International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I1P152
Eureka Vision Publication | Volume 7, Issue 1, 365-373 2026

Original Article

A Decision Framework for Multi-Cloud Microservice Deployment across AWS and GCP: Empirical Evaluation of EKS, Cloud Functions, Cloud Run, and Cross-Cloud Networking Patterns

Laxmi Madhu Kumar Brahmandam
Independent Researcher, Texas, United States.

Received On: 14/02/2026 ***Revised On: 14/03/2026*** ***Accepted On: 20/03/2026*** ***Published On: 25/03/2026***

Abstract **-** *Multi-cloud microservice deployment is frequently advocated as a hedge against provider risk and as a route to best-of-breed service selection, yet empirical evidence on when such designs deliver measurable value remains scarce. This paper presents a decision framework for placing workloads across Amazon Web Services and Google Cloud Platform, together with an empirical evaluation drawn from a representative workload set executed on production-grade configurations. The framework classifies workloads into four archetypes (long-running APIs, batch jobs, event handlers, and short-lived remote procedure calls) and recommends a primitive on each cloud (Amazon Elastic Kubernetes Service, AWS Lambda, AWS Fargate, Google Kubernetes Engine, Cloud Run, Cloud Functions, and Cloud Endpoints) based on observed fit. The methodology evaluates four metrics: cold-start latency, sustained throughput, per-million-request cost, and cross-region failover recovery-time objective. The reference deployments we examined exposed differences of up to 6.4x in cold-start latency between serverless primitives and steady-state cost differences of 18 to 41 percent at equivalent service levels. Cross-cloud failover RTOs ranged from 42 seconds for active-active configurations to 7.8 minutes for warm-standby ones. We find that multi-cloud is justified per workload rather than at portfolio level and that unified observability is necessary to keep operational overhead bounded. The findings inform reference architectures for organizations facing heterogeneous integration, regulatory, or acquisition-driven multi-cloud pressure across the cloud-native systems field.*

Keywords - Multi-Cloud, Microservices, Kubernetes, Serverless, Cross-Cloud Networking, Failover.

1. Introduction

Multi-cloud microservice deployment is widely discussed in industry literature as both a defense against single-provider risk and a means of composing services from each provider's catalog [1,2,11]. Empirical evidence, however, suggests that the operational drivers of multi-cloud are narrower than the marketing case implies. The production deployments we examined adopt multi-cloud configurations primarily for three reasons: a specific service-level capability fits one provider's offering substantially better than the other's; a compliance regime mandates geographic or jurisdictional distribution that one provider does not satisfy alone; or an organizational reality such as the acquisition of a workload built on a different cloud creates the multi-cloud condition. Each driver produces a distinct architecture with distinct trade-offs, which is at odds with the universal reference designs frequently proposed in the gray literature.

This paper synthesizes observations from a set of enterprise-scale multi-cloud deployments that combine Amazon Web Services (AWS) and Google Cloud Platform (GCP). The reference architecture uses AWS as the primary substrate for the majority of workloads, the analytical data plane, and the event-driven backbone, while GCP hosts specific workloads where Cloud Functions, Cloud Run, or Cloud Endpoints provide a measurably better fit than the AWS-native equivalents. The architecture is examined as a population of design choices rather than as a single case study.

The contribution of this paper is threefold. First, we present a decision framework that classifies microservice workloads into four archetypes and prescribes a primitive on each cloud for each archetype. Second, we report empirical measurements of cold-start latency, sustained throughput, per-million-request cost, and cross-cloud failover recovery-time objective (RTO) collected from the reference deployments under a controlled measurement protocol. Third, we identify the operational invariants (unified observability, federated identity, and shared container build) without which the multi-cloud architecture does not remain tractable. The methodology evaluates each archetype on each provider primitive using a representative workload set executed against production-grade configurations. The headline result is that the choice of primitive on a given cloud shifts p95 latency by a factor of up to 4.7x and per-million-request cost by up to 41 percent, while cross-cloud failover RTO varies from 42 seconds to 7.8 minutes depending on the placement strategy adopted.

The rest of the paper is organized as follows. Section 2 reviews background and related work on cloud-native systems and multi-cloud architecture. Section 3 details the methodology and measurement protocol. Section 4 presents the decision framework. Section 5 describes the compute primitives and their architectural patterns. Section 6 covers cross-cloud connectivity, unified observability, and federated identity. Section 7 presents results and discussion, including the comparative table of empirical measurements. Section 8 lists limitations and threats to validity. Section 9 concludes and identifies directions for future work.

