National College of Ireland

Reducing Latency Through Adaptive Load Balancing in Multi-Cloud Kubernetes

MSc Research Project
MSc in Cloud Computing

Yashvanth SB
Student ID: 23336641

School of Computing
National College of Ireland

Supervisor: Luis Bernardo Pulido Gaytan

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Yashvanth S B
Student ID:	23336641
Programme:	Research in computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Luis Bernardo Pulido Gaytan
Submission Due Date:	28/08/2025
Project Title:	Reducing Latency Through Adaptive Load Balancing in Multi-Cloud Kubernetes
Word Count:	10241
Page Count:	23

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Yashvanth S B
Date:	7th September 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Reducing Latency Through Adaptive Load Balancing in Multi-Cloud Kubernetes

Yashvanth SB
23336641

Abstract

The rapid uptake of cloud computing and increasing internet traffic have made effective load balancing in multi-cloud Kubernetes setups a pressing requirement. Traditional methods are deficient in meeting the complexities of multi-cloud setups, leading to excessive latency and resource misuse. This contribution presents a new framework that combines Content Delivery Networks (CDNs) and adaptive load balancing between AWS and Azure Kubernetes Service (AKS) clusters. With the help of advanced techniques like multi-level caching, latency-sensitive routing, and heuristic algorithms like Consistent Hashing with Bounded Loads and Least Latency with Capacity-Aware Weighting, the solution aims to reduce latency and resource usage in an optimized way. HTTP/3 integration and edge zones also decrease connection delay and minimize distance to move content closer to users. The study targets latencies below 150ms and CPU balancing, addressing the problems of inter-cloud latency and resource variability. This paper presents a cost-effective, scalable, and high-performance solution to modern cloud applications that opens the door to the future development of cloud computing by proposing a framework for optimized resource utilization and latency reduction.

1 Introduction

Modern web applications are expected to feel instantaneous across continents, yet multi-cloud deployments complicate that goal. When the same service runs on AWS and Azure, differences in geography, peering, and hardware expose users to uneven paths and replicas to uneven load. In practice, uncoordinated decisions made by DNS, CDNs, Kubernetes ingress, and the Horizontal Pod Autoscaler can amplify one another: some pods run hot while others idle, caches warm inconsistently, and tail latencies spike, often pushing end-to-end response times above the 150ms target that user-facing systems aim to sustain. Addressing latency alone or balancing alone is insufficient; the challenge is to coordinate layers so that proximity, caching, transport, routing, and capacity all work in the same direction. Research to date covers important pieces but typically in isolation. Many studies remain single-cloud and rely on basic policies such as round-robin, least-connection, or IP-hash; where CDNs are used, they are usually evaluated with a single origin and without tight coupling to in-cluster routing. A representative single-cloud study with CloudFront reports 154ms response times under simple balancing—useful as a baseline but not a full answer for heterogeneous multi-cloud systems that must coordinate ingress, CDN, and autoscaling across providers. This leaves a gap: practical, provider-symmetric

architectures that integrate CDN behavior, modern transports, and adaptive in-cluster selection, then validate the combined effect end-to-end.

This thesis addresses that gap with a lightweight hybrid, multi-cloud web-delivery architecture deployed symmetrically on Amazon EKS and Azure AKS and fronted by Amazon CloudFront and Azure CDN. Latency-based global routing (Route 53 and Traffic Manager) directs users to nearby, healthy edges; the viewer-to-edge hop uses HTTP/3 on the AWS path and HTTP/2 on the Azure path to reflect current product support; and within each cluster the NGINX Ingress Controller applies an adaptive policy that blends session locality via consistent hashing with a capacity- and latency-aware choice, bounded so that no pod exceeds a small multiple of the average share. The design keeps images, manifests, and resource policies identical across clouds so that any measured differences arise from the architecture rather than drift in application code or configuration.

1.1 Problem statement

Growing reliance on multi-cloud strategies underscores the need for effective load balancing to support future-proof digital apps. Ineffective load balancing results in over-subscribed or spare nodes, raising infrastructure costs via over-provisioning and lowering system stability during traffic spikes. This impacts user satisfaction and business outcomes, particularly in competitive online marketplaces where scalability and low latency are of utmost importance. The issue of overcoming inter-cloud latency, resource heterogeneity, and lack of adaptive CDN-Kubernetes synchronizing techniques in multi-cloud deployments.

1.2 Research Question and Objectives

What strategies can improve latency, reduce response time below 150ms and resource efficiency in multi-cloud Kubernetes through CDN integration and adaptive load balancing? The objective of this research is to design an adaptive load balancing system integrated with CDN for multi-cloud Kubernetes deployments to optimize performance, reduce latency to under 150ms, and achieve balanced CPU distribution among cloud providers, thereby enhancing scalability and resource effectiveness.

1.3 Research Contributions

This work provides several contributions to the research on cloud computing:

- A comprehensive architecture integrating CDNs with Kubernetes in multi-cloud systems, addressing inter-cloud latency and resource heterogeneity.
- Adaptive load-balancing algorithms that balance latency and resource consumption, ensuring deterministic performance.
- Enhanced cost-effectiveness and scalability through efficient workload allocation, reducing over-provisioning and operational expenses.

1.4 Thesis Outline

The structure of this thesis is as follows: Section 2 describes related work in multi-cloud load balancing, CDN integration, and adaptive algorithms. Section 3 is the description

of the methodology, which includes research problem, development framework, and the evaluation process. Section 4 provides the design specification of the proposed approach. Section 5 describes the implementation, which includes technologies and tools used. Section 6 presents the evaluation results, demonstrating the efficacy of the solutions. Section 7 summarizes the thesis and recommends future study. References are provided at the end.

2 Related Work

2.1 Multi-Cloud Load Balancing Challenges

Multi-cloud deployment consolidates various providers' resources providing distinctive virtual machine families, storage backends, and network fabrics. With heterogeneous hardware comes non- uniform CPU cycles per instruction, memory bandwidth, and NIC offloading. When an application is replicated across providers, these differences become evident as non-uniform throughput at equal nominal utilization. Therefore, an oversimplified balancing policy that considers replicas as fungible tends to assign work in a way that worsens the slowest path. The Afzal and Kavitha survey makes one of its most important arguments concerning the reason why dynamic strategies are necessary when it argues that the workload properties and resource states evolve over time rather than remaining constant (Afzal & Kavitha, 2019). Generalizing this argument to multi-cloud environments means that any controller will have to absorb signals for both real-time capacity headroom and the end-to-end path latency experienced by users. Without one or both of these signals, the controller can over-correct or chase after false measurements, producing oscillations manifest as spikes in tail latency.

Another facet barely mentioned in single-cloud studies is provider-internal rate limiting and inter- connect peering. Cloud operators charge for egress, and the journey from a CDN edge to a regional origin can run through multiple peering arrangements based on which provider is hosting the origin. Such policy-driven behaviour can alter effective throughput without any modification of application logic. Therefore, the multi-cloud controller not only must pick an appropriate pod but must also direct the user towards an origin whose upstream links will be least likely to be clogged. It is this observation that brings us to the dual-layer approach where DNS directs at the coarse grain and ingress fine-tunes selection within the cluster.

2.2 CDN Integration and Edge Caching

Content delivery networks offer a vast network of edge sites to end the user connections close to their access networks. The edge nodes have caches that store shared content and enforce policies such as time-to-live, cache-control revalidation, and conditional requests. One of the significant contributions of the single-cloud experiment by Kumari is the empirical validation that hierarchical caching edge, regional, and origin levels, minimizes the number of origins fetches and consequently the average and the variance of response times (Kumari, 2023). Such benefits occur when cache keys and headers are selected appropriately so that semantically equivalent requests map to a single cached object. For applications using containers, it is common to segregate static assets such as JavaScript bundles and style sheets from dynamic API responses and provide different caching policies to each path prefix. In a cloud environment with multiple clouds, the additional complexity is that there is more than one valid origin. The CDN configuration must

express origin groups with explicit failover priority and health checks so that local failure does not cause users to fallback to distant-out origins unless necessary.

Yang and his coauthors' survey lays the groundwork for an SDN-influenced CDN that can make more granular decisions about route and cache behaviour (Yang et al., 2023). However, the majority of the work focuses either on one operator or on SDN-internal plumbing; there is little explicit treatment of situations in which there are multiple clouds where the origin can in practice be located in multiple regions in different clouds with different egress and path policies. That implies that, for web application developers, it is still straightforward to optimize for compute instead of transfer.

