Minimalistic Monitoring in Cloud Environments: Platform-Agnostic Best Practices

By Aditya Saxena

Abstract

Modern cloud computing environments demand robust monitoring to ensure reliability, yet excessive metrics and tooling can overwhelm engineers with noise. This paper presents a minimalist, platform-agnostic approach to cloud monitoring that focuses on essential indicators while remaining applicable across AWS, Azure, and GCP. We concentrate on Infrastructure-as-a-Service (IaaS) scenarios and distinguish their monitoring needs from Platform-as-a-Service (PaaS). Key metrics – **Latency, Errors, Saturation, and Throughput (L.E.S.T.)** – are highlighted as the minimal **"golden signals"** required to assess system health (Google SRE monitoring ditributed system - sre golden signals). Strategies are discussed for determining reasonable upper and lower bounds for these metrics based on baselines and Service Level Objectives (SLOs). We detail methods for setting up alerts on metric breaches and for automating remediation (self-healing) wherever possible to reduce manual intervention. Both real-time monitoring for immediate incident response and historical analytics for trend analysis are covered, including how machine learning techniques can improve anomaly detection, reliability, and capacity forecasting. We review open-source, cloud-neutral monitoring stacks (Prometheus, Grafana, ELK) alongside commercial cross-platform solutions (e.g., Datadog, New Relic), and provide guidance for monitoring in virtual machine environments and containerized deployments (including Kubernetes). Throughout, we adopt the perspective of a Site Reliability Engineer (SRE) operating in a closed cloud system – leveraging cloud provider interfaces and automation to maintain service health. The paper is structured with an academic lens, including a literature survey of prior approaches, a methodology for minimalist monitoring design, real-world case examples

on major platforms, best practices and logical workflows for common issues, tool comparisons, and conclusions. The goal is to equip cloud engineers and SREs with a concise yet effective monitoring framework that ensures high reliability without unnecessary complexity.

Introduction

Monitoring and alerting are **indispensable for maintaining the performance, availability, and reliability** of cloud-based systems (Monitoring and Alerting in the Cloud: Best Practices for DevOps Teams | Cloud Native Daily). As organizations deploy applications on dynamic cloud infrastructure, they face the challenge of observing system behavior across ephemeral virtual machines, containers, and managed services. A **platform-agnostic monitoring strategy** is desirable to avoid siloed solutions for each provider – it allows a unified view of multi-cloud environments and prevents vendor lock-in (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons). At the same time, a minimalist approach is needed to focus engineers on the signals that matter most, avoiding alert fatigue from too many metrics or overly sensitive alarms (Google SRE monitoring ditributed system - sre golden signals) (Mastering the Four Golden Signals of SRE: Throughput, Latency, Error Rate, and Saturation | by Proyash Paban Sarma Borah | Mar, 2025 | Medium). This work proposes a **minimal yet sufficient set of monitoring parameters and practices** that can be applied uniformly across AWS, Microsoft Azure, and Google Cloud Platform (GCP) IaaS offerings, with adaptability to PaaS contexts.

Infrastructure-as-a-Service (IaaS) provides virtualized resources (VMs, networks, storage) that the cloud customer must monitor and manage. In an IaaS model, although the physical infrastructure is abstracted away, the user retains responsibility for the operating system and application stack. As noted by McDonald (2020), *"Within an IaaS environment, there's far more to manage. The agency doesn't 'own' the infrastructure yet is still responsible for its monitoring and management. Troubleshooting application*

issues is your responsibility, not the cloud provider's." (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter). In contrast, **Platform-as-a-Service (PaaS)** abstracts much of the runtime environment; the cloud provider manages the underlying servers and OS, while the user manages only their application. Consequently, monitoring needs differ: PaaS users focus on application-level metrics and rely on the platform for lower-level monitoring. As a SolarWinds guide explains, *"Within a PaaS environment, the development team manages the applications regardless of which organization (cloud provider or agency) is managing the infrastructure components."* (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter). In other words, PaaS shifts much of the infrastructure monitoring burden to the provider, leaving the user to monitor performance and reliability of their code (often via application performance monitoring and logs provided by the platform). We will clarify these differences and ensure that our recommended practices cover the necessary layers for each model.

A key concept in our minimalist philosophy is the identification of a **critical core of metrics** that provide maximum insight with minimal overhead. Google's SRE doctrine advocates the "Four Golden Signals" of monitoring – latency, traffic (throughput), errors, and saturation – as the highest-value metrics for any user-facing system (Google SRE monitoring ditributed system - sre golden signals). If one can only measure a few things, these are the recommended choices. By monitoring these four dimensions, an operator can capture the user experience (via latency and errors), the load on the system (throughput/demand), and the capacity or stress of the infrastructure (saturation/utilization). In this paper, we adopt these golden signals, referring to them

with the mnemonic **L.E.S.T.** (Latency, Errors, Saturation, Throughput). We discuss each of these metrics in depth in later sections, including how they reflect system health and how to establish **upper and lower bounds** for acceptable values. The notion of upper/lower bounds ties into **SLOs and thresholds** – for example, an SRE team might define an SLO that 99% of requests have latency below 200 ms (upper bound for latency), or that error rate should be virtually zero during normal operation (upper bound for errors is near zero, with any sustained error rate above, say, 1% being a concern). Lower bounds can also be relevant: e.g., a sudden drop in throughput to near zero during peak hours might indicate an outage or lost traffic, triggering an alert if below a certain floor.

Another focus is on **alerting and automated remediation**. Alerts are the mechanism by which monitoring systems notify engineers (or take action) when metrics indicate abnormal conditions. A well-tuned alerting system has *"good signal and very low noise"* (Google SRE monitoring ditributed system - sre golden signals) – it should page a human only when human intervention is truly needed (i.e., when a problem is urgent or cannot be auto-remediated). Wherever possible, if a condition can be detected and a corrective action scripted, an automated fix is preferable to waking up an on-call engineer in the middle of the night. Modern SRE practices heavily emphasize automation to reduce toil and mean-time-to-recovery: *"Automation plays a vital role in incident response. Tools like runbooks and auto-remediation scripts allow systems to recover from failures with minimal human intervention, reducing MTTR."* (The Role of Automation in SRE Success | by Mihir Popat | Medium). In line with this, we explore methods to automatically trigger scaling, healing, or other responses when certain metrics breach their thresholds. For

example, if CPU saturation stays high, an autoscaling policy might add another server instance; if a container in Kubernetes crashes, the orchestrator's liveness probe and restart policy constitute an automatic remedial action; if an application memory leak is detected (memory saturation growing steadily), one could automatically cycle the application or container at a safe interval. We will outline algorithmic approaches to implement such **self-healing systems**, as well as logical workflows for common issues (e.g., what steps to take when latency spikes or error rate increases).

The scope of our discussion spans both **real-time monitoring** and **historical analysis**. Real-time (or near real-time) monitoring is crucial for detecting incidents as they happen and enabling quick responses – this typically involves dashboards updating every few seconds to minutes and alerts that trigger within a short window of a threshold breach. On the other hand, historical monitoring involves storing metrics and logs over long periods to analyze trends, seasonality, and root causes of past incidents. Machine learning (ML) and statistical analysis can be applied to historical data to glean insights that improve reliability. For instance, ML-based anomaly detection can establish dynamic baselines so that alerts trigger on significant deviations from normal behavior rather than static thresholds (Anomaly Detection with Machine Learning: Techniques and Applications | DoiT). ML models can also forecast resource usage (e.g., predicting when a database will run out of disk in advance (Google SRE monitoring ditributed system - sre golden signals) or forecasting daily traffic peaks to schedule capacity accordingly). We will discuss how incorporating ML – often termed AIOps in the industry – can complement traditional monitoring. Notably, while Google's SRE teams caution against over-reliance on "magic" automated thresholding for real-time alerting (Google SRE monitoring

ditributed system - sre golden signals), they do acknowledge that **complex analyses are valuable for capacity planning and long-term trends** that tolerate occasional inaccuracies. This balanced view will inform our approach: keep real-time monitoring simple and robust, but leverage ML for advanced analytics that improve **predictive reliability**.

We also recognize that tool support is a fundamental aspect of any monitoring strategy. Therefore, this paper surveys the landscape of **monitoring tools**, both open-source and commercial, especially those that work across multiple cloud platforms. Open-source stacks such as **Prometheus** (for metrics collection) and **Grafana** (for visualization and alerting) have become de facto standards in cloud-native environments (Multi-Cloud Monitoring and Alerting with Prometheus and Grafana). The popularity of Prometheus+Grafana is due in part to their flexibility and community of integrations (exporters for many systems, Kubernetes support, etc.), and their neutrality (they can run on any cloud or on-premises, scraping metrics via standard protocols). We will illustrate how these tools can be deployed in a cloud-agnostic fashion – for example, running a Prometheus server to scrape metrics from AWS EC2 instances, Azure VMs, or GCP VMs alike, or using Grafana's cloud provider data source plugins to pull in CloudWatch/Azure Monitor metrics into one dashboard. Similarly, the **ELK stack** (Elasticsearch, Logstash, Kibana) is a widely used open-source solution for log aggregation and analysis that can be set up to collect logs from any environment (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons). We will discuss use of ELK (or its variant EFK with Fluentd/FluentBit) for consolidating logs across cloud services for deeper troubleshooting. In addition, we

mention emerging open standards like **OpenTelemetry**, which provides a vendor-neutral way to instrument applications for metrics, traces, and logs – facilitating consistent telemetry across heterogeneous platforms.

Alongside open tools, we review major **commercial monitoring solutions** that support multi-cloud environments. Vendors such as **Datadog** and **New Relic** offer unified monitoring platforms that can ingest data from AWS, Azure, GCP, and on-prem systems into a single pane of glass. For example, Datadog provides *"a comprehensive platform with real-time metrics, security, and log management"* that integrates with a wide array of services (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons), and New Relic similarly *"provides deep insights into application and infrastructure performance"* across environments (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons). These Software-as-a-Service monitoring tools often come with built-in machine learning features for anomaly detection (e.g., Datadog's Watchdog) and sophisticated dashboards and alerts out-of-the-box. We will compare the trade-offs of using such commercial offerings versus building on open-source tools, in terms of ease of use, flexibility, and cost. Additionally, we mention other notable solutions like Splunk (and its cloud observability suite) – which excels in log management with real-time analytics (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons) – and how cloud-native services (like AWS CloudWatch, Azure Monitor, GCP Cloud Monitoring) can be bridged into a multi-cloud strategy.

