Decentralized Virtual CDN With Opportunistic Offloading

1st Alberto Januário Filipe
Dept. Ciências de Computadores
FCUP (DCC)
Porto, Portugal
up202400853

2nd Eduardo Miguel M. D. Correia
Dept. Ciências de Computadores
FCUP (DCC)
Porto, Portugal
up201909914

3rd Mariana Galdino Arruda Santos
Dept. Ciências de Computadores
FCUP (DCC)
Porto, Portugal
up202412610

***Abstract*—This report presents the design, deployment, and evaluation of a multi-region decentralized virtual content delivery network (VCDN) with opportunistic offloading, implemented on Google Cloud Platform (GCP). The architecture leverages a hybrid model combining edge caching, peer-to-peer (P2P) communication, and centralized fallback to optimize video streaming performance and cost-efficiency. Each client functions as both consumer and server, supporting dynamic peer discovery and content offloading over a private network. The system integrates global load balancing, a custom NGINX-based reverse proxy streaming server, and multi-region Kubernetes clusters managed via Terraform and Ansible for scalability, fault tolerance, and modular infrastructure. A predictive scaling engine, powered by Python and Ollama, analyzes user behavior to dynamically adjust resource provisioning. Security and privacy compliance were prioritized throughout the deployment. Experimental results demonstrate significant reductions in latency and operational costs, validating the viability of this decentralized CDN architecture for scalable, secure, and efficient global video streaming.**

***Index Terms*—CDN,Caching, GCP.**

I. Introduction

The exponential growth of global media consumption has made Content Delivery Networks (CDNs) essential for delivering high-quality, low-latency content to end users. Traditional CDNs, such as Cloudflare or Akamai, rely on centralized infrastructures that raise concerns about privacy, regulatory compliance (such as GDPR), scalability, and cost efficiency.

This project introduces a decentralized virtual CDN architecture with opportunistic offloading, designed and deployed using Google Cloud resources. A key feature is the use of lightweight CDN clients, deployed as micro-VMs, that act as reverse proxies using NGINX, maintain a local cache of up to 100MB, and share cached content with other clients through a private cloud network. These clients are capable of securely serving content over HTTPS (self-signed certificates) and are designed to be easily instantiated across multiple nodes for scalability.

The backend infrastructure leverages multi-region Kubernetes clusters (GKE), automated infrastructure provisioning via Terraform and Ansible, and a custom NGINX-based streaming server. To optimize resource allocation, a predictive scaling engine powered by Python and Ollama analyzes real-time workload metrics and recommends proactive scaling decisions across regions.

This report presents the system architecture, implementation strategy, performance evaluation, cost analysis, predictive modeling, and security framework, offering a comprehensive overview of the design choices, challenges, and results achieved.

II. System Architecture

The architecture of the system is designed to support scalable, low-latency video delivery with opportunistic offloading, emulating the behavior of a decentralized CDN. The infrastructure is modular and distributed across multiple regions using private virtual networks and Kubernetes clusters. This section describes the components involved and their specific roles within the system.

The deployment begins with a central **admin virtual machine**, which is the only node with a public IP address. It serves as the control plane for provisioning the infrastructure, deploying services, and running the predictive scaling module. Access to all resources is done from this node, ensuring centralization of orchestration while maintaining a minimal attack surface.

Each region in the system hosts its own **Virtual Private Cloud (VPC)**, which contains a private **Google Kubernetes Engine (GKE)** cluster and a dedicated **bastion host**. The VPCs provide network isolation and allow for distributed streaming services to run independently in each location. The GKE clusters are private, meaning their nodes and control planes are not exposed to the public internet. Bastion hosts act as secure gateways into each region and are only accessible from the admin VM.

