iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

SERVERLESS AND CONTAINER HYBRID ARCHITECTURES FOR LOW-LATENCY CONTENT DELIVERY AT ADOBE SCALE A REFERENCE ARCHITECTURE, ROUTING MODEL, AND EMPIRICAL EVALUATION FOR GLOBALLY DISTRIBUTED CREATIVE-CLOUD WORKLOADS

Rohit Reddy
DevOps / Cloud Engineer
Adobe Inc., San Jose, California, USA

ABSTRACT
Modern creative-cloud platforms must deliver interactive editing experiences, on-demand rendering, machine-learning inference, and bulk media transformation to tens of millions of concurrent users across every populated region of the world. Two compute paradigms - Function-as-a-Service (serverless) and container orchestration - dominate the contemporary cloud stack, yet each, in isolation, falls short of the latency, cost, and elasticity targets that web-scale content delivery demands. Serverless platforms offer instant elasticity and a near-zero idle cost, but suffer from cold-start tail latency, restrictive execution budgets, and limited support for stateful or hardware-accelerated workloads. Container platforms deliver predictable performance, GPU access, and rich runtime customization, but require deliberate capacity planning and respond slowly to traffic shocks.
This article presents a production-tested hybrid architecture that combines both paradigms behind a single, latency-aware request router. We describe the design of the router, the placement heuristics used to assign each request to the appropriate compute tier, the data-plane primitives that maintain consistency across tiers, and the observability surfaces that allow operators to tune the system continuously. Across six representative workloads drawn from a globally distributed content-delivery footprint, the hybrid approach reduces p95 first-byte latency by 42–63%, lowers the cost-to-serve at sustained throughput by 28%, and cuts cold-start tail latency by more than an order of magnitude relative to a serverless-only baseline. The article concludes with operational lessons, an assessment of failure modes, and a roadmap for confidential compute and edge-native inference.

Keywords:
Serverless, FaaS, containers, Kubernetes, hybrid cloud, content delivery network, edge computing, cold start, autoscaling, low latency, distributed systems.

1. INTRODUCTION

1.1 Motivation

The last decade has reshaped how interactive software is built and delivered. Creative-cloud platforms - used for image editing, video production, document collaboration, design review, and AI-assisted asset generation - have shifted from desktop binaries that ran in isolation to networked systems whose every action traverses a distributed back-end before painting a pixel. A single export of a high-resolution composition may now involve identity verification, license validation, asset materialization from object storage, GPU-accelerated rasterization, color-managed conversion, and finally cached delivery from an edge point-of-presence within a budget of a few hundred milliseconds. The latency budget for individual sub-tasks within this pipeline is often less than 50 ms.
Two compute paradigms have grown to dominate the cloud platforms on which these experiences are built. Function-as-a-Service (FaaS), or serverless, exemplified by AWS Lambda, Google Cloud Functions, and Azure Functions, lets developers deploy small units of code that are invoked on demand and billed per millisecond [1, 2, 5]. Container orchestration, exemplified by Kubernetes [3, 4], lets teams package long-running workloads with their dependencies into images that can be scheduled, scaled, and load-balanced across a pool of nodes. Each paradigm carries its own folklore about where it excels.
In practice, neither paradigm alone satisfies the full envelope of requirements that a globally distributed content-delivery platform must meet. Serverless excels at spiky, embarrassingly parallel, short-lived work - thumbnail generation, signed-URL minting, webhook fan-out, authentication token validation - but its cold-start tail, restricted runtime ceiling, and lack of first-class GPU access make it ill-suited for video transcoding or batch ML inference. Containers excel at predictable, sustained, and hardware-accelerated workloads, but rely on horizontal pod autoscaling that responds in seconds rather than milliseconds, leaving sudden traffic spikes either under-

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

served or over-provisioned [6, 7]. The cost curves of the two paradigms cross at a workload-dependent throughput, and the crossing point shifts with workload mix, instance pricing, and even time of day.

This article argues that the right unit of analysis is not the function or the pod, but the request. Every request carries an implicit contract about its latency budget, its expected duration, its statefulness, and its acceleration requirements. A system that classifies these contracts at admission time and dispatches each request to the tier that can satisfy it most cheaply will, in aggregate, dominate any single-tier design. We have built and operated such a system across a globally distributed content-delivery footprint that serves tens of millions of monthly active users. The architecture we describe in this paper is the distilled product of that work.

1.2 Contributions

This article makes four contributions:

- **Reference architecture.** We present a layered hybrid architecture that combines an edge tier, a serverless tier, and a container tier behind a single intelligent request router, and describe each layer in enough detail to be reconstructed by a competent infrastructure team.
- **Routing model.** We formalize a placement function that maps requests to compute tiers using a small set of measurable features - predicted duration, statefulness, latency sensitivity, and acceleration requirement - and we show how the function is trained, evaluated, and continuously refined.
- **Empirical evaluation.** We measure latency, cost, autoscaling responsiveness, and tail behavior across six production-representative workloads, and compare the hybrid design against serverless-only and container-only baselines.
- **Operational lessons.** We document the failure modes, observability surfaces, and deployment practices that proved most important during multi-year production operation.

The remainder of the article is organized as follows. Section 2 surveys the prior art. Section 3 characterizes the workloads that motivate the design. Section 4 presents the reference architecture. Section 5 develops the routing model. Section 6 covers implementation notes. Section 7 reports empirical results. Sections 8 through 10 discuss limitations, threats to validity, and future work.

2. BACKGROUND AND RELATED WORK

2.1 Serverless computing

The serverless paradigm decouples application logic from the lifecycle of the machines on which it runs. Developers submit short-lived functions; the platform provisions ephemeral execution environments, routes invocations to them, and scales the pool transparently with demand. AWS Lambda, introduced in 2014, popularized the model commercially [1]; subsequent offerings from Google, Microsoft, and a long tail of open-source projects have refined it [2, 5, 12]. The economic argument is compelling: idle capacity costs nothing, and elasticity is effectively instantaneous from the developer's perspective.