Finally, we include special consideration for **containerized environments and orchestration platforms** like **Kubernetes**. In modern cloud deployments, Kubernetes is

frequently used to manage containers across clusters of VMs, whether on self-managed clusters on IaaS or via managed services like EKS (AWS), AKS (Azure), or GKE (GCP). Kubernetes introduces an additional abstraction layer and a dynamic topology (pods starting and stopping, nodes joining and leaving) that requires a refined monitoring approach. We will outline best practices for Kubernetes monitoring, such as using the Kubernetes metrics server and API to gather cluster health statistics (e.g., API server latency, scheduler performance), using Prometheus with Kubernetes exporters (cAdvisor, node-exporter, etc.) to collect container and node metrics, and monitoring cluster-level indicators like etcd performance or control-plane availability. We also discuss logging and tracing in Kubernetes (e.g., aggregating pod logs, using tools like Jaeger for distributed tracing if needed). The ephemeral nature of containers means that **traditional host-based monitoring must evolve** – instead of identifying issues by static hostnames, one must track by service or workload, and leverage labels/metadata to aggregate metrics by service. Kubernetes itself has self-healing mechanisms (auto-restarting failed pods, rescheduling on healthy nodes) which we consider part of the automated remediation theme. We will ensure that our minimal metric set and alerting philosophy extends to these orchestrated environments (for example, monitoring pod resource usage to signal when to scale deployments or clusters).

The rest of this paper is organized as follows. In the **Literature Survey**, we review foundational principles and related work in cloud monitoring and SRE practices, including prior frameworks for minimal monitoring and multi-cloud visibility. The **Methodology** section then outlines a systematic approach for implementing a minimalist monitoring solution, from instrumenting the L.E.S.T. metrics to defining thresholds,

alerts, and automated actions. We provide **Case Examples** demonstrating these practices in AWS, Azure, and GCP environments (focusing on IaaS use cases for consistency, but noting PaaS differences). We then enumerate **Best Practices** distilled from the study, serving as a checklist for SREs and cloud engineers. A **Tool Comparison** is presented in tabular form to evaluate open-source and commercial tools against requirements like multi-cloud support, metric scope, and automation features. We conclude with a summary of findings and recommendations for implementing minimalist, reliable monitoring in the cloud. All assertions and recommendations are supported by references to SRE literature, cloud documentation, and industry case studies throughout the text.

Literature Survey

Monitoring in distributed systems and cloud infrastructure has been a topic of extensive discussion in both industry and academia. Early cloud adopters often carried over traditional datacenter monitoring approaches: tracking dozens of low-level system metrics (CPU, memory, disk, network) per server, using simple threshold alerts or SNMP traps, and manually correlating information during outages. Over time, as systems grew more complex and failures more nuanced, engineers recognized the need to focus on higher-level indicators that better capture the end-user experience and overall system health. This gave rise to **monitoring frameworks like the Four Golden Signals**, introduced in Google's **Site Reliability Engineering (SRE) book** as a concise set of metrics that every service should be monitored on (Google SRE monitoring ditributed system - sre golden signals). As quoted earlier, *"The four golden signals of monitoring are latency, traffic, errors, and saturation. If you can only measure four metrics of your user-facing system, focus on these four."* (Google SRE monitoring ditributed system - sre golden signals). This philosophy marks a shift from monitoring everything (which can lead to information overload) to monitoring the most telling symptoms. The Golden Signals framework has been widely embraced; for instance, Splunk's SRE guidance notes that these signals *"define what it means for the system to be 'healthy'"*, since they cover performance (latency), demand (traffic throughput), reliability (error rate), and capacity (saturation/utilization) (SRE Metrics: Core SRE Components, the Four Golden Signals ...). Many subsequent works, such as blogs by New Relic, Gremlin, and others, have further propagated the importance of these metrics in cloud observability strategies.

Another influential concept in monitoring minimalism is the **USE method (Utilization, Saturation, Errors)** by Brendan Gregg, which is more infrastructure-centric, and the **RED method (Rate, Errors, Duration)** which is aimed at microservices. These methods echo similar sentiments: they urge engineers to pick a small number of critical metrics for each resource or service (for example, for a given resource: how busy it is, how utilized its capacity, and how often it errors). In our context, these map closely to saturation (utilization and saturation in USE), errors (common to all), and throughput/latency (rate and duration in RED). The L.E.S.T. signals can be seen as a superset that covers these needs broadly for most systems. We note that while comprehensive **observability** might involve metrics, logs, and traces to diagnose deep issues, there is virtue in starting with **just a few key metrics** as the first line of defense. Multiple sources stress this: **Groundcover**, for example, describes the Golden Signals as *"a reduced set of metrics that offer a wide view of a service from the user's perspective"* (The four Golden Signals of Monitoring - Sysdig). By limiting initial focus to these, one can detect most common problems (like increased latency or error rates) and then leverage logs or traces only as needed to root cause the issue.

Literature also highlights the importance of **alert quality** over quantity. Google SREs warn against triggering alerts for every anomalous data point, advising that *"you should never trigger an alert simply because 'something seems a bit weird'"* (Google SRE monitoring ditributed system - sre golden signals). An alert should signify a problem that **requires action**; otherwise it adds to noise. The human cost of false alarms is well-documented – too many pages lead engineers to become desensitized or to ignore alerts, undermining the whole monitoring system (Google SRE monitoring ditributed system -

sre golden signals). Therefore, a recurrent theme is to carefully choose alert conditions that are strongly indicative of user-facing issues or impending critical failures. The Golden Signals guidance again provides a baseline here: *"If you measure all four golden signals and page a human when one signal is problematic (or, in the case of saturation, nearly problematic), your service will be at least decently covered by monitoring."* (Google SRE monitoring ditributed system - sre golden signals). This statement encapsulates both the selection of metrics and the philosophy of when to alert – e.g., one might alert on high latency only if it exceeds a threshold that threatens the user experience, or on saturation when a resource is near its limit and could soon cause errors.

Several studies and articles also discuss the differences in monitoring needs across **cloud service models (IaaS/PaaS/SaaS)**. A piece by LogicMonitor (2021) explains that with IaaS, since the users manage the OS and runtime, they often deploy their own monitoring agents or systems to gather OS-level metrics and application metrics, in addition to cloud provider metrics (SaaS vs. PaaS vs. IaaS: What's the Difference? | LogicMonitor). PaaS users, by contrast, lean on what the platform offers – e.g., Heroku or Azure App Service automatically provides metrics like request count, response time, and application logs, so the user's focus shifts to interpreting those and instrumenting custom app-level metrics if needed. SaaS (Software-as-a-Service) offers the least control; typically, one can monitor only via the SaaS application's provided APIs or export features, which may limit the visibility primarily to usage metrics and high-level performance indicators (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter). Our interest is mostly in IaaS (and containers on IaaS) because that is where engineers have the most responsibility to set up monitoring. However, we include PaaS in our survey to point out

which monitoring tasks it alleviates (e.g., no need to monitor OS disk space on Azure Functions, since Azure takes care of it; but you *do* need to monitor your function's error counts and execution duration). The literature suggests an **"application-centric approach"** in all cases (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter) – meaning regardless of model, ensure you are measuring the user-visible outcomes (latency, errors) in addition to any internal resource metrics.

Multi-cloud monitoring, where an organization uses more than one cloud provider, has emerged as a challenge identified in recent years' literature. **New Horizons (Taylor, 2024)** emphasizes that *"multi-cloud monitoring goes beyond simply maintaining operations—it's about ensuring data remains secure, accessible, and optimized across different cloud environments"* (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons). It points out difficulties like integrating different vendors' metrics, normalizing data, and avoiding visibility gaps. Best practices recommended include using **vendor-neutral tools** that can pull data via APIs from all clouds, and presenting them in a **unified dashboard** (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons). The same source underscores the need for **automating alerts and responses** in multi-cloud setups to handle issues that may arise in any environment (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons) (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons). The use of open standards (like OpenTelemetry for data collection and common data models) is often suggested as a way to reduce fragmentation. Our literature review did not find a one-size-fits-all solution, but it consistently highlights the trend of

centralizing monitoring for hybrid and multi-cloud, often via third-party services or open-source tools rather than relying on each cloud's native monitoring in isolation.

There is also growing literature on the application of **machine learning to IT operations (AIOps)**, which includes cloud monitoring. Academic research, such as an ensemble-based approach by Gupta et al. (2021), demonstrates improved detection of performance anomalies in cloud applications using AI techniques (Robust and accurate performance anomaly detection and prediction ...). Industry articles (e.g., by Gartner analysts or AIOps vendors) often discuss how ML can automatically baseline metrics for each service and detect deviations that might be missed by static thresholds. For example, an article by DoiT International (2025) notes that *"unlike traditional monitoring systems that rely on static thresholds, today's outlier detection capabilities leverage sophisticated machine learning algorithms to adapt to dynamic environments"* (Anomaly Detection with Machine Learning: Techniques and Applications | DoiT). This adaptability is crucial in cloud environments where "normal" can vary with time of day, deploys, or usage patterns. Similarly, Grafana's recent features for **"Forecasting and outlier detection"** use models (like Holt-Winters, etc.) to predict expected metric ranges and alert if actual values fall outside predicted bounds (Identify anomalies, outlier detection, forecasting - Grafana). Our survey found that while ML is highly promising for reducing false positives and catching subtle issues, it is most effective when combined with human-defined SLOs. In practice, many SRE teams use a mix: they maintain some simple alert rules for clear-cut violations (e.g. CPU > 90% for 5 minutes or error rate > 5%) and augment their monitoring with ML-based anomaly alerts as a secondary signal. We will

incorporate this hybrid perspective, citing literature that advocates a cautious but forward-looking integration of ML in monitoring pipelines.

Lastly, in the context of **Kubernetes and container monitoring**, literature (including blog posts by cloud providers and Kubernetes tool vendors) emphasizes the need for **observability built into the cluster**. The CNCF (Cloud Native Computing Foundation) community has produced standards and tools (Prometheus being a flagship) that align with Kubernetes. For example, Kubernetes documentation itself suggests using Prometheus and Grafana for cluster monitoring, and projects like the **kube-prometheus-stack** provide a ready-made bundle of Prometheus, Alertmanager, and Grafana with Kubernetes dashboards. A Logz.io article on K8s monitoring (2022) discusses best practices such as capturing metrics at multiple layers – cluster (e.g., API server latency), node (CPU, memory, disk usage), namespace, pod, and even application custom metrics – to get a full picture. They note that one should *"establish proactive alerting and incident response"* specific to Kubernetes constructs, for instance alert on CrashLoopBackoff events or on pod eviction rates which have no equivalent in non-container setups (Kubernetes Monitoring Best Practices). The complexity of Kubernetes has led to an emphasis on **monitoring the control plane** as well (something not present in traditional single-VM systems): e.g., etcd health, controller manager logs, and network overlay status. Literature also covers how Kubernetes' auto-scaling mechanisms (Horizontal Pod Autoscaler, Cluster Autoscaler) rely on metrics, and thus the monitoring system can feed not just humans but automation within the cluster. We will reference these sources to reinforce how our recommended metrics (L.E.S.T.) apply in Kubernetes – for example, *latency* might refer to service request latency (an app-level metric) but

also API server latency (a critical platform metric); *saturation* might refer to node CPU or memory pressure that could trigger the cluster autoscaler. In summary, the literature surveyed spans SRE best practices, multi-cloud strategy, AIOps, and container observability, all of which inform the methodology we propose next.