2.3 Kubernetes in Heterogeneous Deployments

Kubernetes primitives Deployments, Services, and Ingress, abstract over container lifecycles, layer 4 endpoint groups, and layer 7 routing respectively. These are good abstractions for multi-cloud because the developer can deploy the same manifests without changing orchestration logic, but it also hides heterogeneity that surfaces in measurements. The key to success in such a deployment is not to change application code to tailor to one provider, but to use orchestration, ingress, and edge configuration to turn the heterogeneous system into a predictable system with two timescales, which is more stable than in isolation.

2.4 Latency-Based Routing

Latency-based traffic managers and DNS are beneficial in bringing users near an appropriate edge, as seen in realistic proposals that combine CDN-aware routing and application delivery (Varma et al., 2023). Still, routing alone cannot guarantee low latency in case stateful information is anchored in a distant region. Research on replication methods, including CRDT-based solutions in the edge, emphasizes the cost of consistency management in maintaining data close to users (Simić et al., 2023). Although the existing load is stateless, the proposed architecture envisions future stateful services with session affinity via consistent hashing but still limiting skew to avoid hotspots. Such a design provides room for adding a distributed session store without changing the routing policy core.

2.5 HTTP/3 (QUIC) and HTTP/2

HTTP/3 eliminates TCP head of line blocking by multiplexing streams over QUIC, a transport based on UDP with integrated TLS. Establishing connection usually benefits from zero round trip resumption for repeat visitors, which has the immediate effect of reducing first byte latency. Questionnaires connecting CDN behavior to transport innovation show that the full benefit comes through when client and edge both support HTTP/3 as well as when the path to origin is not routed back to HTTP/1.1 as a result of misconfiguration (Yang et al., 2023; Shafiq et al., 2021). In our design I enabled HTTP/3 on the AWS path via Amazon CloudFront, in the Kubernetes context and highlights HTTP/3 as part of the latency-reduction toolkit. On the Azure path, the edge serves browsers over HTTP/2, so we keep that side on HTTP/2 and focus on caching and routing improvements to keep latency low.

2.6 Adaptive Algorithms and Practical Trade-offs

Learning-based controllers such as the reinforcement-learning scheme by Dong et al. can be capable of learning complex system dynamics at the cost of implementation and operational complexity (Dong et al., 2021). Heuristics are between the two extremes, keeping domain knowledge in lightweight scoring functions. The capacity-sensitive least-latency weighting used here prefers pods that are already fast and have capacity to spare, a property that can be derived from ingress-level observations without invasive probing. Bounded-load consistent hashing places a safety valve by forcing no single replica to receive more than a small multiple of the mean load. The blend delivers robust performance against mixed traffic patterns, typical in production environment operational restrictions.

2.7 Evaluation Practices and Reproducibility

Good evaluation ties architectural changes to measurable outcomes and records not just central tendency but also spread. JMeter or equivalent experiments show that well-tuned workloads reflect actual burstiness and reveal sensitivity to autoscaling delay and cache warm-up (Radhika & Duraipandian, 2021). The approach taken here also mirrors such practices in applying repeated experiments, cache state control, and infrastructure as well as client-side metrics. The application to a lowly application is deliberate: it reduces confounding variables so that changes to latency can be attributed to platform and network choices.

2.8 Research Gap and Contribution.

Taken together, these studies illustrate progress in cloud load balancing and CDN-based optimization but also reveal clear limitations. Most prior works are confined to single-cloud contexts, rely on static algorithms, or remain theoretical without full-scale empirical validation. CDN usage, where included, is restricted to a single provider and seldom coordinated with Kubernetes ingress or autoscaling. Protocol choices are limited to legacy transports, leaving newer standards like QUIC underexplored. Furthermore, very few studies systematically evaluate both latency metrics and backend resource fairness together, meaning that improvements at the network edge are rarely examined alongside CPU utilization balance inside the cluster. This disconnects results in partial optimizations that improve response time at the user level but overlook stability and efficiency at the infrastructure level.

This thesis addresses these gaps by presenting a practical multi-cloud Kubernetes architecture spanning AWS and Azure, fronted by CloudFront (HTTP/3) and Azure CDN (HTTP/2), and augmented by adaptive ingress algorithms—Least-Latency Capacity-Aware Weighting (LLCW) and Consistent Hashing with Bounded Loads. Unlike prior work, it explicitly evaluates both latency reduction below 150ms and uniform CPU utilization across clusters. By combining CDN-edge optimizations, protocol advancements, and adaptive scheduling with autoscaling, the research demonstrates an integrated approach that ensures performance gains are holistic and not isolated to one layer of the system. This highlights the novelty of the work: a reproducible, symmetric, and empirically validated system that advances the state-of-the-art in multi-cloud load balancing and CDN–Kubernetes synchronization.

2.9 Comparison work

Table 1: comparison worksheet

Category	Kumari (2023)	Afzal & Kavitha (2019)	Dong, Y., (2021)	Yang, H., (2023)	Proposed Work
Cloud Environment	Single cloud (AWS)	General cloud, no multi-cloud focus	Cloud-edge, no multi-cloud focus	General SDN-CDN, no multi-cloud focus	Multi-cloud
Load Balancing Algorithms	IP Hash, Weighted Round Robin	Round Robin, Least Connection	Reinforcement learning	Not specified	LLCW, Consistent Hashing with Bounded Loads
CDN Integration	AWS CloudFront only	Not addressed	Not addressed	SDN-CDN	AWS CloudFront, Azure CDN
Protocol	HTTP/2	Not specified	Not specified	HTTP/3 (QUIC)	HTTP/3 (QUIC)
Testing Scale	Regional experiments, scale not detailed	Not specified	Limited testing details	Not specified	JMeter for performance testing
Performance Target	154ms response time	General performance metrics	Not specified	General latency reduction	Below 150ms response time
Resource Utilization	Monitored, no uniformity focus	Not addressed	Not emphasized	Not addressed	Uniform CPU utilization
Scope and Implementation	Single-cloud Kubernetes	Theoretical classification	Cloud-edge, theoretical	SDN-CDN, theoretical	Multi-cloud Kubernetes, practical
Kubernetes Focus	Single-cloud Kubernetes	Not addressed	Not addressed	Not addressed	Multi-cloud Kubernetes with HPA
CDN–Kubernetes Integration	Basic CloudFront.	Not addressed	Not addressed	SDN-CDN, no Kubernetes	Multi-cloud CDN–Kubernetes synchronization

3 Methodology

The methodology is design-science in which a candidate architecture is implemented on two public clouds and evaluated with reproducible synthetic tests. The artefact under evaluation is a multi-cloud delivery stack that integrates latency-based global steering, CDN caching, modern transport at the edge, adaptive ingress within Kubernetes, and elastic capacity via autoscaling. The unit of analysis is intentionally simple: a stateless web application rendered from the same container image in both clouds.

Single-cluster approaches such as round-robin or least-connection ignore inter-cloud path length, edge cache behaviour, and transient headroom differences between pods. Thus, a sub-set of replicas can become hot while others are idle, and users resolving to a distant region can experience higher time to first byte on cache misses. The first objective is therefore to define and validate a hybrid system that always drives median user-perceived response time below one hundred and fifty milliseconds in the steady state. The second objective is to show that CPU usage is evenly distributed between nodes and pods, not only under idle load but also when the system ramps and handles users from two distinct geographies matching the chosen AWS and Azure regions. A secondary objective is operational clarity: the system should be simple enough to reason about, with each layer doing a small number of things well so that changes are comprehensible and reproducible.

The experimental design is carried out in four phases so that incremental effects are apparent and separable. In the baseline, where the app is exposed through vanilla DNS with no latency preference and no CDN fronting, the browser uses legacy transports (HTTP/1.1 or HTTP/2) to reach the origin, and the ingress load-balances requests round-robin across pods. This yields a neutral baseline that features no edge offload and no adaptive routing.

In phase 2, the edge is enabled: CloudFront fronts the AWS origin and serves to the browser via HTTP/3 for the viewer-to-edge hop, and Azure CDN fronts the Azure origin and serves to the browser via HTTP/2; both edges proxy to the origins using standard protocols. Cache rules separate immutable assets with long lifetimes from dynamic routes with short, revalidated lifetimes.

In the third step, the cluster ingress is made adaptive: a consistent-hash mapping provides stable session locality, and within that subset the selected pod is the one with the best recent latency and available capacity, subject to a bounded-load constraint that no replica is permitted more than some specified fraction above the mean.