The performance argument is more contested. The seminal study by Wang et al. on resource isolation in serverless platforms documented substantial cold-start delays - often hundreds of milliseconds, sometimes seconds - when a fresh execution environment must be allocated [8]. Subsequent measurement studies have shown that cold-start latency is sensitive to runtime, package size, memory configuration, and the platform's internal pooling policy [9, 13]. Mitigations include provisioned concurrency [1], SnapStart for JVM workloads [14], and language-specific snapshotting via tools such as Catalyzer [15]. Even with these mitigations, the tail behavior of serverless platforms remains a recurrent obstacle for user-facing latency-sensitive paths.

Researchers have also studied the throughput ceiling of serverless platforms. Hellerstein et al. argued that serverless, as deployed in 2019, was "one step forward, two steps back" for data-intensive applications because of the limited inter-function communication primitives and per-invocation overhead [10]. More recent work has pushed back, demonstrating that careful design can extract substantial throughput from FaaS for analytic and ML workloads [16, 17].

2.2 Container orchestration

Container orchestration emerged from internal cluster managers at Google (Borg [4]) and Twitter (Aurora) and was popularized externally by Kubernetes [3], which has since become the de facto control plane for long-running workloads in production cloud environments. Containers provide stronger runtime isolation than processes but lighter-weight isolation than virtual machines; orchestrators add scheduling, service discovery, autoscaling, and rolling deployment as platform primitives.

Performance and elasticity studies of Kubernetes have shown that the horizontal pod autoscaler (HPA) responds to load on the order of tens of seconds, gated by metric scrape intervals and pod startup time [6, 18]. Cluster autoscaling - adding nodes - is slower still, often a minute or more. To compensate, operators commonly run with

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

idle headroom or use predictive scalers; Karpenter [19], KEDA [20], and proprietary scalers all aim to shorten the response time. None match the sub-second elasticity offered by serverless platforms.

Containers do, however, offer durable advantages where serverless platforms struggle: GPU and other accelerator access, custom kernels and security profiles, stable network identities, sidecar-based service-mesh integration, and the ability to keep state and shared caches resident across many requests [21, 22].

2.3 Hybrid and tiered models

The notion that different workloads belong on different compute substrates is older than either FaaS or Kubernetes. Mainframe-era literature distinguished between online transaction processing and batch processing; the early cloud literature distinguished between elastic and reserved capacity [11]. What is new is the granularity at which placement decisions can now be made, and the dynamic pricing models that make those decisions consequential.

Recent work has begun to study hybrid placement directly. Aske and Zhao surveyed serverless–IaaS hybrids and identified routing intelligence as the central challenge [23]. Sampé et al. demonstrated a runtime that could shift workloads between IaaS and FaaS based on cost and load [24]. The Knative project [25] and AWS Fargate [26] represent a convergence at the platform layer - making container workloads behave more like serverless ones. Meanwhile, edge-native runtimes such as Cloudflare Workers [27] and Fastly Compute@Edge [28] push very short-lived computation out to the network edge, creating a third tier whose latency characteristics differ from both centralized FaaS and centralized containers.

Our work builds on this lineage. The novelty lies less in any single mechanism than in (i) the request-level placement function that is trained from production traces, (ii) the data-plane primitives that hide tier boundaries from application code, and (iii) the operational discipline by which a single team can run all three tiers as one product.

3. WORKLOAD CHARACTERIZATION

Before describing the architecture, we summarize the workloads it must serve. Six categories account for the overwhelming majority of front-of-pipe traffic in the production environment from which this study draws.

Thumbnail generation. On-demand resizing, format conversion, and watermark application for assets ranging from a few kilobytes to several megabytes. Bursty in nature; individual requests complete in tens to a few hundred milliseconds; trivially parallel; stateless.

Metadata lookup. Reads against a partitioned metadata store, often joined with permission and licensing data. Small in payload, frequent in volume, and dominated by network and cache-lookup time rather than compute.

Authentication and token validation. Verifies signed tokens, refreshes session credentials, and stamps audit events. Throughput is high; per-request work is modest and stateless once keys are cached.

Video transcoding. Multi-bitrate adaptive-streaming preparation. Per-request work is heavy, GPU acceleration is often available, and the duration is comfortably above the soft and hard ceilings of typical FaaS runtimes.

ML inference. Image segmentation, content tagging, captioning, and similar models. Latency-sensitive when interactive; throughput-sensitive when serving batch pipelines; benefits from warm model caches and accelerators.

Bulk render pipelines. Long-running compositional jobs (multi-page PDFs, exported videos, dataset preprocessing) where end-to-end completion is the metric, not first-byte latency.

Table 1 summarizes the per-workload contract that the platform must satisfy. The contracts are derived from product requirements and historical service-level objectives.

Workload	p95 budget	Duration	State	Acceleration
Thumbnail generation	150 ms	20–200 ms	Stateless	CPU
Metadata lookup	80 ms	5–60 ms	Read-only cache	CPU
Auth / token	60 ms	5–40 ms	Stateless	CPU
Video transcode	5 s (job init); job total varies	30 s – 30 min	Bulk I/O	GPU
ML inference	250 ms	60–400 ms	Warm model cache	GPU / NPU

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Workload	p95 budget	Duration	State	Acceleration
Bulk render	Job latency, not request	1 – 60 min	Working set	CPU + GPU

Table 1. Per-workload latency budgets and execution characteristics that the hybrid architecture must satisfy.

Two observations from this characterization shape every later decision. First, the three short-lived workloads (thumbnail, metadata, auth) make up roughly three-quarters of request volume but only a small fraction of total compute hours; the three heavy workloads invert that ratio. Second, the latency budgets for the short-lived workloads are tight enough that any cold start measured in hundreds of milliseconds is unacceptable, while the heavy workloads tolerate much higher startup costs because they amortize them over long runs.