Methodology

In this section, we outline a step-by-step methodology for implementing a minimalist, effective monitoring regime in a cloud environment. The methodology is divided into several phases: **(1) Identifying Key Metrics (L.E.S.T.) and Instrumentation**, **(2) Establishing Baselines and Thresholds**, **(3) Configuring Alerts and Integration with Incident Response**, **(4) Automating Remediation Actions**, and **(5) Leveraging Historical Data and Machine Learning**. This approach is intended to be platform-agnostic, meaning the same high-level process can be applied whether one is using AWS, Azure, GCP or a combination thereof, with differences only in the tools or services used to collect the metrics.

1. Identify Key Metrics (Latency, Errors, Saturation, Throughput) and Instrumentation

The first step is to decide **what to monitor**. Based on the principles discussed, we choose the **L.E.S.T. metrics (Latency, Errors, Saturation, Throughput)** as our primary monitoring targets. Concretely, this means for each critical service or component in the cloud system, we will capture:

- **Latency** – the time taken to service requests or transactions. For web services, this could be HTTP response time (ideally at various percentiles like 50th, 95th, 99th percentile); for database systems, query execution time; for an event processing pipeline, end-to-end processing time. We instrument both *successful* request latency and *error* latency separately (Google SRE monitoring ditributed system - sre golden signals), because a fast error (e.g., immediate failure) and a

slow error (timeout) are both important and can have different causes. Latency directly affects user experience – high latency degrades perceived performance.

- **Errors** – the rate or count of errors/failures in the system. This includes explicit errors (e.g., HTTP 5xx status codes, exception counts, failed transactions) and implicit errors (functionally incorrect results). We gather metrics like error rate (errors per second or as a percentage of requests) for services, and perhaps system-level errors such as OS-level errors or Kubernetes pod crash loops. In cloud infrastructure, this might also include things like number of failed VM instance health checks or container restarts.

- **Saturation** – how utilized the system is relative to its capacity. This often maps to resource utilization: CPU usage percentage, memory usage (and memory pressure), disk space usage, disk I/O and network I/O rates, etc., depending on what resource is the likely bottleneck for the service (Google SRE monitoring ditributed system - sre golden signals). Each component might have a different primary resource to watch: e.g., for a CPU-bound application server, CPU utilization is key; for a database, perhaps disk I/O or memory. We identify those and ensure metrics are collected. Saturation also covers cues of approaching limits, such as queue lengths, thread pool usage, or as the SRE book suggests, even predictive metrics like "time until resource exhaustion" can be derived (Google SRE monitoring ditributed system - sre golden signals).

- **Throughput (Traffic)** – the incoming demand or throughput that the system is handling. For web apps, this is often requests per second or transactions per

second. For data pipelines, it could be messages per minute. This metric is crucial to interpret the other three: for example, latency or errors may spike only when throughput spikes (load-induced issues), and saturation is often a function of how much throughput the system is trying to handle (Google SRE monitoring ditributed system - sre golden signals). Throughput gives context – it helps answer "how busy is the system?".

To implement this, we leverage a combination of cloud provider **built-in metrics** and **custom instrumentation**. All major clouds provide a monitoring service (AWS CloudWatch, Azure Monitor, GCP Cloud Monitoring) that automatically collects certain metrics from their services. For IaaS VM instances, these include basics like CPU utilization, disk I/O, network I/O, etc. For platform services, they include things like request counts and error counts (e.g., AWS Elastic Load Balancer provides request count, latency, HTTP 4xx/5xx counts via CloudWatch). We will use these native metrics wherever available for consistency and low overhead. For example, on AWS we would enable detailed monitoring on EC2 instances to get 1-minute CPU and network metrics (Difference and relationship between CloudWatch "monitoring type ...), and on Azure we would use the Azure Monitor **VM Insights** which installs an agent to collect guest OS metrics and sends them to Azure Monitor (Getting Started with Azure Monitor: Key Features and Best Practices | DataCamp).

In addition, for application-specific latency and error metrics, we often need to instrument the application code or use an APM (Application Performance Monitoring) tool. For instance, using **OpenTelemetry SDK** in an application, we can measure request latency and emit custom metrics to a backend (which could be Prometheus or

CloudWatch via a collector). In PaaS scenarios, the platform might gather some of this: Azure's Application Insights automatically measures HTTP request duration and result codes for web apps (Getting Started with Azure Monitor: Key Features and Best Practices | DataCamp), and GCP's Cloud Run provides request latency distributions out of the box. Where possible, we will turn on these application monitoring features. Otherwise, we deploy a sidecar or agent (like the CloudWatch Agent or Azure Monitor Agent) that can scrape application logs for errors or expose an endpoint for Prometheus to poll metrics like latency histograms.

For containerized apps, **Prometheus** is a common choice to scrape metrics. We would expose metrics from apps (perhaps using the Prometheus client libraries in each app, which can easily track request counts, durations, error counts, etc.). Prometheus will also gather container-level and node-level stats via **exporters**: the *node-exporter* gives host CPU, memory, disk; *cAdvisor* (often integrated in Kubernetes or via the kubelet) provides container resource usage metrics. So in a Kubernetes cluster, simply installing the Prometheus Operator and the kube-prometheus stack can give you instant visibility into saturation metrics (resource usage) and you can add your app's latency and error metrics to that. This approach is platform-agnostic because the same stack can run on EKS, AKS, or GKE.

Thus, **Phase 1** yields a set of dashboards or metric streams for each key service: e.g., for a web service, a time-series for request latency (p95), a time-series for error rate, one for CPU/memory utilization, and one for request throughput. These will serve as the basis for analysis and alerting.

2. Establish Baselines and Thresholds for Metrics

Once the metrics are being collected, the next step is to determine what ranges of values are "normal" and what should be considered alarming. We do this by establishing **baselines** for each metric and then setting **thresholds** that delineate acceptable vs. unacceptable conditions. As recommended in Azure's monitoring best practices, *"Start by defining baseline metrics that represent normal operating conditions for your resources. Use these baselines to set appropriate alert thresholds."* (Getting Started with Azure Monitor: Key Features and Best Practices | DataCamp). Baseline establishment can be done by observing the system during a period of normal operation (e.g., a few weeks including peak and off-peak times) or by referring to SLOs if they are already defined by the business.

For each L.E.S.T metric, consider how to derive its thresholds:

- **Latency:** If we have an SLO (say, 95% of requests under 300 ms), that directly gives an upper threshold for latency. Otherwise, look at historical p95 or p99 latency during healthy operations and choose a threshold somewhat above that (e.g., if p95 is usually 250 ms, one might set an alert at 400 ms p95 latency to catch serious degradation). The lower bound for latency is normally 0 (cannot be negative), so a very low latency isn't an issue unless it indicates something like a calculation error. However, *changes* in latency baseline are important; a sudden drop in latency concurrent with a drop in throughput might mean traffic is not reaching the system (so it appears fast because nothing is being processed). Therefore, latency should be considered in context with throughput. We might set

a rule like: “if latency = 0 and throughput = 0 for a period during expected load, trigger an alert” (meaning no traffic is flowing at all). Typically, though, latency alerts focus on high latency. **Service-level objectives (SLOs)** and **Service-level indicators (SLIs)** come into play here: one might define an SLI for latency (e.g., median and 90th percentile) and tie alerts to breaching the SLO targets over a rolling window.

- **Errors:** Ideally, the baseline error rate is zero or near-zero for a well-functioning service (aside from minor acceptable levels, like perhaps a few known benign errors). An SLO for errors might be, for example, “99.9% of requests succeed” which translates to error rate < 0.1%. We set the error rate threshold accordingly. If historically a service has, say, 0.01% error rate, and it jumps to 1% consistently, that’s a red flag. Thus, an alert threshold could be slightly above normal: e.g., alert if error rate > 0.5% for 5 minutes (tunable depending on how quickly we want to respond and how much false positives we tolerate). For lower bound, an error rate of 0 is obviously fine – though one might consider an alert if *zero traffic* and thus zero errors when traffic is expected, but that’s more of a throughput concern.

- **Saturation:** This metric often benefits from having a **target utilization** defined. As the SRE guidance notes, *“many systems degrade in performance before they achieve 100% utilization, so having a utilization target is essential”* (Google SRE monitoring ditributed system - sre golden signals). For example, one might determine through load testing that a particular server’s optimal throughput is reached at ~70% CPU; beyond that, latency rises sharply. In that case, one could

set 70% as a desired upper bound for CPU in normal conditions, perhaps alert at 85% (leaving some headroom). Similarly, for memory, if using >90% of memory often precedes OOM issues, one might alert at 90%. For disk space, a common practice is to alert when free space < X% (like <20% free) to have time to react before it fills (as suggested, e.g., by Kubernetes best practices: *"alert when available disk space falls below a defined threshold, e.g., 20%"* (Kubernetes Monitoring Best Practices)). The lower bound of saturation metrics (0% utilization) might indicate idleness. That generally isn't an alert (resources being under-utilized is a cost concern more than reliability concern, unless it's unexpected idle during a busy time, which again ties back to throughput). Thus, for saturation, we set **upper bounds** for safe operation. In cloud auto-scaling scenarios, these thresholds might also be tied to scaling actions (e.g., if CPU > 75% for 5 minutes, add an instance – ideally before it hits 100%). We will design thresholds with both alerting and scaling in mind.

- **Throughput (Traffic):** This metric's bounds are highly system-dependent. The upper bound could be defined by contractual limits or known capacity (e.g., a particular API might be designed to handle up to 1000 requests/sec; anything beyond might overwhelm it, so that could be an alert threshold). Also, a sudden spike far above typical peak could indicate a DoS attack or runaway client, thus warranting an alert. More commonly, a **lower bound** on throughput is used to detect outages: if your website normally gets >100 requests/minute during business hours and suddenly drops to 0, that's likely an incident (perhaps the load balancer is misrouting or the app crashed). So we might set an alert like

"throughput < 10 requests/min for 5 min (during 9am-5pm)" to detect a traffic drop. Care must be taken with time-of-day patterns; this is where having a baseline per time window or dynamic thresholds (with ML) helps. For simplicity, one can schedule different static thresholds for different periods (many cloud monitoring systems allow scheduled alerting or at least multiple conditions).

To gather baselines, we can utilize historical data or initial period of monitoring. Visualization tools like Grafana or CloudWatch Metrics Explorer help in plotting the distributions and variations of metrics. One effective method is to run **load tests or chaos engineering experiments** to see how the system behaves at extremes, which informs where the "cliffs" are (e.g., at what throughput does latency shoot up, at what memory usage does swapping start, etc.).

3. Configure Alerts and Integrate with Incident Response

With thresholds defined, we proceed to create **alerting rules** in the monitoring system. The goal is to have alerts that are actionable and timely. Each alert should ideally be tied to a known playbook or automated action (or at least a clear investigation path). Based on best practices (Monitoring and Alerting in the Cloud: Best Practices for DevOps Teams | Cloud Native Daily), we ensure that we configure both **static threshold alerts** and possibly **anomaly detection alerts** for each metric:

- **Static Threshold Alerts:** For each key metric, configure an alert condition that triggers when the metric breaches the threshold for a certain duration. For instance:

- *Latency alert:* if the 95th percentile latency > 400 ms for 5 consecutive minutes, trigger a “High Latency” alert.
- *Error alert:* if error rate (5xx count / total requests) > 1% for 2 minutes, trigger an “Error Rate Spike” alert.
- *Saturation alert:* if CPU usage > 85% for 5 minutes, or if memory usage > 90%, or if disk free < 15%, trigger a “High Utilization” alert (possibly separate alerts for each resource type).
- *Throughput alert:* if requests/sec < 10 (during a time window where normally >100) – this might be implemented via a static threshold with a time schedule or via a more advanced anomaly detection (discussed next).
- *Throughput high alert:* if traffic > some very high value (e.g., double the peak baseline) – though often high traffic will manifest as other issues (like latency or saturation) before we solely alert on the traffic itself.