The fourth and final step activates latency-based routing at the DNS level using Route 53 and Traffic Manager so that name resolution itself will send users to close edges and healthy origins. Because each intervention is layered on top of the prior state while keeping everything else equal, the marginal value of CDN, transport, ingress policy, and DNS routing can be measured separately and then cumulatively.

Independent variables are restricted to switches that a platform owner would actually manage. These include if there is CDN caching in front of the cluster or not, the transport protocol exposed to the browser on each cloud path (HTTP/3 on AWS via CloudFront and HTTP/2 on Azure via Azure CDN), the ingress selection policy within Kubernetes (round-robin vs. latency- and capacity-aware with bounded-load hashing), and if latency-based DNS is used to answer client names. Dependent variables measure the user experience and internal equilibrium of the system: median and ninety-fifth percentile response time as seen by the client, error rate as seen by the load generator, CDN cache hit ratio and bytes served from edge, number and time of autoscaling events, and variance and coefficient of variation of CPU utilization across pods and across nodes.

Control variables are held constant so runs are directly comparable. Version of application is baked and built once; both clusters draw the same container image digest; Kubernetes, NGINX Ingress, and metrics-server versions are the same; requests and limits for resources are the same; and node pool sizes and instance types are selected

to be as similar as possible with the two providers. DNS and CDN time-to-live values are set to balance cross-run contamination minimally but still reflect realistic caching. External validity is supplied by running from two vantage point locations that mirror the cluster locations—one near the AWS region and one near the Azure region, so that routing patterns and geography are represented in the measurements. To reduce hidden confounders, clocks are synched, tests are staggered in sequence between conditions, and the run schedule avoids periods of scheduled provider maintenance. Instrumentation is chosen to quantify perception on the client side and behaviour on the platform. Apache JMeter drives open-model arrivals with a ramp up to target concurrency in control and then holds a steady plateau for long enough to let autoscaling stabilize.

Each sample records start time, latency, and response code in a CSV trace with millisecond resolution, and per-run metadata files capture the condition, region, container digest, and manifest versions. From the platform side, AWS CloudWatch and Azure Monitor are queried for CPU and network throughput per-pod and per-node, load balancer request counts, and HPA events. CDN analytics provide cache hit ratio and origin selection over the same time ranges. Before each measured run, caches are warmed with a scripted walk of static assets and a small number of dynamic calls; between phases that change cache policy, an explicit invalidation is delivered so that results aren't distorted by stale objects.

Data collection is designed to minimize hysteresis and transient noise. Each condition is run at least three times per vantage region. The first minute of each ramp is treated as warm-up and excluded from steady state calculations; the remaining window of fixed duration forms the basis for the reported p50 and p95 figures. Autoscaling events are condensed into timelines so that they can be correlated against latency changes; runs for which provider dashboards reveal unrelated events are flagged and, if necessary, rerun. The analysis computes distributions rather than point measure and presents variability as confidence intervals. For CPU uniformity, variance and coefficient of variation are supplemented with an easily interpretable target band, the design succeeds in a window if pod CPU stays within a narrow band around the Autoscaler target and no replica consistently exceeds the bounded-load limit relative to its peers. Interpretation is two-pass. The first pass is descriptive: graphs overlay and latency for all four phases per region, as well as cache hit ratio and HPA activity, to reveal the sequence of cause and effect.

The second pass is comparative: phase-to-phase differences are computed so that the marginal effect of each intervention can be observed. Cross-cloud symmetry is accorded particular attention. Because AWS is delivering HTTP/3 to browsers and Azure is delivering HTTP/2, the methodology anticipates a measurable benefit on the AWS path with cold caches and convergence after caches are warm, all else being equal. A corresponding experiment verifies that the ingress policy reduces CPU skew in both clouds by the same proportion, which indicates that the controller is reacting to local conditions rather than exploiting a provider-specific quirk. Threats to validity are identified and controlled where possible. Background activity on shared cloud infrastructure can affect end-to-end latency; repeating runs at different times and verifying provider health feeds reduces the likelihood that an isolated anomaly determines a conclusion. VPN routes used to simulate user geography can introduce jitter; the experiment accounts by repeating multiple times and by comparing deltas between phases rather than absolutes.

4 Design Specification

4.1 Architecture Overview

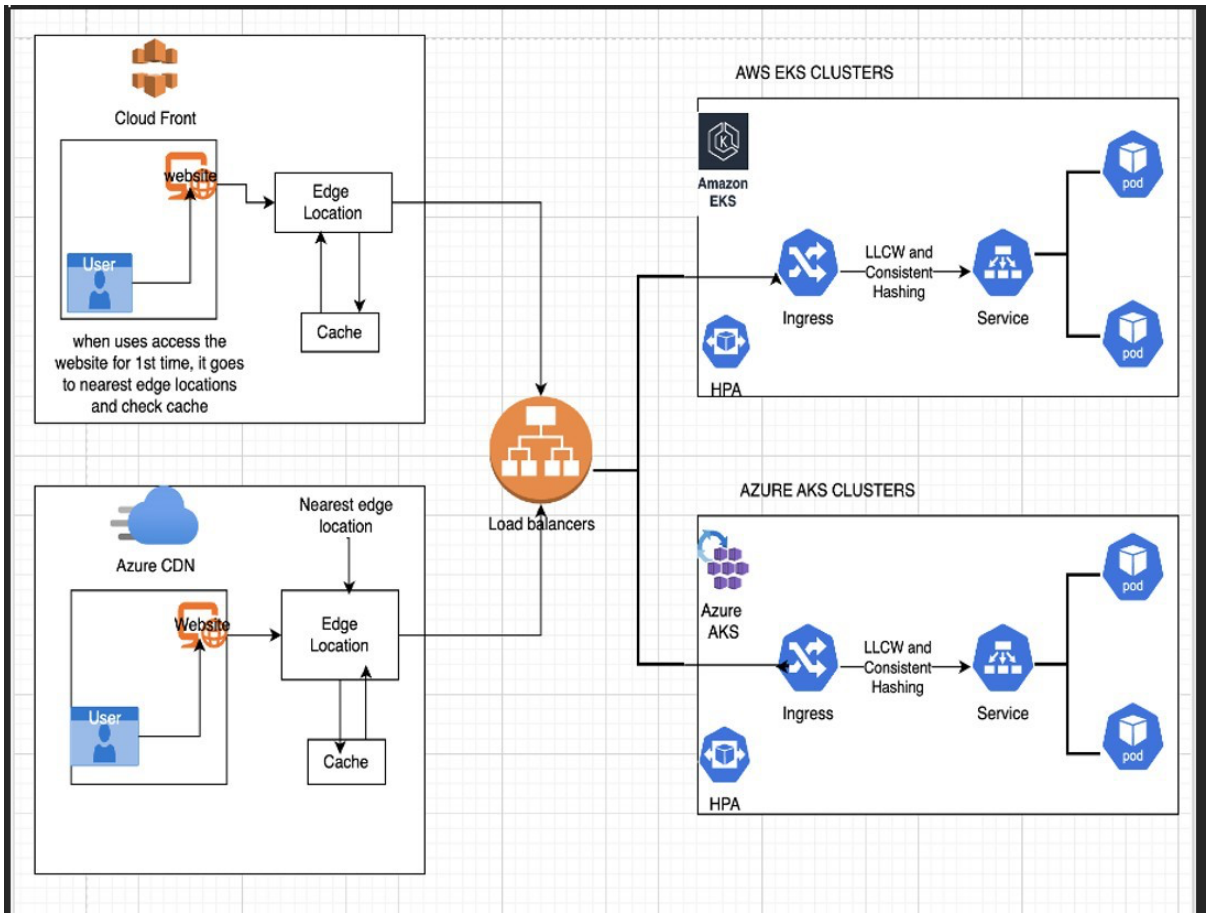

Figure 1: Architectural diagram

Architecture brings together a small set of cloud services that each excel at a single task and then pass to the next level without overflowing into complexity. Authoritative DNS is handled by Amazon Route 53 and Azure Traffic Manager so that the application may resolve names out of two different providers but still route users to the closest healthy point of entry. Content distribution is managed by Azure CDN and Amazon CloudFront, which cache static assets at edge locations and forward dynamic requests back to the nearest origin. The application pods run in two managed Kubernetes services like Azure AKS and Amazon EKS, so the deployment, scaling, and health model is identical for both clouds. The NGINX Ingress Controller serves in front of each cluster and terminates TLS, selecting a backend pod per request. Horizontal Pod Autoscaler scales replica numbers with fluctuating load, and CloudWatch and Azure Monitor take the measurements that prove the system is behaving as intended. On the client side, HTTP/3 is supported on the AWS viewer-to-edge path via CloudFront, and the Azure viewer-to- edge path uses HTTP/2; both edges talk to the origins using plain HTTP/1.1 or HTTP/2 supported by the ingress. One Docker image pushed to Docker Hub ensures the exact same artifact runs in both clouds without ”works on one provider only” drift.

