerts for application telemetry, implementing "adaptive" alerting patterns to reduce manual threshold tuning [38].

Security and compliance controls include role-based access control (RBAC (Table 1)), i.e. permissions assigned according to user roles, for monitoring resources, structured workspace access modes, and encryption/key management guidance for the monitoring pipeline [39]. For data governance, Azure Monitor provides mechanisms to manage personal data in Log Analytics and delete data when required, which is relevant when application logs can contain identifiers [40]. Pricing is dominated by data volume. Azure Monitor Logs charges primarily for ingestion and retention within workspaces. Pricing documentation explicitly describes usage-based cost drivers, reinforcing patterns such as table-tiering, retention minimization, and selective collection. Figure 2 represents high-level architecture view of Azure Monitor and its functionality.

2.2.3 GCP portfolio and roles

Cloud Monitoring collects metric data and provides visualization and monitoring tools; it automatically collects performance information for most Google Cloud services [41]. Metric retention is long-horizon by default for user-defined metrics (24 months), which supports SLO evaluation and historical analysis without exporting telemetry, but also makes metric hygiene (limiting unused ingestion) a cost and governance concern [43]. Cross-project and multi-environment monitoring is implemented through metric scopes and observability scopes; by default a project monitors its own telemetry, but scopes can be configured to chart/alert across multiple projects while keeping data stored at origin [44]. Cloud Logging is a fully managed service to store/search/analyze/monitor/alert on log data; it automatically collects logs from Google Cloud resources and can ingest logs from other environments, supporting hybrid operational models [7]. PaaS services provide "agentless defaults". App Engine automatically sends request and application logs to Logging, and Cloud Run logs are automatically sent to Cloud Logging [45]. Log-

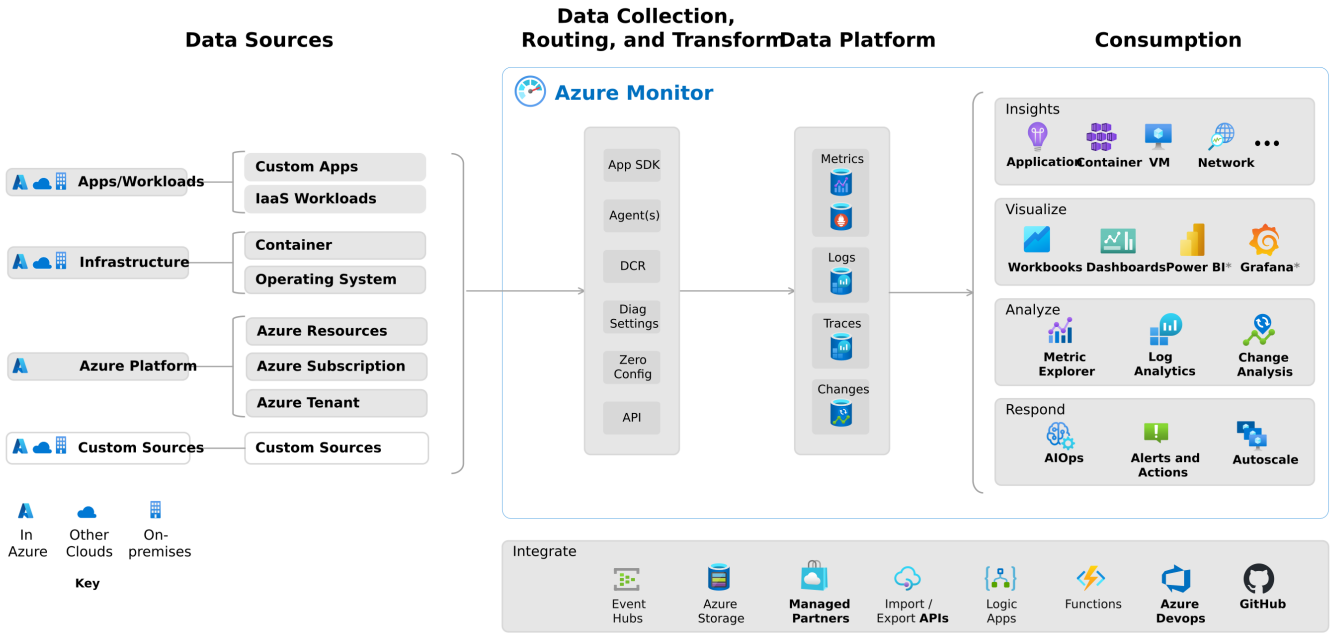

Figure 2: High-level architecture view of Azure Monitor [42]

ging storage is organized into log buckets (Table 1) with explicit retention controls. Default retention is 30 days (configurable 1-3650 days), while the system "Required" bucket retains required audit logs for 400 days and cannot be modified, directly supporting audit/compliance baselines [46].

Change/audit monitoring is implemented via Cloud Audit Logs. Admin Activity logs (and other audit categories) are always written, meaning governance evidence is non-optional at the platform layer, but organizations can control additional audit log categories [47].

For traces and performance diagnosis, Cloud Trace collects latency data and displays it near-real-time; trace retention is 30 days [48]. Cloud Profiler is a statistical low-overhead continuous profiling tool for CPU and memory allocation and is explicitly listed as having no cost in observability pricing materials [49]. Error Reporting aggregates error events from running services and can be automatically integrated with Cloud Run, offering exception-level visibility without bespoke infrastructure [50].

Alerting is incident-centric. Cloud Monitoring creates incidents when alert conditions are met, and notification channels define how on-call teams are notified. Channels are configurable in the console and support multiple routing options [51]. Log-based alerting policies tie Cloud Logging detection into Cloud Monitoring's incident model, forming a unified alert plane across logs and metrics [52]. Pricing is usage-based (data volume/usage), motivating explicit cost controls for monitoring and logging, including alerting cost management guidance.

Security controls rely on Identity and Access Management (IAM) roles and (where needed) customer-managed encryption keys (CMEK) for logging storage. Cloud Logging documents IAM-based access control and CMEK configuration as compliance mechanisms [53]. Figure 3 represents GCP high-level monitoring architecture and describe its main functions

2.3. Cross-provider patterns in professional monitoring

Across AWS, Azure, and GCP, several professional monitoring patterns recur, driven by the same constraints: high telemetry volume, multi-environment separation, compliance requirements, and the need to shorten mean time to detect/resolve [15, 30, 41]. A key architectural pattern is centralized monitoring with logical isolation, typically implemented as a dedicated "monitoring" boundary that can observe multiple accounts/subscriptions/projects. AWS supports cross-account observability and cross-account/cross-Region dashboards and analysis. Azure implements central analysis through Log Analytics workspace design and cross-workspace queries, with recommended architectures that account for tenants, regions, and regulatory data location. GCP uses metric scopes and observability scopes to chart and alert across projects while leaving metric/trace storage at origin, enabling federation-like monitoring at read time.

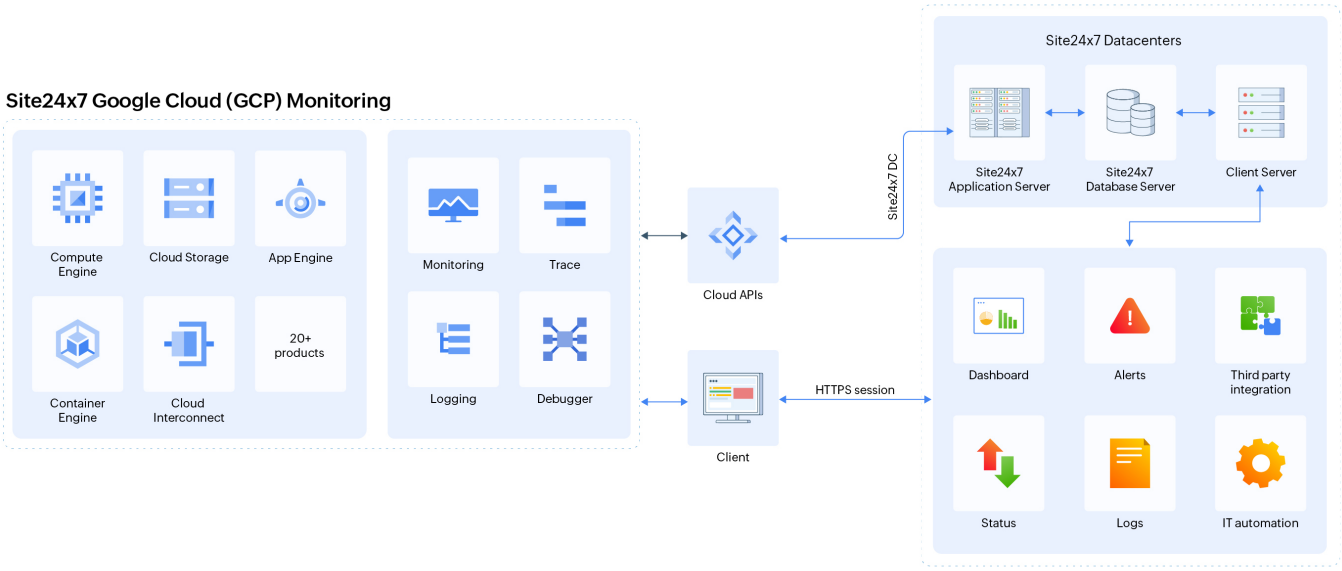

Figure 3: GCP high-level monitoring architecture [54]

A second pattern is agentless defaults with agent augmentation. In all three providers, managed services emit baseline metrics/logs automatically (e.g., AWS Lambda metrics to CloudWatch, App Engine/Cloud Run logs to Cloud Logging), but deeper host/workload visibility requires agents [18, 33, 45]. This reinforces a design principle; PaaS monitoring should treat "agentless telemetry" as mandatory baseline coverage, while agents are reserved for high-value gaps (OS metrics, custom app signals, specialized logs), minimizing operational overhead and cost.