These alerts can be configured in cloud-native monitors (e.g., AWS CloudWatch Alarms, Azure Monitor Alert Rules, GCP Alerting Policies) or in tools like Prometheus Alertmanager or Datadog monitors. Each platform has slightly different syntax and capabilities (for example, GCP allows combining multiple conditions, AWS CloudWatch has “anomaly detection” for alarms which auto-adjusts thresholds based on historical patterns). We will use those capabilities as available. Notably, the **Medium “Cloud Native Daily” guide** suggests: *“Set appropriate thresholds for each metric, indicating when an alert should be triggered. Establish baseline performance levels to identify*

anomalies more accurately." (Monitoring and Alerting in the Cloud: Best Practices for DevOps Teams | Cloud Native Daily) which aligns with our approach here.

- **Dynamic/Anomaly Alerts:** Where available, we enable anomaly-detection-based alerts for metrics that have cyclical patterns. For instance, Azure Monitor and AWS CloudWatch both offer an "anomaly detection" feature on metric alarms that uses statistical models to create a band of expected values and alerts if the metric goes outside that band. This can be very useful for catching issues like "throughput dropped lower than usual for this hour of day" or "latency is higher than usual given the current load". If using Prometheus, we might not have out-of-the-box ML, but we could write recording rules for, say, a rolling average and standard deviation and alert if the current value is 3 std dev away from rolling mean (a simple anomaly heuristic). Additionally, external AIOps tools or services (like Dynatrace's AI engine, or Datadog's Watchdog) can supplement by sending alerts on detected anomalies without explicit thresholds. These should be tuned to ensure they're adding value and not noise. In our methodology, we treat dynamic alerts as a complement to static alerts, not a replacement, especially because setting them up might require more mature operations. Initially, one might proceed with static thresholds, then iterate to add anomaly detection for areas that were hard to static-threshold.

Once the alert rules are set, we configure them to notify the appropriate channels. Typically, alerts should go to an **incident management system** or paging system that the SRE/on-call team uses. This could be an email, Slack/MS Teams message, or integration with PagerDuty/OpsGenie for phone/SMS alerts. Grouping and routing is important: for

example, production alerts go to the on-call, whereas maybe less critical warnings or dev/test environment alerts might just email a team distribution. We also implement **deduplication and suppression logic** if possible (for example, Alertmanager allows grouping alerts so you don't get 100 notifications if 100 nodes all hit high CPU – instead you get one notification saying "100 nodes high CPU").

It's worth noting that as part of alert configuration, one should document a **runbook** or attach instructions to each alert. Many systems let you include links or annotations in the alert payload. The person on-call should immediately know what to do when an alert fires. For instance, an alert "Disk space < 15% on Server X" could link to a runbook that says "Check if logs are filling disk, consider cleaning /var/log or adding disk space or failover to a new instance".

4. Automate Remediation (Self-Healing)

After detecting a problem via an alert, the ideal scenario is that the system can **resolve it automatically (or at least mitigate it) without human intervention**. This step of the methodology involves identifying which alerts or failure modes can be handled programmatically and then implementing those **automation hooks**.

There are multiple avenues for automated remediation in cloud environments:

- **Auto-Scaling:** This is a built-in automation for handling saturation and throughput issues. All major clouds support auto-scaling groups (AWS ASG, Azure VMSS, Google Instance Groups) where you can define policies to add or remove VM instances based on CPU, memory, or custom metrics. Similarly, at the platform level, Azure App Service and AWS Lambda have scaling rules, and

Kubernetes has the Horizontal Pod Autoscaler (HPA) for pods and Cluster Autoscaler for nodes. As a proactive measure, we configure auto-scaling wherever applicable. For example, we might set an auto-scale rule: if average CPU > 70% for 5 minutes across the web server pool, add one instance. Or in Kubernetes, if a deployment's CPU usage > X, increase replicas. This way, even before an alert pages an SRE, the system tries to handle the increased load. We align these with our thresholds: typically, auto-scale triggers at a lower threshold than a critical alert (so it attempts to fix the issue early). This **automation reduces manual intervention** as noted in best practices: *"For example, an autoscaling action can be triggered when CPU utilization exceeds a threshold. Automation reduces manual intervention."* (Monitoring and Alerting in the Cloud: Best Practices for DevOps Teams | Cloud Native Daily).

- **Auto-Healing:** Many cloud services have health checks and can automatically replace or restart failed components. In AWS, if an EC2 instance in an Auto Scaling group fails a health check, it can be terminated and a new one launched. Kubernetes will restart pods that crash or fail liveness probes. We ensure these features are enabled. For instance, for critical workloads on VMs, consider using AWS EC2 Auto Recovery or Azure's VM health checks to auto-reboot or notify if a VM is unresponsive. In Kubernetes, define liveness probes for each service so that hung containers get restarted automatically. These mechanisms address issues without even needing an alert in some cases – they are pre-emptive fixes.

- **Scripted Runbooks & Lambdas:** For scenarios not covered by built-in automation, we can wire custom scripts or serverless functions to trigger on alerts.

Many monitoring systems allow an alert to **trigger an action** (sometimes called Alert -> Action or webhooks). For example, an AWS CloudWatch Alarm can be set to trigger an SNS notification which a Lambda function subscribes to; the Lambda could then execute remediation (like cleaning up temp files if disk space low, or cycling a service, etc.). Azure Monitor similarly has *Action Groups* which can run an Azure Automation Runbook or Logic App when an alert fires. GCP can trigger Cloud Functions or Cloud Run jobs via Cloud Monitoring alerts. Using these, we implement specific fixes:

- If disk space low alert fires on a known issue (say logs filling up), automate a log rotation or cleanup via script.
- If the error rate spikes after a deployment (perhaps indicating a bad release), an automated rollback could be initiated if your deployment pipeline supports it (this is advanced but some organizations do automated canary rollbacks).
- If a particular service goes down, automate a failover (for example, if primary database instance is detected down via monitoring, automatically promote a read-replica to primary).

These are essentially encoding SRE runbooks into code. The **Mihir Popat (2025)** article on SRE automation highlights this practice: using playbooks to guide response teams and automating remediation scripts for known issues (The Role of Automation in SRE Success | by Mihir Popat - Medium) (The Role of Automation in SRE Success | by Mihir

Popat | Medium). Over time, as incidents repeat, one can keep adding to the automated remediation arsenal.

- **Self-Healing Systems Design:** Beyond reactive scripts, we also consider if the system architecture can be made more self-healing. For example, using resilient design patterns – circuit breakers that trip to stop cascading failures (and automatically reset after some time), or queuing mechanisms that can absorb bursts and recover. These are design-time decisions that complement runtime monitoring. Our methodology acknowledges these but focuses on the monitoring-triggered actions primarily.

In implementing automation, it's crucial to **test these automated actions** in non-prod environments to ensure they do what's intended. Also, we put guardrails: for instance, an automated script shouldn't keep rebooting a flapping service endlessly without human review. Often, the best practice is to automate the **safe, well-understood fixes**, and leave complex, uncertain situations to humans. As one source humorously puts it, "Don't automate doing the wrong thing faster." So we wouldn't, say, automatically delete databases because of an alert, but we might automatically expand a disk volume or restart a process.

We also ensure that automated actions themselves emit events or metrics (so we monitor our automation – e.g., count of auto-scaling events, success/failure of runbook executions). This provides feedback loops; if an automated fix fails, that could trigger an alert to a human.

5. Incorporate Historical Analytics and Machine Learning

The final part of the methodology deals with **continuous improvement** of the monitoring system using historical data and, where applicable, machine learning. After the system has been running with the above monitoring and some data has accumulated, we perform the following:

- **Trend Analysis:** Examine historical metrics to see trends such as growth in traffic, gradual increase in resource usage, etc. This is important for **capacity planning**. Many cloud monitoring tools have built-in analytic views (for example, CloudWatch Dashboards or Azure Monitor Workbooks) to view long-term data. We might identify that every day around 8 PM, latency increases slightly – which could correlate with a backup job or something. These insights can lead to adjusting thresholds or adding new metrics to watch (maybe we realize garbage collection time is a factor, so we start monitoring that). If certain metrics show a linear growth week over week (say, storage usage up by 5% each week), we can project when a capacity limit will be hit and plan upgrades proactively. In fact, as the SRE book mentions, *"saturation is also concerned with predictions of impending saturation, such as 'your database will fill its hard drive in 4 hours'."* (Google SRE monitoring ditributed system - sre golden signals). We can create such predictions via simple linear extrapolation or more advanced forecasting.

- **Forecasting and Predictive Scaling:** Using historical throughput data, one could predict future load. For example, e-commerce traffic might be predictable for a sale event. Cloud providers have started offering predictive auto-scaling (AWS Auto Scaling has a predictive mode based on daily patterns). We incorporate these if relevant: maybe configure auto-scaling to pre-warm instances at a

predicted peak time. If doing it manually, we could schedule scale-ups or use a prediction model to trigger a scale event a bit ahead of time.

- **Anomaly Detection (ML):** As discussed in the literature, ML algorithms can help detect anomalies without static thresholds. After collecting a baseline period, one could use tools like Amazon Lookout for Metrics, Azure Cognitive Services anomaly detector, or open-source libraries to train models on the time series. For instance, applying an ARIMA or Prophet model to forecast expected range for a metric and then alert if actual deviates significantly (Anomaly Detection with Machine Learning: Techniques and Applications | DoiT). Grafana's cloud offering and some on-prem versions now let you enable such ML-based alerts (learning seasonality, etc.) (Identify anomalies, outlier detection, forecasting - Grafana). In our methodology, we pilot ML-based alerts for one or two metrics that are noisy or hard to threshold (common cases: CPU that fluctuates or user traffic that has weekly cycles). The ML can flag anomalies like "Tuesday's traffic is 30% lower than the past 8 Tuesdays on average" which might indicate a silent outage or user drop-off.

- **Reliability Analysis:** Over time, use the collected data to compute SLO compliance and refine SLOs. If we set an SLO of 99.9% uptime or certain latency percentiles, we verify those against the monitoring data. If SLOs are repeatedly missed, we either improve the system or adjust SLOs if they were unrealistic. Monitoring data feeds into **error budgets** (if using SRE practices), which guide how much risk (in changes) can be taken. This is more a governance aspect but is enabled by good monitoring.

- **Incident Post-Mortems and Metric Reviews:** Each incident that occurred, we review whether the monitoring was adequate. Did an alert fire in time? If not, why did we miss it – do we need a new metric or alert? Or did an alert fire too much (false positive) – do we need to tweak threshold or add a condition? This iterative process ensures the monitoring set evolves. It's often said that monitoring is never "set and forget" – it requires continuous tuning.