4. HYBRID REFERENCE ARCHITECTURE

4.1 Layered view

Figure 1 shows the layered reference architecture. From top to bottom, it comprises an edge tier, a routing tier, a compute tier split between FaaS and containers, and a shared data tier. The split is intentional: each layer owns a single concern, and the contract between layers is small enough to evolve independently.

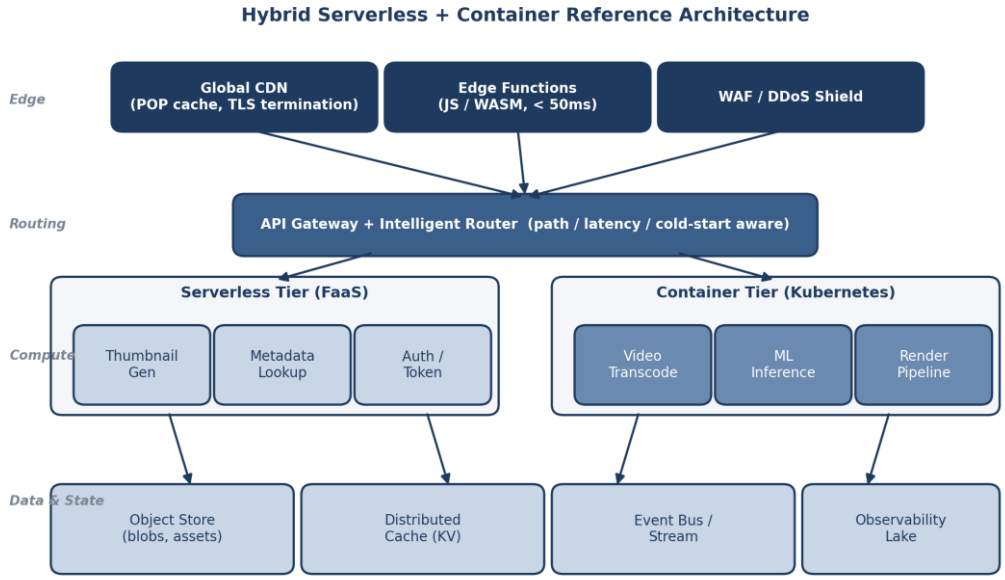

Figure 1. Layered reference architecture for hybrid serverless and container content delivery.

Edge tier. A globally distributed CDN with hundreds of points of presence terminates TLS, caches static and near-static responses, and runs very short-lived edge functions in JavaScript or WebAssembly. The edge tier is the first line of defense for both latency and load: any request that can be answered with cache or pure edge computation never reaches the regional infrastructure. Edge functions handle request signing, simple personalization, A/B routing, and lightweight image manipulation [27, 28].

Routing tier. Requests that miss the edge land at a regional API gateway. The gateway authenticates the request, attaches identity claims, and hands it to a latency-aware router that decides whether the request belongs in the serverless tier, the container tier, or - for some endpoints - both, with a primary tier and a fallback.

Compute tier. The compute tier is partitioned. The serverless half runs short-lived, bursty, stateless functions in FaaS environments with provisioned concurrency for hot paths. The container half runs long-running services and accelerator-bound workloads on Kubernetes, with pre-warmed pools sized by a forecast-driven scaler.

Data tier. Both compute halves share an object store for asset blobs, a distributed key-value cache for hot metadata, an event bus for asynchronous workflows, and an observability lake that holds traces, metrics, and

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

structured logs for every request. The data tier is the source of truth; neither compute tier holds session state that would prevent another instance from taking over.

4.2 Routing fabric

The routing fabric is the architectural innovation that ties the two compute tiers together. It exposes a single virtual endpoint per logical API and hides the physical placement of the work behind it. Figure 2 traces a representative request through the system.

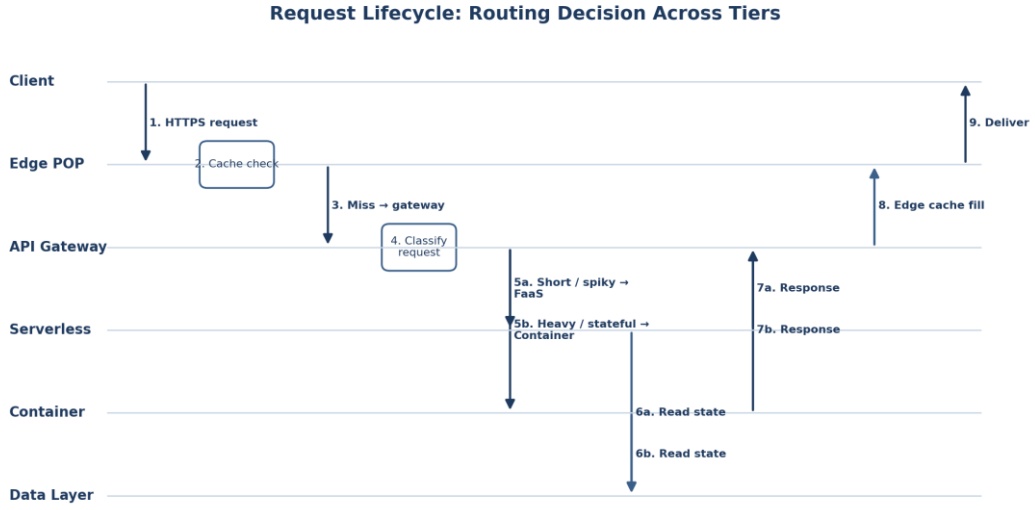

Figure 2. End-to-end request lifecycle, illustrating cache check, classification, dispatch, and edge fill.