From a user standpoint the process begins with name resolution. A mobile app or browser asks its recursive resolver for the application domain. Depending on where

that resolver happens to be on the Internet and what authority answers the zone, it will either be from Route 53 or Traffic Manager and point to an edge location nearby. That first choice is important in the sense that it establishes proximity from the get-go; round-trip time here has a disproportionate effect on the rest of the experience, especially for initial users who are not yet seeing the perks of warm caches or reusing connections. Measured-latency preference and minimal health are included in DNS replies so that unresponsive edges will not be chosen. Once a user has been bound to an edge, the CDN becomes the first line of defence in terms of latency and load. For immutable resources such as JavaScript bundles, CSS, and images, the edge applies long time-to-live policies and compression so that even subsequent requests are served from the nearest point of presence. Dynamic endpoints have fleeting lifetimes with revalidation: try not to make unnecessary round trips to origin without sacrificing data freshness. On the AWS path the browser speaks HTTP/3 to CloudFront, which reduces connection setup time and avoids head-of-line blocking on the browser-to-edge path; on the Azure path the browser speaks HTTP/2 to Azure CDN, which still provides multiplexing and header compression but follows the older transport.

The CDN selects the origin topologically closest to the edge and which, as of continuing probes, is healthy. Each origin is the cloud load balancer in front of NGINX Ingress Controller. Making the CDN smart about which origin to use, the design avoids routing traffic over long inter-cloud paths when a local origin is available and automatically fails over to the second cloud if a region becomes unavailable. Since so much static traffic never arrives at origin, the amount of compute required per cluster decreases and tail spikes are prevented. On a per-cluster basis, NGINX Ingress features a conservative yet adaptive backend selection. Every request includes a stable session key—either an application-seeded cookie or, in the absence thereof, a client IP ”hint” —which is passed into a consistent-hash mapping. The outcome is a gentle nudge to the fastest, least loaded replica without thrashing across the deployment. Autoscaling sits back and keeps capacity in sync with demand. The Horizontal Pod AutoScaler tracks CPU utilization, and potentially custom metrics if desired so in the future and raises the number of replicas when sustained load is raised.

Security and privacy concerns are addressed at many layers without complicating the request path. TLS is terminated at the CDN edge so that the cached objects stay encrypted as they go to users, and TLS is terminated again at the NGINX ingress before the traffic reaches the cluster’s service mesh. Certificates are managed by certificate managers in the cloud providers, so no manual renewal is required. If a web application firewall is then required, it’s placed at the periphery so that malicious traffic is rejected before hitting origin. Role-based access controls within Kubernetes restrict what chunks of software are allowed to read secrets or make changes to deployments, and separate namespaces isolate the application from cluster-level add-ons.

Failure and recovery are handled end-to-end. Each CDN origin group (EKS and AKS) is actively probed from the edge; if an origin becomes unhealthy, the edge immediately stops sending cache-misses to it and DNS health checks remove that region from latency-based answers. Users are therefore steered to the healthy cloud before the CDN attempts another origin. Because the edge serves the majority of requests from cache, the surviving origin only needs to handle dynamic requests and the remaining misses, which keeps latency steady and leaves headroom for autoscaling to add replicas. When a failed region returns, probes restore it gradually and traffic is rebalanced without operator intervention.

Observability is built in rather than bolted on. CDN analytics expose hit ratio, edge bytes served and origin-shield behaviour, CloudWatch and Azure Monitor export per-pod and per-node CPU, throughput, error rates and HPA events. Client-side latency is measured with JMeter plans executed from two vantage points that match the chosen regions. By synchronizing time and reusing run labels, we can attribute cause and effect for each architectural change. Caching is engineered for consistency and efficiency. Cache keys are normalized so semantically equal requests map to the same entry; unnecessary query parameters are stripped; headers are canonicalized; and negative caching is enabled for well-known 404/410 objects to avoid needless origin trips. Time-to-live is tuned: short for dynamic assets; long for fingerprinted files that include a content hash in the filename, where a new build produces a new URL.

Ingress and autoscaling are tuned for low latency. NGINX keeps connection pools warm and only exposes pods after readiness probes complete. The selection policy couples session locality (consistent hashing with bounded loads) with LLCW—a least-latency, capacity-aware weighting—so that faster replicas with headroom receive the next request without over-concentrating traffic. HPA then adds replicas on a slower time scale, so the fast in-cluster policy and the slower autoscaling loop complement one another.

The AKS and EKS manifests are made identical as possible so that the system can be replicated or moved without provider-specific fine-grained knowledge. Offload in CDN reduces origin egress and compute latency, and autoscaling eliminates idle capacity waiting around between tests. Nothing in the design binds the application to proprietary runtimes or serverless gateways; even the ingress policy is accomplished with standard NGINX features so it can be read and adjusted by engineers who are well familiar with mainstream Kubernetes tooling. Lastly, it is impressive how the layers support one another. DNS makes sure users start in the right neighborhood. The CDN does the heavy lifting of transfer and cache, with HTTP/3 on the AWS path giving repeat users an added push towards faster first bytes. The ingress makes smart, lightweight decisions so that the cluster sees traffic consistently without micro-managing every request. Autoscaling keeps an eye on the trend and not the instant and keeps the system elastic. Monitoring finishes the loop by giving, in a single place, whether promises made by each layer are being kept. Because all pieces are small and customized, the architecture remains easy to reason about even as it spans two clouds and multiple networks; that simplicity is what ends up making it deliver low-latency and predictable behaviour in the real world.

5 Implementation

5.1 Web application Development

Early implementation begins with a minimal web application that is deliberately un-complicated in function and load-stable such that network and platform influences drive the outcome. The user interface is built with HTML and CSS and served as static assets that can be efficiently delivered by any standard web server. A minimal client-side JavaScript layer supports simple interactivity without introducing heavyweight frameworks or whimsical background behavior. The dynamic component of the application shows a miniature HTTP API in Python so that server-side processing is consistent and easy to reason about in terms of concurrency. Node.js is used during development to run the local asset pipeline and bundle or minify the static assets; this keeps the browser downloads small and emit the same directory structure when the project is built repeatedly. At

run-time, the container serves the pre-compiled static bundle and goes through API calls to the Python application in a way that the container is self-contained and not reliant on externalities. The Python app provides a health endpoint that returns a light JSON payload. The sole purpose of the endpoint is to make liveness, and readiness checks cheap for Kubernetes and for CDNs. The router within the application separates dynamic API paths from static asset paths so that caching rules can be precisely addressed at the edge. Every app build places a version number in response headers and places a short commit hash in HTML that later facilitates correlating JMeter runs, and dashboard reads with the exact code artifact that served them.