A third pattern is the emergence of unified telemetry planes, where metrics, logs, and traces are correlated under a common console and alerting model. AWS explicitly correlates CloudWatch metrics/logs with X-Ray traces in trace maps and ServiceLens-type views, and can route audit logs (CloudTrail) into the same log analytics plane [19]. Azure Monitor describes a shared data platform enabling correlation across data types and stores distributed traces within the same logs backend [29]. GCP integrates logging and monitoring so that incidents and alerts can be driven both by metrics and by log-based conditions [7].

A fourth pattern is topology-aware diagnosis. AWS X-Ray's service graph/trace map and CloudWatch application maps operationalize dependencies as first-class diagnostic objects, supporting causal investigation across microservices [21]. GCP exposes application/service topology views within its operations suite and strengthens diagnosis via systematic correlation of service metrics and traces.

A fifth pattern is adaptive detection to combat alert fatigue (Table 1) and threshold complexity. AWS provides metric outlier detection using machine learning (ML) models and anomaly-informed alarms. It also provides baseline-based anomaly detection over audit event rates (CloudTrail Insights) and log anomaly detection [24]. Azure provides dynamic thresholds (learning seasonal patterns from historical data) and smart detection alerts, reducing manual tuning for application telemetry [38]. GCP emphasizes incident creation and structured alert policy behavior. While not framed primarily as ML in core docs, it provides cost-aware alerting guidance and SLO-based service monitoring as a low-noise reliability signal [51].

Finally, provider-native solutions increasingly support observability-as-code: defining dashboards, alarms, and SLOs through APIs or declarative templates. AWS publishes infrastructure template resources for metric filters, anomaly detectors, and SLO objects, enabling version-controlled monitoring configuration [55]. Azure supports programmatic configuration of key alerting primitives such as action groups and encourages using platform APIs and templates for repeatable deployment [56]. GCP exposes alerting policy configuration via APIs and supports inspecting JSON representations of policies, enabling automation and consistency across environments [57].

These patterns also carry trade-offs relevant to provider-native selection for PaaS monitoring systems. Provider-native telemetry is highly integrated and operationally efficient (low setup (Table 1) for managed services), but coupling telemetry formats, dashboards, and alerting semantics to a provider's ecosystem can increase switching costs (vendor lock-in (Table 1)). Conversely, the tight integration of audit/compliance capabilities (e.g., mandatory audit logs, encryption and IAM coupling) can be a decisive advantage for regulated PaaS workloads. Cost is structurally usage-based across

providers, so monitoring architectures that do not explicitly encode sampling, aggregation, and retention controls risk becoming "unbounded cost centers."

3. Evaluation of open-source monitoring systems

The evaluation of open-source monitoring systems was based on five functional categories required in a complete PaaS observability stack: metrics monitoring, visualization, log monitoring, distributed tracing, and alerting/incident response. These categories were selected because they represent the main functional areas needed to observe, diagnose, and operate cloud-native platforms. Metrics provide quantitative information about system behavior, logs give detailed event-level context, traces make it possible to analyze request paths across distributed services, visualization supports interpretation of collected telemetry, and alerting transforms monitoring data into operational actions.

On this basis, evaluation criteria were defined for each functional category. The criteria describe both technical capabilities and operational requirements, such as integration with cloud-native environments, scalability, high availability, multi-tenancy, interoperability, maintainability, and support for declarative configuration. For every category, monitoring systems that provide the corresponding functionality were identified and compared against the defined criteria. This made it possible to assess not only whether a given system supports a specific function, but also how mature and suitable this support is for PaaS environments.

The assessment was performed using a Mean Opinion Score (Table 1) (MOS)-style scale from 1 to 5. In this work, MOS is used as an ordinal expert-based scoring method that allows heterogeneous tools and criteria to be compared in a consistent way. A score of 1 means that the evaluated functionality is missing, immature, or not practically usable in the analyzed context. A score of 2 indicates partial support, where the functionality exists but requires significant manual configuration, external components, or operational workarounds. A score of 3 represents acceptable basic support that can be used in practice, although with visible limitations. A score of 4 means that the functionality is well supported, mature, and suitable for production use in most PaaS scenarios. A score of 5 indicates strong, well-integrated, and production-ready support, with good alignment to the requirements of scalable and maintainable PaaS monitoring.

All criteria within a given functional category were treated with equal weight. For each functional category, the criteria were derived from professional monitoring patterns identified in commercial cloud solutions, such as AWS, Azure, and Google Cloud. These commercial platforms were treated as reference models because they represent mature, production-grade approaches to monitoring, observability, alerting, and operational governance.

For each evaluated open-source tool, every criterion was assessed by comparing the tool's documented capabilities with the corresponding capabilities observed in the reference models. A score of 5 was assigned when the evaluated tool provided functionality comparable to professional solutions and covered the criterion in a mature, integrated, and production-ready way. Lower scores were assigned when only part of the expected functionality was available, when additional components were required, or when the support was limited in terms of scalability, integration, automation, or operational maturity.

Thus, the scores should be understood as functionality coverage coefficients mapped to a 1–5 scale. For a given criterion, the score was determined proportionally to the ratio of fulfilled requirements to all requirements expected for that criterion. For example, if a tool fulfilled approximately 80% of the relevant functional requirements, the assigned score was 4. In general, the score can be interpreted as: *score* $\approx 5 \times$ *functionality coverage ratio* with the result mapped to the 1–5 MOS-style scale. A score of 1 indicates missing or very limited support, while a score of 5 indicates coverage close to the reference professional solutions.

The average score (Table 1) for each tool in a given category was then calculated as the arithmetic mean of the criterion-level scores. All criteria within a category were treated with equal weight. This approach makes the evaluation transparent and repeatable; the final score is not a single arbitrary opinion, but a structured estimate of how completely a tool covers the required functionality compared with professional monitoring systems. The global assessment made it possible to compare the overall suitability of the systems and to indicate which tools fulfill the required functional areas of a PaaS monitoring architecture.

The proposed scoring approach can be interpreted as a simplified multi-criteria evaluation method. Similar multi-criteria decision-analysis approaches have been used for cloud service selection, where alternatives are compared against several technical and operational criteria. In this paper, the same general principle is applied to open-source monitoring tools, with criteria derived from professional monitoring patterns and PaaS observability requirements [58].

3.1. Metrics monitoring

3.1.1 Category overview

Metrics are numeric time-series measurements that serve as the primary signal for detection, trend analysis, and SLO/SLA tracking. They typically encompass infrastructure metrics (CPU, memory, disk, network) for host/node capacity and saturation, application metrics (latency, throughput, error rate) for service behavior and user impact, and custom or derived metrics such as ratios and burn-rate indicators (Table 1) for platform-specific control loops and reliability objectives. From a Functionality perspective, metrics provide the foundation for monitoring coverage and alerting; rule-based, SLO-based, and event-based alerts convert metric telemetry into actionable notifications. Commercial providers consistently reflect this. Azure Monitor defines metrics as numeric data collected into a time-series database and treats them as the primary input to alerting and SLO tracking [6], while GCP Cloud Monitoring automatically collects performance metrics for most managed services and supports multi-month retention for user-defined metrics to enable long-horizon SLO evaluation [43]. From a Responsibility perspective, multi-tenancy support, cost controls, and declarative configuration are essential for operating the metrics platform sustainably at scale.

3.1.2 Selection criteria

The following criteria define the capabilities required to replicate and extend commercial metrics patterns with an open-source backend.

Multi-dimensional data model: evaluates whether the candidate can represent metrics as time series with labeled or attributed dimensions that enable segmentation by service, tenant, environment, and region while preserving aggregation correctness at scale. A mature data model is crucial in PaaS because workloads are elastic and multi-instance.

SLI/SLO query expressiveness: measures how directly the metrics query layer supports computing ratio-based service-level indicators, error budget burn rates, and latency percentiles natively. AWS CloudWatch Application Signals and GCP Cloud Monitoring both expose SLOs as first-class configuration objects [10, 11].