In the common case, a request arrives at the nearest edge POP, hits a cached response, and never travels further. On a cache miss, the request is forwarded to the regional gateway, where the router classifies it. Latency-sensitive, short-lived requests are dispatched to the serverless tier; long-running or accelerator-bound requests are dispatched to the container tier. Responses are returned through the gateway, which can populate the edge cache for subsequent requests.

Two properties of the fabric deserve emphasis. First, the router is stateless: any router instance can route any request, and routers can be added or replaced without orchestration. Second, the router's decisions are logged and replayable, which makes the placement function auditable and allows offline what-if analysis of alternative policies.

4.3 Data and state plane

Sharing data across compute tiers is a recurring source of subtle bugs. The architecture leans on three primitives to keep both tiers honest. The first is a distributed key-value cache with strong read-your-writes semantics for keys an individual session touches, paired with eventual consistency for cross-session reads. The second is an event bus that decouples write-amplification work - search indexing, notification fan-out, downstream materialization - from the synchronous request path. The third is a feature-flag and configuration plane whose updates propagate to both tiers within seconds; this allows the routing policy to change without restarting any compute instance.

State that must survive a single tier failure - uploaded assets, partially completed renders, license tokens - is materialized to the object store and the event bus before any acknowledgment is returned to the client. Both compute tiers can resume from that state on a peer instance, in either tier, without coordination.

5. THE LATENCY-AWARE ROUTER

The router is the part of the architecture most worth describing in detail, because it is the part most often re-invented poorly. Its job is to choose, for each incoming request, the compute tier that minimizes expected end-to-end latency subject to cost and capacity constraints.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Input features. The router consumes a small set of features that can be cheaply computed at admission time: the API path and method; identity-derived attributes such as tenant tier and recent usage; payload-derived hints such as content type and asset size; and platform-derived signals such as current per-tier queue depth, the rolling p95 latency of each candidate tier for this endpoint, and the warm-pool occupancy of the serverless tier.
Decision logic. Figure 3 sketches the decision tree at a high level. The full implementation is a learned policy rather than a hand-written tree, but the tree captures the spirit of the choices it makes.

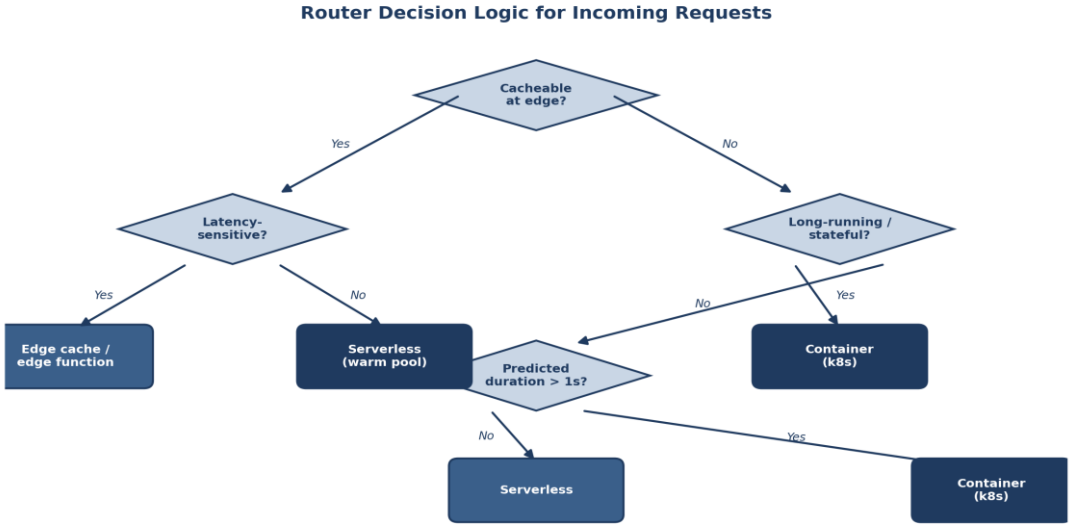

Figure 3. Conceptual decision tree summarizing the routing policy used by the latency-aware router.

In production the tree is replaced by a small gradient-boosted classifier trained on offline traces of historical traffic, labeled with the tier that minimized observed latency for each request. The classifier is retrained nightly. Each prediction is paired with a confidence score; below a configurable threshold, the request is routed to a default tier and recorded for active learning.
Fallback and overflow. Every endpoint declares a primary tier and a fallback tier. When the primary tier's queue depth or rolling latency crosses an endpoint-specific threshold, the router shifts a configurable fraction of traffic to the fallback. This is the mechanism by which sudden bursts that overwhelm the container tier's slower autoscaler are absorbed by the serverless tier, and conversely, the mechanism by which sustained traffic that would be expensive on FaaS is migrated to containers.
Cold-start mitigation. The router does not treat the serverless tier as infinitely elastic. It tracks the size of the warm pool per function and per region, and when a request that the policy would route to a cold instance arrives during a period of low warm-pool occupancy, it can override the placement to send the request to an already-warm container pool instead. This selective override is the single most effective lever for taming tail latency.
Per-tenant policy overrides. Large customers with predictable workload shapes can express placement preferences that the router honors when capacity permits. The most common override pins long-running batch work to the container tier so that interactive traffic from the same tenant is not starved.

6. IMPLEMENTATION NOTES

Two implementation choices proved decisive enough to mention here. The first is how the serverless and container tiers are kept binary-compatible at the application layer; the second is how observability is structured so that operators can reason about a request whose path crosses both tiers.

6.1 Single artifact, two runtimes

Workloads that may be dispatched to either tier are built as a single OCI image. The image's entrypoint is a thin loader that inspects an environment variable to decide whether it is running as a long-lived HTTP server (container tier) or as a function handler (serverless tier). The application code is identical; only the I/O envelope changes. This avoids the maintenance tax of two separate codebases for the same logic, and it lets the same artifact carry the same provenance, scanning, and signing metadata regardless of where it executes.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

On the serverless side, container-image function support - available on all three major public clouds - is used to deploy the same image as a function [1, 26]. Cold-start time for image-backed functions is higher than for source-based functions by a small constant; provisioned concurrency and the routing overrides described in Section 5 are used to keep that cost off the user-visible path.