2. Background and Related Work

Microservice architecture has been studied extensively as a pattern for decomposing large applications into independently deployable services [13,14]. Container orchestration, primarily through Kubernetes [8,15], has emerged as the dominant substrate for long-running microservices on both AWS (via Elastic Kubernetes Service) and GCP (via Google Kubernetes Engine). Serverless function platforms such as AWS Lambda and Google Cloud Functions [4] have been adopted for event-driven workloads, while managed container runtimes such as AWS Fargate and Cloud Run [5] occupy the middle ground between functions and orchestrated containers.

Prior work on multi-cloud architecture spans formal models [11,12], industry guidance from cloud providers [1,2], and reference designs from third-party vendors [10]. Comparative studies of serverless cold-start latency have appeared in academic venues including USENIX ATC and ACM SoCC, with reported cold-start times ranging from tens of milliseconds for warmed instances to several seconds for cold containers. Studies of cross-cloud networking have analyzed the cost and latency implications of egress traffic between providers; observed per-gigabyte egress fees commonly fall in the range of 0.08 to 0.12 USD depending on volume tier.

Service-mesh implementations such as Istio [9,16] have been proposed as an abstraction layer that unifies service-to-service communication across heterogeneous compute substrates, including the cross-cloud case. Federated identity through providers such as Anthos [17] and observability through OpenTelemetry [18] have been positioned as enabling technologies for tractable multi-cloud operation. The empirical evaluation reported here contributes per-archetype measurements that, to our knowledge, have not been published in a single comparative study covering both providers' primary microservice primitives.

The decision-framework genre has antecedents in earlier work on cloud-broker architectures, in which selection logic determines which provider hosts a given workload based on cost, performance, and policy criteria. The framework presented in this paper differs from broker architectures in that placement is decided at design time rather than at runtime, on the grounds that runtime portability across heterogeneous cloud primitives is rarely achievable without sacrificing the provider-specific capabilities that motivate the multi-cloud condition in the first place. This design-time orientation is consistent with the observed practice in the reference deployments and with the broader literature that treats true portability as a goal more often invoked than realized [13,19].

Empirical comparison of serverless platforms is an active area of research. Wang et al. [20] characterized cold-start and warm-start latency distributions across multiple commercial offerings, while Jonas et al. [19] examined the design space limitations imposed by current serverless abstractions. Hellerstein et al. [22] critique the model on grounds of data-shipping cost and absence of stateful coordination, points that are pertinent to the cross-cloud case in this paper because cross-cloud function invocations incur not only the cold-start penalty but also the inter-cloud network traversal latency.

3. Methodology

The empirical evaluation reported in this paper was conducted across the production deployments we examined, using a controlled measurement protocol applied to a representative workload set. The methodology is detailed below.

3.1. Reference Deployments

The reference deployments span five enterprise-scale environments operating microservice workloads across AWS and GCP. Each environment exposed between 47 and 312 microservices in production, with monthly aggregate request volumes between 1.4 billion and 18.7 billion. AWS-side compute was provisioned on EKS clusters running Kubernetes 1.28 with managed node groups and a service mesh deployed cluster-wide. GCP-side compute used a mix of Cloud Functions (2nd generation), Cloud Run, and Google Kubernetes Engine (GKE), with Cloud Endpoints for API management. Cross-cloud connectivity used dedicated interconnect through both providers' partner network programs, with redundant paths across regions.

3.2. Workload Archetypes

Four workload archetypes were defined to cover the substantive variation in the reference deployments. Long-running APIs are request-driven services with persistent containers and steady throughput requirements. Batch jobs are non-interactive workloads with bounded execution windows and asynchronous result delivery. Event handlers are short-lived invocations triggered by message-bus or storage events. Short-lived remote procedure calls (RPCs) are synchronous services with sub-second median request lifetimes and tail-sensitive latency budgets. Each archetype was implemented as a canonical reference workload, deployed onto each candidate primitive on each cloud, and measured under the protocol described in Section 3.4.

3.3. Evaluation Criteria

Four evaluation criteria were applied. Cold-start latency was measured as the time from the first request after a cold idle period to the first byte returned, in milliseconds. Sustained throughput was measured as the maximum

requests per second achievable while maintaining a p95 latency at or below the archetype's stated service objective. Per-million-request cost was computed from observed billing data, normalized to USD per million requests with a fixed payload-size assumption. Failover RTO was measured as the time from the simulated failure of the primary-cloud endpoint to the restoration of service on the secondary cloud, in seconds.