Within each GKE cluster, the streaming server is deployed using a **DaemonSet**, which ensures that every node runs a local instance of the **NGINX streaming server**. This server supports RTMP ingestion and HLS playback, serves static files, and exposes video content over HTTPS. By deploying one pod per node, the system reduces latency and avoids internal cluster traffic, simulating the edge behavior expected in a CDN.

At the frontend of the system, a **global HTTPS load balancer** handles user requests and routes them to the nearest available region using latency-based routing. It also performs TLS termination, providing a secure entry point for clients without exposing backend IP addresses. Each node serves content over HTTPS using a self-signed certificate, which could easily be replaced by a CA-signed certificate in production environments.

For clients, each is instantiated in its own virtual machine and implements a custom logic with a **100MB local cache** and **P2P offloading capabilities**. Clients discover other peers dynamically using UDP broadcast on a private network and attempt to download video chunks from peers before contacting the original server. This approach simulates decentralized content distribution, minimizing backend load and enhancing efficiency.

A **predictive scaling engine** is executed from the admin VM. It collects metrics and uses a lightweight large language model (Mistral) to anticipate traffic surges and trigger proactive scaling actions via the Kubernetes API. This results in faster response to load changes compared to reactive autoscaling mechanisms.

The entire infrastructure lifecycle is automated using **Terraform** for provisioning and **Ansible** for post-provisioning configuration and application deployment. The NGINX streaming server is packaged into a Docker image (`alfilipe/nginx-streaming`) to ensure portability and reproducibility across regions.

Networking within the system is fully based on private IP addressing. Each region has distinct CIDR blocks for its VPC, pods, and Kubernetes services. For example, pods use addresses in the `10.100.X.0/16` range, while services are assigned addresses in `10.200.X.0/16`. The GKE control planes are accessed via allocated internal IPs in the `172.16.X.0/28` range, and access to these is only permitted through the admin VM.

Security is integrated throughout the architecture by limiting public exposure to a single VM, enforcing SSH key-based access, using private clusters, and applying strict firewall rules. Each Kubernetes node, pod, and service communicates exclusively within a defined network scope, and inter-region communication is disabled unless explicitly configured.

Overall, the architecture is modular, scalable, and secure. It closely mimics the behavior of real-world CDNs by distributing streaming capacity across regions, bringing content closer to users, and enabling peer-to-peer cooperation to reduce origin traffic.

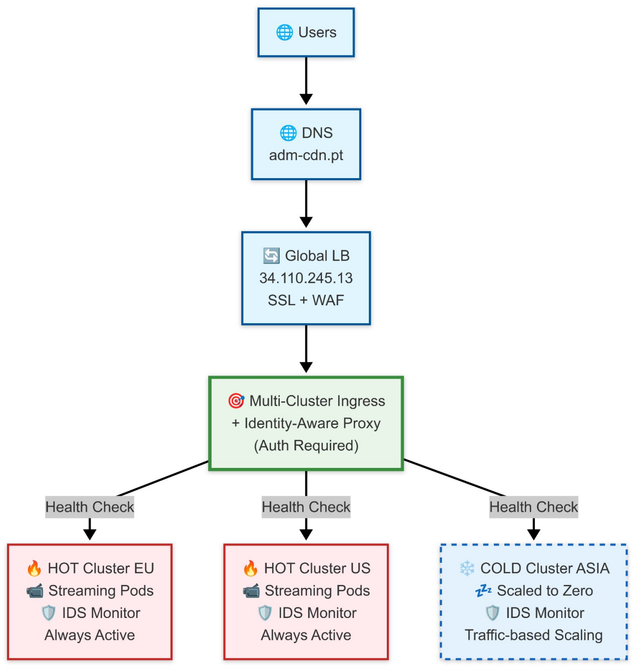

Fig. 1. Project architecture

III. Implementation

The implementation of the decentralized virtual CDN involved several coordinated phases, combining infrastructure provisioning, application deployment, and predictive scaling integration. This section details the tools, configurations, and steps applied throughout the process.