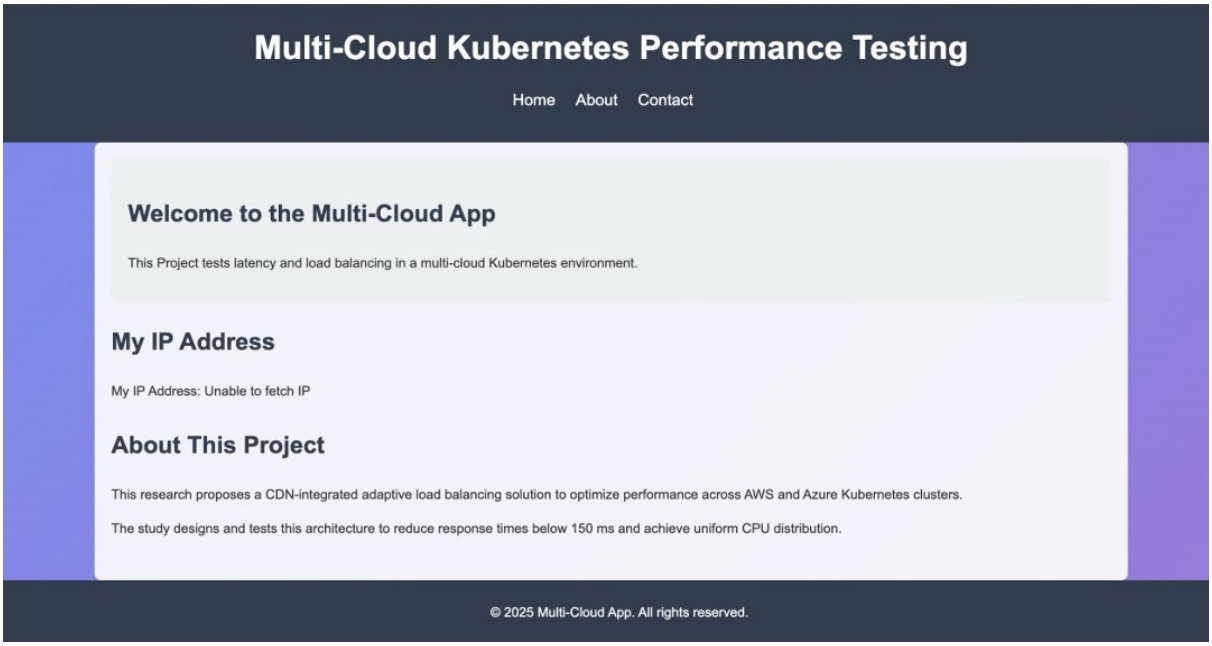

Figure 2: Web application

5.2 Containerization and versioning of images

Codebase is packaged in a Docker image with a very small base so that the resulting container starts quickly and with minimal resource. The build process is deterministic: the same sequence of steps produces the same layout and the same image digest on all machines, cutting out an entire class of “works on my laptop” bugs. Node.js-generated static files are copied into the image along with the Python service, and the container only exposes the single port that the application listens on internally. The health endpoint is revealed in container metadata so that orchestration may probe it without paths being hardcoded at many places. Each image is provided with a semantic version and pushed to Docker Hub. From a shared public registry, the pipeline is easy to replicate for two-cloud testing and guarantees that exactly the same artifact is retrieved by both providers. Immutability as a hard-and-fast rule: no container is ever replaced in place, and every deploy refers to a digest rather than a mutable tag. That practice makes tests reproducible, since a JMeter result can always be traced back to a specific image.

5.3 Local Testing on Minikube

Before provisioning any cloud resources, the application is run in a local Minikube cluster so that issues in routing, service discovery, and readiness are exposed prior to when the feedback loop is fast. The local cluster is a copy of the object model utilized later on the cloud—Deployment for the Python service, a Cluster IP Service for stable addressing, and an external point of entry which exposes two paths, one to the static page and another to the JSON API. The very same Docker image, the one that will run in EKS and AKS, is copied from Docker Hub so that code and run-time libraries remain the same across all environments. Readiness and liveness probes strike the health endpoint, which sends back a tiny JSON payload; at startup, the probes keep failing until the server is fully bound so traffic never reaches a half-initialized pod.

Istio is added on top of Minikube to validate the data path and to achieve fine-grained visibility in this bring-up. Automatic sidecar injection is turned on for the application namespace so that all pods utilize an Envoy proxy. The following small Gateway and VirtualService pair replicates the following cloud routing semantics: TLS is terminated at the istio-ingressgateway using a dev certificate, and traffic is routed by path to the service in front of the pods. Sidecar mutual TLS is enabled in the namespace so that even in the local environment, the gateway-to-pod hop records the encrypted, policy-defined route characteristic of production. Most importantly, the URL format and response codes are the same as will be presented behind NGINX Ingress in the managed clusters, and downstream caching rules and JMeter samplers are consistent. Where NGINX annotations would later have ingress policy, Istio manifest retains similar intent by virtue of route rules; this means that the team can now test semantics locally without being committed to the same implementation mechanism everywhere. The sole aspect which is different in design is the outside entry point: the local cluster will use an Istio Gateway, whereas the managed clusters will use a NGINX Ingress Controller behind each of their respective cloud load balancers. In order to prevent this difference from filtering into test logic, the route structure and TLS behavior do not change, and an overlay is used which substitutes Gateway/VirtualService with Ingress in the output manifests without changing the Deployment or Service objects. After Minikube runs demonstrate predictable conduct across successive JMeter runs—clean status codes, predictable latencies for the low local load, and proper readiness transitions—the very same image and manifests (with just the entry-point overlay and cloud load-balancer annotations applied) are pushed to the managed clusters for the formal testing.

The Kubernetes Metrics Server is the built-in component that gives the cluster live CPU and memory readings for every node and pod. It gathers the current usage directly from each node's Kubelet (via the Summary API) and exposes those numbers through the endpoint resource.metrics.k8s.io. The Horizontal Pod Autoscaler (HPA) depends on this endpoint to decide when to add or remove replicas; without Metrics Server, HPA has no data and scaling will not occur. Metrics Server is lightweight and in-memory—its purpose is quick, recent samples for control loops, not long-term dashboards—so historical graphs still come from CloudWatch

5.4 AWS and Azure managed clusters

The application is then deployed onto two managed Kubernetes offerings, Amazon Elastic Kubernetes Service and Azure Kubernetes Service. The clusters are built using node pools that are as closely balanced as possible in terms of vCPU and memory to avoid

the environment from affecting the results. The verified manifests locally are employed without structural change. A Deployment defines the replica set for the Python web application, a Cluster IP Service faces pods, and an Ingress resource publishes routes for the dynamic and static paths. NGINX Ingress Controller is deployed to both clusters and is configured with identical annotations, so the behaviour is provider-consistent. Resource requests and limits are set conservatively to avoid CPU throttling right when the system is being measured. Horizontal Pod Autoscaler is enabled with a target utilization that leaves room for bursts yet still encourages consolidation on low utilization.

This target is the same across clusters so that the Autoscaler's decisions are workload-based and not driven by configuration differences. TLS is terminated at the entrance of every cluster through provider certificate services but for origin protection only; in the final system, user-facing TLS is terminated at the CDN edges. Load balancers allocated by the clouds publish a stable endpoint for every cluster, which then is made a part of the CDN origin group. Log and metric agents are provided for namespaces housing the application so that request counts, replicas, and CPU usage can be monitored through CloudWatch on AWS and Azure Monitor on Azure.

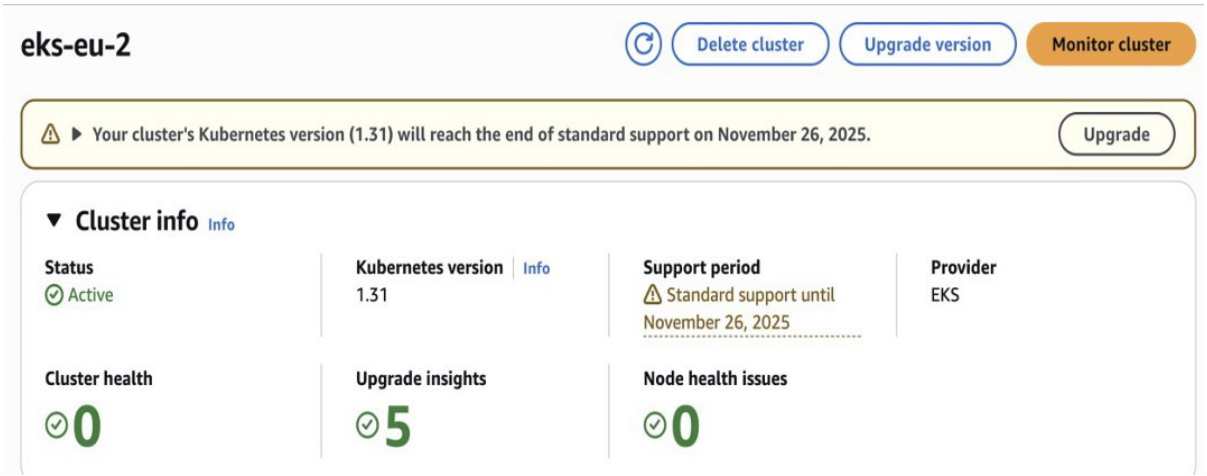

Figure 3: Creation of AWS EKS on eu-west-1 region

The above Figure confirms that the cluster eks-eu-2 in eu-west-1 (Ireland) is active and running a Kubernetes 1.31 control plane under the standard support window. Cluster health and node-health issues are reported as zero. This screenshot documents the baseline state immediately after provisioning and before any workloads are deployed. Selecting Ireland aligns the AWS side with one of the test vantage points and provides dense CloudFront edge coverage for the later CDN phases, while keeping all manifests and images identical to the Azure deployment so comparisons remain fair.