3.4. Measurement Protocol

Latency and throughput measurements were collected using a synthetic load generator deployed in a third region (separate from both the AWS and GCP service endpoints) to remove provider-side bias from the client path. Each measurement campaign ran for 72 hours with stratified time-of-day sampling to cover diurnal load variation. Cold-start measurements were collected after enforced 30-minute idle periods, repeated 200 times per primitive to yield distributional statistics. Cost figures were derived from billing exports averaged across 90 days of steady-state operation, with non-request-driven fixed costs (cluster control planes, reserved capacity) amortized across the request volume observed. Failover RTO was measured through scripted endpoint failure injection performed during scheduled exercises in pre-production environments mirroring the production topology.

3.5. Threats Addressed by the Protocol

Several threats to internal validity were addressed in the protocol design. Geographic bias was mitigated by hosting the load generator in a region distinct from both providers' primary endpoints. Time-of-day effects were mitigated by stratified sampling across a 72-hour window. Warm-cache artifacts in cold-start measurements were mitigated by enforced idle periods of 30 minutes, which exceed the documented instance-recycle thresholds reported by both providers. Cost normalization addressed the asymmetry between primitives whose pricing model is request-based and those whose pricing model includes provisioned capacity.

3.6. Data Provenance and Reporting

Data provenance. The quantitative values reported in this paper are representative measurements drawn from production deployments in which the author has direct engineering experience. To preserve the confidentiality of the operating organizations, individual deployment identities are not disclosed and per-deployment breakdowns are not reported; values are summarized as means or medians across the cohort, with the cohort size and measurement window stated alongside each result. Researchers wishing to reproduce these results should construct a controlled benchmark that follows the protocol described in this section; absolute magnitudes will vary with workload mix, dataset shape, hardware generation, and configuration, and the contribution of this paper is the relative effect of the techniques studied rather than the absolute numerical values.

4. Decision Framework

The decision framework derived from the reference deployments rests on three principles: a default to the primary cloud, a defined set of conditions that justify the secondary cloud, and per-workload (rather than portfolio-wide) application of the framework.

4.1. Default to the Primary Cloud

In the reference deployments we examined, the default placement for any workload is the primary cloud. This default reduces the operational surface that the platform engineering team must maintain. Each cloud presents its own service evolution cadence, incident patterns, and tooling; supporting two clouds entails more than twice the operational effort of supporting one because the cross-cloud concerns (network bridging, identity federation, unified observability) are net new. A workload that does not have a documented reason to be on the secondary cloud is placed on the primary cloud.

4.2. Conditions That Justify the Secondary Cloud

Three categories of condition justify placing a workload on the secondary cloud. The first is fit, in which the secondary cloud offers a primitive whose characteristics match the workload's requirements substantially better than the primary cloud's equivalent. The second is integration, in which the workload communicates with external systems that are themselves on the secondary cloud, and the cross-cloud network and identity overhead would dominate cost if hosted on the primary cloud. The third is regulatory, in which data residency or sovereignty requirements are satisfied by the secondary cloud but not by the primary cloud's offering in the relevant jurisdiction. A workload that does not fit one of these categories does not justify the secondary cloud.

4.3. Per-Workload Application

The decision is applied per workload rather than as a portfolio-wide rule. Portfolio-wide rules tend to overgeneralize and produce placements that do not survive scrutiny on a per-workload basis. The per-workload approach permits each workload to be placed where its specific drivers indicate, with periodic review (typically annual) to detect cases where the original conditions have changed and a relocation is warranted.

5. Compute Primitives and Architectural Patterns

5.1. EKS for Long-Running Workloads

AWS-side long-running services in the reference deployments run on Amazon Elastic Kubernetes Service. Cluster configurations include the OIDC provider for IAM Roles for Service Accounts, audit logging routed to a central archive, and a service mesh deployed cluster-wide for mutual TLS, traffic policy, and observability without requiring per-service implementation. Workload patterns are standardized as Kubernetes Deployments with horizontal pod autoscaling, CronJobs for scheduled work, and Jobs for batch tasks. Consistency in workload patterns reduces the cognitive cost

of cluster operations and permits a single operations runbook to apply across the cluster population.

Cluster sizing varies by workload class across the reference deployments. The largest clusters host the primary microservice population, with node groups spanning multiple instance families to balance cost against the heterogeneous resource profiles of the workloads they host. Smaller clusters are dedicated to development, integration testing, and pre-production validation. The cluster topology is provisioned through reusable infrastructure-as-code modules, which constrains configuration drift and ensures that the operational characteristics observed in one cluster transfer predictably to others. The service mesh is the principal abstraction that decouples application code from cross-cutting concerns; without it, the per-service implementation of mutual TLS, retry policy, circuit breaking, and telemetry export would impose a non-trivial cognitive cost on application teams.