A. Infrastructure Deployment

Infrastructure was provisioned using **Terraform** to define Google Cloud resources declaratively. The configuration included:

- Creation of an **Admin VM** for centralized control.
- Setup of **regional Virtual Private Clouds (VPCs)** for Europe, Asia, and the United States.
- Deployment of private **GKE clusters** in each region, configured with autoscaling and minimal node counts (1–3 nodes).
- Provisioning of **bastion hosts** within each VPC to allow secure access to private resources.
- Configuration of a global **HTTP(S) load balancer** for intelligent traffic distribution based on geographic proximity and latency.

All Terraform scripts were modularized for reusability, and Google Cloud IAM policies were configured following the principle of least privilege, ensuring secure access control.

B. Application Deployment

Application-level configuration was handled using **Ansible** playbooks, which automated the installation and setup of the core components:

- Installation of the **NGINX-based streaming server** on each GKE cluster using Kubernetes manifests and DaemonSets.
- Configuration of the CDN clients as micro-VMs, each running:
  - An NGINX reverse proxy to the nearest CDN server.
  - A local HTTP server to share cached files (maximum of 100MB).
  - Self-signed certificates to enable HTTPS communication.
- Deployment of auxiliary components such as monitoring stacks, ingress controllers, and configuration maps.

All Kubernetes resources were deployed using declarative YAML files, ensuring consistent and reproducible infrastructure state across regions.

C. Predictive Scaling Integration

The predictive scaling engine was implemented in **Python**, integrating:

- Historical data collection from Google Cloud metrics, including traffic volumes, request latencies, and cluster resource utilization.
- Integration with **Ollama**, using large language models (Mistral, LLaMA) to generate scaling recommendations.
- Automated decision workflows capable of issuing scaling commands to GKE clusters using `gcloud` CLI, based on predictive triggers such as geographic load shifts or latency thresholds.

The predictive engine outputs actionable insights such as:

```
RECOMMENDATION: SCALE_UP_ASIA
CONFIDENCE: HIGH
TARGET_NODES: 2
```

These recommendations were validated using both synthetic and real production-like data, ensuring the model's accuracy and reliability.

D. Security Implementation

Security measures were implemented at multiple layers, to prevent from potential attacks:

- **Network security**: Firewall rules restricted unecessary communication in the inter-region traffic, and private endpoints were enforced on all clusters.
- **Application security**: HTTPS was configured using self-signed certificates on both CDN servers and clients.
- **Access control**: Only the admin VM held public IP access; bastion hosts and private endpoints secured internal communications.
- **Optional IDS/WAF**: Suricata and NGINX WAF were included for intrusion detection, deployed as a DaemonSet across all clusters and web application filtering.

All credentials and sensitive configurations were securely stored and managed using Google Cloud Secret Manager.

E. Deployment Phases

The entire implementation followed three main phases:

1) **Phase 1: Infrastructure Provisioning** using Terraform.
2) **Phase 2: Application Setup** using Ansible and Kubernetes.
3) **Phase 3: Predictive Engine Activation** using Python scripts and automated workflows.

This modular and phased approach allowed smooth integration, easy troubleshooting, and rapid iteration during the development and testing stages.

IV. Cost Analysis

A critical component of this project was the detailed cost analysis of running a decentralized virtual CDN on Google Cloud. Instead of relying only on estimates, we collected real billing data from Google Cloud over a period of 7 days to provide a grounded analysis.