Infrastructure and application metrics coverage: assesses native availability of infrastructure signals alongside application-level signals, without requiring heavy custom instrumentation. All three major cloud providers treat infrastructure coverage as a mandatory baseline collected automatically [16, 18].

Custom and derived metrics: evaluates support for recording rules or computed metrics-ratios, aggregations, and SLO burn-rate indicators (Table 1) derived from raw signals without requiring an external processing layer. Derived metrics are essential for SLO-oriented operations because user-facing reliability is rarely captured by a single raw counter.

Cloud-native service discovery: evaluates how well metrics collection integrates with dynamic deployment environments typical for PaaS, where instances and endpoints are created and removed automatically. Kubernetes-oriented platforms drive discovery through label selectors and stable service abstractions.

OpenTelemetry and OpenTelemetry protocol ingestion: measures the ability to receive metrics via OpenTelemetry protocol (OTLP) or to integrate natively with an OpenTelemetry Collector, enabling vendor-neutral instrumentation pipelines. A common export layer eliminates per-backend software development kit (SDK) coupling and simplifies multi-signal pipeline governance.

Horizontal scalability and high availability: evaluates the capacity to scale ingestion and query horizontally, handle cardinality growth, and avoid single points of failure in the metrics path. Monitoring components must remain resilient precisely during the outages they are intended to detect.

Multi-tenancy and cost controls: captures per-tenant isolation, cardinality limits, metric aggregation rules, and configurable retention to prevent unbounded storage growth in shared PaaS environments. Commercial providers structurally enforce this through usage-based pricing that incentivizes sampling, aggregation, and retention scoping decisions [18, 43].

Long-term retention and remote storage: evaluates support for tiered or external storage backends-such as remote write and read interfaces-enabling long-horizon SLO analysis. Azure Monitor stores platform and custom metrics for 93 days with explicit guidance to export or aggregate for longer-horizon analysis [6].

Declarative configuration ("as code"): measures whether all scrape targets, recording rules, and alert rules can be defined in version-controlled files or APIs and reproduced across environments without manual UI steps. Commercial solutions increasingly support observability-as-code through infrastructure template resources and API-driven configuration [55].

3.1.3 Evaluated tools

Prometheus [59] is a metrics monitoring system built around a pull-based scraping model, a label-based time-series data model, and PromQL, a query language designed for operational analysis. It stores scraped samples (Table 1) locally and evaluates rules over that

data, including rules that produce alerts [60]. It is difficult to match Prometheus's combination of ecosystem adoption (exporters, service discovery patterns, Kubernetes conventions), query expressiveness, and operational simplicity for single-cluster and moderately sized environments [60, 61]. Its principal constraints-single-server time-series database (TSDB) model and cardinality sensitivity-can be addressed via remote write/read interfaces and external long-term stores, though enabling remote write has operational cost [62]. VictoriaMetrics enters the metrics category as both a time-series database in its own right and as long-term remote storage for Prometheus. Its documentation explicitly positions it as a fast, cost-effective, scalable TSDB [63]. It can reduce pressure on Prometheus's local retention and enable longer historical analysis without changing the Prometheus collection model. The trade-off is architectural; introducing a second metrics store increases the number of interdependent operational components, such as runtimes, controllers, service meshes, collectors, storage backends, and alerting services. and changes operational failure modes [62, 63]. Zabbix covers metrics monitoring through a different philosophy: a highly integrated monitoring solution with broad data-gathering options including Simple Network Management Protocol (SNMP), Intelligent Platform Management Interface (IPMI), Java Management Extensions (JMX), VMware monitoring, agents, and proxy-based collection [64]. This breadth maps well to heterogeneous infrastructure estates where protocol coverage matters more than cloud-native label conventions. However, Zabbix's strengths can become weaknesses in cloud-native environments emphasizing ephemeral workloads and monitoring-as-code patterns. OpenTelemetry should be viewed as a metrics enabler rather than a metrics store; it defines and implements how metrics are produced, processed, and exported as one of several signals [65]. It becomes critical when evaluation emphasizes instrumentation consistency across services and portability across backends.

3.1.4 Evaluation results

The evaluation results are shown in Table 2. Prometheus remains the top metrics backend because it best matches the PaaS-oriented metrics-first operational model: a mature label-based metric model, an expressive query language for operational analysis and SLI/SLO derivation, and native rule evaluation as a consistent control point for detection [59–61]. These properties directly support professional monitoring patterns where metrics are the primary input for alerting and SLO tracking [6, 10, 11]. Prometheus scores particularly well on cloud-native discovery and metadata conventions, which are essential to approximate the agentless-by-default augmentation pattern in dynamic environments [16, 18, 60]. The evaluation also clarifies role separation; OpenTelemetry scores highest on OTLP alignment but is structurally weaker as a metrics backend because it does not natively provide the storage/query semantics expected from this pillar [65].

Table 2: MOS evaluation for metrics monitoring

Criterion	P	V	Z	O
Multi-dimensional data model	5	5	3	4
SLI/SLO query expressiveness	5	4	3	2
Infrastructure and application metrics coverage	5	4	4	2
Custom and derived metrics	5	4	3	3
Cloud-native service discovery	5	4	2	2
OpenTelemetry and OTLP ingestion	3	3	2	5
Horizontal scalability and high availability	4	5	4	4
Multi-tenancy and cost controls	3	4	3	2
Long-term retention and remote storage	4	5	3	1
Declarative configuration ("as code")	5	4	3	3
Avg	4.40	4.20	3.00	2.80

P - Prometheus | V - VictoriaMetrics | Z - Zabbix | O - OpenTelemetry

3.2. Visualization

3.2.1 Category overview

Visualization covers the presentation and exploration layer that turns telemetry into shared understanding across operators, developers, and stakeholders. From a Visibility perspective, this encompasses operational, executive/key performance indicator (KPI), infrastructure/capacity, and APM-oriented dashboards, as well as persona-specific views and single-pane-of-glass summaries. Service maps, topology maps, and service health summaries support impact assessment and dependency-aware troubleshooting. From a Functionality perspective, visualization enables cross-signal correlation workflows-engineers pivoting between correlated signals without leaving the main console, as operationalized in AWS CloudWatch ServiceLens [27], Azure Monitor's unified data platform [29], and GCP's integrated operations suite [51]. From a Responsibility perspective, role-based access control that scopes dashboard visibility by team, environment, or tenant is necessary to prevent cross-tenant data exposure. Both Azure Monitor and GCP Cloud Logging document RBAC (Table 1) and workspace access control as compliance mechanisms [39, 53]. Provisioning dashboards and data source configurations as version-controlled code aligns with provider emphasis on observability-as-code [55–57].

3.2.2 Selection criteria

Multi-signal dashboard composition: measures the ability to query and combine metrics, logs, and traces in a single dashboard using multiple data source plugins with flexible panel types. In PaaS, teams frequently need to pivot between platform-level views and tenant or application views without switching tools.

Cross-signal pivot workflows: evaluates whether the visualization layer supports navigation from a metric anomaly to correlated logs or traces using shared context such as labels, trace IDs, and time ranges, without switching tools. AWS explicitly operationalizes this through CloudWatch ServiceLens [27].

SLO and error budget visualization: assesses whether the visualization layer provides native panels or templates for displaying SLO compliance status, error budget burn rate, and availability trends per service or tenant. GCP Cloud Monitoring treats SLOs as first-class configuration objects with dedicated visualization surfaces [11].

Service and dependency topology views: evaluates support for service maps and dependency graphs derived from trace or metric topology. AWS X-Ray builds service graphs by merging nodes from all services participating in a trace [21, 27].

Golden signals dashboards and templating: measures support for parameterized, reusable dashboard templates covering latency, traffic, errors, and saturation applicable across services and environments. Template driven dashboards reduce the overhead of establishing consistent operational views across a multi-tenant PaaS platform.

Alert lifecycle surfacing: measures how well the visualization experience surfaces active alerts, silences, firing history, and ownership context to support rapid triage.

Multi-tenancy and RBAC (Table 1) for dashboards: evaluates role-based access control that scopes dashboard visibility by team, environment, or tenant, preventing cross-tenant data exposure. Both Azure Monitor and GCP Cloud Logging document RBAC (Table 1) and workspace access control as compliance mechanisms [39, 53].

Provisioning and "as code" configuration: measures whether dashboard definitions and data source configurations can be managed as version-controlled files, deployed reproducibly without manual UI editing. Azure supports programmatic configuration via platform APIs [56]; GCP exposes alerting policy configuration via APIs [57].

Data source plugin ecosystem: assesses the breadth and quality of supported backends–Prometheus, Loki, Jaeger, and others via native integrations or plugins. A rich plugin ecosystem allows unifying multiple specialized backends into shared dashboards and operator workflows.