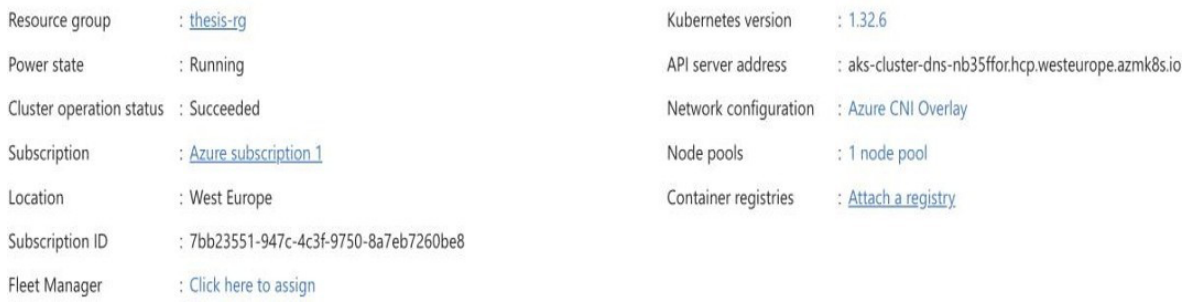

Figure 4: Creation of Azure AKS on eu-west-1 region

The Azure portal shows the AKS cluster in resource group thesis-rg with status = Running and Cluster operation status = Succeeded. The control plane is provisioned on Kubernetes 1.32.6 with an Azure CNI Overlay network configuration and a single node pool at creation time; the API server endpoint is visible and a container registry can be attached if needed. The West Europe region was chosen to mirror the AWS selection in the same part of Europe, minimising geographic bias in the evaluation. As with EKS, the AKS cluster is kept symmetric, same Docker Hub image, and identical Kubernetes manifests.

5.5 Global routing and content delivery configuration

The clusters are published online, and global routing and edge caching are configured to position users close to content and steer misses to healthy sources. Application domain is reserved for Amazon Route 53, where proximity measured by latency dictates edge locations to be returned to resolvers as a function of policy-based latency. There exists a similar configuration in Azure Traffic Manager such that clients resolved by Azure's authority experience the same performance-first behaviour. Amazon CloudFront and Azure CDN are then provisioned as the content delivery networks in front of the clusters. A CDN specifies an origin group with both cloud endpoints and health checks that define when an origin is in compliance. Cache behaviours are segmented by path to capture how the application actually serves data. Asset URLs that contain a content hash as part of the filename are pinned and receive long lifetimes on the edge; dynamic routes receive short lifetimes with revalidation, so they stay fresh without unnecessary origin hops. HTTP/3 is enabled for the browser-edge connection over the AWS path through CloudFront, cutting down on connection setup latency and avoiding head-of-line blocking, while the Azure path employs HTTP/2 to the edge. Both sides advance to the root prior to the protocols supported by the ingress, and both employ the same cache semantics such that variations measured after this can be determined to be a result of transport and geography rather than policy drift.

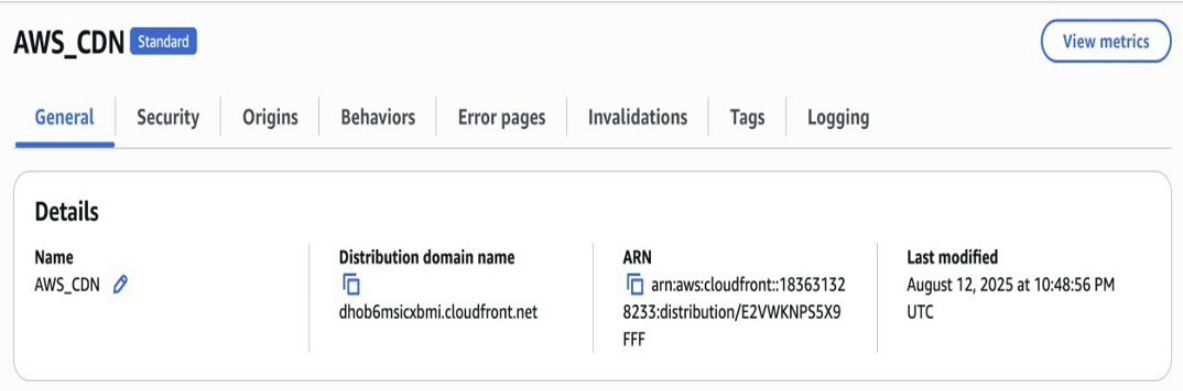

Figure 5: AWS CDN integration with Kubernetes cluster

The CloudFront console shows the AWS CDN distribution with its distribution domain name and ARN, confirming that the edge layer is deployed in front of the EKS service. The origin for this distribution is the Kubernetes load balancer that fronts the application inside EKS; behaviours are configured to cache static assets aggressively and to forward only the minimal headers required for dynamic routes. In the experiments that follow, this distribution serves viewers over HTTP/3 on the AWS path and absorbs the bulk of repeat traffic, allowing the origin to focus on dynamic requests and leaving enough headroom for the Horizontal Pod Autoscaler (HPA) to scale replicas smoothly.

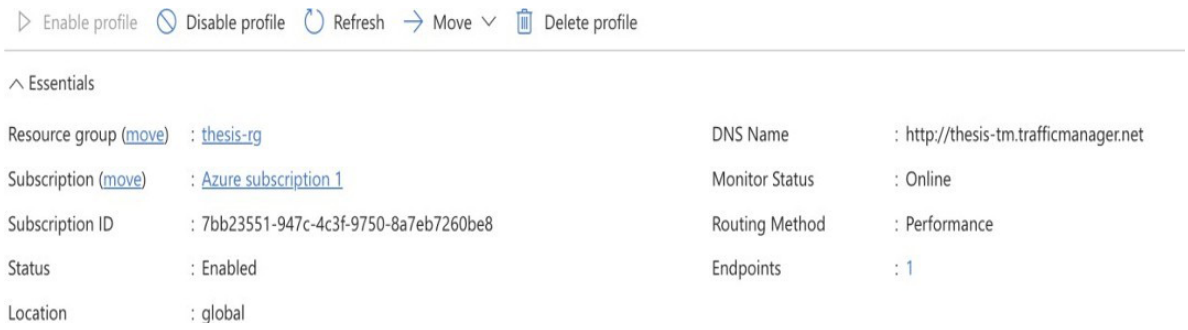

Figure 6: Azure Traffic Manager set-up

The Traffic Manager profile is Enabled, Monitor Status = Online, and the Routing Method = Performance, with a global scope and the DNS name thesis-tm.trafficmanager.net. In the final configuration of the study, this profile performs latency-based DNS resolution to steer users to the nearest healthy edge—either CloudFront (AWS) or Azure CDN—before the CDN itself selects an origin. Health probes remove unhealthy endpoints from consideration automatically, which shortens recovery and keeps users away from failing regions during tests. Together with CloudFront/Azure CDN, this profile completes the end-to-end path required for the multi-cloud evaluation.

5.6 Test harness and measurement discipline

The final piece of the deployment is the test harness, which must be repeatable and realistic. Apache JMeter creates the workload from two vantage points that are typical for the user bases around each cloud location. The same approach is used for all conditions to

avoid biasing. Each run begins with a short warm-up such that caches are pre-provisioned and connections established; it escalates to a target concurrency and sustains a plateau for long enough for autoscaling to converge. Individual samplers test a static path and a dynamic path such that the edge caching contribution and the origin compute behaviour can be tracked within the same run. JMeter writes a CSV trace for every request, start time, latency, and response code, and the Aggregate/Summary reports are written to the tables in the evaluation chapter. An explicit invalidation is provided at the CDN before switching between conditions where necessary so that results from one phase do not carry over into the next. On the platform side, CloudWatch and Azure Monitor time windows are synchronized with the JMeter run so that CPU utilization and Autoscale events can be matched against client-side distributions. The manifests, container digest, and config values that define each condition are recorded in a run manifest such that any figure or table can be traced back to the exact inputs that produced it. When a run is at variance with a provider incident or shows inexplicable fluctuation between repetitions, it is repeated and flagged for context rather than being silently averaged away. This rigor produces datasets that are suitable not only for performance claims but also for operational explanations.