By following these phases, one can build a monitoring system that is minimal (only key metrics), but robust (has good coverage of failure modes) and smart (uses automation and analytics to handle issues proactively). In practice, one would document this entire setup in a monitoring strategy document for the organization, and periodically audit it. For example, **TechTarget recommends** to *"continuously test your cloud monitoring tools to ensure they are fully functional"* and through regular testing and audits, adjust them (Cloud Monitoring: 8 Best Practices, Benefits & More | CrowdStrike). We incorporate that advice by scheduling quarterly drills where we simulate an incident to see if alerts trigger and if runbooks (manual or automated) work. This ensures our confidence in the monitoring.

Having described the methodology abstractly, we now proceed to concrete examples in real cloud environments, demonstrating how these principles can be applied on AWS, Azure, and GCP.

Case Examples

To illustrate the above methodology in action, we present a series of hypothetical case examples across AWS, Azure, and GCP. Each example focuses on an IaaS deployment (or a container deployment on IaaS) and shows how the minimal metrics and practices can be implemented using that cloud's native services combined with platform-agnostic tools.

Case 1: Web Application on AWS EC2 with Auto Scaling

Scenario: A web application running on a cluster of Amazon EC2 instances behind an Elastic Load Balancer (ELB). The application stack is a typical Linux/Apache/Tomcat server. The team uses AWS for infrastructure but wants to keep the monitoring minimal and possibly integrate with an open-source stack.

- **Metrics Collection:** Out of the box, AWS CloudWatch provides metrics for EC2 instances such as CPUUtilization, NetworkIn/Out, DiskRead/Write (for EBS volumes), etc., reported at 1-minute intervals (with detailed monitoring enabled). These cover saturation (CPU, network, disk) and throughput (network can be proxy for throughput, or ELB provides request count). AWS ELB provides metrics like RequestCount, Latency (avg), HTTPCode_ELB_4XX_Count, HTTPCode_ELB_5XX_Count (which are error counts). We enable these metrics. For application-level latency distribution and error rate, we decide to use a custom metric: the team instruments the application to record request processing time and emits an aggregate metric (e.g., average or p90 latency every minute) to CloudWatch via the CloudWatch agent or AWS Embedded Metrics Format.

Alternatively, they push detailed request logs to CloudWatch Logs and use CloudWatch Logs Insights to extract latency percentiles, but that's more advanced. For simplicity, suppose they instrument to send a custom CloudWatch metric "AppLatency" and "AppErrors" count. These metrics give us L (latency) and E (errors). Saturation is captured by CPU and maybe memory (for memory, CloudWatch needs the agent since default EC2 metrics don't include memory – we install the CloudWatch Agent on the instances to report MemoryUtilization). Throughput is captured by ELB's RequestCount and perhaps a custom metric "AppRequests" if we want an exact count at app level.

- **Baselines & Thresholds:** After a week of running, assume we see that at peak (100 req/s), CPU on each instance goes ~60%, and p95 latency ~200 ms, error rate near 0%. We set thresholds: CPUAlarm if >80% for 5 min, LatencyAlarm if p95 > 500 ms for 5 min, ErrorAlarm if 5xx count > 5 per minute for 5 min (since normally it's 0). Also, a LowTraffic alarm if RequestCount < 1 for 5 min (during business hours) to catch unexpected drops. These thresholds reflect some headroom above the observed baseline.

- **Alerts:** We create CloudWatch Alarms for those metrics. For example, CPUAlarm on each Auto Scaling group (or per instance, but better aggregated) with threshold 80%. CloudWatch Alarms are set to trigger an Amazon SNS topic. The SNS topic is subscribed to the team's Slack via a webhook and to an email for on-call. Each alarm message includes a brief description and CloudWatch dashboard link. We also use AWS's anomaly detection for the RequestCount metric to alert if traffic is significantly lower or higher than expected (this might

catch issues more dynamically than a fixed <1 threshold, adjusting for time of day).

- **Automated Remediation:** We configure an Auto Scaling policy: when CPU > 70% on average, add one EC2 instance to the group (scale-out), and perhaps scale-in when CPU < 30% for a while. This should handle sudden load increases without manual intervention. For errors, suppose a spike might indicate one instance is bad – ELB has health checks that if an instance fails health checks (could be due to many errors), it will remove it from rotation; we also enable EC2 Auto-Recovery so if an instance becomes impaired it restarts. Additionally, we use AWS Systems Manager Automation: for Disk space, we create a Automation document that clears a tmp directory and attach it to a CloudWatch alarm on low disk space, so it triggers automatically. The team also has a Lambda function subscribed to the ErrorAlarm SNS topic; the Lambda will automatically grab some debug info (like instance IDs with most errors) and restart the application service on those instances (as a temporary fix), then send a message that it attempted this. This Lambda acts as a first-aid: if it resolves the issue (e.g., app was stuck), great; if not, the on-call will see errors persist and take further action.

- **KPI and SRE considerations:** The team defined an SLO: 99% of requests under 400 ms latency. The monitoring will track this and we configure a CloudWatch SLO dashboard (maybe using a Metric Math to compute the percentage of requests under 400ms). If SLO is violated over a day, an alert notifies the SRE team (not paging, but as an FYI to investigate). The system leverages AWS's

reliability features but remains minimal in metrics – primarily CPU, memory, error counts, latency, and traffic.

Outcome: Using this setup, when traffic increases, auto-scaling adds instances (saturation handled automatically). If a deployment causes errors, the ErrorAlarm triggers; the Lambda attempts a restart; if errors continue, the on-call gets an alert with actionable information (since the Lambda might include logs). If latency grows gradually, perhaps indicating saturation beyond scaling or a code issue, the LatencyAlarm pages an engineer to investigate (who might find a DB query is slow via further analysis). Meanwhile, all metrics are logged to CloudWatch, and weekly the team reviews them. They notice one instance type is consistently high on CPU – perhaps they then decide to use a larger instance size, adjusting capacity (capacity planning based on those metrics). This case shows AWS native tooling plus a bit of custom logic achieving our monitoring goals.

Case 2: Microservice on Azure Kubernetes Service (AKS) with Prometheus and Grafana

Scenario: A microservices-based application running on Azure Kubernetes Service. There are multiple containerized services communicating via REST. The team wants a cloud-agnostic monitoring solution, so they deploy Prometheus and Grafana in the cluster instead of relying only on Azure Monitor, but they will use Azure Monitor for some infrastructure aspects.

- **Metrics Collection:** Inside the AKS cluster, they install the **Prometheus Operator (kube-prometheus-stack)** which sets up Prometheus, Alertmanager,

and Grafana with a default scrape configuration. Immediately, this will collect Kubernetes metrics: node metrics (CPU, memory, disk via node-exporter), pod container metrics (via cAdvisor metrics that the kubelet exposes), and Kubernetes API server metrics (like request rates, latency of the API server), etc. Grafana comes with built-in dashboards for Kubernetes components. For application services, the team uses the Prometheus client library in their code. For example, each service exposes an /metrics endpoint giving metrics like http_request_duration_seconds (a histogram of request latencies) and http_requests_total (count of requests and their statuses). Prometheus is configured to scrape these from each pod. This yields latency distributions, request throughput, and error counts (which can be derived from status codes) for each service – fulfilling L, E, T. Saturation is covered by node metrics and perhaps some app-specific ones (like queue length if any internal queue, which they also expose). Additionally, Azure Monitor can be set to capture node VM metrics too, but since AKS nodes are Azure VMs, we might rely on Prometheus for everything to keep it self-contained.

- **Baselines & Thresholds:** After deploying and running for some time, they identify normal ranges: e.g., Service A normally handles 50 req/s with 95th latency 100 ms; Service B handles fewer requests but is CPU heavy, using ~70% of a 4-core node under normal load. They set thresholds accordingly in Prometheus Alertmanager rules (could also integrate Azure Alerting via Container insights, but they choose Prometheus for consistency):

- If process_cpu_seconds_total (rate) for a container exceeds, say, 0.8 * #cores (80% CPU) for 5 minutes, alert “High CPU in container X”.
- If http_request_duration_seconds_bucket shows >5% of requests taking >1s (perhaps by calculating the rate of requests over 1s), alert “High Latency service X”.
- If http_requests_total error count increases – e.g., more than 5 errors in 1 minute, alert “Error spike service X”.
- If a node’s memory usage > 90% or disk space < 10%, alert “Node resource exhaustion”.
- Also if any service’s throughput drops to 0 when it previously was >0 (and pod is still running), that could mean the service is stuck; they set an alert if a certain key operation hasn’t happened in a while.

These thresholds are encoded as PromQL expressions in Prometheus alerting rules YAML. For example, an error alert might use something like: increase(http_requests_total{status=~"5.."}[5m]) > 0 to catch any 5xx in last 5m for a service that usually has none.

- **Alerts:** The Alertmanager is configured to route alerts to an email or Teams webhook. Also, since this is Azure, they integrate Alertmanager with Azure Monitor by sending critical alerts to an Event Hub or Azure Function that then creates an Azure alert incident (this is optional; one could just use Alertmanager

alone). The key is they have a single place (Grafana or Alertmanager UI) to see all firing alerts across the cluster.

- **Automated Remediation:** In Kubernetes, much is automatically handled: if a container crashes, it restarts. They also set up the Horizontal Pod Autoscaler for a couple of services: e.g., scale out the "frontend" deployment if CPU > 60% or if request per second per pod > some value. This uses Kubernetes metrics (which Prometheus also can see via adapter). Additionally, they employ a self-healing mechanism: a simple CronJob runs daily to clean up unused Docker images and free disk on nodes (to prevent node disk saturation). For specific issues, they use K8s Operators or tooling; for instance, if they know that sometimes a stuck pod needs deletion, they could automate that by detecting via an alert and then using Kubernetes API to force delete or restart it. Since AKS is managed, they rely on Azure's SLA for the control plane but still monitor API server latency; if the API server is slow, they can alert Microsoft support via an automated ticket (just to illustrate completeness – this might not be fully automated but could be).

Another automation: integrate Azure Functions with Alertmanager – e.g., if an alert "High Latency service X" fires and they suspect it's due to pod saturation, an Azure Function could automatically trigger scaling by increasing the replica count of that service by 1 (via Kubernetes API). This is doable but they might prefer using the built-in HPA as mentioned.

- **Historical Analysis:** All metrics in Prometheus are stored (say 15 days by default). They also configure long-term storage by using the Thanos project or

remote write to Azure Data Explorer, so they can keep months of data. Grafana is used to plot weekly trends. They notice one service's latency creeping up over weeks – that informs them to allocate more memory to it. They also use Grafana's **alerting with prediction**: Grafana can evaluate if a metric's forecast shows it crossing threshold soon (Identify anomalies, outlier detection, forecasting - Grafana), though if not using Grafana Cloud, they might skip this.