Horizontal scalability under query load: evaluates the ability to scale rendering and query concurrency to support multiple teams, Network Operations Center (NOC) screens, and on-call dashboards simultaneously without degraded response times.

3.2.3 Evaluated tools

Grafana is the dominant visualization and exploration layer, explicitly designed to query, visualize, alert on, and explore metrics, logs, and traces wherever they are stored, with a large data source plugin model [66–69]. The practical significance is that Grafana can unify multiple specialized backends (Prometheus for metrics, Loki for logs, Jaeger for traces) into shared dashboards and operator workflows, reducing the human cost of multi-tool observability while preserving backend specialization. Grafana provides built-in support for Prometheus as a data source and includes explicit guidance for Jaeger data source configuration and provisioning [66–68]. Grafana's alerting subsystem supports creating and managing alert rules spanning multiple data sources [70]. Grafana's principal trade-offs cluster around operational governance-managing dashboards at scale, multi tenancy, and plugin sprawl-and licensing: Grafana Labs' core projects moved from Apache 2.0 to AGPLv3, which affects redistribution and network-use obligations [71]. Zabbix provides built-in visualization capabilities as part of its integrated suite, which can be attractive where a single vendor like UI is desired. Yet Grafana is typically the more flexible front end when many heterogeneous data sources must be queried side-by-side [64].

3.2.4 Evaluation results

Table 3: MOS evaluation for visualization

Criterion	G	Z
Multi-signal dashboard composition	5	3
Cross-signal pivot workflows	5	2
SLO and error budget visualization	4	3
Service and dependency topology views	4	2
Golden signals dashboards and templating	5	3
Alert lifecycle surfacing	4	3
Multi-tenancy and RBAC (Table 1) for dashboards	4	3
Provisioning and "as code" configuration	5	2
Data source plugin ecosystem	5	3
Horizontal scalability under query load	4	4
Avg	4.50	2.80
G - Grafana | Z - Zabbix		

The evaluation results are shown in Table 2. Grafana is selected as the visualization and operator workflow layer because it is explicitly designed to sit above specialized backends and unify investigation in a single interface through a broad data source model and dashboard com-

position features [66–68]. This aligns directly with the unified telemetry plane pattern observed in commercial monitoring [7, 27, 29]. Grafana scores strongly on multi-signal dashboards (single pane of glass across the stack), cross-signal pivot workflows (investigative navigation across data sources), and provisioning as code [55–57, 66]. In contrast, Zabbix provides visualization within an integrated suite, but the criteria weigh cross-signal correlation and plugin-backed heterogeneity more heavily-capabilities central to PaaS observability stacks assembled from specialized components [64, 66–68].

3.3. Log monitoring

3.3.1 Category overview

Logs are discrete event records used for investigation, audit, and detailed runtime context. Key capabilities include centralized collection and parsing, structured logging (e.g., JSON) with enrichment, search and retention, and log-based metrics and pattern detection. In PaaS, logs present characteristic risks: unbounded volume growth, inconsistent structure across teams, and potentially high cost of full-text indexing and long-term retention. GCP Cloud Logging defines its scope as storing, searching, analyzing, monitoring, and alerting on log data, with explicit retention configuration from 1 to 3650 days per log bucket (Table 1) [7, 46]; AWS CloudWatch Logs similarly centralizes logs with configurable retention and at-rest encryption (Table 1) [19, 20]. From a Responsibility perspective, per-tenant log stream separation enforced at the API and gateway level, configurable retention, and declarative pipeline configuration are essential governance mechanisms. GCP Cloud Logging enforces IAM-based access control as a compliance mechanism [53].

3.3.2 Selection criteria

Structured log ingestion and parsing: evaluates support for JSON and structured log formats, field extraction, label-based stream routing, and semantic normalization across heterogeneous workload sources. Treating logs as time-ordered event streams with stable semantic mapping determines whether logs become reliably queryable data rather than unstructured text parsed under incident pressure.

Log search and field-based filtering: measures efficient filtering by structured fields and label values, time-bounded search, and predictable scan performance under incident-time pressure. Query ergonomics and predictable performance under scan-heavy workloads are core selection requirements.

Log-based metrics and pattern detection: assesses the ability to derive metric counters from log events-such as error rate from log lines- and to surface anomalous log patterns without requiring manual query construction. AWS CloudWatch supports log anomaly detection that scans incoming log events at ingestion time [26].

Cross-signal correlation via trace IDs and labels: evaluates whether log entries can be linked to distributed traces via propagated trace IDs and to metrics via shared label context, enabling pivot workflows from alert to root-cause log evidence. This is the log layer's contribution to cross-signal correlation: enabling engineers to move quickly from symptom to evidence.

OpenTelemetry Collector compatibility: measures native support for receiving log streams from an OpenTelemetry Collector via OTLP, enabling a unified, vendor-agnostic ingestion pipeline across all telemetry signals. A Collector-compatible log backend can consolidate agent sprawl (Table 1), reduce per-backend configuration surface, and improve ingestion consistency across heterogeneous PaaS workloads.

Multi-tenancy and tenant isolation: assesses per-tenant log stream separation enforced at the API and gateway level, preventing cross-tenant log access in shared PaaS infrastructure. PaaS log backends must provide equivalent isolation primitives to the IAM-based access control documented in GCP Cloud Logging [53].

Horizontal scalability and storage efficiency: captures the ability to handle high ingestion rates and long retention without degradation or prohibitive overhead, using an object-storage-backed model or equivalent that scales ingest horizontally and keeps storage cost proportional to volume rather than index size.

Retention and cost controls per stream: evaluates configurable retention periods per log stream or tenant, with deletion and archiving policies that satisfy data governance requirements. GCP explicitly supports log bucket (Table 1) retention from 1 to 3650 days [46].

Label cardinality governance: evaluates operational controls-limits, guidelines, and tooling-to prevent high-cardinality label dimensions that degrade query performance and inflate index size. This is particularly important in log systems that index by label metadata rather than full-text content.

Declarative pipeline configuration: measures whether log routing, parsing rules, and retention policies can be expressed as version controlled configuration files, reproducible without manual UI intervention. Commercial platforms expose configuration via APIs and templates [38, 57].

3.3.3 Evaluated tools

Loki is a log aggregation system designed for horizontal scalability and multi-tenancy, explicitly inspired

by Prometheus but for logs. Critically, Loki is engineered to be cost-effective by indexing only metadata labels, not the full log contents, while storing compressed log data in chunks commonly in object storage [72–74]. This design dramatically reduces indexing cost and simplifies storage economics in high-volume environments. Loki's label model is also its primary risk surface; over-labeling or high-cardinality label dimensions can harm performance and cost, while underlabeling can make queries less precise [72–74]. Loki's documentation treats tenant isolation as first-class via a mandatory tenant identifier header in multi-tenant mode [72–74]. Zabbix can monitor log files as part of its integrated suite and can be a pragmatic fit where log monitoring means detecting patterns in local/system logs on hosts already managed by Zabbix agents. However, as a centralized log analytics backend with retention, multi-tenant query experience, and correlation with traces, Loki plus Grafana typically provides a more specialized and scalable log-analysis path [64]. OpenTelemetry is the most relevant complement in log monitoring because log backends are only as effective as the ingestion pipelines feeding them. The OpenTelemetry Collector is explicitly positioned as a vendor-agnostic way to receive, process, and export telemetry to multiple targets [65, 75].

3.3.4 Evaluation results

Table 4: MOS evaluation for log monitoring

Criterion	L	Z
Structured log ingestion and parsing	4	3
Log search and field-based filtering	4	2
Log-based metrics and pattern detection	4	3
Cross-signal correlation via trace IDs and labels	5	2
OpenTelemetry Collector compatibility	4	1
Multi-tenancy and tenant isolation	4	3
Horizontal scalability and storage efficiency	5	4
Retention and cost controls per stream	4	3
Label cardinality governance	3	2
Declarative pipeline configuration	5	2
Avg	4.20	2.50
L - Loki | Z - Zabbix		

The evaluation results are shown in Table 4. Loki is selected as the log backend because its storage/indexing design is aligned with the PaaS reality of high log volume and cost pressure: it indexes primarily label metadata and stores compressed log chunks in object storage, making cost growth more predictable when labeling is governed and consistent [72–74]. This directly mirrors professional monitoring patterns that treat log retention and data management as first-class governance concerns [19, 46]. The most important constraint captured in the scoring is label cardinality risk: uncontrolled high-cardinality labels can create excessive streams and index growth, degrading performance under incident load [72–74]. This evaluation outcome links back to the centralized monitoring with logical isolation pattern: multi-tenant log backends must enforce isolation and governance, because PaaS environments combine heterogeneous tenants and workloads in shared telemetry infrastructure [53, 72–74].