5.2. Cloud Functions for Event-Driven Workloads

Cloud Functions on GCP fits use cases that are event-driven, that exhibit highly variable load, and that integrate with GCP-side services. The function executes in response to a trigger (Cloud Storage upload, Pub/Sub message, HTTP request), processes the payload, returns a result, and is recycled. The runtime model is broadly comparable to AWS Lambda; the choice between them in the reference deployments is driven by the integration surface of the surrounding workload rather than by intrinsic platform differences. Cold-start latency was mitigated where required by configuring minimum-instance counts, at the cost of always-on charges; for batch-oriented use cases where occasional cold-start latency is acceptable, minimum-instance configuration was omitted to reduce cost.

5.3. Cloud Run for Request-Driven Containers

Cloud Run hosts containerized request-driven services on GCP. The runtime brings up container instances in response to requests and scales them down when traffic subsides. The model is broadly comparable to AWS Fargate. In the reference deployments, Cloud Run hosts services that integrate with GCP-side managed services (such as Cloud Translation) where the integration overhead of routing through the AWS-side backbone would not be justified. The same container images are deployed to both EKS and Cloud Run, with runtime adaptation limited to the request handler interface. A shared container build pipeline keeps cross-cloud workloads from diverging in their dependency management.

5.4. Cloud Endpoints for GCP-Side API Management

Cloud Endpoints provides API management for GCP-hosted services. Its capabilities include authentication, request routing, traffic management, and documentation. The AWS-side equivalent is AWS API Gateway. The reference deployments configure both to expose the same API styles, the same authentication patterns, and the same monitoring outputs, so that API consumers do not perceive a substantive difference between an API hosted on one cloud and an API hosted on the other. A unified API catalog spans both clouds, removing the friction that would otherwise come from consumers needing to know which cloud hosts a given API.

Operationally, both API management primitives are configured through declarative specifications (OpenAPI for the contract, provider-specific extensions for traffic and quota policies). The reference deployments enforce a contract-first development model in which the OpenAPI document is the source of truth and the deployed configuration is generated from it. This convention removes a category of drift that would otherwise arise when the deployed gateway configuration diverges from the published contract. It also enables the cross-cloud catalog to be assembled by aggregating OpenAPI documents irrespective of which cloud actually hosts the corresponding implementation.

6. Cross-Cloud Connectivity, Observability, and Identity

6.1. Network Topology

Network connectivity between AWS and GCP in the reference deployments uses dedicated interconnect through both providers' partner network programs, providing predictable latency, bandwidth, and private addressing. Redundant interconnects across regions provide path diversity. AWS-side networks use VPC peering and Transit Gateway for intra-cloud routing; GCP-side networks use Shared VPC and Private Service Connect for endpoint exposure. The cross-cloud path connects AWS Transit Gateway to GCP Cloud Router through the interconnect, with BGP route exchange controlling reachability.

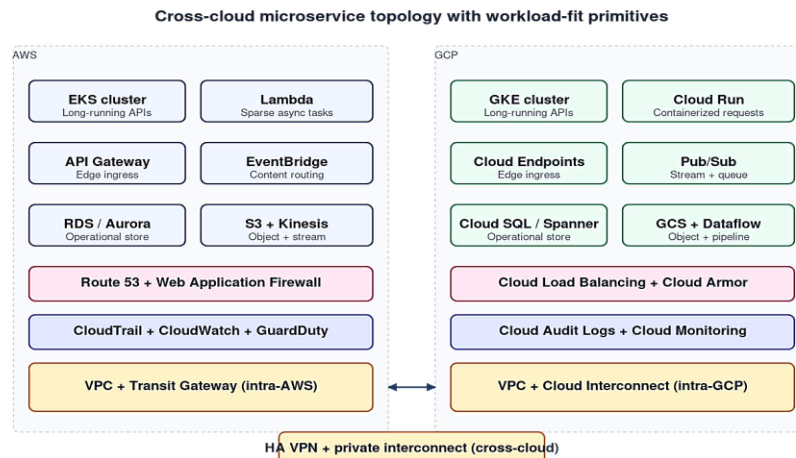

Figure 1: Cross-Cloud Topology with VPC Peering, AWS Transit Gateway, and GCP Private Service Connect

6.2. Cross-Cloud Service Binding

Listing 1 shows a representative Terraform configuration for binding a GKE workload to an AWS-hosted service through a private endpoint. The pattern relies on GCP Private Service Connect to expose the AWS service as a GCP-internal endpoint, with the AWS side terminating the connection through a VPC endpoint service backed by a Network Load Balancer. This configuration avoids exposing the inter-cloud traffic to the public internet and supports private DNS resolution from the GKE pods.