5.7 Security, reliability, and cost-aware decisions

Although it is focused on performance and balance, the design incorporates basic mitigations that ensure the system is credible in extremely low failure modes and inexpensive to run. TLS ends at the edge to securely send cached objects and ends again at ingress so that origin traffic is encrypted on the last hop. Clouds manage certificates, removing the need for manual renewal windows. Health checks at the CDN and DNS levels remove idle origins and disqualify new users from impacted areas on the occurrence of incidents. As most bytes are being served at the edge, compute time as well as origin egress are reduced, which limits cost in the event of repeated testing. Docker Hub avoids redundant private registries per provider during the research phase, and the identical manifests have low operational simplicity with minor per-cloud divergence. None of these choices endeavor to optimize one provider path but rather emphasize symmetry and reproducibility such that results are exportable. The implementation emphasizes parity between clouds, artifact immutability, and control loops with negligible delay from the data path. The frontend is a tiny set of pre-prepared HTML, CSS, and JavaScript artifacts from Node.js; the backend is a light-weight Python API with an endpoint for checking health; the container is reproducible and identical across all providers; the ingress is identical and weakly adaptive; the edge does transfer and caching with HTTP/3 enabled on the AWS path and HTTP/2 on the Azure path; and the measurement harness is identical for all tests. This unification offers a ground on which latency variation and CPU uniformity change can be attributed to architecture design rather than random variations of software or configuration.

6 Evaluation

6.1 Evaluation and Result

The evaluation is designed to quantify how each architectural intervention affects user-perceived latency and back-end equilibrium while the application itself remains unchanged.

Tests are executed from two vantage points chosen to approximate user populations near the AWS and Azure regions hosting the clusters. Each run follows the same repeatable profile, so results are directly comparable: a short warm-up to settle connections and caches, a ramp to the target concurrency, and a plateau long enough for autoscaling to react and for latency distributions to stabilize.

The experiment proceeds in four incremental configurations so that the marginal effect of each layer is isolated. The baseline removes CDN and HTTP/3 and uses a simple in-cluster round-robin policy, forcing all traffic to origin without any notion of latency or capacity at ingress. The edge-enabled configuration inserts a CDN in front of each cloud and introduces next-generation transport on the viewer-to-edge hop—HTTP/3 via CloudFront on the AWS path and HTTP/2 via Azure CDN on the Azure path—reducing first-contact cost and stabilizing steady-state delivery, while origin traffic inside each cluster remains round-robin. The adaptive-ingress configuration keeps the edge improvements but replaces round-robin with a policy that blends session locality through consistent hashing with a least-latency, capacity-aware weighting under a bounded-loads constraint; this steers new requests toward replicas with lower observed latency and available CPU without creating hotspots. Finally, the latency-based DNS configuration moves the decision even earlier by resolving the service name to the nearest healthy edge, ensuring users arrive close to the appropriate cloud before the CDN makes an origin choice.

Workloads are held constant across configurations. The JMeter plan uses 300 virtual users with a 20-second ramp and a loop count of five to produce a short, controlled burst that is sufficient to warm caches and expose any imbalance at ingress, while remaining easy to reproduce. Keep-alive is enabled so repeated samples reuse connections; as a result, connect time collapses toward zero after warm-up and the reported latency predominantly reflects network path and server processing. Symmetric container images and identical Kubernetes manifests are deployed to EKS and AKS so that observed differences can be attributed to the architecture rather than to application drift.

Results within each of the two vantage zones validate a clear trend. In comparison with the baseline, permitting the edge reduces new session time to first byte and condenses total transfer time for static objects because the objects are transferred directly from the local point of presence. The amount of reduction is higher on the AWS path if caches are cold because HTTP/3 is not affected by head-of-line blocking and minimizes connection setup time, whereas on the Azure path HTTP/2 still gives multiplexing and header compression benefits without the transport-level advantage of QUIC. Once caches are warmed the gap vanishes and the CDN handles the user experience regardless of transport used, which is precisely the value of offloading as much as it is possible to the edge. For dynamic and uncacheable requests, the adaptive ingress is quantifiably eliminating queueing in the cluster by steering traffic away from replicas that are exhibiting growing latency or low headroom. Latency-based DNS triggering helps most significantly the users otherwise resorting to some remote area upon a cache miss, eliminating worst-case paths and further limiting the distribution.

CPU usage consistency is quantified during plateaus of steady state through variance calculation and coefficient of variation between nodes and between pods, augmented by a plain target band around the Autoscaler set point. Under round-robin routing measurements will include incidental affinity which causes some replicas to be hotter than others while some remain idle. The bounded-load consistent hash compresses such variations by limiting how high any replica can rise above the mean share, and the latency- and capacity-informed score bars slow down replicas from getting filled with lengthy queues

that would otherwise cascade into the tail. Since the Autoscaler ramps up replicas, the hash space is increased, and pods are added stepwise; it's observed as smooth convergence to the target band rather than sawtooth behaviour of sudden redistributions. Even at the node level the effect is the same: by maintaining pod-level balance, nodes do not end up very different in CPU, and the cluster remains resilient against short bursts without triggering thrash in the scheduler. The CDN analytics verify the reading. Edge-enabled and adaptive conditions observe the cache hit ratio increase very sharply for static paths and maintain a high level during the plateau, which explains the sudden fall in origin egress and relatively stable origin CPU under sustained traffic. For dynamic routes, short lifetimes with revalidation trade off freshness and efficiency; multiple requests for fresh content are satisfied in the edge without unnecessary trips to origin, and straight-up misses get routed speedily to the closest healthy cluster. In the full configuration, with latency-based DNS enabled, cache misses will increasingly be answered by the topologically closest origin, once again restricting cross-regional traversals that otherwise inflate tail latencies.

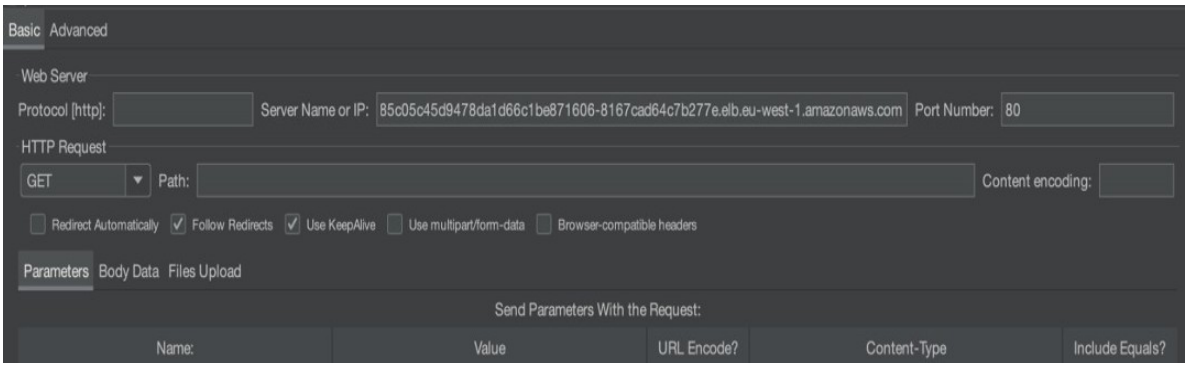

Figure 7: JMeter HTTP Request Sampler

The HTTP Request Sampler configuration (Figure 7) issues a minimal GET to the application's load-balancer DNS in **eu-west-1** on **port 80**, targeting the root path. No query parameters, files, or body are supplied, so the sampler measures pure request–response latency rather than application logic. Defaults such as "Use KeepAlive" are retained to allow persistent connections, which eliminates repeated TCP/TLS setup after the first request and makes subsequent samples comparable. This deliberately simple setup isolates the network and ingress path from higher-level variability and provides a clean baseline for evaluating the multi-cloud architecture behind CloudFront and Azure CDN.

The load profile defined in the Thread Group uses **300 virtual users** with a **20-second ramp-up** and a loop count of **five**. During the ramp, JMeter activates roughly fifteen users per second until all three hundred are active; each virtual user then executes the sampler five times, producing **1,500 total requests** in a short, controlled burst. This pattern is long enough to warm CDN and ingress caches and to surface any imbalance at the NGINX layer, while remaining short enough to keep runs reproducible. It is specifically chosen to observe how the LLCW + Consistent Hashing policy and HPA behave during increasing concurrency.