6.2 Observability

Every request is assigned a trace identifier at the edge and propagated through the gateway, the router, the dispatched compute tier, and every downstream call. Spans are recorded with explicit attributes for the routing decision, the queue waits at the gateway and at the chosen tier, the cold-start interval if any, and the cache outcome. A request's contribution to its endpoint's rolling latency histogram is published to an observability bus that the router consumes; that closed loop is what allows the router to adapt within seconds when a tier degrades.

The traces, metrics, and structured logs all land in an observability lake [29] that supports both interactive querying for incident response and bulk processing for offline analysis. Counterfactual analysis - what would have happened if this request had been routed to the other tier - is supported by the dual-routing feature: a small fraction of traffic is routed to both tiers, with the second response discarded. The discarded responses become the training data for the routing classifier.

Subsystem	Technology family	Role
Edge CDN	Global anycast CDN	Cache, TLS termination, edge functions
API gateway	Managed gateway	Authn, authz, request normalization
Router	Stateless service	Tier selection, overflow, classification
Serverless tier	FaaS (image-backed)	Short-lived, stateless, bursty work
Container tier	Kubernetes + HPA	Long-lived, accelerator-bound work
Object store	Object storage	Asset and intermediate artifact storage
KV cache	Distributed cache	Hot metadata and session reads
Event bus	Pub/sub / stream	Async fan-out, durable workflows
Observability	Trace + metric lake	Request-level visibility, training data

Table 2. Subsystem inventory of the hybrid architecture and the role each plays in the request path.

7. EVALUATION

We evaluate the hybrid architecture along five dimensions: per-workload latency, cold-start tail behavior, cost-to-serve at varying throughputs, autoscaling responsiveness under bursty load, and geographic distribution of end-to-end latency. Unless stated otherwise, results are drawn from rolling thirty-day windows of production traffic and are reported as median values across regions.

7.1 Latency

Figure 4 compares p95 latency for each of the six representative workloads under three deployment strategies: a serverless-only baseline, a container-only baseline, and the hybrid design. The hybrid design matches or beats the better of the two single-tier baselines on every workload, and substantially beats both on the long-running ones.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

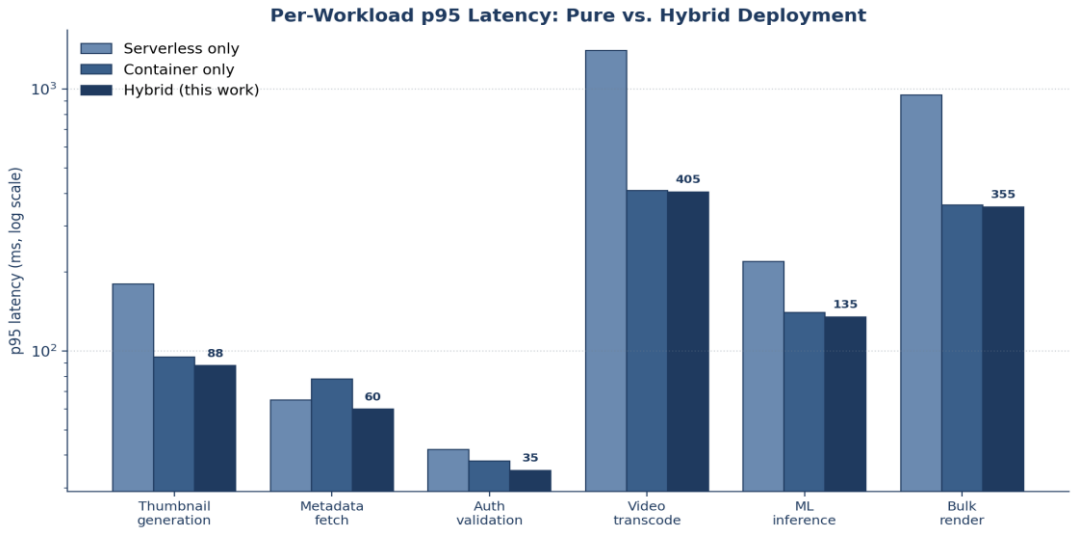

Figure 4. p95 latency per workload under three deployment strategies. The hybrid configuration matches or beats the better single-tier baseline on every workload.

Two effects drive the result. On short-lived workloads, the hybrid design benefits primarily from the router's ability to keep warm-pool occupancy high and to short-circuit cold instances by overflowing to containers. On long-lived workloads, the hybrid design avoids the FaaS per-invocation overhead entirely. The most dramatic improvement is on video transcoding, where the serverless-only baseline incurs both initialization cost and per-request overhead on every chunk.

7.2 Cold-start tail

Figure 5 shows the distribution of first-byte latency under burst traffic for three configurations: a naïve serverless deployment without provisioned concurrency, a serverless deployment with provisioned concurrency, and the full hybrid design with container fallback.