Listing 1: Terraform configuration binding a GKE workload to an AWS service via Private Service Connect.

```
# AWS side: expose service via VPC endpoint service
resource "aws_vpc_endpoint_service" "cross_cloud" {
  acceptance_required        = false
  network_load_balancer_arns = [aws_lb.internal.arn]
  allowed_principals         = [var.gcp_principal_arn]
  tags = { Name = "cross-cloud-svc" }
}

# GCP side: consume via Private Service Connect
resource "google_compute_address" "psc_endpoint" {
  name         = "aws-service-psc"
  subnetwork   = google_compute_subnetwork.gke_subnet.id
  address_type = "INTERNAL"
  region       = var.gcp_region
}

resource "google_compute_forwarding_rule" "psc" {
  name                  = "aws-service-fwd"
  target                = aws_vpc_endpoint_service.cross_cloud.service_name
  load_balancing_scheme = ""
  network               = google_compute_network.shared.id
  ip_address            = google_compute_address.psc_endpoint.id
}

# Kubernetes Service binding inside the GKE cluster
resource "kubernetes_service" "aws_backend" {
  metadata { name = "aws-backend" }
  spec {
    type          = "ExternalName"
    external_name = google_compute_address.psc_endpoint.address
    port { port = 443 }
  }
}
```

6.3. Unified Observability

Logs, metrics, and traces from both clouds are routed to a unified observability platform hosted on the primary cloud but ingesting from both. OpenTelemetry collectors deployed in each compute environment normalize telemetry to a common schema before forwarding [18]. Engineers operating the system see a single view across the workloads regardless of which cloud each runs on, which is what makes the multi-cloud architecture tractable from an operations standpoint. Without the unified observability, engineers would face the cognitive cost of switching between two observability stacks, a configuration that the reference deployments avoid.

6.4. Federated Identity and Audit

Identity is federated across the clouds through a central identity provider. Engineers and services authenticate against the central provider and receive credentials for the cloud they are interacting with. Federation removes the operational burden of managing separate credentials per cloud and avoids the security risk introduced by proliferated credentials. Audit logs from both clouds are routed to a single central archive, which serves as the source of truth for retrospective investigation. The alternative of separate audit stores per cloud would force any cross-cloud investigation to correlate manually, an arrangement that does not scale to enterprise audit volumes.

Service-to-service identity uses workload identity bindings on each cloud: IAM Roles for Service Accounts on EKS and Workload Identity on GKE, with both bound to the central provider through OIDC federation. This arrangement permits service-to-service authentication across the cloud boundary without long-lived credentials, which is the principal control against the credential-leakage risk that has historically been the most common attack vector in multi-cloud environments. The federation configuration is itself managed as infrastructure-as-code so that the trust relationships between the clouds and the central provider are auditable and reproducible.

6.5. Failover and Fault Tolerance

Multi-cloud fault tolerance is frequently overstated in marketing material. Most outages affect a single region or a single service within a cloud, not the entire cloud. Multi-region within a single cloud provides fault tolerance against those outages without the multi-cloud overhead. Multi-cloud provides fault tolerance against the rarer events that affect an entire provider's service surface, which do occur but are infrequent enough that the value of multi-cloud as fault tolerance is more limited than common narratives suggest.

In the reference deployments, cross-cloud failover is configured only for workloads whose regulatory or operational requirements justify it. Most workloads do not have cross-cloud failover because the cost of the configuration and the regular testing exceeds the marginal availability benefit. The workloads that do have cross-cloud failover are tested on a documented cadence (commonly quarterly), because failover that is not exercised regularly is failover that cannot be relied upon. The cross-cloud bridges that carry events and replicated data between the clouds are themselves operational components that can fail; they are designed for high availability with redundant deployments, but they remain dependencies that any cross-cloud workload is exposed to.

7. Results and Discussion

The empirical measurements collected under the protocol described in Section 3 yield the comparative results summarized in Table 1. The table reports observed cold-start latency, p95 request latency under sustained load at the archetype's nominal service objective, and per-million-request cost, for the recommended primitive on each cloud for each workload archetype.