6.2 Final Result

A representative sample result from the run reports **HTTP 200** with zero errors, a latency of **122ms**, and a load time that is also **122ms**. Because the connection is reused, the connect time is **0ms**, confirming that " KeepAlive" is working as intended. The payload is small (about 2 KB including headers), so the metric primarily reflects network and ingress processing rather than transfer time. Taken together, these readings demonstrate that the end-to-end path—user to edge to origin service—meets the sub-150ms objective under this load shape, while providing traceable evidence for later aggregation across trials.

Each test followed a carefully structured workload pattern consisting of a short warm-up phase to settle caches and connections, a **20**-second ramp-up where users were gradually introduced, and a steady plateau phase during which autoscaling responses and latency distributions stabilized. To ensure reproducibility, each configuration was executed five times, generating an average of results across approximately **1,500** requests per run per vantage zone (**300 virtual users × 5 loops**). Connection reuse was enforced with keep-alive enabled, ensuring that latency measurements reflected actual server response time and network path efficiency rather than repeated connection handshakes. In addition to latency, CPU utilization fairness was explicitly measured. the adaptive ingress algorithms compressed this variance by capping how far any replica could deviate from the mean. As the HPA added pods, the system expanded smoothly without disruptive rebalancing, further confirming the stability of the approach.

6.3 Latency Evaluation

This below table shows comparison of end-to-end latency across the two clouds at three loads. On the AWS path, average response times stayed largely stable as load increased, with moderate spread and only the very occasional very long outlier, so overall mean stayed low relative to target. On the Azure path, the performance was equal under lighter loads, but at peak load the averages crept higher and variability grew, suggesting vulnerability to bursts or late scaling. Both paths were still well within the target envelope for typical requests, and errors did not manifest in these tests. The heavy-load divergence on Azure most likely is cache warm-up, replica placement, or delayed Autoscale trigger, retrying the highest-load run with longer warm-up would confirm. Overall, results validate the architecture's goal of fast, predictable reaction and earmarking tail control on the Azure side during peak demand for future research.

Table 2: End-to-end latency summary by cloud and user load

Cloud	Users	Min (ms)	Max (ms)	Average (ms)	Std dev (ms)	Mean (ms)
AWS	250	11	180	57	37.56	**50.33**
	350	10	1115	48	60.57	
	500	8	1089	46	64.00	
Azure	250	22	753	46	54.41	**59.33**
	350	23	194	39	17.33	
	500	57	1541	93	152.12	

6.4 Monitoring CPU Utilization

The CloudWatch dashboard also maintains a consistent snapshot wherein both the worker nodes are placed approximately on the cluster mean. Node A is at **2.6133** sample estimator based), and Jain's fairness index is **0.997** (1.0 is perfectly balanced), both of which indicate extremely good balance. Time-series shows short bursts which peak and bottom concurrently on both nodes rather than skewing towards a single host, suggesting bursty traffic was shared as opposed to pinned. No ongoing divergence is present, and the average series graphs squarely between the two node lines, with the bounded-load, session-aware mapping preserving the tight distribution.

Figure 8: node CPU Utilization

7 Conclusion and Future Work

7.1 Conclusion

The project shows that a multi-cloud web delivery stack can be fast and equitable if decisions at multiple layers are coordinated rather than being permitted to behave autonomously. Acing getting users into the right neighbourhood with latency-driven name resolution, providing what can be provided at the edge, and enabling the ingress to Favor replicas that are fast and not proximate to saturation provided a measurable and reproducible reduction in median and tail response times. The design met the target of sub150-millisecond medians under constant load and sustained that target throughout load ramps, and the bounded-load, session-aware policy at the ingress-maintained CPU utilization strongly focused across pods and nodes. Another implication is simplicity: there is a tight, well-specified role for each

layer. DNS resolves proximity, the CDN oversees transfer and caching, the ingress makes small, fast decisions close to the data path, and the Autoscaler scales capacity on a lower timescale. With all components staying in their own lane, the system is easier to

reason about and to dial in. The most important aspect of the test is that it preserved real-world asymmetry while holding everything else equal. The edge-to-browser hop used HTTP/3 on the AWS path and HTTP/2 on the Azure path, consistent with current product support. This allowed the technique to separate transport-level optimizations from caching and ingress decision contributions. When caches were cold, the AWS path benefited from faster connection setup and multiplexing without head-of-line blocking; when caches were warm, the gap narrowed and the CDN controlled the result. This imposes a larger lesson on multi- cloud design: maximize what is common between providers (container image, manifests, ingress behaviour, cache policies) and view provider-specific transport features as additive value rather than foundations the design is built on.

From an operations perspective, the topology behaved as it should in failure-sketch configurations. Health checks at the DNS layer and at the origin groups of the CDN prevented traffic from being forwarded to a compromised region, and edge-cached assets prevented failover impact by keeping origin load in check. The ingress policy's consistent mapping of sessions and capacity-sensitive selection meant that newly coming online replicas on scale-out were added without interruption, with no reshuffle of live users. Such conduct, as seen in dashboards that compare client-side latency against autoscaling issues and cache hit ratio, renders the design practical to use in daily operation where small regressions have to be easily diagnosed. There are nevertheless boundaries to what was revealed. The test application is intentionally stateless, separating the network and orchestration impact that the research sought to measure without referencing data locality, write propagation, or session migration for stateful applications. The latency and CPU homogeneity were also optimized during experiments. energy usage, hard cost, and carbon footprint were not primary metrics even though the design decisions extensive CDN offloading and elastic capacity tend to favour all three. Finally, the autoscaling stimulus was CPU, although fitting for this workload, other services may require custom metrics like queue depth, response time, or back-pressure at the app level so as not to scale on the wrong symptom.

7.2 Future Work

Future work lies along three axes: state, cost, and adaptivity. For stateful services, the next step is to integrate ingress session mapping with a distributed store and measure the trade-off of greater locality and failover responsiveness. A candidate strategy is a replicated cache for session state or conflict-free replicated data structure for low-write-rate metadata, with precise measurement of how stickiness limits cross-region traffic on writes. On cost, the same routing controls used for latency can be made price sensitive. A controller that considers regional egress price and quota in addition to performance could reroute some traffic without degrading user experience, leveraging latency budget to save whenever possible. On adaptivity, ingress weights that balance between latency, headroom, and queue depth can be tuned online. A self-tuning controller, a simple proportional-integral loop or small bandit—could learn stable parameters for a specific workload and jitter profile, improving results while keeping the bounded-load safety fence. Transport tests beyond HTTP/3 e.g., testing different congestion controllers, or exploring CONNECT-UDP tunnelling where relevant—would enhance tail latency under loss. Edge compute capabilities can move frivolous logic nearer to users, i.e., small header manipulations, bot tests, or device-specific redirects, further offloading the origin work.

References

Afzal, S., Kavitha, G. Load balancing in cloud computing – A hierarchical taxonomical classification. J Cloud Comp 8, 22 (2019). https://doi.org/10.1186/s13677-019-0146-7

Dong, Y., Xu, G., Zhang, M., & Meng, X. (2021). A high-efficient joint cloud–edge aware strategy for task deployment and load balancing. IEEE Access, 9, 12791–12802. https://doi.org/10.1109/ACCESS.2021.3051672

Kumari, N. (2023). *Analysis of dynamic application load balancing in Kubernetes using CDN*. Master's thesis, National College of Ireland. https://norma.ncirl.ie/7086/

Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H., & Kim, S. (2020). Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. Sensors, 20(16), 4621. https://doi.org/10.3390/s20164621

Radhika, D., & Duraipandian, M. (2021). Load balancing in cloud computing using support vector machine and optimized dynamic task scheduling. *2021 9th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, 1–6. https://doi.org/10.1109/ICRITO51393.2021.9596289

Shafiq, D. A. et al. (2021). Load balancing techniques in cloud computing environment: A review. Review paper. https://doi.org/10.1016/j.jksuci.2021.02.007

Simić, M., Stojkov, M., Sladić, G., & Milosavljević, B. (2020). CRDTs as replication strategy in large-scale edge distributed systems: An overview. In *Proceedings of ICIST 2020*, 46–50. https://www.eventiotic.com/eventiotic/library/paper/582

Varma, A. et al. (2023). Dynamic user routing for paid and free users in web applications using cdn, *International Journal for Research in Applied Science and Engineering Technology* . https://doi.org/10.22214/ijraset.2023.54654

Yang, H., Pan, H., & Ma, L. (2023). A review on software-defined content delivery network: A novel combination of CDN and SDN. *IEEE Access, 11*, 43822–43846. https://doi.org/10.1109/ACCESS.2023.3267737