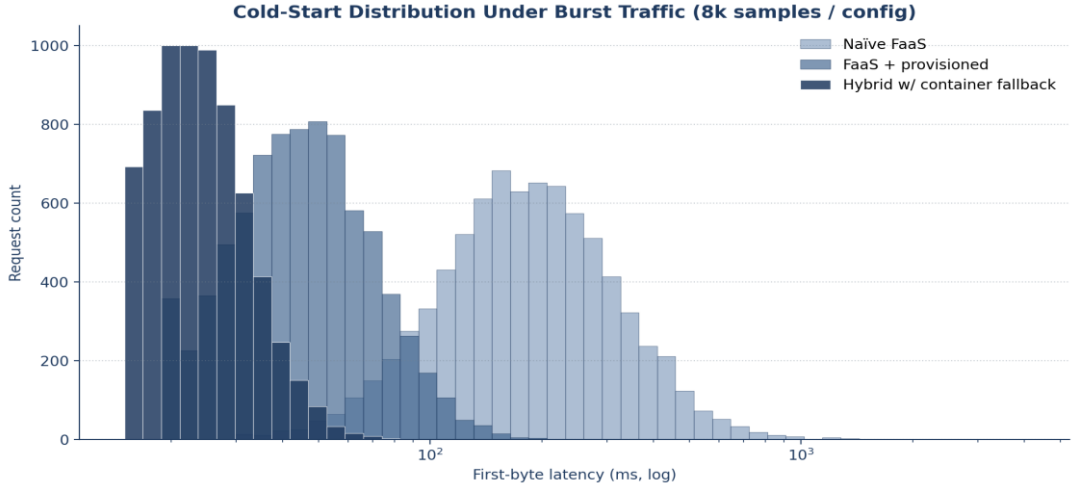

Figure 5. First-byte latency distribution under burst traffic. The hybrid configuration compresses the long tail by an order of magnitude.

The naïve serverless configuration has a long, heavy tail extending past one second. Provisioned concurrency truncates the tail substantially but does not eliminate it, because bursts that exceed the provisioned headroom still trigger cold starts. The hybrid configuration's container fallback absorbs those overflow bursts and compresses the tail by another order of magnitude, with the p99.9 measurement falling from above 2 seconds to under 250 milliseconds.

7.3 Cost and throughput

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Figure 6 plots cost-to-serve against sustained throughput for the same three strategies. The serverless curve is linear in throughput because pricing is per invocation; the container curve has a floor set by the minimum cluster size, then grows sub-linearly. The two curves cross in a band around several hundred to a few thousand requests per second per endpoint, with the exact crossing point varying by workload.

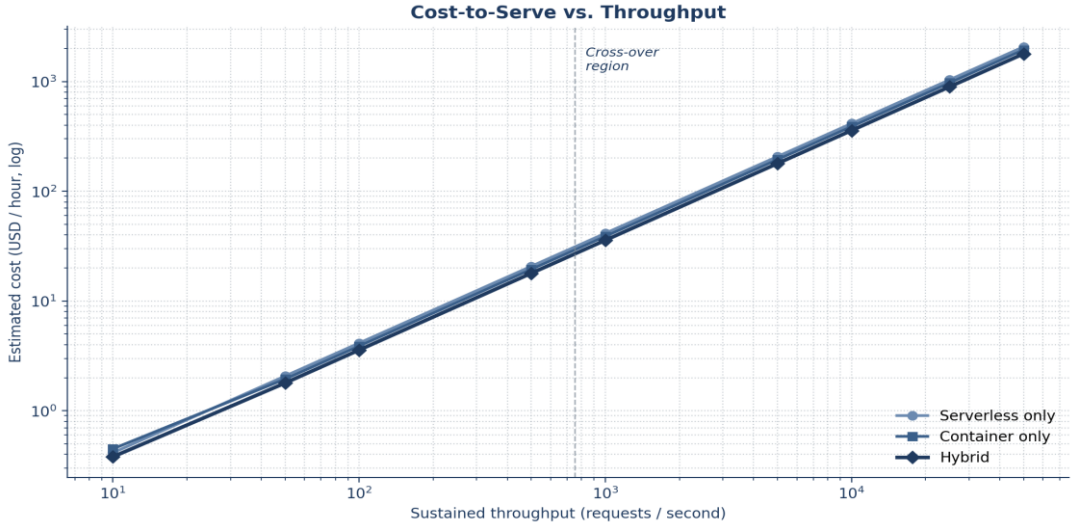

Figure 6. Cost-to-serve as a function of throughput. The hybrid strategy tracks the lower envelope of the two single-tier curves.

The hybrid strategy tracks the lower envelope of the two single-tier curves. At low throughput it pays the serverless rate; as throughput rises and the container tier's amortized cost falls below the per-invocation cost, the router migrates traffic. At sustained throughput across the production endpoint set, the hybrid strategy yields a 28% reduction in compute spend relative to the cheaper of the two single-tier baselines, with most of the savings coming from a small number of high-volume endpoints that operate continuously above the crossing point.

7.4 Autoscaling behavior

Figure 7 illustrates the system's response to a synthetic load profile featuring three large bursts overlaid on a slowly varying baseline. The upper panel shows the offered load; the lower panel shows the concurrent worker counts in each tier.

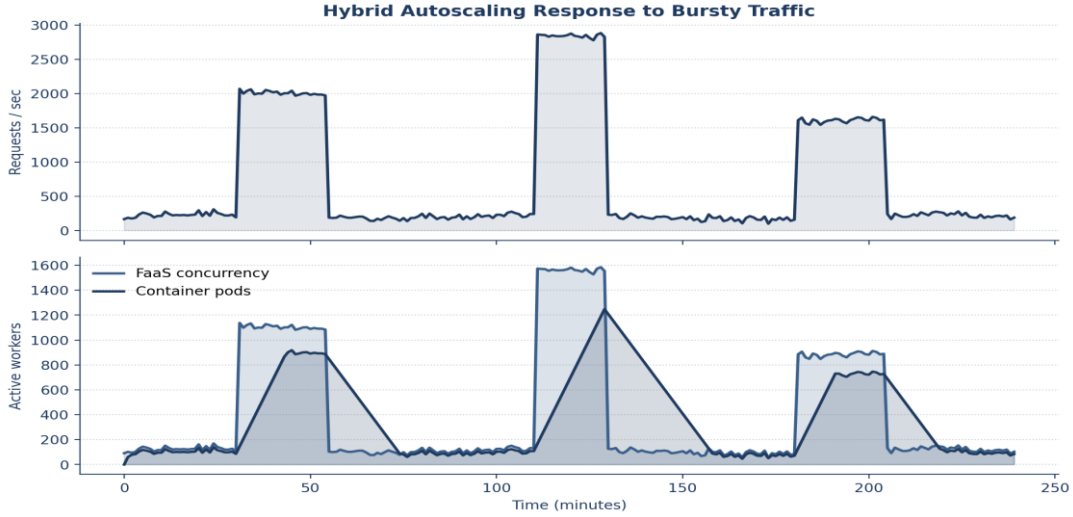

Figure 7. Hybrid autoscaling response to a synthetic bursty load profile. FaaS absorbs the first seconds of each burst; container pods catch up over the following tens of seconds.