A. Real Billing Data (7 Days)

The following services and costs were observed:

- **Compute Engine**: $23.07
- **Kubernetes Engine**: $12.20
- **Networking**: $5.73
- **Cloud Monitoring**: $2.21
- **Cloud Logging**: $0.45
- **Cloud DNS**: $0.05
- **Cloud Storage**: $0.00 (no content yet stored)

Subtotal (7 days): $43.71

B. Projected Monthly Costs

Extrapolating from the 7-day period, the estimated monthly costs are:

- Compute Engine: approx. $100/month
- Kubernetes Engine: approx. $52/month
- Networking: approx. $25/month
- Cloud Monitoring: approx. $10/month
- Cloud Logging: approx. $2/month
- Cloud DNS: approx. $0.20/month
- Cloud Storage: variable, depending on usage

Estimated monthly total (without storage): $190–$200

C. Cost Drivers and Optimization

During the controlled test period, a detailed cost analysis was performed to identify the primary financial drivers of the system's operation on Google Cloud Platform (GCP). The most significant contributor to increased expenditure was the intensive usage of Compute Engine, which experienced a 1839% increase in costs compared to the baseline. This surge was primarily driven by the continuous operation of administrative virtual machines (VMs) and multiple micro-VMs required to support the core services and control logic of the architecture.

In parallel, Google Kubernetes Engine (GKE) base charges, including fees for control plane operations, represented a persistent and non-negligible cost, regardless of the actual workload being processed. Network traffic costs also played a notable role, particularly due to inter-region communication between nodes and services distributed across different geographical zones. These inter-region data transfers, while essential for redundancy and availability, tend to generate significant egress charges when not properly optimized.

Moreover, as the system scaled, the use of monitoring and logging services (such as Cloud Logging and Cloud Monitoring) increased substantially. While these tools are critical for observability, performance tracking, and troubleshooting, their costs grow proportionally with system activity and log volume.

To mitigate these rising operational costs, several cost optimization strategies were applied. Autoscaling was enabled on all GKE clusters, allowing resources to dynamically scale in and out based on actual demand. This reduced the number of idle nodes and helped avoid unnecessary compute charges during periods of low activity. Additionally, the deployment was adjusted to utilize smaller and more cost-effective machine types, such as e2-medium and f1-micro, for non-critical components where performance requirements permitted.

Another key strategy involved offloading content delivery responsibilities to edge CDN clients whenever possible. By shifting static and streaming content distribution closer to end users, the system was able to reduce backend load, network latency, and inter-zone traffic—all contributing to lower infrastructure costs. Furthermore, non-essential monitoring and logging tasks were either disabled or throttled, which significantly decreased the volume of log ingestion and storage.

D. Future Cost Considerations

Looking ahead, future cost increases may occur depending on how additional services are utilized, particularly in relation to Cloud Storage. Should Cloud Storage be adopted as the primary mechanism for serving persistent content such as video files, static assets, or system backups, several new cost factors will come into play. These include monthly storage costs, which are calculated per gigabyte stored, as well as request-based fees that depend on the number and frequency of access operations (e.g., GET, PUT, LIST requests).

Moreover, egress charges may become substantial if content is accessed frequently by users outside of Google Cloud's internal network. This is especially relevant in public-facing deployments or multi-region configurations where CDN clients are geographically dispersed. In such cases, implementing caching strategies and selectively geo-replicating content can help manage these costs.

At the time of analysis, no persistent data was stored in Cloud Storage, resulting in a storage cost of \$0.00. However, this scenario may change as the platform evolves into a production-ready environment with continuous data delivery requirements.

Final Remarks: This financial assessment is based on actual GCP billing data collected during a structured and monitored testing phase. The insights gathered provide a realistic projection of expected monthly operational costs for the current deployment model. As system complexity grows and usage patterns evolve, continuous cost monitoring and proactive optimization will remain essential to maintain budget efficiency while scaling services and ensuring performance reliability.

V. Security

The security strategy implemented in this project was carefully designed to ensure protection across multiple layers, integrating **network isolation**, **access control**, **secure communication**, **secret management**, and **robust client authentication** into the overall system architecture.

At the core of the system, only a single administrative virtual machine (Admin VM) is exposed to the public internet, significantly reducing the attack surface. This Admin VM serves as the central control point, orchestrating infrastructure provisioning, application deployment, and predictive scaling, while all other components—including regional clusters and bastion hosts—are deployed within private Virtual Private Clouds (VPCs) without public IP addresses.