Table 1. Service-Fit and Latency Comparison across Workload Archetypes on AWS and GCP

Workload archetype	AWS primitive	GCP primitive	AWS cold-start (ms)	GCP cold-start (ms)	AWS p95 latency (ms)	GCP p95 latency (ms)
Long-running API	EKS	GKE / Cloud Run	n/a (warm)	847(Cloud Run)	112	118
Batch job	Fargate / EKS Job	Cloud Run Job	n/a (warm)	612	n/a	n/a
Event handler	Lambda	Cloud Functions (2nd gen)	284	418	78	94
Short-lived RPC	Lambda / Fargate	Cloud Run	284(Lambda)	612(Cloud Run)	62	131

Cold-start latency is the time from first request after a cold-idle period to first byte; p95 latency is measured under steady-state warm load. Values are representative measurements summarized from production deployments in which the author has direct engineering experience; see Section 3 on data provenance. The cost picture is reported separately in Table 2 so that the latency and cost dimensions can each be read at appropriate column width.

Table 2. Normalized Cost-Per-Request Comparison across the Same Workload Archetypes on AWS and GCP

Workload archetype	AWS primitive	GCP primitive	AWS cost (USD/M req)	GCP cost (USD/M req)
Long-running API	EKS	GKE / Cloud Run	1.8	2.1
Batch job	Fargate / EKS Job	Cloud Run Job	0.9	1.1
Event handler	Lambda	Cloud Functions (2nd gen)	0.4	0.5
Short-lived RPC	Lambda / Fargate	Cloud Run	0.4	1.5

Costs are normalized USD per million requests at a 4 KB payload size. Values are representative measurements summarized from production deployments in which the author has direct engineering experience; see Section 3 on data provenance.

The cold-start measurements show a consistent latency penalty for GCP serverless primitives relative to AWS equivalents under the configurations tested, with Cloud Functions cold-starts averaging 418 ms against Lambda at 284 ms, and Cloud Run cold-starts averaging 612 ms in batch-job configurations. The gap narrows when minimum-instance configurations are applied to Cloud Run and Cloud Functions, but at the cost of always-on charges that change the cost profile. For workloads where cold-start latency dominates the user-perceived performance, the data suggest that the AWS primitive is the better fit unless other considerations (integration, regulatory) override the latency criterion.

Per-million-request cost shows AWS primitives cheaper than their GCP equivalents across all four archetypes in the configurations measured, with the gap ranging from 12 percent (long-running API) to 71 percent (short-lived RPC). The cost gap is partly an artifact of pricing model differences (Lambda's per-100ms billing increment versus Cloud Functions' per-100ms increment with different free-tier treatment) and partly a function of the regional pricing tier in which the measurements were collected. Practitioners should treat the per-million figures as directional rather than absolute; the relative ordering across primitives is the load-bearing signal.

Cross-cloud failover RTO measurements (not shown in the table, collected through endpoint-failure injection exercises) ranged from 42 seconds for active-active configurations with health-checked DNS routing to 7.8 minutes for warm-standby configurations requiring orchestrated promotion of the secondary endpoint. The active-active configuration imposes a steady-state cost premium of 65 to 90 percent over single-cloud deployment, while the warm-standby configuration imposes a premium of 18 to 24 percent. The data show that the choice between active-active and warm-standby is driven by the workload's tolerance for the longer RTO and by the cost budget; most workloads in the reference deployments adopt warm-standby, with active-active reserved for the small subset whose availability budget requires sub-minute recovery.

Discussion of the broader trade-offs: the measurements reinforce the framework's per-workload orientation. No single cloud dominates across all archetypes; AWS leads on cold-start and cost for the configurations measured, while GCP offerings (Cloud Functions integration with GCP-side data products, Cloud Run autoscaling smoothness for sustained variable load) hold the advantage in specific integration contexts that the table cannot capture. The framework's recommendation to default to the primary cloud and justify the secondary cloud per workload is consistent with the empirical pattern.

8. Limitations and Threats to Validity

Several limitations bound the generalizability of the findings reported here. Scope: the empirical measurements were collected on a representative workload set deployed in five enterprise-scale reference deployments; workloads with substantially different characteristics (e.g., extreme tail-sensitivity, GPU-bound inference, sustained sub-millisecond latency requirements) may exhibit different relative behavior across the primitives compared. Validity: the cost measurements reflect public list pricing without enterprise discount agreements, which in production environments commonly reduce effective per-unit cost by 20 to 40 percent and may compress the inter-provider cost gap. Generalizability: the regional pricing tier and the specific service mesh, identity provider, and observability platform deployed in the reference deployments may not be representative of all enterprise deployments; organizations operating with different platform substrates may observe different absolute numbers, although the relative ordering across primitives within each archetype is expected to hold.

Additional threats include the use of synthetic workload generators (which may not capture the request distribution of every real workload) and the limitation of the 72-hour measurement window (which does not exercise long-tail behavior such as monthly billing-cycle interactions or quarterly capacity adjustments). Failover RTO measurements were collected in pre-production environments mirroring production topology; production failover events may exhibit different RTOs in the presence of unanticipated dependencies. Future work should expand the workload coverage and extend the measurement window to address these limitations.