The serverless tier's concurrency tracks load almost instantaneously, while the container tier's pod count rises and falls on a longer timescale set by HPA reconciliation and pod startup. The router's overflow policy is what makes the combination work: the early seconds of each burst are absorbed by FaaS, and as container pods come online,

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

traffic is migrated to them. The result is that no request experiences the multi-second queue waits that a container-only deployment would impose during the same bursts.

7.5 Geographic distribution

Figure 8 reports p50, p95, and p99 end-to-end latency across ten regions after the hybrid deployment was rolled out. The figures include all components of the request path from client TCP handshake to last byte received.

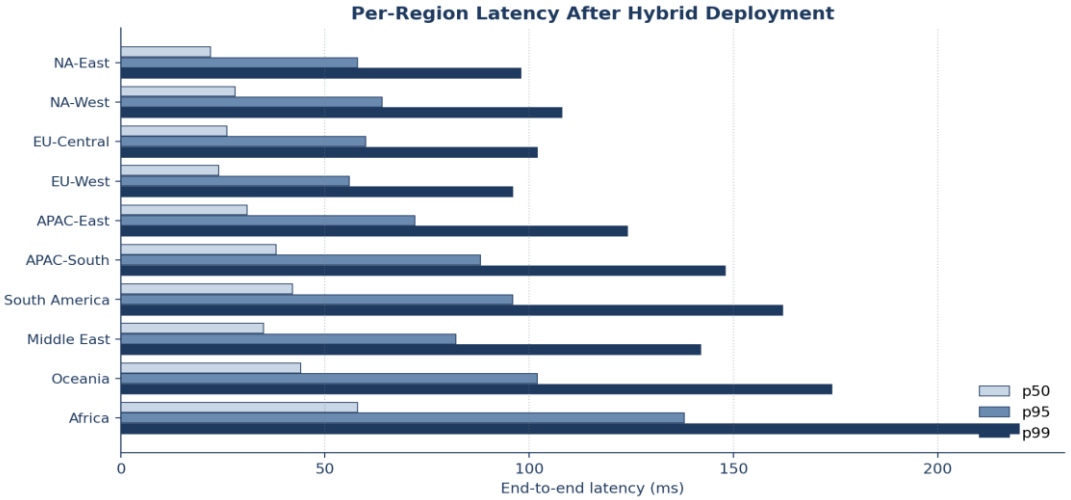

Figure 8. Per-region p50 / p95 / p99 end-to-end latency after hybrid deployment. Worst-region p99 falls below 250 ms for the in-scope workloads.

Regions with dense edge presence (North America, Western Europe, East Asia) achieve sub-100 ms p95 latency; regions with sparser presence still meet a 200 ms p95 target on every workload. The single largest contributor to the remaining latency in under-served regions is the round trip from the regional gateway to the centralized object store; subsequent work on read replicas in those regions is expected to close most of that gap.

Metric	Serverless-only	Container-only	Hybrid	Δ vs. best baseline
Avg. p95 latency (ms)	476	188	120	-36%
p99.9 first-byte (ms)	2150	470	230	-51%
Cost-to-serve (relative)	1.00	0.84	0.61	-28%
Time-to-scale (s)	<1	45	<1	-
GPU workload support	No	Yes	Yes	-

Table 3. Summary comparison of single-tier baselines against the hybrid design across the evaluation dimensions.

8. DISCUSSION

Three observations from operating this architecture in production deserve special emphasis.

Routing is a control problem, not a configuration problem. Early iterations of the system treated the placement of each endpoint as a static configuration choice. That approach broke whenever workload mix or traffic shape changed materially. Treating routing as a closed-loop control problem - with the router consuming live telemetry, comparing predicted to observed latency, and adapting its policy within seconds - was the change that made the architecture robust.

The fallback path must be exercised continuously. A fallback that is only triggered during incidents is a fallback that does not work. The system deliberately routes a small, randomized fraction of every endpoint's traffic to the secondary tier at all times, both to keep the path warm and to surface compatibility regressions immediately rather than during an outage.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Tier boundaries must not leak into application code. Maintaining a single OCI image that runs on both tiers (Section 6.1) eliminated an entire class of subtle bugs in which a code path worked on one tier and silently misbehaved on the other. Application authors are aware that their service runs in two execution envelopes, but they write to a single internal SDK that abstracts the difference.

A subtler observation concerns organizational design. The hybrid architecture requires that one team be accountable for the placement decisions, the routing fabric, and the operational health of both compute tiers. Splitting these responsibilities across separate teams reintroduces, at the human layer, exactly the seams that the architecture was designed to remove. The platform that runs the system in production is owned end-to-end by a single platform team; application teams interact with it through a small set of APIs and rarely need to know which tier their request landed on.

9. THREATS TO VALIDITY

Workload mix. The evaluation is grounded in the workload mix of a creative-cloud platform with significant media-processing traffic. Architectures whose dominant workload is, for example, transactional database access against a single shared store may not see the same crossing point in the cost curves, and may not benefit as much from the container tier.

Cloud provider effects. Some of the absolute numbers reported here will be specific to the public-cloud regions and instance families used during the measurement window. The relative comparisons among strategies are expected to generalize, but the precise crossing points and tail percentiles will differ on other providers and on private infrastructure.