3.4. Distributed tracing

3.4.1 Category overview

Traces are end-to-end request path representations across distributed components, especially important in microservice-based PaaS environments. They support distributed tracing across services, service maps inferred from trace topology, and span attributes and sampling strategies for balancing fidelity and cost. Tracing becomes particularly valuable in PaaS because the abstraction of underlying infrastructure and the prevalence of microservice-style decomposition can make purely host-centric diagnosis insufficient; tracing reconstructs the causal path of a request even when components are ephemeral. AWS X-Ray implements distributed tracing by collecting request data, building service graphs from all services participating in a trace ID, and retaining trace data for 30 days with sampling controls [8, 21]. GCP Cloud Trace collects latency data near-real-time with a 30-day retention window [48]. From a Responsibility perspective, head-based and tail-based sampling strategies, storage backend flexibility, and declarative deployment configuration are essential cost-governance levers. From an Observability perspective, W3C Trace Context compatibility [76] and OpenTelemetry OTLP ingestion are critical to ensure interoperable context propagation across heterogeneous PaaS components.

3.4.2 Selection criteria

Trace and span data model completeness: evaluates whether the tracing solution supports a robust trace and span model: spans with timing, parent-child relationships, attributes, events, links, and status codes covering both synchronous and asynchronous request workflows. **End-to-end service map generation:** evaluates automatic derivation of service topology and dependency graphs from trace data. AWS X-Ray merges nodes from all services participating in a trace ID into a service graph, enabling topology-aware analysis of latency and dependencies [21, 27]. **W3C Trace Context and OpenTelemetry compatibility:** measures support for standard context propagation across language and runtime boundaries, including W3C Trace Context

headers [76] and OTLP ingestion, which prevents broken traces in heterogeneous PaaS components.

Cross-signal correlation via exemplars and trace IDs: measures whether individual traces can be linked to metric data points via exemplars and to log entries via propagated trace IDs, enabling investigation workflows such as jumping from a latency spike directly to an example trace that contributed to it.

Span-level performance analysis: evaluates support for critical path identification, slowest span detection, and latency hotspot ranking across a distributed request flow. User experience and incident symptoms are often dominated by tail latency rather than averages.

Head and tail-based sampling strategies: assesses whether the tracing pipeline offers effective mechanisms to control cost while preserving diagnostic value. AWS X-Ray retention is coupled to sampling controls [21], reinforcing that retention and sampling must be considered together as cost-governance levers.

Storage backend flexibility and retention control: measures the choice of scalable storage backends-such as Elasticsearch/ OpenSearch, Cassandra, or object storage-with configurable retention windows aligned to cost and compliance requirements.

Horizontal scalability of collector and query: evaluates independent horizontal scaling of the collector (ingestion) and query components to handle traffic bursts from autoscaling PaaS workloads without data loss.

Declarative deployment and configuration: assesses whether collector pipelines, sampling policies, and storage backend configuration can be defined as version-controlled files or Helm/operator manifests, reproducible across environments [38, 57].

Integrations with continuous integration and continuous delivery (CI/CD) and deploy markers: evaluates the ability to correlate trace latency changes with deployment events, enabling change correlation between deployment activity and performance impact in the trace timeline.

3.4.3 Evaluated tools

Jaeger is a distributed tracing platform designed for end-to-end visibility in microservice architectures, including topology/service dependency visualization and features such as adaptive sampling [77]. Its documentation highlights OpenTelemetry compatibility and multiple built-in storage backend options, including Elasticsearch/ OpenSearch, Cassandra, and local/single-node storage backends, plus extensibility through remote storage APIs [77]. Current Jaeger documentation explicitly recommends using the standard OpenTelemetry Collector in many deployments because it is likely needed to handle other telemetry types beyond traces [77]. Jaeger's challenges tend to appear at high scale and when aligning retention and sampling controls with cost-of-ownership objectives; storage backend choice and sampling strategy drive index/storage cost and query responsiveness [77]. OpenTelemetry is central in the tracing category because traces are only useful when context propagation and instrumentation are consistent. OpenTelemetry formalizes a multi-signal model (traces, metrics, logs) and defines OTLP, a stable specification for encoding and transporting these signals between sources, collectors, and backends [65, 75]. In practical evaluation terms, OpenTelemetry reduces lock-in risk and improves interoperability; the same instrumentation can feed Jaeger today and another backend later, subject to signal fidelity, sampling strategy, and semantic conventions.

The importance of comparing tracing tools is also reflected in the work of Janes et al., who provide a systematic comparison of open tracing tools according to their features, popularity, benefits, and issues. Their findings support the view that tracing tools differ not only in functionality, but also in operational trade-offs and adoption context [78].

3.4.4 Evaluation results

Table 5: MOS evaluation for distributed tracing

Criterion	J	O
Trace and span data model completeness	5	3
End-to-end service map generation	5	2
W3C Trace Context and OpenTelemetry compatibility	5	5
Cross-signal correlation via exemplars and trace IDs	4	4
Span-level performance analysis	4	2
Head and tail-based sampling strategies	5	5
Storage backend flexibility and retention control	4	2
Horizontal scalability of collector and query	4	5
Declarative deployment and configuration	4	5
Integrations with CI/CD and deploy markers	4	4
Avg	4.40	3.70
J - Jaeger | O - OpenTelemetry		

The evaluation results are shown in Table 4. Jaeger is selected as the tracing backend because it represents a complete operational tracing system: ingestion, query, visualization, topology/dependency representation, and flexible storage backend choices in architectures that scale collector and query responsibilities independently [77]. This aligns with the professional topology-aware diagnosis pattern, where service maps and dependency context become first-class diagnostic objects [21, 27]. The evaluation intentionally treats OpenTelemetry as a critical enabling layer rather than a direct replacement for a tracing backend: it standardizes instrumentation, context propagation, and OTLP-based transport, reduc-

ing coupling between application code and any specific tracing store [65, 75]. This is consistent with the unified telemetry plane and agentless default augmentation patterns; standardized collectors/pipelines provide a controlled point where telemetry can be processed, sampled, enriched, and exported coherently across signals [27, 29, 65, 75].

3.5. Alerting and incident response

3.5.1 Category overview

Alerting and incident response cover how telemetry is converted into operational action: detection, notification, routing, and noise control. From a Functionality perspective, rule-based, SLO-based, and event-based alerts convert telemetry into actionable notifications; noise reduction (deduplication, grouping, suppression, maintenance windows (Table 1)) is critical to prevent alert fatigue (Table 1); and routing/escalation mechanisms should align alerts to responsible teams and on-call schedules. From an Incident Response Workflow perspective, monitoring platforms increasingly support incident lifecycle management (acknowledge, mitigate, resolve) with runbook/ playbook integration. All three major cloud providers integrate alerting tightly with incident management workflows; Azure Monitor structures alerting around action groups that can notify and trigger automation including runbooks and functions, and supports dynamic thresholds [38, 56], GCP Cloud Monitoring creates incidents when alert conditions are met and supports log-based alerting policies that tie Cloud Logging detection into the Cloud Monitoring incident model [51, 52], while AWS CloudWatch alarms can create incidents in Systems Manager Incident Manager, linking detection directly to operational workflows [14, 28].

3.5.2 Selection criteria

SLO-based and multi-window alerting: evaluates native support for error budget burn rate rules and multi-window evaluation-for example, evaluating alert conditions simultaneously over a 1-hour and a 6-hour window- targeting user-impacting events rather than transient threshold breaches.

Noise reduction mechanisms: evaluates first-class support for deduplication, inhibition (suppressing downstream alerts when a higher-level root-cause alert is already firing), grouping of related alerts, silencing (Table 1), and maintenance windows. Azure Monitor's dynamic thresholds and smart detection alerts specifically reduce manual threshold tuning to combat alert fatigue (Table 1) [38, 56].

Tag and label-based alert routing: measures routing of alerts to the correct team or on-call rotation based on service, environment, and tenant labels without manual triage intervention.

On-call escalation and scheduling integration: assesses integration with paging and scheduling tools-such as PagerDuty and OpsGenie-for escalation policies, on-call rotation handoffs, and severity-based paging. The notification path must be treated as a critical production subsystem.

Incident lifecycle management: evaluates support for an acknowledge-mitigate-resolve workflow, including incident creation, status tracking, and integration with Information Technology Service Management (ITSM) or war-room tooling (Table 1). AWS Systems Manager Incident Manager and GCP Cloud Monitoring incidents exemplify this pattern [14, 28, 51].

Notification channel breadth and delivery reliability: measures coverage of email, chat platforms (Slack, Teams), webhook, and paging channels with reliable delivery semantics. GCP Cloud Monitoring supports multiple notification channels [51]. Azure Monitor action groups can notify and trigger automation across a comparable set of channels [38, 56].

Automated remediation hooks: measures whether webhook or API integration enables controlled auto-remediation actions, such as restart, scale, or rollback-with traceability and human approval gates. Azure Monitor action groups can trigger runbooks and functions [38, 56]. AWS CloudWatch alarms can trigger Systems Manager Automation runbooks [14, 28].