This case demonstrates how a cloud-agnostic open-source toolset (Prometheus/Grafana) can be used on a managed cloud cluster. The solution is portable – it could run on any Kubernetes, not just AKS. It leverages Kubernetes's own capabilities (HPA, self-healing) extensively, aligning with the SRE best practice of automating reliability as much as possible. The Golden Signals are well-covered: each service's latency, errors, throughput are known, and saturation at node and pod level is tracked. If the company ever moved to a different cloud or hybrid, this monitoring could move with the workloads, fulfilling the platform-agnostic requirement.

Case 3: Multi-Cloud Deployment with a Unified Monitoring Stack (GCP and AWS)

Scenario: A company runs some services on AWS and others on GCP for redundancy. They want a single monitoring view. They decide to use a combination of **ELK stack for logs** and **Grafana for metrics** across both platforms, along with a commercial tool (Datadog) as a secondary layer (to illustrate options).

- **Metrics Collection:** They deploy the Datadog agent on all their AWS EC2 and GCP Compute Engine instances. The agent automatically sends system metrics (CPU, memory, disk, network) and can also scrape application endpoints for

metrics (Datadog can ingest Prometheus metrics or JMX, etc.). This gives them a uniform set of metrics in Datadog across clouds. In parallel, they set up **Grafana Cloud (or self-hosted Grafana)** connected to both CloudWatch and GCP Monitoring via plugins/APIs. Grafana can query AWS CloudWatch metrics (using read-only API keys) and GCP metrics, so they build dashboards that show e.g., "Frontend service CPU on AWS vs GCP" side by side. This demonstrates platform-agnostic viewing. They also use **Elastic Stack** for logs: each VM sends logs to a centralized Elastic Cloud cluster. Logs are used for drilling down when metrics show a problem (for instance, high error rate metric leads them to search logs for error details).

- **Baselines & Thresholds:** They coordinate thresholds such that whether an app is on AWS or GCP, the alert criteria are similar. For example, both environments have a web server component, one in AWS, one in GCP (active-active). They set an alert if either location's latency > X or error rate > Y. Baselines are determined per environment though, as there might be slight differences. If GCP VMs are smaller, their CPU might run higher normally, so maybe threshold CPU 85% on AWS and 90% on GCP, etc. The idea is to prevent false alarms from differences in environment.

- **Alerts:** They choose to rely on **Datadog's unified alerting** for critical alerts because it covers both. In Datadog, they create monitors that use a query like "avg:service.error_rate{env:prod} by {cloud} > 1%" – this will trigger if error rate in either cloud's prod environment goes above 1%. The alert message will include which cloud. Datadog's machine learning features (Watchdog)

automatically flag anomalies; for example, it might notify them “This host’s memory usage is 3x higher than usual” without a preset threshold. They treat those as informational alerts to verify if something strange is happening. Additionally, they set up Grafana alerts on some panels as a backup (Grafana can alert if a CloudWatch metric goes out of range, though Datadog already covers it – redundancy in monitoring can be helpful but one must ensure they don’t double-page).

- **Automated Remediation:** Because this is multi-cloud, they implement cloud-specific automation for each but trigger them from a central logic. For instance, they have an automation controller (could be a Jenkins or custom orchestrator) listening for alerts. If an alert says “AWS web tier high CPU”, it calls the AWS Auto Scaling API to scale out the ASG for that tier. If an alert says “GCP database disk near full”, it triggers a Terraform script or gcloud command to resize the disk. These automations are maintained in a runbook repository. Alternatively, they could use a multi-cloud automation platform like HashiCorp Nomad/Consul or cloud-agnostic tools, but here they likely script each because the environments are separate. The key is they have logic in place for both sides. They might also utilize cloud-native automation: AWS auto-scaling and GCP’s instance group auto-scaling both on, coordinated such that each environment scales independently. That handles a lot automatically.

- **Observability Enhancements:** Since they use Elastic for logs, they set up alerts in Elastic too, like if a particular error message appears more than N times in 5 minutes (indicating maybe a new kind of error). They also use distributed tracing

(with OpenTelemetry) to track requests across AWS and GCP services – sending traces to a Jaeger or Datadog APM. This goes beyond basic monitoring but helps pinpoint which service in which cloud is slow when latency is high.

- **Historical and AI:** Over months, they feed data to an AI ops tool (Datadog's AI or an external one) to predict incidents. For example, it might correlate that "whenever GCP cache hit rate drops below 80%, we see error rate go up in 30 minutes" – an insight they then codify into a new metric or alert. They also conduct monthly capacity reviews: the Grafana dashboards show that AWS side is underutilized compared to GCP, so they might save cost by adjusting how traffic is routed.

This multi-cloud case underscores the need for **unified monitoring tools**. By using Datadog (commercial unified platform) (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons) and Grafana/Elastic (open source tools that work on any cloud), the team achieves a single coherent monitoring view. It validates that our minimal metrics (L.E.S.T.) are applicable no matter the environment – the team monitors latency, errors, saturation, throughput for each service, and just aggregates or compares between clouds. The specific values might differ but the concept holds. The case also shows that even with multiple providers, one can automate responses (though the automation might need to interact with different APIs under the hood).

Each of these case studies demonstrates the practicality of the minimalist monitoring framework. They show that by focusing on the key metrics and leveraging cloud capabilities (like auto-scaling and managed metrics) plus possibly augmenting with open-

source tools, one can achieve robust monitoring without deploying overly complex or numerous tools. Next, we summarize best practices gleaned from these scenarios and the earlier analysis.

Best Practices

Based on the above methodology and examples, we compile the following best practices for implementing monitoring tools in a minimalist yet effective way in cloud environments:

Focus on the "Golden" Metrics (L.E.S.T.)

Concentrate on Latency, Error rates, Saturation (resource use), and Throughput as primary indicators of system health. These four metrics should be collected and watched for every significant service or component. They cover both user experience and system capacity. Avoid the trap of collecting hundreds of metrics that aren't actionable; start with this core and expand only if a clear gap is identified. As an SRE principle: *"If you measure all four golden signals and page a human when one signal is problematic... your service will be at least decently covered by monitoring."* (Google SRE monitoring ditributed system - sre golden signals). Use these as Service Level Indicators – e.g., define SLOs like "<1% error rate" or "99th percentile latency under 500ms" and monitor against them.

Platform-Agnostic Implementation

Use tools and standards that work across environments. **Leverage open-source monitoring stacks (Prometheus, Grafana, ELK, OpenTelemetry)** which can be deployed on any cloud or on-premise. This ensures consistency when you have a hybrid or multi-cloud deployment. For example, instrument your application with OpenTelemetry APIs for metrics/traces – then whether it runs on AWS or Azure, the same data can be collected. If you use managed services (like AWS RDS or Azure

Service Bus), use their provided metrics but export or federate them into your central system (e.g., use CloudWatch Metric Streams or Azure Monitor exports to pull data into a unified database). Prefer solutions that are **vendor-neutral**: a blog on multi-cloud monitoring notes that *"a good monitoring solution will provide an integrated view across both in-house and cloud-based applications… ideally from a single view"* (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter). For example, Grafana can combine data from CloudWatch, Azure Monitor, and GCP operations into one dashboard – consider such approaches to avoid siloed views.

Keep Monitoring Architecture Minimal

Adopt a **simple architecture** for monitoring itself. One or two systems should handle most needs: e.g., Prometheus for metrics and Elastic for logs, or Datadog for everything. Having multiple parallel monitoring systems can lead to confusion (which source of truth to trust). At the same time, ensure redundancy for critical monitoring paths (Prometheus should be highly available or use a cluster, etc., because the monitor should not be a single point of failure). Google SRE's experience shows preference for simpler monitoring pipelines (Google SRE monitoring ditributed system - sre golden signals) – they avoid overly complex dependency hierarchies in alerts. We can emulate that by avoiding alerts that depend on too many layers ("if service A is slow and cause is B then do C" – instead alert on the symptom, and have playbook to check causes).

Establish Baselines and Use Thresholds Intelligently

Before jumping to set alert thresholds, gather baseline data. **Profiling normal operation** helps distinguish real issues from natural fluctuations. When setting static thresholds,

choose values that reflect a clear deviation from normal (to reduce false alarms). Also implement **hysteresis or multiple data points** for alerts: require the condition to hold for a few minutes or multiple samples before alerting, to avoid flapping alerts on single spikes. All cloud monitoring services support this (e.g., requiring 3 out of 5 datapoints breaching) (Recommended alarms - Amazon CloudWatch). Combine multiple related signals if it improves fidelity – e.g., alert when latency high **and** throughput drop together (meaning latency high not just due to high load but maybe an internal issue). Ensure thresholds are revisited periodically as the system evolves.

Automated Alerts and Runbooks

For each alert, have a documented **response procedure**. Whether automated or not, the action on alert should be known. Implement **automated responses for well-understood triggers**: *"Implement automation for certain remediation actions in response to specific alerts… Automation reduces manual intervention."* (Monitoring and Alerting in the Cloud: Best Practices for DevOps Teams | Cloud Native Daily). Examples: auto-scaling on high CPU, auto restart on memory leak, etc., as we detailed. Use tagging in alerts to indicate severity and owner (so the right team is notified). Maintain an on-call rotation for critical alerts and route alerts appropriately (using something like PagerDuty schedules or Opsgenie). Non-critical alerts can just create tickets or emails that are handled in working hours.

Integrate Machine Learning Judiciously

Incorporate ML-based analytics **after** establishing solid basic monitoring. Use ML primarily to enhance anomaly detection and forecasting, not as a black box for

everything. For instance, enable anomaly detection on metrics that have seasonal patterns; use predictive alerts for things like disk growth. Always review ML suggestions with SRE intuition before acting. The goal is to catch things like "saturation creeping up" or "subtle error rate increase" early. According to one guide, *"Integrating AI will enhance the ability to predict issues before they occur, offering more proactive monitoring and decision-making."* (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons) – leverage this for capacity planning (predicting when you'll need more resources) and for detecting out-of-family behavior. But keep critical, user-impacting alerts on simple, proven triggers to ensure reliability of the alerting system (as per Google SRE, don't solely rely on "magic" ML for core alerting (Google SRE monitoring ditributed system - sre golden signals)).

Unified View and Correlation

Strive for a **single pane of glass** where on-call engineers can see the state of the system at a glance. This often means setting up a dashboard showing the four golden signals for each major service in one place. Also include business-level checks (like homepage availability from an external probe) on the same board if possible. The New Horizons guide emphasizes unified dashboards for multi-cloud (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons). Additionally, correlate data: use tools that let you pivot from an alert to related data (e.g., from an error rate spike alert to the log entries of errors in that timeframe, or from a high latency alert to a trace showing what took long). This reduces time to diagnose. Modern observability stacks (like Splunk or Datadog) excel at this correlation – open source can achieve it too with custom links between Grafana and Kibana, for example.

Design for Resilience and Self-Healing

Wherever feasible, design the system such that it **recovers on its own**, with monitoring just confirming the recovery. Kubernetes deployments with liveness probes, auto-scaling groups replacing unhealthy instances, load balancers shifting traffic – use these features. Monitoring then serves to verify these mechanisms worked and to highlight when they cannot (e.g., if auto-scaling hits a max limit). A best practice from Edge Delta suggests: *"Implement automated remediation that relies on multiple indicators… from zero to self-healing"* (Self-Healing SRE & DevOps Napkin Sketch - Edge Delta), meaning use combined signals to trigger safe automation. However, always have monitoring to catch when automation fails or when human decision is needed (for example, if auto-healing restarts a pod 5 times and it still fails, alert a human).