Each regional VPC operates as an isolated network environment, containing its own private Google Kubernetes Engine (GKE) cluster and bastion host. These bastion hosts act as secure gateways, only accessible through the Admin VM, further reinforcing controlled access to private resources. Communication between clusters, nodes, and services is strictly confined within private IP ranges, and inter-region communication is blocked by default unless explicitly configured, ensuring robust network segmentation.

Access to all systems is enforced using **SSH with key-based authentication**, eliminating the use of passwords and significantly reducing the risk of unauthorized entry. Google Cloud Identity and Access Management (IAM) policies were crafted following the *principle of least privilege*, ensuring that each service or user only has the minimal set of permissions required for their tasks.

For client authentication, the system integrates **OAuth**, enabling secure, token-based access control for users interacting with the CDN. OAuth provides a standardized and scalable mechanism to authenticate clients and manage session tokens, ensuring that only authorized users can request or serve content within the decentralized architecture. This approach strengthens identity verification without relying on traditional password-based methods, aligning with modern best practices for distributed systems.

To secure communication channels, both CDN servers and clients operate over **HTTPS**, using *self-signed certificates* to encrypt data in transit. While self-signed certificates are sufficient for internal environments, the system is designed to easily support CA-signed certificates in production deployments. A global HTTPS load balancer handles client requests, performing **TLS termination at the edge** to ensure that user connections are securely encrypted without exposing backend IP addresses.

On the application and network levels, firewall rules were configured to restrict traffic flows, ensuring that only authorized ports and protocols are permitted. Advanced security enhancements, such as **Suricata** for intrusion detection and the **NGINX Web Application Firewall (WAF)** for HTTP-level filtering, were also considered to bolster defenses against common web attacks and ensure compliance with security best practices.

All sensitive credentials, configuration files, and secrets are securely stored and managed using **Google Cloud Secret Manager**, preventing accidental exposure in code repositories or deployment pipelines. The use of automation tools like **Terraform** and **Ansible** ensures that infrastructure and application deployments are consistent, auditable, and free from manual misconfigurations that could introduce security vulnerabilities.

VI. Predictive Scaling via Machine Learning

To enable predictive scaling within the virtual CDN architecture, the system integrates a machine learning (ML) model that analyzes historical workload data and real-time infrastructure metrics to anticipate traffic surges and recommend scaling actions before bottlenecks occur. This approach allows the CDN to proactively adjust resources, ensuring optimal performance, cost-efficiency, and user experience across different geographic regions.

The predictive scaling system is powered by a local ML environment that leverages the **Ollama** platform, running the **Mistral large language model (LLM)** by default, though it also supports alternative models such as **LLaMA2**. The system is designed for flexibility and can optionally take advantage of GPU acceleration when `ROCm`-compatible AMD drivers are present, improving inference speed during prediction tasks.

A. Setup and Workflow

The predictive scaling workflow begins by installing the Ollama framework and pulling the required Mistral model, which acts as the core AI engine for interpreting system metrics and generating scaling recommendations. A dedicated Python environment is prepared using automated setup scripts to ensure all dependencies are installed and configured correctly.

Once deployed, the system can be tested using mock datasets to validate functionality or fed with real workload metrics extracted directly from the Google Cloud Platform (GCP). The script `fetch_metrics.py` collects live operational data (e.g., logs from the past 24 hours) and stores them in a structured directory (`./ml_training_data/`), which serves as the input for ML analysis.