Cross-signal alert composition: assesses the ability to define alert rules spanning metrics and logs from a unified rule plane, avoiding siloed alert ownership. GCP integrates logging and monitoring so that incidents and alerts can be driven both by metrics and by log-based conditions [7, 52].

Declarative alert policy configuration: evaluates whether alert rules, routing trees, and inhibition rules can be expressed as version controlled configuration files, reviewed and deployed as code. Commercial solutions expose alerting policy configuration via APIs and declarative templates [55–57].

High availability of the alert delivery path: evaluates HA clustering or peer replication of the alerting tier to ensure alert delivery continues precisely during the infrastructure outages that trigger the most alerts. The alerting subsystem must remain operational during the exact failure conditions it is designed to detect.

3.5.3 Evaluation results

The evaluation results are shown in Table 5. Alertmanager is selected because it cleanly separates detection

Table 6: MOS evaluation for alerting and incident response

Criterion	A	G	Z
SLO-based and multi-window alerting	4	4	3
Noise reduction mechanisms	5	3	3
Tag and label-based alert routing	5	4	3
On-call escalation and scheduling integration	5	4	3
Incident lifecycle management	3	3	3
Notification channel breadth and delivery reliability	5	4	3
Automated remediation hooks	4	4	3
Cross-signal alert composition	3	5	3
Declarative alert policy configuration	5	4	3
High availability of the alert delivery path	4	4	4
Avg	4.30	3.90	3.10
A - Alertmanager | G - Grafana | Z - Zabbix			

logic (rule evaluation upstream at the metrics layer) from the incident-facing responsibilities that determine on-call effectiveness: grouping, deduplication, inhibition, silencing (Table 1), and routing are first-class primitives but does not itself evaluate complex multi-signal conditions across metrics, logs, and traces; this requires upstream systems such as Prometheus, Loki/Grafana, or other rule engines. [79, 80]. Under PaaS operational conditions-multi-tenant noise, shared dependencies, and frequent topology change-these noise-control semantics are central to maintaining trust in alerting and preventing alert fatigue (Table 1). This result connects to the professional monitoring pattern of adaptive detection and noise control: the core operational objective is reducing non-actionable alerts while preserving timely detection of user-impacting issues [24, 26, 38, 56]. The evaluated open source alerting pipeline (Prometheus rule evaluation + Alertmanager lifecycle management) achieves this via robust suppression and routing semantics, and fits naturally into observability-as-code workflows [38, 80]. Grafana alerting scores strongly on cross-signal alert composition but its governance model can become more complex when organizations split ownership between metrics-native rule evaluation and dashboard-centric alert definitions [69, 70, 79].

4. General monitoring framework

4.1. Cross-cutting criteria

Several criteria apply across all categories because they determine operational sustainability regardless of signal type. Their engineering implications differ by category (e.g., scalability for log storage is ingestion/index-dominated, whereas scalability for visualization is primarily a concurrency and query-fanout concern), so they are evaluated separately within each pillar. Scalability and high availability: evaluates whether a component can grow with workload volume and organizational adoption while remaining resilient to failures. Prometheus is strong as a per-environment/per-cluster metrics engine, but long retention and global query often require extensions such as remote storage (e.g., VictoriaMetrics) or HA/long-term systems like Thanos [62, 63]. Loki is designed for horizontal scalability and multi-tenancy, but label governance is a scalability prerequisite [72–74]. Jaeger's scalability is heavily influenced by chosen storage backend and sampling strategy [77]. Zabbix scales via server/proxy/agent patterns but can become operationally heavy for highly ephemeral workloads [64]. Integrations ecosystem and interoperability: assesses how well the component interoperates with adjacent layers via stable interfaces, protocols, and conventions. Grafana's data source model and built-in integrations make it a strong front end for heterogeneous telemetry backends [66–69]. OpenTelemetry and OTLP enable a standardized export layer that reduces vendor/tool lock-in and promotes consistent context propagation [65, 75]. Prometheus's ecosystem density (exporters, conventions, integrations) remains a major differentiator in metrics monitoring [60, 61]. Configuration complexity and "as code" maintainability: measures whether configuration can be expressed declaratively, versioned, reviewed, and reproduced across environments rather than being an accumulation of manual UI steps. The best-of-breed stack (Prometheus + Loki + Jaeger + Alertmanager + Grafana) increases component count, so configuration management, upgrades, backups, and runbooks require notable operational focus. Prometheus configuration is file- and flag-driven and tends to be automation-friendly [62]. Grafana supports provisioning patterns for data sources and operational resources [66–68]. OpenTelemetry Collector can reduce agent sprawl (Table 1) by consolidating ingestion and export [65, 75]. Zabbix reduces component diversity but introduces its own operational complexity as an all-in-one suite [64]. Resource overhead: evaluates the compute, memory, and storage cost introduced by the component at typical and peak loads. Overhead matters in PaaS because monitoring is itself a platform service competing for shared capacity. Loki's index-by-label design is explicitly intended to lower storage/indexing cost but requires discipline in labels [72–74]. Prometheus's remote write guidance highlights resource overhead trade-offs when pushing metrics to long-term storage [62]. Documentation quality: captures whether official documentation enables correct installation, configuration, scaling, and troubleshooting without relying on tribal knowledge. For open-source monitoring components deployed as platform services, documentation

quality is a practical proxy for implementation risk. Community maturity: evaluates project sustainability signals such as governance stability, contribution activity, and evidence of production adoption. Cloud Native Computing Foundation (CNCF) notes that graduated and incubating projects are used successfully in production, and both Prometheus and Jaeger are CNCF graduated, which is a useful maturity signal [61]. Zabbix maintains commercial support offerings alongside open-source distribution [64].

4.2. Final proposed toolchain composition

Based on the highest average MOS scores in each functional category, the proposed stack is composed of the resulting integrated stack can be expressed as:

- Metrics backend: Prometheus
- Visualization / operator workflow layer: Grafana
- Log backend: Loki
- Tracing backend: Jaeger
- Alert routing & incident notification: Alertmanager

The final architecture combines five specialized components, Prometheus, Loki, Jaeger, Alertmanager, and Grafana, to cover the three core telemetry signals (metrics, logs, traces) and to translate them into an operational workflow (visualization and incident response). Each tool focuses on a single responsibility boundary, which reduces functional overlap while still enabling end-to-end correlation during troubleshooting and on-call response.

4.3. Contributions and integrated toolchain

The central aim of this publication was to move open-source tool selection for PaaS monitoring from ad hoc preference to defensible engineering rationale. This was achieved by first clarifying the operational needs of PaaS monitoring—multi-signal observability (metrics, logs, traces), correlation across layers, and incident-oriented alerting—and then translating these needs into a category-based criteria taxonomy covering metrics monitoring, visualization, log monitoring, distributed tracing, and alerting/incident response, complemented by cross-cutting criteria such as scalability/HA, interoperability, configuration maintainability "as code," overhead, documentation quality, and community maturity. The evaluation method used a MOS-style 1-5 ordinal rubric to keep scoring interpretable and comparable across diverse criteria while remaining grounded in verifiable technical characteristics.

Applying this framework led to a clear recommended composition: Prometheus as the metrics backbone for SLI/SLO derivation and rule evaluation; Grafana as the unified exploration and dashboarding layer; Loki as the log backend optimized for scalable ingestion with controlled indexing; Jaeger as the tracing backend for end-to-end request visibility; and Alertmanager as the incident-facing routing and noise-reduction layer. Alongside these "winners," OpenTelemetry was positioned as a critical standardization layer for consistent signal production and transport, strengthening interoperability and future substitutability.

4.3.1 The proposed monitoring schema

The diagram shown in Figure 4 illustrates an intentionally simplified observability deployment with three layers (plus external notification channels in the fourth layer):

1. Application Layer: a representative set of workloads (Background Workers, API Service, Web Application) emits three telemetry streams.
2. Data Collection:
   (a) Trace data from all application components flows into Jaeger (distributed tracing).
   (b) Application logs from each component flow into Loki (log aggregation).
   (c) Metrics endpoints (metrics) are scraped by Prometheus (metrics collection).
3. Visualization & Alerting:
   (a) Grafana queries all three backends: it queries traces from Jaeger, logs from Loki, and metrics from Prometheus, unifying investigation in a single UI.
   (b) Prometheus fires alerts to Alertmanager, which then notifies external channels (Email, Slack, PagerDuty).