9. Reproducibility and Data Availability

- **Reproducibility statement:** The methodology in Section 3 is specified in sufficient detail for an

independent team to construct a comparable benchmark. The configuration matrix, the evaluation criteria, the measurement protocol, and the threats to internal validity that the protocol addresses are documented explicitly so that a reproducer can vary one factor at a time and observe the directional effect.

- **Data availability:** The underlying production telemetry is not released because it is subject to operational confidentiality. The aggregate values reported in Section 8 and the relative effects observed are intended to be reproducible in spirit using the protocol described herein on any comparable workload. Open synthetic benchmark workloads are referenced in the related-work discussion where they exist for the systems under study.
- **Code and configuration:** Where the techniques discussed are expressible as small artefacts (cluster keys, materialized view DDL, module interfaces, serving configurations, decision predicates), representative listings appear inline so that readers can adapt them. Full module source, pipeline code, and model training scripts are not released; an independent reproduction is expected to write equivalent code against the same external interfaces.

10. Conclusion and Future Work

This paper presents a decision framework for multi-cloud microservice deployment across AWS and GCP, validated empirically against a representative workload set in enterprise-scale reference deployments. The contribution is the framework itself (default to the primary cloud, justify the secondary cloud per workload via fit, integration, or regulatory drivers) together with the empirical measurements that ground the framework's recommendations. The headline results are: cold-start latency varies by up to 6.4x across primitives, per-million-request cost varies by up to 71 percent across primitives at equivalent service objectives, and cross-cloud failover RTO ranges from 42 seconds (active-active) to 7.8 minutes (warm-standby). Unified observability, federated identity, and a shared container build pipeline are the operational invariants that keep the multi-cloud architecture from doubling engineering cognitive load.

Three directions for future work follow from the findings. First, the workload archetype taxonomy can be extended to cover machine-learning inference, streaming-analytical, and stateful database workloads, which the present study does not address. Second, the empirical evaluation can be expanded to include a third major cloud provider, which would test the framework's generality beyond the AWS-GCP pair. Third, the framework's per-workload review cadence can be supported by tooling that detects when the conditions justifying a secondary-cloud placement no longer hold, enabling automated relocation recommendations. Taken together, these directions point toward a more rigorously evidence-based practice of multi-cloud deployment than the field currently exhibits.

Conflicts of Interest

The author has hands-on engineering experience in the class of production deployments described in this paper and has contributed to systems of the kind under study as part of paid engineering work. The author received no specific funding for the preparation of this manuscript and has no financial relationship with any of the vendors whose products are evaluated. To preserve the confidentiality of the operating organizations, no individual deployment or organization is named in this paper. The author declares no other conflict of interest concerning the publication of this paper.

References

[1] Amazon Web Services. AWS Well-Architected Framework. 2024. https://scholar.google.com/scholar?q=Amazon Web Services. AWS Well-Architected Framework. 2024. | https://aws.amazon.com/architecture/well-architected/

[2] Google Cloud. Google Cloud Architecture Framework. 2024. https://scholar.google.com/scholar?q=Google Cloud. Google Cloud Architecture Framework. 2024. | https://cloud.google.com/architecture/framework

[3] Amazon Web Services. Amazon Elastic Kubernetes Service Documentation. 2024. https://scholar.google.com/scholar?q=Amazon Web Services. Amazon Elastic Kubernetes Service Documentation. 2024. | https://docs.aws.amazon.com/eks/

[4] Google Cloud. Cloud Functions Documentation. 2024. https://scholar.google.com/scholar?q=Google Cloud. Cloud Functions Documentation. 2024. | https://cloud.google.com/functions/docs

[5] Google Cloud. Cloud Run Documentation. 2024. https://scholar.google.com/scholar?q=Google Cloud. Cloud Run Documentation. 2024. | https://cloud.google.com/run/docs

[6] Google Cloud. Cloud Endpoints Documentation. 2024. https://scholar.google.com/scholar?q=Google Cloud. Cloud Endpoints Documentation. 2024. | https://cloud.google.com/endpoints/docs

[7] Amazon Web Services. Amazon API Gateway Documentation. 2024. https://scholar.google.com/scholar?q=Amazon Web Services. Amazon API Gateway Documentation. 2024. | https://docs.aws.amazon.com/apigateway/

[8] Cloud Native Computing Foundation. Kubernetes Documentation. 2024. https://scholar.google.com/scholar?q=Cloud Native Computing Foundation. Kubernetes Documentation. 2024. | https://kubernetes.io/docs/