Minimal Overhead and Cost

Monitor your monitoring. Keep an eye on how much data you are collecting and storing. It's easy for log volumes or high-frequency metrics to become expensive in cloud monitoring services. Use sampling for traces, and filter out metrics that aren't used. An example best practice: only enable detailed (1-min) metrics on critical components; use 5-min interval for others to save cost. If using open-source, ensure the monitoring components themselves are not using too many resources on your cluster. Also ensure security of monitoring endpoints (don't expose Prometheus metrics publicly without auth, etc., since it can leak internal info).

Adapt for IaaS vs PaaS Differences

In IaaS, you'll need to monitor the OS and middleware (CPU, memory, disk, services status). In PaaS, focus on application performance because the provider handles the OS. For example, with Azure App Services, use Application Insights for app metrics and rely on Azure's service health for platform issues. You might not get disk metrics in PaaS (because you can't see the disk), but you get things like consumption quotas. So adjust metrics: e.g., monitor Azure Function execution times and failures (as errors) instead of CPU. The key practice is: **know your responsibility** in the cloud model and monitor accordingly. If in doubt, an *"application-centric approach to monitoring"* works for all (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter): always verify the application is delivering the expected experience, regardless of underlying infrastructure.

Regular Drills and Updates

Finally, treat the monitoring system as living. **Test it regularly** – simulate failures (e.g., kill a service in a staging environment) and see if alerts fire and if automation triggers. Conduct game days where SREs practice responding to alerts using only the dashboards and runbooks. This will reveal any missing visibility or unclear instructions. Keep documentation (dashboard explanations, runbooks, SLO definitions) up to date as things change. Moreover, as new features or tools become available, evaluate if they can simplify your stack. For example, if a new managed service can replace a self-run Prometheus without loss of fidelity, consider it to offload toil.

By adhering to these best practices, a cloud team can ensure robust monitoring with a lean setup. These practices encapsulate an SRE-oriented mindset: measure what matters,

automate whenever possible, and continuously improve the system's reliability with data-driven insights.

Tool Comparison

To support the implementation of the above best practices, we compare several popular monitoring tools and platforms. We look at their features in the context of minimal, multi-cloud monitoring, considering whether they cover the L.E.S.T. metrics, how they support alerts and automation, and their applicability across different environments. The comparison is divided into **Open-Source Tooling** and **Commercial Multi-Cloud Solutions**:

Open-Source Monitoring Tools (Multi-Cloud Compatible)

Tool	Type	Key Features	Multi-Cloud Usage
Prometheus	Metrics Collector/DB	Time-series database and query language (PromQL); pull-based scraping of metrics (supports many exporters); Alertmanager for alerting rules.	Runs on any cloud (container or VM). Scrapes targets via HTTP – can monitor cloud VMs, containers, or platform services if endpoints exposed. Often used in Kubernetes. Integrates with Grafana for dashboards. (Multi-Cloud Monitoring and Alerting with Prometheus and Grafana)The cloud-native world relies on Prometheus for metrics collection (Multi-Cloud Monitoring and

Tool	Type	Key Features	Multi-Cloud Usage
			Alerting with Prometheus and Grafana).
Grafana	Visualization & Alerting	Dashboard builder supporting many data sources; alerting (can trigger notifications); plugins for CloudWatch, Azure Monitor, GCP, etc.	Fully multi-platform: can pull in metrics from various sources (Prometheus, cloud APIs, databases). Ideal as a unified interface across clouds. Many pre-built dashboards (e.g., for Kubernetes, Linux, etc.). Used as the front-end for both open-source and commercial backends.
ELK Stack (Elastic Stack)	Logging and Analytics	Elasticsearch (data store), Logstash/Beats (data ingestion), Kibana (visualization). Used for centralized log management and can store metrics as well.	Open-source and deployable anywhere (or use Elastic Cloud service). Commonly used to aggregate logs from multiple clouds into one index. Provides search and alerting on logs. ([

Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons

](https://www.newhorizons.com/resources/blog/multi-cloud-monitoring#:~:text=Splunk))Described as “an open-source solution for searching, analyzing, and visualizing log data.” (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons) Good for multi-cloud as logs from different sources can be normalized and analyzed together. | | **OpenTelemetry** (Collector & SDK) | Instrumentation Standard | Not a monitoring tool by itself, but a standard and set of libraries to collect traces, metrics, and logs from applications. The Collector can export data to various backends (Prometheus, Elastic, etc.). | Allows a single instrumentation to work on any cloud and send data to any backend. Promotes consistency in metrics names and tracing across microservices. Useful in multi-cloud microservice deployments to ensure all services have the same telemetry format. | | **Nagios / Zabbix** (mention legacy) | Host Monitoring (legacy) | Actively test and monitor host resources and availability (Nagios) or agent-based metric collection (Zabbix). Provide alerts and some dashboards. | Can be configured for any host reachable over network. However, less cloud-native; they don’t integrate as well with modern APIs or auto-discovery (one must configure each host). Typically used on-prem, but some use in cloud via deployed agents. Minimalistic in metrics (CPU, memory, ping, etc.), but scaling to dynamic cloud environments is labor-intensive. |

Notes: Open-source tools often require manual setup but give flexibility. Prometheus and Grafana are a powerful combo for metric-focused monitoring in cloud; ELK covers the log side. They are all proven at scale (e.g., companies run Prometheus with millions of time-series). The choice may depend on expertise and whether the organization prefers to manage infrastructure or use managed equivalents (like Amazon Managed Prometheus or

Elastic Cloud). The advantage is no licensing cost and full control; the challenge is the operational overhead of running them. However, their cloud-neutral nature makes them ideal for avoiding lock-in and achieving a single view. Grafana's ability to interface with cloud vendor metrics makes it a bridge between open-source and cloud-proprietary.

Commercial Cross-Platform Monitoring Solutions

Tool/Service	Description	Strengths	Multi-Cloud Capabilities
Datadog	Software-as-a-Service (SaaS) monitoring platform. Provides infrastructure monitoring, APM (application performance monitoring), log management, and security monitoring in one product.	Turnkey integrations (400+ integrations for various technologies), real-time dashboards, ML-based anomaly detection (Watchdog), powerful alerting and collaboration features. Minimal setup: install agent and get metrics. **Automated discovery** of L.E.S.T. metrics for common systems (e.g., it knows to track HTTP latency if using its APM).	Designed for multi-cloud and hybrid. The Datadog agent works on any host; it can pull metrics from AWS, Azure, GCP through their APIs. So you see all your servers and services in one pane. **Unified tagging** allows grouping by cloud or region. **Reference**: *"Datadog – Comprehensive platform with real-time metrics, security, and log management."* ([

Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons

](https://www.newhorizons.com/resources/blog/multi-cloud-monitoring#:~:text=Datadog)). Many companies use it to monitor across cloud and on-prem simultaneously. | | **New Relic One** | SaaS platform for observability: infrastructure monitoring, APM, synthetics, logs. It offers a unified telemetry database. | Rich APM for deep dive into application performance (transaction traces, database queries), user experience monitoring. Good support for custom metrics and big data queries (NRQL). | Multi-cloud support through agents and API integrations. It can ingest CloudWatch, Azure Monitor data, etc. New Relic provides cloud-specific dashboards out-of-the-box. It is often used in environments with multiple cloud deployments to compare performance. (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons)Noted for insights into app and infra performance. | | **Dynatrace** | An AI-powered monitoring solution that deploys OneAgent for full-stack monitoring. Offers infrastructure, APM, and network monitoring with a strong AI engine (Davis) for problem root-cause. | Automatic topology discovery (it maps dependencies between services automatically), very minimal manual configuration. Davis AI pinpoints the likely cause among many alerts. Good for large enterprise environments. | Supports AWS, Azure, GCP, and others – OneAgent will detect whatever environment it's in. Has integration for cloud services via APIs. All monitored data goes to its central cluster which can be cloud-hosted or on-prem. Particularly suited for complex multi-cloud microservices because of the dependency mapping (e.g., can show if an error in Azure service is caused by latency in an AWS DB call, etc.). | | **Splunk Observability**

(formerly SignalFx) | A SaaS platform from Splunk focusing on metrics (SignalFx) and APM (Omnition) and logs (Splunk). It's often sold as part of Splunk Cloud. | Very high-scale metrics ingestion, streaming analytics (detecting conditions on the fly). Strong in logs (Splunk's legacy) so good for log-heavy environments. | Multi-cloud: supports ingesting data from any source. Many use Splunk to aggregate logs from all clouds for compliance. The Observability suite specifically can take metrics/traces from cloud-native sources. However, Splunk's strength is more on logs and event analytics, while metrics/APM side competes with Datadog/New Relic. | | **Azure/AWS/GCP Native** (CloudWatch, Azure Monitor, GCP Operations) | Each cloud's native monitoring service. While not a single tool across clouds, we list them to compare using each vs a unified tool. | Deep integration with respective platform (e.g., AWS CloudWatch knows about every AWS service's metrics automatically; Azure Monitor can enable diagnostics on any Azure resource). No extra cost for basic metrics (already included), and usually highly reliable and available within that cloud. | By default, limited to its own cloud. However, they have options to extend: e.g., Azure Monitor can ingest logs from AWS/GCP, CloudWatch can receive custom metrics from anywhere. Some third-party bridging exists (or one can export metrics from one cloud and import to another's monitor). Generally, if you use only one cloud, the native tools are fine; for multi-cloud, relying on three separate systems is not ideal. Many organizations export native metrics to a central system (like using Grafana or a data lake) to achieve a unified view. |

Notes: Commercial tools often provide more **polish and advanced features** at the expense of licensing cost. They can significantly reduce the engineering effort to set up monitoring (important for minimalism if the team is small). For example, Datadog or

Dynatrace can quickly instrument an environment without the team building a lot of plumbing – this aligns with minimal overhead, though not minimal cost. On the other hand, open-source requires more initial work but could be cheaper and avoid vendor lock. Some companies even use a combination: for instance, use Prometheus for real-time critical metrics (no external dependency), but also feed data to a SaaS for long-term analysis.

In summary, the choice of tools should be guided by: **support for essential metrics**, **ability to integrate across cloud boundaries**, and **capabilities for alerting and automation**. The above table shows all the listed tools meet the basic needs (monitor metrics, set alerts). The open-source stack is very capable and flexible (and used in many SRE setups globally), whereas commercial solutions can accelerate deployment and add intelligence (like AI ops). A minimal approach can succeed with either – one could even consider a fully managed version of open-source (e.g., Grafana Cloud, Amazon Managed Prometheus, etc.) to get the best of both worlds.

Conclusion

Monitoring in cloud computing environments is both critical and challenging: critical, because the dynamic and distributed nature of cloud systems demands constant vigilance to maintain reliability; challenging, because the plethora of services and metrics can lead to complexity and overwhelm. In this paper, we advocated for a **minimalist yet powerful approach to cloud monitoring**, focusing on universal principles that transcend specific cloud vendors.