4.3.2 Data flow and integration

These four levels of the monitoring system must cooperate properly to ensure efficient and reliable operation of the whole architecture. In practice, this cooperation is achieved by integrating selected open-source tools and enabling structured data exchange between them. The application layer exposes observability data through instrumentation: metrics can be published on HTTP /metrics endpoints and periodically scraped by Prometheus [59], logs can be written to standard output, files, or log streams and then forwarded to Loki using agents such as Promtail, Fluent Bit, or the OpenTelemetry Collector, while trace data can be generated by instrumented services and sent to Jaeger using OpenTelemetry SDKs, collectors, or compatible tracing protocols. The data collection layer stores each telemetry type in a dedicated backend and makes it available through query interfaces. Grafana then communicates with Prometheus,

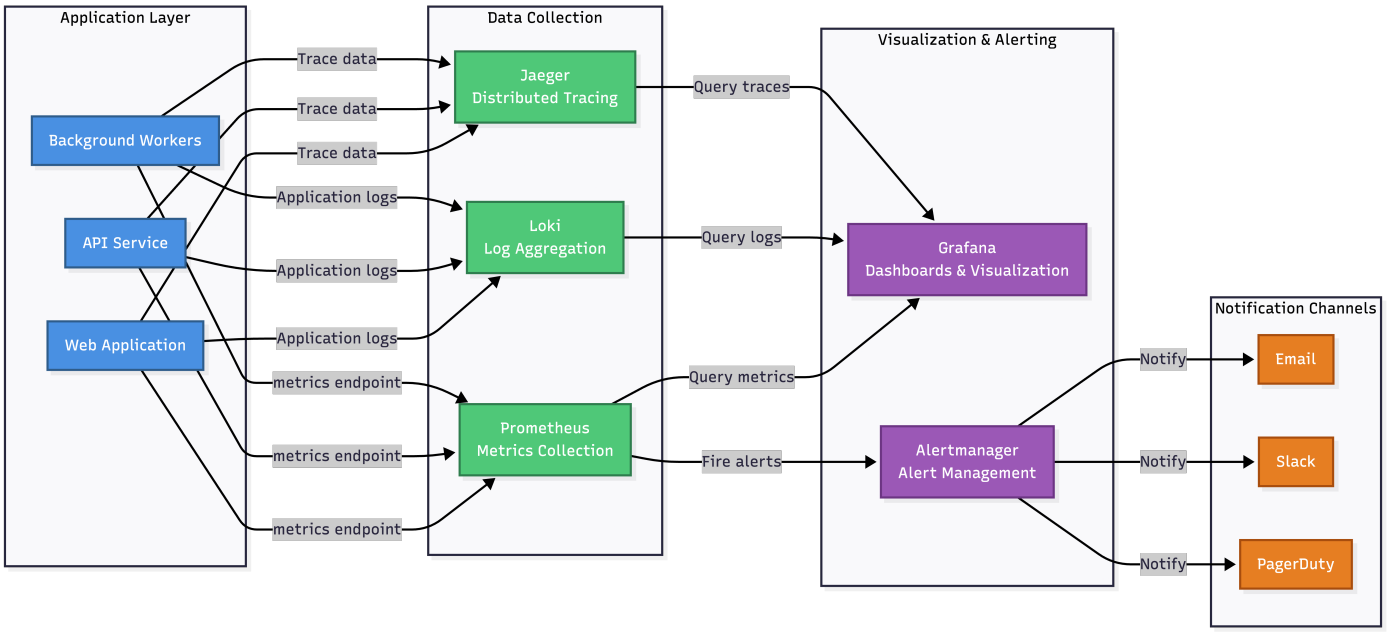

Figure 4: Diagram presenting general cloud monitoring schema

Loki, and Jaeger through their APIs and data source integrations, allowing users to visualize metrics, search logs, and inspect distributed traces in one place. Alert exchange is handled mainly between Prometheus and Alertmanager. Prometheus evaluates alerting rules based on collected metrics and sends firing alerts to Alertmanager, which groups, deduplicates, silences, and routes them according to configured policies. Finally, Alertmanager exchanges data with notification channels through webhooks, email protocols, or service-specific integrations, delivering alerts to tools such as Email, Slack, or PagerDuty. As a result, the architecture depends not only on individual monitoring tools, but also on consistent metadata, labels, timestamps, service names, and trace identifiers that allow telemetry to be correlated across layers.

5. Final remarks

This publication has addressed the practical challenge of selecting an open-source observability stack suitable for Platform as a Service environments, where elasticity, multi-tenancy, and layered dependencies make troubleshooting inherently cross-cutting. It introduces a structured, criteria-driven method that normalizes heterogeneous requirements through a MOS-style scoring model, and it applies that method to a curated toolset to derive a coherent, minimally overlapping "best-fit" toolchain. The resulting proposal is technically grounded, operationally oriented, and intentionally designed to be reproducible and adaptable as platform constraints evolve.

5.1. Implications for PaaS monitoring practice

A key implication of the proposed approach is architectural clarity: by selecting one primary component per observability pillar, the stack achieves full coverage while limiting redundant feature overlap that often increases operational burden. Equally important, the publication frames monitoring as an operational workflow rather than a data lake. The simplified reference architecture explicitly links telemetry capture (metrics scraping, log ingestion, trace collection) to a single investigative surface and to an incident-response path that is designed to reduce alert fatigue (Table 1) through grouping, deduplication, inhibition, and silencing (Table 1). This promotes faster diagnosis under continuous platform change, especially when correlation is treated as a first-class property rather than an afterthought.

5.2. Limitations and scope boundaries

At the same time, the publication's results must be interpreted as a well-justified selection baseline, not a final proof of fitness under all production conditions. The MOS rubric provides transparency and comparability, but it remains an ordinal, criteria-based assessment that can embed implicit judgment, particularly where "maturity" and "operability" depend on organizational experience. Equal weighting was adopted to keep the first version replicable; however, this assumption can underrepresent platform-specific priorities (for example, cost sensitivity, extreme burstiness, or stringent data residency requirements). Finally, the evaluation is necessarily constrained by what can be inferred from documented capabilities

and architectural characteristics; it cannot fully predict emergent behaviors such as cardinality-driven instability, query fan-out bottlenecks, noisy alert dynamics, or the real operational cost of upgrades and governance at scale.

5.3. The need for experimental testing: use method in Cloud Artificial Intelligence Service Engineering platform

The CAISE platform is used in this paper as the intended environment for further validation of the proposed monitoring framework. Its architecture and operational requirements make it a suitable case study for verifying whether the selected open-source toolchain can support practical PaaS monitoring needs in a real platform context. [81]. To convert the proposed toolchain from a logically consistent design into an evidence-backed operational recommendation, practical validation should be performed directly on the CAISE platform with representative PaaS workloads and realistic failure modes. The primary objectives should include: (1) verifying end-to-end correlation workflows (metric anomaly -> scoped logs -> trace-level root cause) under time pressure; (2) measuring performance and scalability limits across ingestion, storage, and query paths; and (3) assessing alert quality and incident workflow effectiveness in real on-call conditions. A robust experimental design would combine controlled load scenarios (steady-state, burst traffic, tenant spikes), topology change scenarios (autoscaling, rolling deployments, ephemeral instances), and fault-injection scenarios (dependency latency, partial outages, misconfiguration, log storms). Metrics should include at minimum: telemetry completeness (sampling impact, trace continuity), time-to-detection and time-to-diagnosis proxies, query latency percentiles, ingestion throughput, storage growth rates, and resource overhead on both collectors and backends; alerting should be assessed through precision/recall-oriented indicators (false positives, missed incidents), deduplication effectiveness, and operator-perceived actionability. Sample sizes need not be fixed a priori, but each scenario should be repeated enough times to characterize variance and to separate transient noise from systematic bottlenecks. Expected outcomes include identifying the operational "breaking points" (e.g., label cardinality in logs, retention/sampling trade-offs in tracing, dashboard/query fan-out under concurrency) and validating whether the chosen separation of responsibilities truly reduces toil. Key risks include uncontrolled cost growth from high-volume tenants, emergent coupling between components through shared metadata conventions, and security/governance gaps in multi-tenant access patterns. The results should feed back into refining the selection method itself (e.g., introducing weight profiles for CAISE-specific priorities, tightening criteria definitions) and into refining the practical reference architecture (recommended defaults, guardrails, and runbooks for day-2 operations).

5.4. Closing outlook

In summary, this publication delivers a defensible selection methodology and a cohesive, best-of-breed open-source monitoring architecture for PaaS, while correctly acknowledging that operational reality ultimately validates architectural intent. The most valuable next step is to institutionalize experimental validation on CAISE, using measured outcomes to harden both the proposed toolchain and the selection framework into a repeatable, platform-specific engineering standard. This concludes the publication.

5.5. Acknowledgements

The research was supported [in part] by project "Cloud Artificial Intelligence Service Engineering (CAISE) platform to create universal and smart services for various application areas", No. KPOD.05.10-IW.10-0005/24, as part of the European IPCEI-CIS program, financed by NRRP (National Recovery and Resilience Plan) funds.

References

[1] *https://csrc.nist.gov/glossary/term/Platform_as_a_Service*, 05.02.2026.

[2] O. Tomarchio, D. Calcaterra, and G. D. Modica, *Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks.* Journal of Cloud Computing, 2020.

[3] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, *Cloud monitoring: A survey.* Computer Networks, 2013.