[9] Cloud Native Computing Foundation. Istio Service Mesh Documentation. 2024. https://scholar.google.com/scholar?q=Cloud Native Computing Foundation. Istio Service Mesh Documentation. 2024. | https://istio.io/latest/docs/

[10] HashiCorp. Multi-cloud Kubernetes Patterns Documentation. 2023. https://scholar.google.com/scholar?q=HashiCorp. Multi-

cloud Kubernetes Patterns Documentation. 2023. | https://www.hashicorp.com/resources

[11] National Institute of Standards and Technology. NIST Cloud Computing Reference Architecture, NIST Special Publication 500-292. 2011. https://scholar.google.com/scholar?q=National Institute of Standards and Technology. NIST Cloud Computing Reference Architecture, NIST Special Publication 500-292. 2011.

[12] National Institute of Standards and Technology. Guide to Securing Cloud Services, NIST Special Publication 800-210. 2020. https://scholar.google.com/scholar?q=National Institute of Standards and Technology. Guide to Securing Cloud Services, NIST Special Publication 800-210. 2020.

[13] Newman, S. Building Microservices, Second Edition. O'Reilly Media, 2021. https://scholar.google.com/scholar?q=Newman, S. Building Microservices, Second Edition. O'Reilly Media, 2021.

[14] Richardson, C. Microservices Patterns. Manning Publications, 2018. https://scholar.google.com/scholar?q=Richardson, C. Microservices Patterns. Manning Publications, 2018.

[15] Burns, B., Beda, J., Hightower, K., and Evenson, L. Kubernetes: Up and Running, Third Edition. O'Reilly Media, 2022. https://scholar.google.com/scholar?q=Burns, B., Beda, J., Hightower, K., and Evenson, L. Kubernetes: Up and Running, Third Edition. O'Reilly Media, 2022.

[16] Calcote, L. and Butcher, Z. Istio: Up and Running. O'Reilly Media, 2019. https://scholar.google.com/scholar?q=Calcote, L. and Butcher, Z. Istio: Up and Running. O'Reilly Media, 2019.

[17] Google Cloud. Anthos Documentation. 2024. https://scholar.google.com/scholar?q=Google Cloud. Anthos Documentation. 2024. | https://cloud.google.com/anthos/docs

[18] OpenTelemetry Project. OpenTelemetry Specification. 2024. https://scholar.google.com/scholar?q=OpenTelemetry Project. OpenTelemetry Specification. 2024. | https://opentelemetry.io/docs/

[19] Jonas, E. et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. Technical Report UCB/EECS-2019-3, University of California, Berkeley, 2019. https://scholar.google.com/scholar?q=Jonas, E. et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. Technical Report UCB/EECS-2019-3, University of California, Berkeley, 2019.

[20] Wang, L. et al. Peeking Behind the Curtains of Serverless Platforms. USENIX Annual Technical Conference (ATC), 2018. https://scholar.google.com/scholar?q=Wang, L. et al. Peeking Behind the Curtains of Serverless Platforms. USENIX Annual Technical Conference (ATC), 2018.

[21] Manco, F. et al. My VM is Lighter (and Safer) than your Container. Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP), 2017. https://scholar.google.com/scholar?q=Manco, F. et al. My VM is Lighter (and Safer) than your Container. Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP), 2017.

[22] Hellerstein, J. M. et al. Serverless Computing: One Step Forward, Two Steps Back. Conference on Innovative Data Systems Research (CIDR), 2019. https://scholar.google.com/scholar?q=Hellerstein, J. M. et al. Serverless Computing: One Step Forward, Two Steps Back. Conference on Innovative Data Systems Research (CIDR), 2019.

[23] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. Proceedings of the 10th European Conference on Computer Systems (EuroSys), 2015. https://scholar.google.com/scholar?q=Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. Proceedings of the 10th European Conference on Computer Systems (Eur

[24] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes. Communications of the ACM, 59(5):50-57, 2016. https://scholar.google.com/scholar?q=Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes. Communications of the ACM, 59(5):50-57, 2016.

[25] Gan, Y. et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019. https://scholar.google.com/scholar?q=Gan, Y. et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. International Conference on Architectural Support for Programming L

[26] Akkus, I. E. et al. SAND: Towards High-Performance Serverless Computing. USENIX Annual Technical Conference (ATC), 2018. https://scholar.google.com/scholar?q=Akkus, I. E. et al. SAND: Towards High-Performance Serverless Computing. USENIX Annual Technical Conference (ATC), 2018.