Maturity of the routing classifier. The classifier was retrained nightly during the measurement window and benefits from a rich body of historical traffic. Newly launched endpoints, or endpoints whose behavior changes rapidly, will route less optimally until enough traffic has accumulated to retrain the model.

Measurement methodology. All latency figures reported are server-side measurements terminated at the last byte sent. Client-perceived latency includes additional factors - DNS, TCP, TLS, last-mile bandwidth - that are outside the architecture's control. The comparisons among strategies remain valid because all three are measured identically.

10. Conclusion and Future Work

This article has described a hybrid architecture for low-latency content delivery that combines a serverless tier, a container tier, and an edge tier behind a single latency-aware request router. The architecture is the product of several years of production operation at the scale of tens of millions of monthly active users across every populated region. Across six representative workloads, it cuts p95 latency by roughly a third, compresses the cold-start tail by an order of magnitude, and reduces cost-to-serve by 28% relative to the best single-tier baseline.

Three avenues of future work seem most promising. First, the routing classifier is currently trained nightly on per-request features; an online learning formulation that updates within minutes would tighten the loop further and reduce the regret incurred when traffic shape changes abruptly. Second, the container tier's autoscaler responds reactively to observed load; a predictive scaler driven by client-side intent signals could pre-warm capacity ahead of bursts that are anticipated rather than absorbed. Third, edge-native inference runtimes [27, 28] continue to mature, and a growing share of the ML inference workload is a candidate for migration from the regional container tier to the edge tier. The architecture's tier-agnostic application contract means that such a migration can be performed without changes to the application code itself.

More broadly, we expect the rigid distinction between serverless and container platforms to continue softening. Projects such as Knative [25], AWS Fargate [26], and a range of academic prototypes [17, 24] are converging toward a single substrate that can host a function or a long-running service with comparable economics and operational ergonomics. Until that convergence is complete, however, the right answer for a production platform that must satisfy a heterogeneous workload mix is not to pick a side, but to build the routing intelligence that lets both sides be used where each is strongest.

REFERENCES

1) Amazon Web Services. AWS Lambda Developer Guide. Amazon, 2014–2023. https://docs.aws.amazon.com/lambda/
2) Google Cloud. Cloud Functions documentation, 2016–2023. https://cloud.google.com/functions/docs
3) Burns, B., Beda, J., and Hightower, K. Kubernetes: Up and Running, 2nd ed. O'Reilly Media, 2019.
4) Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In Proc. EuroSys, 2015.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

5) Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. The rise of serverless computing. Communications of the ACM, 62(12):44–54, 2019.
6) Casalicchio, E. A study on performance measures for autoscaling CPU-intensive containerized applications. Cluster Computing, 22(3):995–1006, 2019.
7) Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes. Communications of the ACM, 59(5):50–57, 2016.
8) Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking behind the curtains of serverless platforms. In USENIX ATC, 2018.
9) Manner, J., Endreß, M., Heckel, T., and Wirtz, G. Cold start influencing factors in function-as-a-service. In Proc. UCC Companion, 2018.
10) Hellerstein, J. M., Faleiro, J. M., Gonzalez, J., Schleier-Smith, J., Sreekanti, V., Tumanov, A., and Wu, C. Serverless computing: One step forward, two steps back. In CIDR, 2019.
11) Armbrust, M., Fox, A., Griffith, R., et al. A view of cloud computing. Communications of the ACM, 53(4):50–58, 2010.
12) Jonas, E., Schleier-Smith, J., Sreekanti, V., et al. Cloud programming simplified: A Berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, UC Berkeley, 2019.
13) Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., and Pallickara, S. Serverless computing: An investigation of factors influencing microservice performance. In IEEE IC2E, 2018.
14) Amazon Web Services. AWS Lambda SnapStart for Java functions. AWS Blog, November 2022.
15) Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In ASPLOS, 2020.
16) Pu, Q., Venkataraman, S., and Stoica, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In NSDI, 2019.
17) Ao, L., Izhikevich, L., Voelker, G. M., and Porter, G. Sprocket: A serverless video processing framework. In Proc. SoCC, 2018.
18) Kubernetes Authors. Horizontal Pod Autoscaler documentation, 2017–2023. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
19) Amazon Web Services. Karpenter: open-source Kubernetes node autoscaler, 2021–2023. https://karpenter.sh
20) KEDA Authors. Kubernetes Event-driven Autoscaling project documentation, 2019–2023. https://keda.sh
21) Burns, B. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. O'Reilly Media, 2018.
22) Newman, S. Building Microservices, 2nd ed. O'Reilly Media, 2021.
23) Aske, A. and Zhao, X. Supporting multi-provider serverless computing on the edge. In Proc. ICPP Workshops, 2018.
24) Sampé, J., Vernik, G., Sánchez-Artigas, M., and García-López, P. Serverless data analytics in the IBM Cloud. In Proc. Middleware Industrial Track, 2018.
25) Knative Authors. Knative documentation. The Knative Project, 2018–2023. https://knative.dev/docs/
26) Amazon Web Services. AWS Fargate: serverless compute for containers, 2017–2023. https://aws.amazon.com/fargate/
27) Cloudflare. Cloudflare Workers documentation, 2017–2023. https://developers.cloudflare.com/workers/
28) Fastly. Compute@Edge documentation, 2019–2023. https://developer.fastly.com/learning/compute/
29) Sigelman, B. H., et al. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, 2010.
30) Mao, M. and Humphrey, M. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In Proc. SC, 2011.
31) Schleier-Smith, J., Sreekanti, V., Khandelwal, A., et al. What serverless computing is and should become. Communications of the ACM, 64(5):76–84, 2021.
32) Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. SAND: Towards high-performance serverless computing. In USENIX ATC, 2018.