[4] J. Kosińska, B. Baliś, M. Konieczny, M. Malawski, and S. Zieliński, *Toward the Observability of Cloud-Native Applications: The Overview of the State-of-the-Art.* IEEE Access, 2023.

[5] *https://learn.microsoft.com/en-us/azure/azure-monitor/*, 10.02.2026.

[6] *https://learn.microsoft.com/en-us/azure/azure-monitor/metrics/data-platform-metrics*, 10.02.2026.

[7] *https://docs.cloud.google.com/logging/docs*, 10.02.2026.

[8] *https://docs.aws.amazon.com/xray/latest/devguide/aws-xray.html*, 10.02.2026.

[9] *https://docs.aws.amazon.com/awscloudtrail/latest/userguide/view-cloudtrail-events.html*, 10.02.2026.

[10] *https://docs.cloud.google.com/stackdriver/docs/solutions/slo-monitoring*, 10.02.2026.

[11] *https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-ServiceLevelObjectives.html*, 10.02.2026.

[12] K. Alhamazani, R. Ranjan, K. Mitra, F. Rabhi, P. P. Jayaraman, S. U. Khan, A. Guabtni, and V. Bhatnagar, *An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art.* Computing, 2015.

[13] *https://docs.aws.amazon.com/images/xray/latest/devguide/images/xray-how-it-works.png*, 10.02.2026.

[14] *https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html*, 10.02.2026.

[15] P. Lakhera, *AWS for System Administrators: Build, automate, and manage your infrastructure on the most popular cloud platform.* Helion, 2023.

[16] *https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Install-CloudWatch-Agent.html*, 10.02.2026.

[17] *https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html*, 10.02.2026.

[18] *https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_concepts.html*, 10.02.2026.

[19] *https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html*, 10.02.2026.

[20] *https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/encrypt-log-data-kms.html*, 10.02.2026.

[21] *https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html*, 10.02.2026.

[22] *https://docs.aws.amazon.com/awscloudtrail/latest/userguide/send-cloudtrail-events-to-cloudwatch-logs.html*, 10.02.2026.

[23] *https://docs.aws.amazon.com/config/latest/developerguide/delete-config-data-with-retention-period.html*, 10.02.2026.

[24] *https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Anomaly_Detection.html*, 10.02.2026.

[25] *https://docs.aws.amazon.com/awscloudtrail/latest/userguide/logging-insights-events-with-cloudtrail.html*, 10.02.2026.

[26] *https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/LogsAnomalyDetection.html*, 10.02.2026.

[27] *https://docs.aws.amazon.com/xray/latest/devguide/xray-console-servicemap.html*, 10.02.2026.

[28] *https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/AlarmThatSendsEmail.html*, 10.02.2026.

[29] *https://learn.microsoft.com/en-us/azure/azure-monitor/fundamentals/overview*, 10.02.2026.

[30] M. Toroman, *Hands-On Administration in Azure: implement, monitor, and manage important Azure services and components including Iaas and Paas.* Helion, 2020.

[31] *https://learn.microsoft.com/en-us/azure/azure-monitor/logs/data-platform-logs*, 10.02.2026.

[32] *https://learn.microsoft.com/en-us/azure/azure-monitor/app/application-insights-faq*, 10.02.2026.

[33] *https://learn.microsoft.com/en-us/azure/azure-monitor/app/metrics-overview*, 10.02.2026.

[34] *https://learn.microsoft.com/en-us/azure/app-service/monitor-app-service*, 10.02.2026.

[35] *https://learn.microsoft.com/en-us/azure/azure-functions/functions-monitoring*, 10.02.2026.

[36] *https://learn.microsoft.com/en-us/azure/network-watcher/network-watcher-overview*, 10.02.2026.

[37] *https://learn.microsoft.com/en-us/azure/service-health/overview*, 10.02.2026.

[38] *https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-overview*, 10.02.2026.

[39] *https://learn.microsoft.com/en-us/azure/azure-monitor/fundamentals/roles-permissions-security*, 10.02.2026.

[40] *https://learn.microsoft.com/en-us/azure/azure-monitor/logs/personal-data-mgmt*, 10.02.2026.

[41] R. Blum, *GCP in Action: A practical guide to building and deploying secure, scalable applications using Google Cloud Platform.* BPB Publications, 2025.

[42] *https://learn.microsoft.com/en-us/azure/azure-monitor/fundamentals/media/overview/overview-blowout-20230707-opt.svg#lightbox*, 10.02.2026.

[43] *https://docs.cloud.google.com/monitoring/compliance/data-at-rest*, 10.02.2026.

[44] *https://docs.cloud.google.com/monitoring/settings*, 10.02.2026.

[45] *https://docs.cloud.google.com/appengine/docs/standard/writing-application-logs*, 10.02.2026.

[46] *https://docs.cloud.google.com/logging/docs/buckets*, 10.02.2026.

[47] *https://docs.cloud.google.com/logging/docs/audit*, 10.02.2026.

[48] *https://docs.cloud.google.com/trace/docs*, 10.02.2026.

[49] *https://docs.cloud.google.com/profiler/docs*, 10.02.2026.

[50] *https://docs.cloud.google.com/run/docs/error-reporting*, 10.02.2026.

[51] *https://docs.cloud.google.com/monitoring/alerts*, 10.02.2026.

[52] *https://docs.cloud.google.com/logging/docs/alerting/log-based-alerts*, 10.02.2026.

[53] *https://docs.cloud.google.com/logging/docs/access-control*, 10.02.2026.

[54] https://desk.zoho.com/support/ImageDisplay?downloadType=uploadedFile&fileName=1744105660431.jpg&blockId=edbsnb646363f03dae4172cd8fd818c6d20a0648bd0bc3bc0ca5019\be7da856579a93&zgId=edbsn7119390a921fea32bd7af73a56e02333&mode=view, 10.02.2026.

[55] *https://docs.aws.amazon.com/AWSCloudFormation/latest/TemplateReference/aws-resource-logs-metricfilter.html*, 10.02.2026.

[56] *https://learn.microsoft.com/en-us/rest/api/monitor/action-groups?view=rest-monitor-2021-09-01&viewFallbackFrom=rest-monitor-2024-03-11*, 10.02.2026.

[57] *https://docs.cloud.google.com/monitoring/alerts/manage-alerts*, 10.02.2026.

[58] M. Whaiduzzaman, A. Gani, N. B. Anuar, M. Shiraz, M. N. Haque, and I. T. Haque, *Cloud Service Selection Using Multicriteria Decision Analysis.* Scientific World Journal, 2014.

[59] J. Pivotto and B. Brazil, *Prometheus: up & running : infrastructure and application performance monitoring.* Helion, 2023.

[60] *https://prometheus.io/docs/introduction/overview/*, 11.02.2026.

[61] *https://www.cncf.io/projects/prometheus/*, 11.02.2026.

[62] *https://prometheus.io/docs/prometheus/latest/storage*, 11.02.2026.

[63] *https://docs.victoriametrics.com/victoriametrics/cluster-victoriametrics*, 11.02.2026.

[64] *https://www.zabbix.com/documentation/current/en/manual/introduction/features*, 11.02.2026.

[65] *https://opentelemetry.io/docs*, 11.02.2026.

[66] *https://grafana.com/docs/grafana/latest*, 11.02.2026.

[67] *https://grafana.com/docs/grafana/latest/datasources*, 12.02.2026.

[68] *https://grafana.com/docs/grafana/latest/administration/provisioning*, 12.02.2026.

[69] E. Salituro, *Learn Grafana 10.x: A beginner's guide to practical data analytics, interactive dashboards, and observability.* Packt Publishing, 2023.

[70] *https://grafana.com/docs/grafana/latest/alerting*, 11.02.2026.

[71] *https://grafana.com/licensing*, 11.02.2026.

[72] *https://grafana.com/docs/loki/latest/get-started/overview*, 11.02.2026.

[73] *https://grafana.com/docs/loki/latest/configure/storage*, 12.02.2026.

[74] *https://grafana.com/docs/loki/latest/get-started/labels/cardinality*, 12.02.2026.

[75] *https://opentelemetry.io/docs/collector*, 12.02.2026.

[76] *https://www.w3.org/TR/trace-context/*, 30.03.2026.

[77] *https://www.jaegertracing.io/docs/2.15*, 11.02.2026.

[78] A. Janes, X. Li, and V. Lenarduzzi, *Open Tracing Tools: Overview and Critical Comparison.* Journal of Systems and Software, 2023.

[79] *https://prometheus.io/docs/alerting/latest/overview*, 11.02.2026.

[80] *https://prometheus.io/docs/alerting/latest/alertmanager*, 11.02.2026.

[81] H. Krawczyk, P. Orzechowski, and B. Wiszniewski, *A Cloud Platform for the Development of Resilient and Intelligent Services for the Public Sector.* INTL Journal of Electronics and Telecommunication, to be appeared 2026.