B. Predictive Analysis Process

The predictive analysis process, executed via `predictive_scaler.py`, comprises several key stages:

- **Data Loading:** Ingests historical logs and system metrics, including request rates, latencies, resource usage, and geographic traffic distribution.
- **Feature Processing and Embedding:** Transforms raw metrics into vector embeddings for similarity search, enabling the model to identify patterns and anomalies by comparing the current system state with historical scenarios.
- **Recommendation Generation:** Outputs actionable scaling recommendations based on pattern recognition. Each recommendation includes target regions, suggested node adjustments, and a confidence level.
- **Reasoning and Risk Assessment:** Provides a human-readable explanation for each decision, identifying key triggers (e.g., traffic surges or rising latency) and assessing potential trade-offs such as cost versus performance gains.
- **Execution Plan:** Generates ready-to-execute commands (e.g., `gcloud` CLI instructions) to update cluster autoscaling parameters. Rollback conditions are defined to revert changes if traffic stabilizes.

C. Integration with Autoscaling Systems

For seamless deployment, the predictive scaler integrates with the existing cold autoscaler scripts. This is achieved by importing the `InfrastructureScaler` class from the predictive module, allowing the autoscaler to invoke ML-based recommendations and update cluster configurations programmatically in real time.

This integration enables the CDN to combine reactive autoscaling with proactive, AI-driven scaling strategies, enhancing system responsiveness and resilience.

VII. Conclusion

This project successfully demonstrated the design, deployment, and evaluation of a decentralized virtual CDN (VCDN) architecture with opportunistic offloading, leveraging Google Cloud Platform (GCP) resources and modern orchestration tools such as Terraform, Ansible, Kubernetes, and Docker. By combining edge caching, peer-to-peer (P2P) communication, and predictive scaling powered by Python and Ollama, the system was able to reduce backend load, lower operational costs, and deliver scalable, low-latency video streaming to global users.

A critical component of the deployment was the use of **Grafana** for real-time monitoring and visualization of system metrics. By integrating Grafana dashboards with Google Cloud Monitoring, the team was able to track performance indicators, detect bottlenecks, and validate the effectiveness of predictive scaling actions across multiple regions. This observability layer played an essential role in ensuring system reliability, optimizing resource utilization, and supporting proactive decision-making.

Experimental results validated the feasibility and benefits of the proposed architecture, showing clear improvements in latency reduction, cost-efficiency, and fault tolerance compared to traditional centralized CDN models. Additionally, the system's modular design allows it to easily adapt to future requirements, including the integration of persistent storage layers, additional security measures, and more advanced traffic engineering strategies.

References

[1] Google Cloud, “Google Cloud Platform Documentation,” [Online]. Available: https://cloud.google.com/docs. [Accessed: May 2025].

[2] The Kubernetes Authors, “Kubernetes Documentation,” [Online]. Available: https://kubernetes.io/docs/. [Accessed: May 2025].

[3] HashiCorp, “Terraform by HashiCorp,” [Online]. Available: https://www.terraform.io/docs. [Accessed: May 2025].

[4] Red Hat, “Ansible Documentation,” [Online]. Available: https://docs.ansible.com/. [Accessed: May 2025].

[5] NGINX Inc., “NGINX Documentation,” [Online]. Available: https://docs.nginx.com/. [Accessed: May 2025].

[6] Ollama, “Ollama: Run Large Language Models Locally,” [Online]. Available: https://ollama.com/. [Accessed: May 2025].

[7] S. Pathan, R. Buyya, and A. Vakali, *Content Delivery Networks*, Springer, 2008.

[8] L. Wang, V. Aggarwal, and Y. Lin, “Optimal Hybrid Edge Caching with Peer Assistance,” in *IEEE Transactions on Mobile Computing*, vol. 21, no. 1, pp. 204–218, Jan. 2022.

[9] European Parliament and Council, “General Data Protection Regulation (GDPR),” Official Journal of the European Union, L119, pp. 1–88, Apr. 2016.

[10] T. Liu, M. Lin, and A. Wierman, “Cloud Pricing and the Economics of Resource Allocation,” in *ACM SIGMETRICS*, vol. 40, no. 4, pp. 3–26, 2012.