We identified **Latency, Errors, Saturation, and Throughput (L.E.S.T.)** as the essential metrics that serve as **leading indicators of system health** (Google SRE monitoring ditributed system - sre golden signals). By prioritizing these four "golden signals" (Google SRE monitoring ditributed system - sre golden signals) in an Infrastructure-as-a-Service context, an engineering team can cover the vast majority of performance and reliability concerns without boiling the ocean of data. We discussed how monitoring needs differ for Platform-as-a-Service, where the cloud provider takes on more responsibility – in PaaS, the emphasis shifts even more to application-level metrics (latency, errors) since lower-level saturation is often abstracted away (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter). Nonetheless, the fundamental signals remain relevant regardless of the service model, ensuring our approach is broadly applicable.

A major theme of our discussion was **platform-agnosticism**. We explored how to design monitoring that isn't siloed to one cloud – through the use of open-source tools (Prometheus, Grafana, ELK) and standards (OpenTelemetry), as well as noting

commercial solutions that intentionally support multi-cloud environments (Datadog, New Relic, and others). The benefit of this approach is not just avoiding lock-in, but also enabling a **unified operations perspective**. Modern SREs often have to manage hybrid environments; a consistent monitoring strategy means fewer tools to learn and fewer gaps in coverage. The literature underscored this need for unified views (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons) (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons), and our case studies demonstrated how it can be achieved in practice.

We put a strong emphasis on turning monitoring data into **actionable insights and automated actions**. Effective monitoring is not just about gathering data, but about responding to it. We outlined methods for setting meaningful thresholds – grounded in baseline analysis – and coupling them with alerting rules that have high signal-to-noise ratio (Google SRE monitoring ditributed system - sre golden signals) (Monitoring and Alerting in the Cloud: Best Practices for DevOps Teams | Cloud Native Daily). The goal is to ensure that when an alert fires, it indicates a problem that either already impacts users or will soon, thus warranting attention. Furthermore, we delved into the realm of **auto-remediation**: using cloud capabilities like auto-scaling groups, health checks, and serverless functions to automatically heal common issues. This aligns with the SRE objective of reducing toil and improving MTTR through automation (The Role of Automation in SRE Success | by Mihir Popat | Medium). We saw that many routine issues (e.g., transient high load, single-instance failure) can be handled without human intervention, allowing engineers to focus on more complex anomalies.

The integration of **real-time monitoring with historical analysis** provides the best of both worlds: immediate detection and response, plus deep understanding and prevention. We discussed how real-time monitoring catches incidents, while historical data, especially when processed with machine learning, can forecast and even preempt incidents (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons). By implementing techniques like anomaly detection (to adapt to changing baselines) (Anomaly Detection with Machine Learning: Techniques and Applications | DoiT) and predictive alerts (for capacity saturation forecasts) (Google SRE monitoring ditributed system - sre golden signals), reliability can be improved proactively. However, we also echoed caution from SRE experts about over-relying on opaque ML systems for critical decisions (Google SRE monitoring ditributed system - sre golden signals); a balanced approach uses ML as an assistant to human-defined rules, not a complete replacement.

Our examination of **tools** revealed that there is no one-size-fits-all solution, but there is a rich ecosystem to choose from. Open-source options give the ultimate flexibility to craft a bespoke monitoring system aligned with our minimalist metrics and can certainly be used across AWS, Azure, GCP uniformly. On the other hand, commercial tools can simplify multi-cloud monitoring by providing a ready-made integrated solution – at the cost of entrusting a third-party and incurring licensing fees. We provided a comparison to help inform decisions, noting how each option supports (or can be configured to support) the core pillars of minimal monitoring and automation.

From the perspective of an SRE or cloud engineer in a “closed” all-cloud environment (no on-prem), the entire solution space is within reach via cloud APIs and managed

services. This means that everything we monitor can also be controlled or adjusted via those same cloud interfaces, which is a powerful synergy. We highlighted this synergy in our algorithmic approach to incident response: monitors detect an issue, and then often the cloud itself (through auto-scaling, auto-healing, or triggered scripts) can attempt to fix it. This creates a **closed-loop system** where cloud services monitor and manage other cloud services under the supervision of SREs who set the policies. Such closed-loop automation is a hallmark of advanced SRE practice, inching towards the ideal of self-maintaining systems.

In conclusion, implementing monitoring in cloud environments with a focus on minimalism is highly feasible and, indeed, advantageous. By zeroing in on the most telling metrics, we avoid drowning in data and can design clearer alerts and dashboards. By staying platform-agnostic, we ensure the monitoring strategy is resilient to architectural or provider changes, and by employing intelligent alerting and automation, we significantly reduce the operational load and mean time to recovery. The end result is a lean monitoring setup that punches above its weight – small in scope but big in impact – effectively safeguarding application reliability and performance.

The research and cases presented here serve as a blueprint for practitioners to follow. An SRE team can adopt these practices incrementally: start with the golden signals, set up basic alerts, then iteratively add automation and analytics. Over time, this minimal core can be expanded or tuned as needed, but even on its own, it provides a strong safety net for cloud-deployed systems. The academic and industry sources cited reinforce each recommendation, showing that this approach is grounded in both theory (SRE principles) and real-world experience. Going forward, as cloud technology evolves (with trends like

serverless computing, edge computing, and even more managed services), the principles of minimal, platform-neutral monitoring will remain relevant – possibly extending to new metrics (for instance, cost could be considered another metric to monitor in clouds) – and the continuous improvement loop with human oversight and machine assistance will continue to be the model for reliable operations.

In summary, effective cloud monitoring does not require monitoring everything under the sun. It requires monitoring the right things, in the right way. **Minimalism in monitoring is not about having fewer data points; it's about having the most meaningful data points and using them wisely.** With the guidance provided in this paper, engineers can implement a monitoring framework that is streamlined yet comprehensive, ensuring robust observability and quick mitigations, ultimately delivering highly reliable cloud services.

References

1. Google SRE Book – *Monitoring Distributed Systems*, especially the section on the four golden signals and principles of alerting (Google SRE monitoring ditributed system - sre golden signals) (Google SRE monitoring ditributed system - sre golden signals) (Google SRE monitoring ditributed system - sre golden signals).

2. SolarWinds (Craig McDonald, 2020) – *Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS*, on Orange Matter blog, which contrasts monitoring responsibilities in IaaS vs PaaS vs SaaS (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter) (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter) (Suggested Cloud Monitoring Strategies for IaaS, PaaS, and SaaS - Orange Matter).

3. Medium (Cloud Native Daily, 2023) – *Monitoring and Alerting in the Cloud: Best Practices for DevOps Teams*, which lists best practices like defining key metrics, setting thresholds, and automating remediation (Monitoring and Alerting in the Cloud: Best Practices for DevOps Teams | Cloud Native Daily) (Monitoring and Alerting in the Cloud: Best Practices for DevOps Teams | Cloud Native Daily).

4. DoiT International Blog (2025) – *Anomaly Detection with Machine Learning: Techniques and Applications*, explains how ML-based anomaly detection adapts to dynamic cloud environments, replacing static thresholds (Anomaly Detection with Machine Learning: Techniques and Applications | DoiT).

5. New Horizons (Taylor Karl, 2024) – *Effective Multi-Cloud Monitoring: Tools and Best Practices*, highlights the need for unified monitoring, AI-driven analytics,

and increased automation in multi-cloud setups (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons) (Effective Multi-Cloud Monitoring: Tools and Best Practices - New Horizons - Blog | New Horizons).

6. Kubecost (2023) – *Kubernetes Monitoring Best Practices*, provides insight into key metrics on Kubernetes (cluster, node, pod levels) and advice on alerting (e.g., CPU > 80% triggers) (Kubernetes Monitoring Best Practices) (Kubernetes Monitoring Best Practices).

7. Mihir Popat (2025) on Medium – *The Role of Automation in SRE Success*, emphasizes reducing toil through automation, including automating monitoring, scaling, and incident response (The Role of Automation in SRE Success | by Mihir Popat | Medium) (The Role of Automation in SRE Success | by Mihir Popat | Medium).

8. Min.io Blog (Matt Sarrel, 2022) – *Multi-Cloud Monitoring and Alerting with Prometheus and Grafana*, confirms Prometheus+Grafana as a de facto observability stack in cloud-native environments (Multi-Cloud Monitoring and Alerting with Prometheus and Grafana).

9. DataCamp (2023) – *Getting Started with Azure Monitor: Key Features and Best Practices*, which gives examples of defining KPIs (request rate, response time, error rate) and setting up alerts in Azure (Getting Started with Azure Monitor: Key Features and Best Practices | DataCamp) (Getting Started with Azure Monitor: Key Features and Best Practices | DataCamp).

10. AWS Documentation – *Recommended CloudWatch Alarms*, which provides guidance on which metrics to alarm on for AWS services (e.g., CPU, latency, status check failures), reinforcing selection of essential metrics (Recommended alarms - Amazon CloudWatch).

11. Gremlin and New Relic Blogs – *The Four Golden Signals* explanations, underlining why latency, traffic (throughput), errors, and saturation are vital to monitor (used conceptually in this paper) (The four Golden Signals of Monitoring - Sysdig) (Setting better SLOs using Google's Golden Signals - Gremlin).

12. Grafana Documentation (2025) – *Forecasting and Outlier Detection in Grafana*, for the mention of using ML to learn expected metric values and doing dynamic alerting (Identify anomalies, outlier detection, forecasting - Grafana).

13. EdgeDelta (2022) – *Self-Healing SRE & DevOps Napkin Sketch*, discusses moving from reactive monitoring to automated remediation triggers in an SRE context (Self-Healing SRE & DevOps Napkin Sketch - Edge Delta).

14. LogicMonitor (2021) – *SaaS vs PaaS vs IaaS: Monitoring differences*, which helped inform the distinctions drawn in how monitoring needs differ by service model.

15. Splunk (2022) – *SRE's Golden Signals*, to reinforce the idea that golden signals define system health from the SRE viewpoint (SRE Metrics: Core SRE Components, the Four Golden Signals ...).

16. Microsoft Learn – *Azure Monitor best practices*, for guidance on data collection and alert configuration in Azure environments (used for Azure-specific insights like Azure Monitor integration with Prometheus (Getting Started with Azure Monitor: Key Features and Best Practices | DataCamp)).

17. TechTarget (2021) – *Best practices for defining a cloud monitoring strategy*, which emphasizes regular testing of monitoring tools and alignment with breach response (aligns with our recommendation for drills and audits) (Best practices for defining a cloud monitoring strategy - TechTarget) (Cloud Monitoring: 8 Best Practices, Benefits & More | CrowdStrike).

18. Cisco DevNet (2020) – *Golden Signals for SRE*, additional industry perspective on focusing on those four key indicators (supporting our minimal metrics rationale).

19. Brendan Gregg (2013) – *USE Method* (though not directly cited above, conceptually influenced the focus on utilization/saturation and errors as key metrics per resource).

20. AWS Well-Architected Framework – *Reliability Pillar*, which recommends monitoring key metrics and automating recovery (provided general guidance consistent with our approach).